Sistemas Operativos

The FurfiOS Corporation

21 de agosto de 2022

Índice general

Este apunte fue hecho en base a las clases teóricas de los profesores Fernando Schapachnik y Rodolfo Baader del Segundo Cuatrimestre 2020, completando principalmente con [SGG18] y [TB18]. Otra bibliografía complementaria:

- Capítulo 5 de [Sta11].
- Capítulo 2.2, 3.7, 4, 5, 7, 8.4 de [HS08].
- Capítulo 8, 10 de [Lyn97].
- Capítulo 4, 5 de [Dow].
- Capítulo 6 de [Sta16].
- Capítulo 1, 4, 7, 12 de [Ste99].
- Capítulo 3, 10 de [LB98].
- Capítulo 13 de [Mau08].
- Capítulo 22 de [Com00]

Importante. Este apunte es una versión **recortada** y no incluye todos los temas de la materia. En el link siguiente está la versión completa (sin termnar)— incluyendo Memoria, E/S, FS, Distribuidos, Seguridad y Virtualización. Se agradece modificar, ampliar y/o corregir el apunte.

`https://www.overleaf.com/4969394927mgmkzxfzshrf`

Capítulo 1

Introducción

1.1. ¿Qué es un sistema operativo?

En los años 60's, para poder ejecutar un programa, no se podía simplemente cargar el programa en la computadora; sino que era necesario entregar las tarjetas perforadas con nuestro programa a un operador, y era éste quien se encargaba de colocarlas en la entrada del sistema. Este operador debía gestionar adecuadamente la E/S, así como también operar correctamente el hardware para que éste no se dañara. A partir de la figura de este operador físico, se deriva el concepto de **sistema operativo**.

Desde el punto de vista de los programas de usuario, el kernel es una capa transparente: las system calls se ven iguales y se comportan de la misma manera que cualquier otro procedimiento. Toman valores, y retornan resultados. Los procesos no saben si el kernel está corriendo o no. Tampoco saben qué parte de su espacio de direcciones virtuales está realmente en memoria y qué partes todavía están en disco. En cualquier caso, los procesos interactúan permanentemente con el kernel para pedir acceso a recursos del sistema: hacer E/S, comunicación con otros procesos, etc. Para todos estos propósitos, los procesos utilizan las funciones provistas por bibliotecas estándares que, a su vez, terminan llamando a las system calls del kernel. En última instancia, el kernel es responsable de compartir recursos y servicios de manera justa y eficiente entre procesos.

Es decir, el sistema operativo se encarga de proveer una **interfaz de programación**. Durante el desarrollo de software, necesitamos de un intermediario entre nuestro código y el hardware. Esta capa intermedia nos permite abstraernos de los detalles *de alta complejidad* del hardware. Además, el sistema operativo se encarga de **administrar los recursos** del sistema, controlando el acceso al hardware y coordinando su uso. Se necesita de un intermediario que se encargue de reservar los recursos, evitando un uso inadecuado de los mismos. Debemos ser capaces de resolver numerosos pedidos de las distintas aplicaciones (posiblemente conflictivos), para que podamos hacer uso de la computadora de forma eficiente, segura y justa (*fairness*).

En general, cuando hablamos de un sistema operativo, nos referimos al kernel del sistema junto con los drivers del sistema (Fig. 1.1). El **kernel** es la parte central el sistema operativo. Se encarga de las tareas fundamentales y contiene los diversos subsistemas que iremos viendo en la materia. Los **drivers** son programas que manejan los detalles de bajo nivel relacionados con la operación de los distintos dispositivos de E/S.

Hoy en día, las distribuciones de un sistema operativo vienen con muchas aplicaciones que no son esenciales para el funcionamiento del sistema que no son parte del sistema operativo. Entre ellas, podemos distinguir a las utilidades del sistema (*system programs*), que son programas específicos que suelen encargarse de monitorear u operar el sistema— por ejemplo, un intérprete de comandos.

Por último, tenemos a las aplicaciones de usuario, que incluye a todos los programas que no están asociados con la operación del sistema. Los programas de usuario puede interactuar tanto directamente con el kernel, como indirectamente a través de las utilidades.

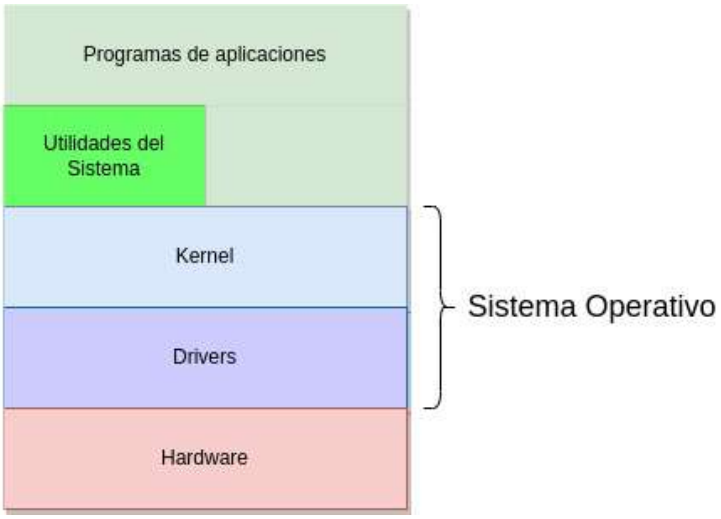

Figura 1.1: ¿Qué es un Sistema Operativo?

1.2. System Calls

Las aplicaciones ven al kernel como una gran colección de funciones que realizan una gran cantidad de servicios. Esto lo hace a través de las **system calls** (o llamadas a sistema). Estas *system calls* son herramientas que nos brinda el sistema operativo para poder solicitar acceso a algún recurso privilegiado, aprovechando las capacidades especiales del kernel. En particular, proveen una interfaz a los servicios que ofrece el sistema operativo, por ejemplo, realizar una escritura sobre un dispositivo de E/S. Las system calls suelen ser procedimientos escritos en C o en lenguaje de ensamblador.

Las system calls disponibles varían de un sistema operativo a otro, aunque la mayoría de conceptos detrás de las system calls suelen ser similares. Claramente, si escribimos un código en C que utiliza system calls que solo existen UNIX, no podremos compilar el mismo código en un sistema Windows. Buscando que los programas sean portables entre los distintos sistemas operativos, se creó un estándar llamado **POSIX**[1] (IEEE 1003.1/2008). Decimos que un sistema es POSIX compatible si implementa la interfaz de programación de aplicaciones (**API**) descrita en el estándar POSIX— no es necesario que las implementaciones concretas de la interfaz sean incluidas en el kernel. Este estándar define algunos servicios como la creación y control de procesos, pipes, señales, operaciones de archivos y directorios, excepciones, errores del bus, biblioteca estándar C, etc. Esta capa intermedia sirve para estandarizar y simplificar la administración de rutinas del kernel entre distintas arquitecturas y sistemas. Otras de las APIs más comunes son la API de Windows y la API de Java.

Normalmente, podemos acceder a una API a través del código de una biblioteca provista por el sistema operativo. A las funciones provistas por una biblioteca de usuario se las conoce como **library calls** (en oposición a las *system calls*). En el caso de UNIX para los programas escritos en C, la biblioteca es conocida como **libc**. Detrás de escena, las funciones que forman parte de la API suelen realizar system calls en nombre del programador de aplicaciones.

Otro componente importante al momento de administrar las system calls es el **run-time system**– todo el software necesario para ejecutar aplicaciones escritas en cierto lenguaje de programación, incluyendo compiladores o intérpretes al igual que otro software como bibliotecas y loaders. El run-time system provee una *sytem call interface* que funciona como enlace a las system calls disponibles. Esta interfaz intercepta llamados a funciones en la API e invoca las system calls necesarias dentro del sistema operativo. La mayoría de los detalles de la interfaz del sistema operativo son ocultas por la API y son administradas por el run-time system. Luego, al momento de desarrollar código de aplicaciones de usuario, no es necesario saber cómo se implementa cada system call particular. En cambio, tan solo necesita conocer la API y entender qué es lo que el sistema operativo hará como resultado de la ejecución de la system call.

[1] Portable Operating System Interface (for UniX).

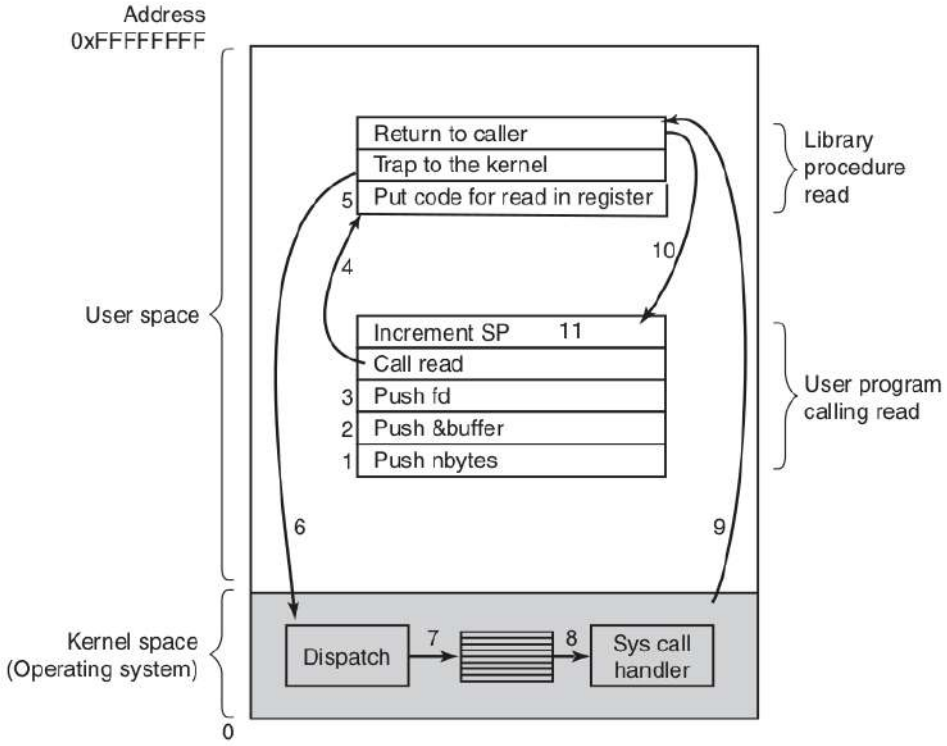

Figura 1.2: Los 11 pasos en realiza la operación read(fd, buffer, nbytes).

A continuación, vamos a dar una explicación más detallada sobre el proceso para llamar y ejecutar una system call. Supongamos que un programa de usuario quiere hacer la siguiente operación:

```
count = read(fd, buffer, nbytes);
```

Notemos que, en este código, `read` es una library call perteneciente a la libc (API POSIX) y no se trata de la system call `read`.

Antes de hacer un CALL a la función *read*, se pushean los parámetros de esta función en el stack (pasos 1-3). Los compiladores de C y C++ pasan los parámetros en el stack en el orden inverso a la firma de la función. El primer y tercer parámetro son pasados por copia, mientras que el segundo es pasado por referencia. Luego, se ejecuta CALL para llamar a la library call *read* (paso 4).

La library call, posiblemente escrita en lenguaje ensamblador, llama a TRAP, indicando el número de system call correspondiente a la system call *read* (paso 5). Esta instrucción cambia a modo kernel.

El código del kernel que le sigue a TRAP examina el número de system call[2] y luego salta al handler adecuado (paso 7). En ese punto, el handler asociado al número de la system call es ejecutado (paso 8). Una vez se haya completado, existen dos posibilidades. O bien se devuelve el control a la library call en user space, regresando a la instrucción siguiente a TRAP (paso 9) o bien se bloquea el proceso para que, eventualmente, sea despertado y continúe con su ejecución. Una vez se volvió a la library call *read*, ésta retorna al programa de usuario de la manera habitual (paso 10).

Para terminar el trabajo, el programa de usuario tiene que limpiar el stack, al igual que lo hace para cualquier llamado a función (paso 11). El programa ahora es libre de hacer lo que sea que quiera hacer. En la Fig. 1.2 se muestra un esquema que resume todos los 11 pasos recién descritos.

En el paso 9 de arriba, dijimos que la system call puede retornar inmediatamente o bloquear al proceso invocador, previniendo que continúe ejecutando. Por ejemplo, si un proceso intenta leer n bytes

[2]Cada system call es identificada por medio constante. No todas las system calls son soportadas en todas las arquitecturas, por lo que el número de system calls varía de plataforma en plataforma.

de un teclado, pero todavía no se presionó ninguna tecla, el proceso (en principio) será bloqueado. En general, podemos clasificar las distintas system calls de E/S en tres grupos: bloqueantes, no bloqueantes y asincrónicas. Decimos que una system call es **bloqueante** si suspende al proceso al momento de invocarla. Por otro lado, decimos que una system call **no bloqueante** si no detiene la ejecución del proceso por un tiempo extendido de tiempo y, en su lugar, retorna rápidamente con un valor que indica cuántos bytes fueron transferidos (potencialmente menos de los que se había pedido).

La tercer alternativa son las system call de E/S **asincrónica**. Una system call asincrónica retorna inmediatamente, sin tener que esperar a que la E/S se complete. El thread continúa ejecutando su código. En algún tiempo futuro, el sistema operativo le avisa al thread que se completó la E/S— escribiendo sobre alguna variable, enviando una señal, una interrupción o algún otro mecanismo. Normalmente, para enterarse de cuándo terminó la E/S, lo que se hace es devolver un identificador de la llamada y utilizar la syscall `select` para revisar (cada tanto) el estado de la E/S. Lo que hace esta syscall es decirle al proceso invocador si la llamada seleccionada bloqueará al proceso. De esta manera, un procedimiento de una biblioteca puede revisar si la llamada a *read* es o no bloqueante y, en base a esta información, decidir si la hace en ese momento o si espera para cuando los datos estén disponibles.

La diferencia con las system calls no bloqueantes es que éstas regresan inmediatamente con lo que haya disponible en el momento— todos los bytes pedidos, menos o ninguno. En cambio, una system call asincrónica realiza toda la transferencia, pero se completa en algún tiempo futuro.

1.2.1. Señales

Las señales son un mecanismo que incorporan los sistemas operativos POSIX, y permiten resolver una serie de problemas relacionadas con eventos asíncronos. La pregunta básica es ¿cómo puede un programa hacer su tarea y, al mismo tiempo, pueda responder a eventos inesperados de manera oportuna? Una **señal** es una notificación destinada a un proceso, indicando la ocurrencia de cierto evento. Para esto, las señales interrumpen la ejecución del programa, saltando a cierto código (*handler*) que se encarga de manejar el evento en cuestión.

Una señal puede ser generada por una interrupción de teclado, por un error en un proceso o por algún evento asincrónico. Además, contamos con la library call

```
int pthread_kill(pthread_t thread, int sig);
```

Esta función termina llamando a la system call `kill`, que nos permite enviar una señal a un proceso. Las *dispositions* son recursos compartidos entre threads de un mismo proceso: si un handler está instalado, será invocado desde el thread *thread*, pero si la *disposition* es una señal de *stop*, *continue* o *terminate*, afectará a todo el proceso.

Hay muchas señales distintas, cada una correspondiente a un evento distinto. Las señales pueden ser enviadas por un proceso a otro proceso (incluso a sí mismo) o del kernel a un proceso. Por ejemplo, cuando un proceso termina, el kernel le envía una señal SIGCHLD al proceso padre. Si el proceso padre estaba esperando a que termine su hijo, por anteriormente haber llamado a `wait` (o similares), esta señal lo termina despertando. Aprovechando el mecanismo de señales, es posible modificar el handler asociado a SIGCHLD para que el proceso padre, al momento de recibir la señal SIGCHLD, pueda ejecutar `wait` e inmediatamente continuar.

El modelo básico de señales en UNIX es que, al principio del programa, se declare cuáles van a ser los handlers para cada tipo de señal. Si no se declara ningún handler para cierto tipo de señal, se utiliza el handler por defecto específico para esa señal. Podemos configurar el handler de un tipo de señal llamando a

```
int sigaction(int signum,
    const struct sigaction *act,
    struct sigaction *oldact);
```

La system call `sigaction` nos permite cambiar la *acción* (o *disposition*) que se asocia a cada señal específica de un proceso.

- *signum* especifica el número de señal y puede ser cualquier señal válida, salvo por SIGKILL y SIGSTOP.
- Si *act* es no nulo, el nuevo handler para la señal *signum* es instalado.
- Si *oldact* es no nulo, el handler original es guardado en *oldact*.

La estructura `sigaction` se define de la siguiente manera (o similar):

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

Claramente, llamar a `sigaction` directamente puede ser demasiado complicado: hay que crear una estructura `struct sigaction` y rellenarla. Para poder modificar los handlers de una señal de forma más sencilla, podemos llamar a la función `signal`. El primer parámetro es el número de la señal y el segundo especifica el nuevo handler.

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

En particular, el *handler* puede tomar uno de los siguientes valores:

- SIG_DFL para el handler por defecto específico al tipo de señal. Típicamente, se termina el proceso, pudiendo generar un archivo con la imagen core del proceso, o simplemente ser ignoradas— por ejemplo, es el caso de SIGCHILD.
- SIG_IGN para ignorar esta señal.
- Un puntero a un handler definida en user space. Esta función tan solo recibe como parámetro el número de señal.

En general, no se recomienda utilizar la system call `signal`, ya que su comportamiento varía entre las distintas versiones de UNIX (y también de Linux). Por este motivo, se suele definir nuestro propio wrapper que se encargue de llamar a `sigaction` con los parámetros adecuados. De esta manera, proveemos una interfaz simple, mientras que se preserva la semántica POSIX.

Si varias señales del mismo tipo son enviadas a un mismo proceso, UNIX garantiza que al menos una señal será entregada, en algún tiempo futuro. Es decir, las señales pueden perderse y no es posible contar la cantidad de señales recibidas. La lógica es que una señal le dice al programa que algo necesita de cierta atención y depende del programa decidir qué cosa se debe hacer.

Las señales de UNIX son usadas para tres propósitos distintos: reportar errores, reportar situación e interrupciones. Si bien esta distinción no es necesaria en programas secuenciales, en un programa paralelo (multithreaded) cada caso requiere de métodos distintos para ser manejados.

- **Error Reporting**. Ocurre cuando un programa ha ejecutado una instrucción ilegal. Posiblemente, intentó dividir por cero o acceder a una dirección de memoria inválida. En estos casos, el hardware por sí mismo descubre la instrucción ilegal, y termina llamando al trap handler del kernel. El kernel se da cuenta qué es lo que había pasado, y envía una señal al proceso (SIGFPE para la división por cero).

  Como el proceso evidentemente ha sido detenido en esa instrucción, puede estar seguro que el handler de esa señal será ejecutado con esa dirección de retorno. Las señales que son generadas traps son conocidas como señales **sincrónicas**. Para este tipo de señales, el run-time system garantiza que la señal será entregada al thread que generó el error.

```c
#include "unp.h"
#define void (*sighandler_t )(int);

sighandler_t signal(int signum, sighandler_t handler)
{
    struct sigaction act, oact;

    act.sa_handler = handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signum == SIGALARM) {
#ifdef SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT; /* SunOS 4.x */
#endif
    } else {
#ifdef SA_RESTART
        act.sa_flags |= SA_RESTART; /* SVR4, 4.4 BSD */
#endif
    }
    if (sigaction(signum, &act, &oact) < 0){
        return (SIG_ERR);
    }
    return (oact.sa_handler);
}
```

Figura 1.3: Función *signal* que llama a la system call *sigaction*.

- **Situation Reporting**. Es una señal **asincrónica** que es entregada cuando se tiene como objetivo informar al proceso que cierta situación ha cambiado y necesita ser atendida. En estos casos, no ocurrió ningún error; simplemente queremos que el programa haga otra cosa adicional.

  Las señales generadas externamente (como las asincrónicas) son direccionadas al *proceso*. En el capítulo de procesos vamos a ver que dipatch table[3] es un recurso compartido entre threads de un mismo proceso, por lo que tan solo puede haber un conjunto de handlers para todo el proceso. Normalmente, el run-time system decide cuál de todos los threads debe recibir la señal y arregla que ese thread ejecute el handler instalado. La selección del thread particular depende de la implementación y no está garantizado que sea siempre el mismo.

  Como usuarios, el único control que tenemos sobre cuál de todos los threads será seleccionado es mediante el uso de **máscaras de señales**. Este es un recurso propio de cada thread, y permite que un thread desactive aquellas señales que no le interesan (pudiendo incluso reactivarlas más adelante). Esto se puede hacer por medio de la *library call*

  ```c
  int pthread_sigmask(int how, sigset_t *set, sigset_t *oldset);
  ```

- **Interruptions**. Es una señal **asincrónica** que es enviada cuando se tiene como objetivo detener el programa de hacer lo que está haciendo y darle otra cosa para hacer.

  Por ejemplo, supongamos que acabamos de crear un dispositivo poco confiable y deseamos hacer un read sobre el mismo, pero no queremos perder control en caso de que el dispositivo falle. Si la función `read` no tiene un tiempo límite, deberíamos configurar una *alarma* para que suene dentro de, por ejemplo, 10 segundos. Pasados los 10 segundos, se enviará una señal SIGALARM al proceso, pudiendo recuperar el control en caso de que el dispositivo haya fallado.

  En este caso, la definición de POSIX realmente no hace lo que esperamos. Una SIGALARM será enviada al proceso, y no al thread que la pidió. No hay ningún método general, confiable para

[3] Tabla de punteros a funciones (*handlers*).

asegurar que sea redirigida al thread adecuado.

En general, todos estos detalles son manejados por el run-time system.

Si un proceso es multithreaded, la mayoría de las necesidades para el uso de señales se vuelven innecesarias; un programa multithreaded puede simplemente crear un nuevo thread para que se quede esperando por cualquier evento de interés. Aún así, todavía hay situaciones donde necesitamos lidiar con señales. Las razones más típicas son:

- El programa necesita lidiar con programas viejos que enviaban señales
- Si queremos poder interrumpir threads individuales.

1.2.2. Tracing System Calls

Como dijimos anteriormente, la mayoría de programas de usuario no utilizan llaman directamente las system calls, sino que utilizan library calls que actúan como wrappers de las system calls específicas a cada sistema operativo. Para analizar en detalle la estructura interna de un programa y, en particular, cuáles son las system calls que efectivamente utiliza, podemos utilizar el comando `strace`. Esta herramienta nos genera un log de todas las system calls invocadas por un programa:

```
strace -o log.txt ./main [...]
```

Si utilizamos este comando con el típico programa *Hola, Mundo!*, nos vamos a encontrar que éste realiza una gran cantidad de system calls que no son realizadas explícitamente en nuestro programa. Estas system calls son generadas automáticamente por el run-time system que se encarga de lanzar y ejecutar el programa. Típicamente, nos vamos a encontrar con la system call `brk`, que se encarga de administrar la memoria dinámica utilizada por la aplicación. Notemos que podemos utilizar la herramienta `strace` incluso si no tenemos acceso al código fuente de la aplicación.

La herramienta `strace` ha sido desarrollada para monitorear las system calls de procesos. Para implementar este comando, se utiliza una system call más general `ptrace`, que recibe cuatro parámetros.

```
long ptrace(enum __ptrace_request request,
            pid_t pid, void *addr, void *data);
```

- `request` es una constante que indica la operación a realizar por ptrace.
- `pid` identifica al proceso objetivo— salvo que *request* sea PTHREAD_TRACEME y, en ese caso, este parámetro es ignorado.
- `addr` y `data` hacen referencia a una dirección de memoria. Sus significados puntales dependen de la operación seleccionada.

Esta system call nos ofrece un mecanismo para que un proceso (*tracer*) pueda monitorear el comportamiento de otro proceso (*tracee*), pudiendo examinar y modificar la ejecución del proceso monitoreado. Esta función opera a través de *requests*, que actúan en el proceso objetivo. El significado de cada parámetro no es general, sino que varía de acuerdo con el tipo de *request* especificado.

Iniciando *tracing*. Antes de poder enviar el resto de *requests*, es necesario enganchar al proceso monitor con su proceso objetivo. Para esto, tenemos dos opciones.

- La primera opción es que un proceso quiera ser monitoreado por su proceso padre. En ese caso, el proceso hijo pide ser monitoreado por medio del *request* PTHREAD_TRACEME. Típicamente, esto se hace justo después del `fork` y antes de llamar a `execve`. Todos los parámetros se ignoran.
- La segunda opción es que un proceso inicie un monitoreo sobre otro proceso por medio del *request* PTHREAD_ATTACH, indicando en *pid* el identificador del proceso objetivo. El proceso objetivo recibirá una señal SIGSTOP; para asegurarse que el proceso objetivo realmente fue detenido, el proceso monitor debe llamar a `waitpid` (se le enviará una señal SIGCHILD). El resto de parámetros

se ignoran.

Obteniendo Información. Mientras sea monitoreado, el proceso será **detenido** cuando se reciba una **señal**— independientemente de su máscara de señales. Además, se genera un evento a ser enviado al proceso monitor, quien puede detectar el evento si previamente había llamado a la system call `waitpid` (o alguna de las system calls relacionadas). Esa llamada retornará un valor de *status* que contiene información acerca de la causa de detención en el proceso monitoreado. En este contexto, detenido significa que el proceso monitoreado admite comandos del proceso monitor. Es decir, siempre que sea posible realizar los *requests* por medio de *ptrace*. No tiene ninguna relación con el estado del proceso.

Mientras que el proceso monitoreado siga detenido, el proceso monitor puede utilizar varios *requests* para inspeccionar y modificar el espacio de direcciones y los registros del proceso monitoreado:

- PTRACE_PEEKTEXT, PTRACE_PEEKDATA, PTRACE_PEEKUSR. Nos permiten leer datos del espacio de direcciones del proceso apuntados por *addr*. En todos los casos, el parámetro *data* es ignorado.

  En particular, PEEKUSR permite leer los registros normales de CPU y, además, cualquier otro registro de debugging usado por el proceso monitoreado. Para seleccionar el registro deseado, ponemos en *addr* una constante numérica que indica el registro deseado.

  Por otro lado, PEEKDATA y PEEKTEXT permiten leer una palabra en la dirección *addr* del proceso *pid*, retornando este valor. Ambas operaciones son equivalentes; sin embargo, si los procesos tuvieran espacios de direcciones distintos para el segmento de datos y para el segmento de texto, las operaciones serían distintas.

  En realidad, en el caso de PTRACE_PEEKUSR, esa constante se interpreta como un offset dentro del *user area* del proceso monitoreado; el *user area* está dentro del espacio de memoria del kernel y guarda cierta información acerca del proceso correspondiente— para acceder a esta área, la PCB del proceso tiene un campo con un puntero a esta estructura, llamado `struct user *user`. En particular, esta estructura almacena los registros de propósito general, los registros de operaciones de punto flotante, el número de la system call llamada, los registros de debugging, etc.

  El número de system call el valor del registro RAX antes de haber llamado a la system call, llamado ORIG_RAX, y tener acceso a este valor nos permite determinar cuál fue la system call llamada por el proceso monitoreado.

- PTRACE_POKETEXT, PTRACE_POKEDATA, PTRACE_POKEUSR. Nos permiten copiar el valor en *data* sobre la dirección indicada en *addr*.

Continuar *tracing*. Eventualmente, el proceso monitor permite que el proceso monitoreado continúe ejecutando por medio de un *request* adecuado:

- PTRACE_CONT. Reinicia el proceso monitoreado (*pid*), quien había sido detenido. El parámetro *data*, si es distinto de cero, se interpreta como un identificador numérico de la señal que se le debe enviar al proceso monitoreado. De esta manera, el monitor puede controlar si una señal es enviada al proceso monitoreado, si se ignora o si se envía alguna otra señal en su lugar. *addr* se ignora.

- PTRACE_SINGLESTEP. Reinicia el proceso monitoreado igual que PTRACE_CONT, pero establece que el proceso monitoreado sea detenido tras haber ejecutado una sola instrucción de lenguaje ensamblador. Si el proceso monitoreado recibe una señal, también es detenido. El parámetro *data* es tratado igual que en PTRACE_CONT y *addr* es ignorado. Cada vez que el proceso monitoreado ejecuta una instrucción, se lo detiene y se envía una señal SIGCHILD al proceso monitor. De esta manera, el proceso monitor puede examinar y modificar el estado del proceso monitoreado, y puede volver a llamar a `ptrace` (con los parámetros adecuados) para continuar este ciclo.

- PTRACE_SYSCALL. Reinicia el proceso monitoreado de la misma forma que PTRACE_CONT, pero establece que el proceso monitoreado sea detenido tras haber ejecutado una system call. Desde el punto de vista del proceso monitor, el proceso monitoreado es detenido por haber recibido una SIGTRAP. Luego, el proceso monitor puede inspeccionar los parámetros de la system call

durante la primer detención, llamar nuevamente a PTRACE_SYSCALL e inspeccionar el valor de retorno de la system call durante la segunda detención. El parámetro *data* es tratado igual que en PTRACE_CONT y *addr* es ignorado.

Finalizar *tracing*. Cuando el proceso monitor termine de monitorear al proceso objetivo, puede decidir dejar de monitorearlo por medio del *request* PTRACE_DETACH. Este *request* reinicia el proceso monitoreado de la misma forma que PTRACE_CONT; sin embargo, antes de hacerlo, desengancha el proceso monitor del proceso monitoreado. El parámetro *data* es tratado igual que en PTRACE_CONT y *addr* es ignorado.

1.3. Administración de Recursos

Como mencionamos anteriormente, una de las tareas de un sistema operativo es **administrar los recursos** del hardware. La CPU, el espacio de memoria, el espacio de almacenamiento, los dispositivos de E/S forman parte de los recursos que el sistema operativo debe manejar.

Manejo de Procesos. Un programa no puede hacer nada por sí mismo. Se necesita que sus instrucciones sean ejecutadas por una CPU. Decimos que un programa en ejecución es un **proceso**. Un proceso necesita de ciertos recursos— incluyendo tiempo de cómputo, memoria, archivos y acceso a dispositivos de E/S— para realizar sus tareas. Cada proceso single-threaded tiene un program counter que especifica la próxima instrucción a ejecutar. La CPU ejecuta una instrucción del proceso de forma secuencial, hasta que el proceso termine. En todo momento, a lo sumo una instrucción es ejecutada en nombre del proceso. Notemos que, a pesar de que dos procesos estén asociados a un mismo programa, se consideran como dos secuencias de ejecución separadas.

Los procesos terminan funcionando como la unidad de trabajo en el sistema. Por lo tanto, podemos pensar al sistema como una colección de procesos. Todos estos procesos puede ejecutar de forma concurrente (o en paralelo).

Manejo de Memoria. Para que un programa sea ejecutado, primero debe mapearse a direcciones absolutas y cargarse a memoria. Durante la ejecución del programa, se va accediendo a sus instrucciones y sus datos. Eventualmente, el programa termina, y el espacio de memoria que ocupaba es declarado disponible. De esta manera, el siguiente programa puede ser cargado y ejecutado.

Para mejorar tanto la utilización de la CPU como la velocidad de respuesta de la computadora, se mantienen múltiples programas activos en memoria, creando la necesidad de un **manejador de memoria**. El sistema operativo es responsable de recordar qué partes de la memoria están siendo utilizadas por qué procesos, reservar y liberar memoria; y decidir qué procesos, partes de procesos o datos deben bajarse a disco y cuáles traerse a memoria.

Manejo de File-System. Una de las tareas del sistema operativo es la de brindar una interfaz de programación. En particular, el sistema operativo provee una visión lógica y uniforme del almacenamiento. El sistema operativo nos abstrae de las propiedades físicas de los dispositivos de almacenamiento, definiendo una unidad de almacenamiento lógico: el **archivo**. Un archivo es una colección de información relacionada, representando programas o datos.

El sistema operativo implementa el concepto abstracto de un archivo a través del manejo de los medios de almacenamiento y los dispositivos que los controlan. Además, los archivos suelen organizarse en directorios para facilitar su uso. Finalmente, cuando múltiples usuarios tienen acceso a los archivos, es deseable poder controlar qué usuarios tienen acceso a cada archivo particular y de qué manera pueden acceder al mismo (read, write, append). Además, el sistema operativo es responsable de montar y desmontar el dispositivo de almacenamiento secundario, manejar la reserva y liberación del espacio, planificar las lecturas del disco, manejar particiones y brindar protección.

Manejo de Cache. El caching es un principio fundamental de las computadoras. La información normalmente se mantiene en algún sistema de almacenamiento, como la memoria principal. Mientras es usada, se copia a sistemas de almacenamiento más rápidos de manera temporal, como los registros y la

cache. Cuando necesitamos una pieza particular de información, primero revisamos si se encuentra en la cache. Si lo está, utilizamos la información directamente de la cache. Sino, usamos la información de la fuente, colocando una copia en la cache bajo la asunción de que se necesitará en un futuro cercano.

Como las caches tienen un tamaño limitado, el manejo de cache es un problema importante de diseño. Una elección adecuada del tamaño de la cache y de la política de reemplazo puede generar una gran mejora en la eficiencia del sistema. El movimiento de la información entre los distintos niveles de la jerarquía de almacenamiento puede ser explícita o implícita, dependiendo del diseño del hardware y el software del sistema operativo. Por ejemplo, la transferencia de datos entre la cache y la CPU suele ser una función del hardware, sin intervención del sistema operativo. En cambio, la transferencia entre la memoria y el disco suele ser controlada por el sistema operativo.

En una estructura de almacenamiento jerárquico, los mismos datos pueden aparecer en distintos niveles del almacenamiento. Esto se vuelve un problema aún más complejo cuando trabajamos en un ambiente distribuido, ya que varias copias de un mismo archivo pueden encontrarse en distintas computadoras, por lo que aparecen problemas asociados a mantener la **consistencia** entre las distintas copias.

Manejo de E/S. Uno de los propósitos del sistema operativo es ocultar las particularidades del hardware específico a los programas de usuario. Incluso, es posible que estos detalles se le oculten al propio kernel, a través del **subsistema de E/S**. Este subsistema consiste de un manejador de memoria que incluye buffering, caching y spooling; una interfaz general entre dispositivo y driver; y los drivers para dispositivos de hardware específico. De esta manera, solo es necesario que cada driver sepa las características específicas de sus dispositivos.

1.4. Organización del Computador

Una computadora moderna consiste de una (o varias) CPUs y **controladores** de dispositivos conectados a través de un **bus** común que provee acceso entre las distintas componentes y la memoria compartida (Fig. 1.4).

Cada controlador está a cargo de un tipo específico de dispositivo. Dependiendo del controlador, uno o más dispositivos pueden estar conectados. Un controlador mantiene un buffer para almacenamiento local y un conjunto de registros de uso específico. Los controladores son responsables de mover los datos entre los periféricos que controla y su buffer local.

Típicamente, los sistemas operativos tienen un **driver** para cada controlador. Este driver sabe hablar con el controlador y provee al resto del sistema operativo una interfaz (uniforme) al dispositivo. Existen varios métodos de comunicación entre un driver y un controlador. Entre los principales, encontramos el mecanismo de interrupciones y DMA.

Interrupciones. Para inicializar una operación de E/S, el driver escribe sobre los registros de uso específicos del controlador. El controlador, en consecuencia, examina el contenido de estos registros para determinar qué acción debe llevar a cabo. Luego, el controlador comienza a transferir datos desde el dispositivo a su buffer local. Una vez la trasferencia se haya completado, el controlador le informa al driver, a través de una **interrupción**, que ha terminado de procesar la operación. Entonces, el driver devuelve el control a otras partes del sistema operativo, posiblemente retornando datos o un puntero a los mismos (si la operación fue una lectura). Para otras operaciones, el driver devuelve una información de estado como "*escritura completada satisfactoriamente*" o "*dispositivo ocupado*".

El hardware podría disparar una interrupción en cualquier momento, enviando una señal a la CPU (típicamente) a través del bus del sistema. Cuando la CPU es interrumpida, deja lo que estaba haciendo e inmediatamente transfiere la ejecución a una ubicación fija. En esta ubicación fija se suele tener la dirección inicial donde se ubica la rutina de atención de interrupciones (*Interrupt Service Routine*). La ISR ejecuta y, una vez se termina de ejecutar, la CPU reanuda la ejecución del programa interrumpido.

Otro tipo de interrupciones son las **traps** (o excepciones), que son interrupciones generadas por software causadas por un error (división por cero, dirección de memoria inválida, etc.), o por un pedido específico de un programa que quiere utilizar un servicio del sistema operativo, a través de una **system**

call (*syscall*).

DMA. Esta forma de manejar E/S funciona bien para mover pequeñas cantidades de datos, pero puede producir un alto overhead cuando se usa para el movimiento de grandes cantidades de datos (como los discos). Para resolver este problema, se utiliza el método de acceso directo a memoria (**DMA**). Luego de configurar los buffers, punteros y contadores para el dispositivo de E/S, el controlador transfiere un bloque entero de datos directamente del dispositivo a la memoria principal, sin la intervención de la CPU. Solo se genera una interrupción por bloque generado, para decirle al driver que la operación fue completada, en lugar de tener una interrupción por cada byte generado. Mientras el controlador se encarga de realizar estas operaciones, la CPU está disponible para realizar otro trabajo.

Notemos que la CPU y el controlador pueden ejecutar en paralelo, por lo que compiten por el uso de la memoria. Para asegurar un acceso ordenado a la memoria compartida, la memoria cuenta con un controlador propio que se encarga de la sincronización sobre el acceso a memoria.

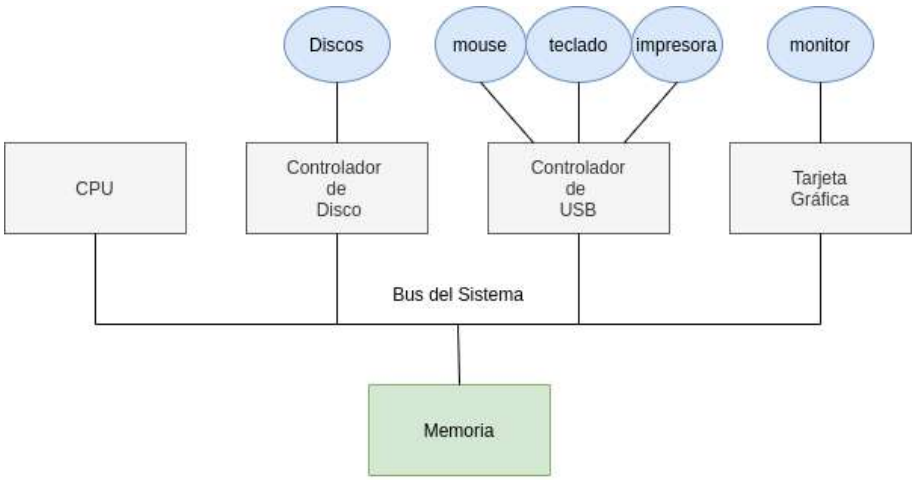

Figura 1.4: Arquitectura monoprocesador.

1.5. Arquitectura del Sistema

1.5.1. UMA

Hace muchos años, las computadoras usaban un único procesador. Hoy en día, la mayoría de las computadoras modernas son sistemas **multiprocesadores**, donde los distintos procesadores comparten el bus del sistema, la memoria y los periféricos.

La principal ventaja de los sistemas multiprocesador es que el *throughput* se ve incrementado. Al incrementar la cantidad de procesadores, podemos realizar más trabajo en menor tiempo. Sin embargo, la relación no es uno a uno, ya que las tareas a realizar generalmente no se pueden paralelizar completamente— también hay que tener en cuenta que se tiene un cierto overhead asociado a mantener a todas las partes funcionando correctamente, sumado a la *contención* sobre los recursos compartidos.

La definición de *multiprocesador* ha evolucionado con el tiempo y ahora incluye a los sistemas **multicore**, en donde tenemos múltiples cores dentro de un mismo chip. Un core está compuesto por componentes que ejecutan instrucciones (ALU, FPU, etc.) y registros que almacenan datos localmente. La comunicación entre cores de un mismo chip suele ser más rápida que la comunicación entre chips, por lo que los sistemas multicore suelen ser más eficientes que los sistemas multiprocesador.

La arquitectura más común para sistemas multiprocesador es la arquitectura **UMA** (Uniform Memory Access) (Fig. 1.5). Se caracteriza por el hecho de que varias unidades de procesamiento comparten el acceso a una memoria central, a través de un bus compartido. Cada procesador compite en igualdad de

condiciones por el acceso a la memoria, y puede realizar cualquier tipo de tarea, incluyendo funciones del sistema operativo y procesos de usuario. Este tipo de procesamiento se conoce como multi-procesamiento simétrico (**SMP**).

En principio, si aumentamos la cantidad de CPUs en un sistema UMA, se incrementará la capacidad de cómputo. Sin embargo, esta estrategia no escala demasiado bien. Es posible que si tenemos demasiadas CPUs, la contención sobre el bus de sistema termine siendo un cuello de botella— degradando la eficiencia del sistema.

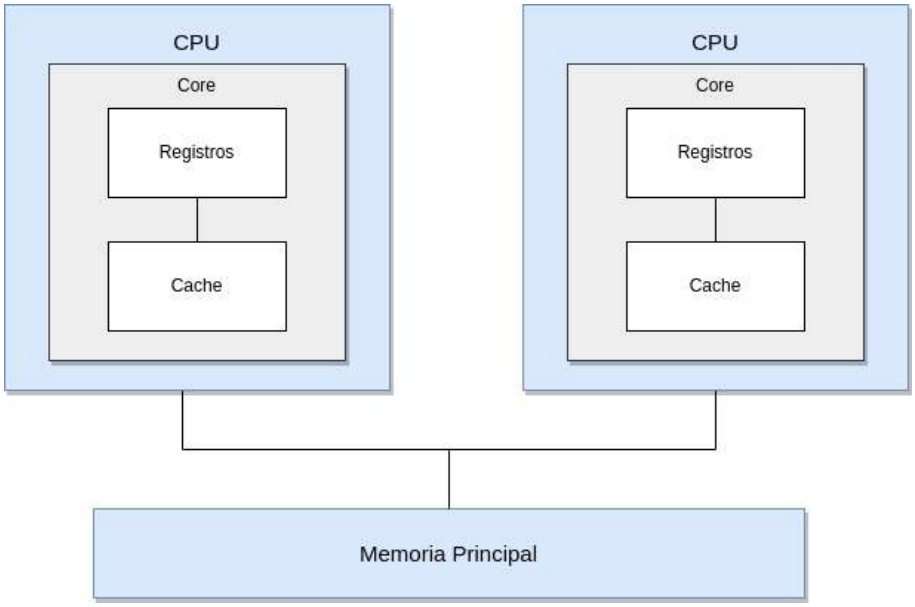

Figura 1.5: Arquitectura de multi-procesamiento UMA (SMP).

1.5.2. NUMA

Una arquitectura alternativa, conocida como arquitectura **NUMA** (Non-Uniform Memory Access), consiste en proveer a cada CPU (o grupo de CPUs) una memoria local propia que sea accedida a través de un pequeño y rápido bus local (Fig. 1.6). Las CPUs pueden acceder al resto de la memoria a través de un **shared system interconnect**, por lo que todas las CPUs terminan compartiendo un mismo espacio físico de direcciones.

De esta manera, una CPU puede acceder a su memoria local más rápido, y se reduce la contención sobre el sistema de interconexión. Por lo tanto, los sistemas NUMA escalan de forma más eficiente a medida que agregamos procesadores.

1.5.3. Clustered Systems

Otro tipo de sistema multiprocesador son los **clustered systems**, que están compuestos por dos o más sistemas individuales (o nodos) débilmente acoplados (**loosely coupled**). Estos están conectados por una conexión LAN o algún otro método de interconexión más rápido— por ejemplo, InfiniBand. Normalmente, se utilizan para aplicaciones que requieren de una alta disponibilidad del servicio, es decir, que el servicio siga disponible incluso si uno o más nodos en el cluster fallan. Cada nodo puede monitorear a uno o más nodos en la red. Si una máquina falla, la máquina que la estaba monitoreando puede adueñarse de su almacenamiento y reiniciar la aplicación que estaba corriendo antes de fallar. Como los clusters consisten de múltiples sistemas conectados, proveen una alta capacidad de cómputo, ya que las aplicaciones pueden correr *concurrentemente* en todas las computadoras del cluster.

Los clusters permiten que cada máquina pueda acceder a un mismo almacenamiento compartido (Fig. 1.7). Para ello, el sistema debe proveer un control de acceso para asegurar que las distintas operaciones no entren en conflicto. Muchas de las mejoras en la tecnología de clusters es posible gracias a las

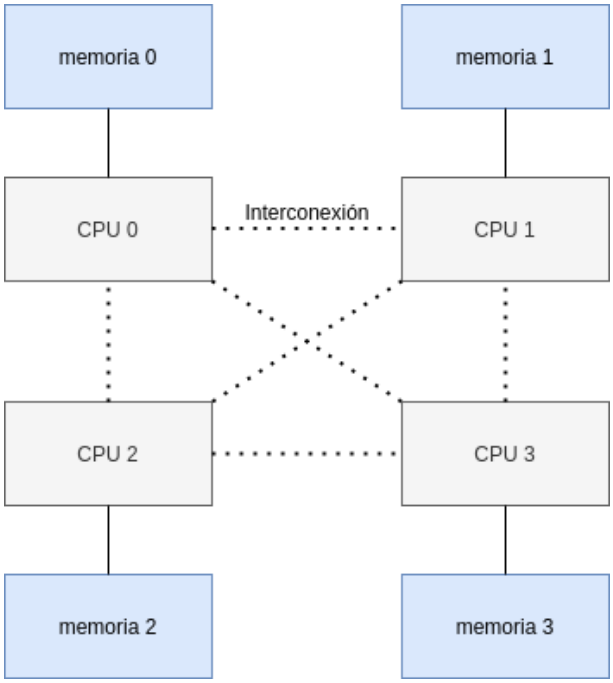

Figura 1.6: Arquitectura de multi-procesamiento NUMA.

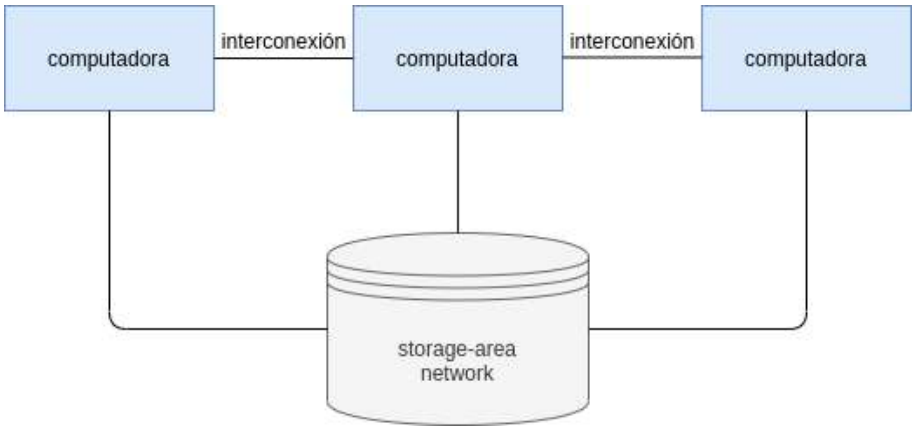

Figura 1.7: Estructura general de un cluster.

storage-area networks (**SAN**), que permiten que muchos sistemas puedan acceder a un mismo pool de almacenamiento.

1.6. Estructura de Almacenamiento

Las computadoras de propósito general corren sus programas en la **memoria principal** (implementada con tecnología DRAM). La memoria provee un arreglo de bytes, accesibles por medio de una dirección. Podemos interactuar con la memoria por medio de instrucciones `load` y `store`.

Idealmente, queremos que los programas y datos residan en la memoria principal permanentemente. Esto no suele ser posible por dos motivos:

- La memoria principal suele ser demasiado chica como para almacenar todos los programas y los datos.
- La memoria principal es volátil, por lo que pierde su contenido cada vez que apagamos el sistema.

Por lo tanto, la mayoría de las computadores proveen **almacenamiento secundario** como extensión

de la memoria principal, que es capaz de almacenar grandes cantidades de datos de forma permanente. Los dispositivos de memoria secundaria más comunes son los discos rígidos (HDDs) y las memorias no volátiles (SSD, memoria flash, etc.). Notemos que un sistema no puede prescindir de una memoria principal, dado que las memorias secundarias es que son mucho más lentas que éstas. Por lo tanto, lo que se termina haciendo es almacenar a la mayoría de programas en la memoria secundaria y, solo cuando sea necesario, se van cargando en la memoria principal.

Aquellos dispositivos que son lo suficientemente lentos y tienen la suficiente capacidad de almacenamiento como para que tengan un uso específico, por ejemplo para hacer *backups*, son conocidos como **memoria terciaria**— discos ópticos, cintas magnéticas, etc.

Capítulo 2

Procesos y Threads

Un sistema operativo provee un ambiente donde los programas son ejecutados. Podemos ver al sistema operativo dese varios puntos de vista. Una vista se concentra en los servicios que el sistema provee; otra, en la interfaz de programación que ofrece a programas de usuario; una tercera, sobre sus componentes e interconexiones. En este capítulo, vamos estudiar una de las abstracciones centrales de los sistemas operativos: los **procesos**. Los sistemas modernos permiten que tengamos múltiples programas cargados en memoria y, de esta manera, es posible ejecutarlos de manera concurrente— a diferencia de los sistemas antiguos, donde solo se podía ejecutar un programa a la vez. Al trabajar con múltiples programas de manera concurrente, se requiere de un mayor control y una mayor compartimentación de los mismos. Estas necesidades resultaron en la creación del **modelo de procesos**.

2.1. Modelo de Procesos

En este modelo, todo el software ejecutable en la computadora es organizado en **procesos secuenciales** (o procesos). Un proceso a un programa en ejecución, que incluye a los valores actuales de su program counter, registros y variables, al igual que su espacio de memoria asociado (y otros atributos). Conceptualmente, cada proceso tiene su propia CPU virtual. Es decir, cada proceso piensa que tiene una CPU propia. En realidad, el sistema operativo se encarga de reservar la CPU real a cada uno de los procesos, cargando y desalojando sucesivamente a los distintos procesos en la única CPU— técnica conocida como **multiprogramación**. En este sentido, los procesos son las unidades de trabajo en una computadora. Necesitan recursos como la CPU, memoria, archivos y dispositivos de E/S para poder realizar su trabajo.

Es importante enfatizar la diferencia entre un proceso y el programa. Un programa es una **entidad pasiva**, como un archivo conteniendo una secuencia de instrucciones almacenado en el disco y que no hace nada. En cambio, un proceso es una **entidad activa** que tiene asociado una serie de recursos, entre ellos, un programa, una entrada, una salida y un estado. Notemos que, incluso en el caso en el que dos procesos ejecuten el mismo programa, se tratan de dos secuencias de ejecución independientes entre sí.

2.1.1. Scheduling de Procesos

Una de las características más importantes de los sistemas operativos es su capacidad de correr múltiples programas de manera concurrente. Normalmente, un solo programa no utiliza todos los recursos del sistema todo el tiempo. Es decir, si tenemos corriendo un solo programa a la vez, terminamos dejando sin utilizar gran parte de los recursos del sistema. La técnica de multiprogramación nos permite mejorar la utilización de la CPU y de los dispositivos de E/S al aumentar la cantidad de procesos disponibles para ejecutar. La idea es la siguiente.

El sistema operativo mantiene simultáneamente varios procesos en memoria. Luego, selecciona y comienza a ejecutar alguno de estos procesos. (Recordemos que tan solo un proceso puede ejecutar en un mismo procesador en un determinado momento.) Eventualmente, el proceso podría tener que bloquearse por algún evento— por ejemplo, podría quedarse esperando por una operación de E/S. En un sistema sin

multiprogramación, la CPU no se podría usar hasta terminar con la E/S. En un sistema multiprogramado, el sistema operativo puede pasar a ejecutar otro proceso. Por lo tanto, se mejora la utilización de la CPU y se mejoran los tiempos de respuesta.

Una métrica importante en cuanto a la utilización del sistema es la cantidad de procesos que actualmente están en memoria, conocida como el **grado de multiprogramación**. Cuando tenemos múltiples procesos listos para ejecutar, es responsabilidad del sistema operativo elegir cuál de todos estos procesos poner a correr— esto va a estar determinado por la *política de scheduling* del sistema operativo.

Cuando un proceso es creado, (su PCB) se coloca en una *ready queue*. Éste espera allí hasta que sea seleccionado para su ejecución. Una vez que el proceso está ejecutando, uno de varios eventos puede ocurrir (Fig. 2.1):

- Hace un pedido de E/S, y es colocado en una *wait queue* de E/S.
- Crea un nuevo proceso hijo, y es colocado en una *wait queue* hasta que el proceso hijo termine de ejecutarse.
- El proceso podría ser desalojado, y es colocado devuelta en la *ready queue*.

Cuando se desaloja un proceso de la CPU, hay que guardar los datos suficientes para poder reanudar la ejecución del mismo más tarde. Para ello, nos vamos a guardar en el PCB del proceso en ejecución su **contexto de ejecución**: los registros y el program counter. Cuando ponemos a ejecutar un proceso, primero recuperamos su contexto de ejecución directamente de el PCB asociada al ese proceso. Notemos que el tiempo utilizado para realizar el cambio de contexto es **tiempo muerto**.

Los procesos continúan este ciclo hasta que terminen su trabajo. Luego, se remueven de todas las colas y se liberan todos los recursos asociados al proceso (incluida su PCB).

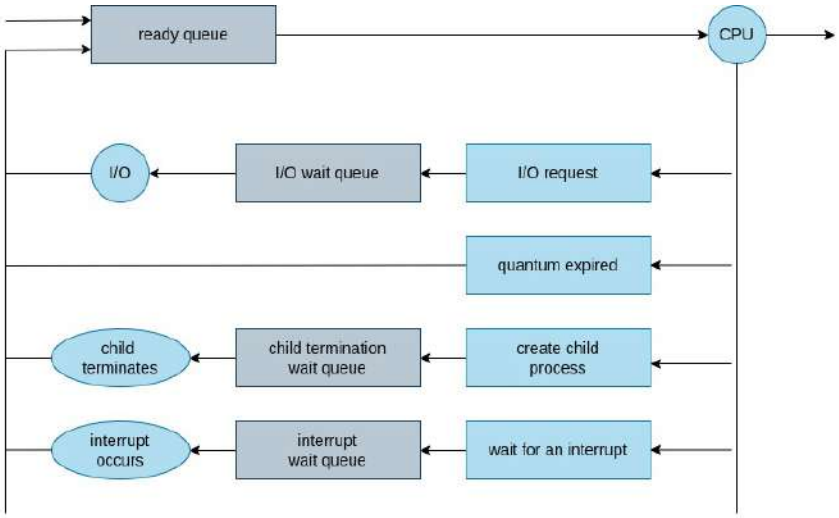

Figura 2.1: Representación de las colas en el scheduling de procesos.

2.1.2. Implementación de Procesos

Para implementar el modelo de procesos, el sistema operativo mantiene una tabla conocida como **tabla de procesos**, con una entrada por proceso. Cada entrada de esta tabla se conoce como **process control block** (PCB), y contiene información importante acerca del estado del proceso, incluyendo su program counter, stack pointer, reserva de memoria, estado de los archivos abiertos, información de auditoría e información de scheduling, entre otros. Es decir, el PCB almacena toda la información necesaria para que cuando el proceso sea desalojado de la CPU, pueda ser recuperado más adelante como si nunca hubiera sido desalojado.

La memoria de un proceso se suele dividir en múltiples secciones para facilitar el manejo de memoria

de los procesos (Fig. 2.2):

- **Sección de texto**. Es el código ejecutable.
- **Sección de datos** (globales). Es donde se almacenan las variables globales. En el caso de los programas en C, esta sección se divide dos: variables globales inicializadas (data) y variables globales no inicializadas (bss).
- **Sección de heap**. Es la memoria que se reserva de forma dinámica en tiempo de ejecución. El heap crece a medida que se reserva memoria dinámica, y se achica cuando esta memoria se libera.
- **Sección de stack**. Es la memoria que se reserva al momento de invocar funciones (parámetros, dirección de retorno y variables locales). Cada vez que una función es llamada, se guardan en el stack los parámetros, variables locales y la dirección de retorno de la función.
- En los programas en C, se reserva un espacio adicional para los parámetros arc y argv que se le pasan a la función `main`.

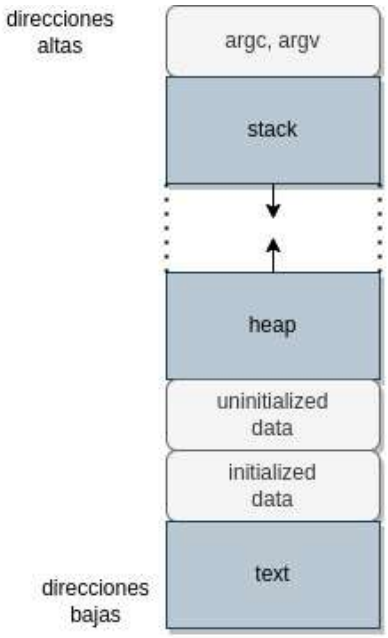

Figura 2.2: Vista de la memoria de un proceso.

Naturalmente, toda esta información debe ser almacenada en el PCB para poder restaurar la ejecución del proceso. En la Fig. 2.3 se muestran algunos de los campos principales de el PCB en un sistema genérico. Los campos de la primer columna se relacionan con el manejo y administración del proceso. Las otras dos se relacionan con la administración de memoria y de archivos, respectivamente. Claramente, los campos particulares en la tabla de procesos son específicos a cada sistema, pero nos podemos dar una idea de la información que se necesita.

- **Información del scheduling**. Esta información incluye la prioridad de un proceso, punteros a scheduling queues, entre otros.
- **Información sobre el estado de E/S**. Una lista de dispositivos de E/S reservados por el proceso, una lista de archivos abiertos, etc.

Ahora que hemos visto el concepto de la tabla de procesos, es posible explicar un poco más en detalle cómo funciona la técnica de multiprogramación. Recordemos que hay una ubicación fija en memoria donde se almacena un arreglo con punteros a las ISRs asociadas a los distintos tipos de interrupciones (conocido como **interrupt vector**).

Supongamos que un proceso de usuario está ejecutando cuando ocurre una interrupción del disco.

Administración de Procesos	Administración de Memoria	Administración de Archivos
Registros	Puntero a tabla de segmentos	Directorio root
Program counter	Puntero a tablas de páginas	Directorio actual
Estado del Proceso		Información de E/S
pid		uid
pid del padre, hijos y hermanos		gid
Información de Scheduling		
Información de Señales		

Figura 2.3: Campos típicos de un PCB.

El program counter, el PSW y algunos otros registros son almacenados en el stack por el hardware que maneja las interrupciones. Luego, la computadora salta a la dirección especificada en el interrupt vector. Esto es todo lo que hace el hardware. De aquí en adelante, todo va a depender del software— en particular, de la ISR invocada.

Todas las interrupciones inician guardando los registros, típicamente en la PCB asociada al proceso actual. Luego, la información que había sido guardada en el stack es removida y el stack pointer se apunta a un stack temporal usado por el handler del proceso. Este trabajo de guardar los registros es idéntico para todos los tipos de interrupciones.

Cuando se termina esta rutina, se llama a un procedimiento específico al tipo de interrupción. Cuando se termina este procedimiento, se llama al scheduler para saber a qué proceso le toca ejecutar. Después de esto, se cargan los registros y las tablas de mapeo a memoria del proceso elegido por el scheduler y se lo pone a ejecutar.

A continuación, resumimos el esqueleto de lo que hace el sistema operativo cuando ocurre una interrupción.

1. El hardware almacena en el stack el program counter, etc.
2. El hardware carga el nuevo program counter desde el interrupt vector.
3. Un procedimiento en lenguaje ensamblador guarda los registros.
4. Un procedimiento en lenguaje ensamblador prepara el nuevo stack.
5. Se ejecuta la ISR específica.
6. El scheduler decide qué proceso le toca correr.
7. La ISR retorna al código en lenguaje ensamblador.
8. El procedimiento en lenguaje ensamblador inicia la ejecución del proceso elegido por el scheduler.

Para hacer que una llamada al sistema sea bloqueante— por ejemplo, cuando hacemos lectura a un dispositivo de E/S—necesitamos algún mecanismo para bloquear al proceso y, una vez completada la E/S, despertarlo. Normalmente, cada dispositivo de E/S tiene asociado un **semáforo**, inicializado en 0. Justo después de enviar la información necesaria para realizar la E/S, el proceso administrador del dispositivo hace un `wait` en el semáforo asociado, bloqueándose inmediatamente. Cuando llega una interrupción, el handler hace un `signal` sobre el semáforo asociado, lo que termina despertando al proceso y éste pasa al estado *ready*. En particular, la operación de `signal` se realiza en el paso 5, para que el proceso que administra al dispositivo pueda ser seleccionado para ejecutar en la CPU por el scheduler en el paso 6.

Claramente, los detalles específicos de implementación varían dependiendo del sistema. La clave es

que, si bien un proceso puede ser interrumpido muchas veces durante su ejecución, siempre se recupera al mismo estado que estaba antes de que ocurriera la interrupción.

2.1.3. Estados de un Proceso

Si bien cada proceso es una entidad independiente, con su propio estado interno, los procesos suelen interactuar entre sí. Un proceso puede generar cierta salida que otro utilice como entrada. Consideremos el siguiente ejemplo.

```
cat texto1 texto2 | grep palabra
```

Un primer proceso va a ejecutar *cat*, que concatena los dos archivos. Un segundo proceso va a ejecutar *grep*, y va a seleccionar todas las líneas que contengan la palabra "palabra". Recordemos que ambos procesos ejecutan concurrentemente, ya que estamos en un sistema multiprogramado. Dependiendo de la velocidad relativa de ambos procesos, es posible que *grep* esté disponible para ejecutar, pero que no haya ninguna entrada disponible. En ese caso, debe bloquearse hasta que haya alguna entrada disponible.

Cuando un proceso se bloquea, lo hace porque lógicamente no puede continuar (típicamente, porque se queda esperando por una entrada). También es posible que un proceso esté conceptualmente listo y disponible para ejecutar, pero que sea detenido debido a que el sistema operativo decidió reservar la CPU a otro proceso. Estas dos condiciones son completamente distintas. En el primer caso, la suspensión es inherente al problema (no se puede procesar datos que no se tienen disponibles). En el segundo, es un problema técnico del sistema: no hay suficiente CPU para darle a cada proceso su propio procesador privado.

En resumen, un proceso cambia de estado a medida que se avanza en su ejecución. El **estado** del proceso nos dice si está ejecutando, esperando algún evento o si está listo para ejecutar, y solo espera a que el procesador se libere. El estado de un proceso es un concepto fundamental, y lo vamos a estar usando en varios de los subsistemas del sistema operativo.

Un proceso puede estar en uno de los siguientes estados:

- **New**. Un proceso que está siendo creado, pero todavía no fue creado. (Cuando se intenta crear un proceso, el sistema operativo decide si lo admite o no lo admite.)
- **Running**. Son los procesos que actualmente están ejecutando. Es importante notar que tan solo un proceso puede estar corriendo en un procesador al mismo tiempo.
- **Waiting**. El proceso está esperando a que cierto evento ocurra— por ejemplo, recibir una señal o a que se complete E/S. (Una señal es usada en los sistemas UNIX para notificar a un proceso que un evento particular ha ocurrido, mediante la system call `kill`.)
- **Ready**. Son los procesos que están listos para ejecutar: el único recurso que necesitan para seguir ejecutando es la CPU.
- **Terminated**. El proceso terminó su ejecución. (Está esperando que el proceso padre haga `wait`.)

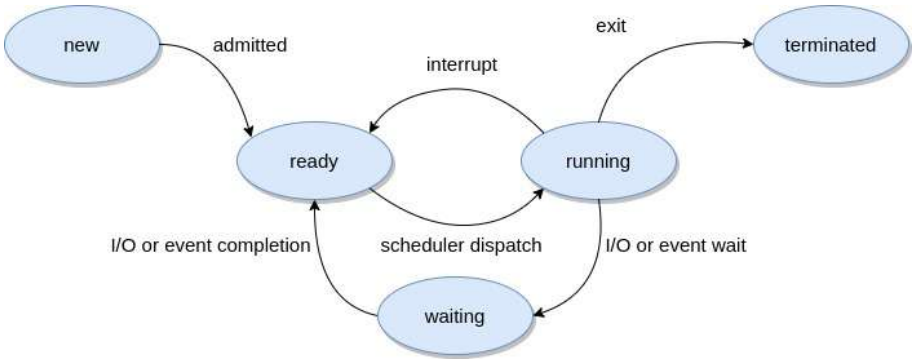

Figura 2.4: Diagrama de estados de un proceso.

Lógicamente, los estados de *running* y *ready* son similares. En ambos casos, el proceso está listo para ejecutar, solo que en el segundo caso no tenemos suficiente CPU para correrlo. El Estado *waiting* es fundamentalmente distinto de los dos anteriores en cuanto a que el proceso no puede ejecutar, incluso si la CPU está en reposo y no tiene nada más que hacer.

Tomando en cuenta solo estos tres estados, hay cuatro transiciones posibles:

- **Scheduler dispatch**. El proceso pasa de estado *ready* a *running*. El scheduler, una parte del sistema operativo, decide que todos los demás procesos ya tuvieron su turno de ejecutar y es momento de reservar la CPU para este proceso.
- **Interrupción**. El proceso pasa de estado *running* a *ready*. El scheduler decide que el proceso que está corriendo ya estuvo ejecutando por demasiado tiempo, y es momento de que pase el siguiente proceso.
- **Se espera por E/S o evento**. El proceso pasa de estado *running* a *waiting*. Ocurre cuando el sistema operativo descubre que el proceso no puede continuar en este momento. Por ejemplo, si un proceso intenta leer de un pipe (archivo especial) y no hay ninguna entrada disponible, el proceso es bloqueado automáticamente.
- **Termina E/S o evento**. El proceso pasa de estado *waiting* a *ready*. Ocurre cuando este proceso estaba esperando por un cierto evento externo y éste ocurre.

Otra métrica importante en cuanto a la utilización del sistema es la cantidad de procesos en estado *ready*, conocida como **carga de un sistema**. Lo ideal es que la carga del sistema sea lo más baja posible, relativo a la cantidad de procesadores en el sistema. Notemos que si la carga del sistema es muy alta, significa que estamos escasos en cuanto a CPU; si este no fuera el caso, los procesos *ready* deberían pasar a *running*, reduciendo la carga del sistema.

2.2. Operaciones sobre Procesos

2.2.1. Jerarquía de Procesos

La mayoría de los sistemas operativos identifican a los procesos mediante un único process identifier (**pid**), que suele ser un número entero mayor o igual que 1. El pid provee un valor único para cada proceso en el sistema, y puede ser usado como índice para acceder a varios atributos de un proceso dentro del kernel.

Cuando el sistema operativo bootea, lanza un proceso inicial que suele llamarse *init*— en los sistemas UNIX se lo suele llamar *systemd* (y su pid siempre es 1). Este proceso inicial puede crear nuevos procesos, formando una relación padre-hijo. Los procesos hijos, a su vez, también pueden crear más procesos, haciendo que todos los procesos se organicen jerárquicamente. En particular, todos los procesos en el sistema forman un único **árbol de procesos**, cuya raíz es el proceso inicial (systemd)[1].

2.2.2. Creación de Procesos

Los sistemas operativos de propósito general necesitan una manera de crear y terminar procesos dinámicamente. En **UNIX**, podemos crear un nuevo proceso utilizando la system call `fork`. El nuevo proceso es una **copia exacta** del proceso padre: tienen el mismo espacio de direcciones, las mismas variables de ambiente y los mismos archivos abiertos. Sin embargo, tanto padre como hijo tienen espacios de direcciones distintos. Esto quiere decir que si cualquier proceso escribe en su espacio de memoria, los cambios no serán visibles para el otro proceso.

Ambos procesos continúan su ejecución en la instrucción siguiente a la invocación de `fork`, con una diferencia fundamental: `fork` retorna 0 al nuevo proceso, mientras que devuelve el pid del proceso hijo al proceso padre— recordemos que 0 no es un pid válido. Esto nos permite diferenciar entre un proceso

[1] En Windows es distinto. No hay concepto de jerarquía de procesos: todos los procesos son iguales. Cuando un proceso es creado, su padre recibe un token especial (handle) que puede usarse para controlar al proceso hijo. Sin embargo, es posible entregar este token a otro proceso, invalidando la jerarquía.

padre y un proceso hijo en tiempo de ejecución:

```
1    pid_hijo = fork();
2    if (pid_hijo == 0){//codigo del hijo;}
3    else{ //codigo del padre;}
```

Luego de llamar a `fork`, el proceso hijo tiene el mismo código binario que el padre, por lo que necesitamos algún mecanismo para reemplazar este código por otro para poder ejecutar otros programas. Esto se logra a través de la syscall `execve` (o similar) para cambiar el espacio de direcciones. Esta system call recibe el path al binario del nuevo programa y lo carga en memoria, destruyendo el espacio de direcciones del programa actual. Por ejemplo, cuando utilizamos en la consola un comando, digamos, `sort`, la terminal hace un `fork`, creando un nuevo proceso, y es este proceso hijo quien termina ejecutando el comando. La razón para tener este proceso de dos pasos es para permitir que el proceso hijo manipule sus file descriptors luego del `fork`, pero antes del `execve` y, de esta manera, poder redireccionar la entrada, salida y salida de error estándar.

En **Windows**, en cambio, una sola llamada a `CreateProcess` se encarga tanto de la creación del proceso como de cargar el programa correcto dentro del nuevo proceso. Por lo tanto, el proceso hijo nunca hereda el espacio de direcciones del padre. Esta llamada tiene 10 parámetros, a diferencia de `fork` que no tiene ningún parámetro, que incluyen el programa a ser ejecutado, los parámetros que recibe por línea de comando, varios atributos de seguridad, bits de control, información de prioridad, etc.

Normalmente, luego de hacer `fork`, el proceso padre llama a la system call `wait` para quedarse en la *ready queue* hasta que termine su proceso hijo. Esta system call le permite obtener el estado de terminación del proceso hijo y su pid (para poder saber cuál de todos sus procesos hijos fue quien terminó), y así poder actuar en consecuencia.

En principio, parecería que es necesario copiar todo el espacio de direcciones de un proceso antes de poder ejecutar en el proceso hijo, luego de hacer un `fork`. Esto convertiría a la creación de un nuevo proceso en un procedimiento extremadamente costo. Sin embargo, en la práctica, esto no es necesario. Más adelante vamos a ver cómo podemos copiar el espacio de memoria del proceso padre de manera eficiente, utilizando la técnica de **copy-on-write**, tan solo es necesario copiar las tablas de páginas para establecer los mapeos entre las direcciones virtuales y las físicas (las direcciones virtuales de ambos procesos apuntarán a la misma dirección física).

Algunas versiones de UNIX— por ejemplo, Linux— proveen otras system calls que también permiten crear procesos. `vfork` es similar a fork, pero no necesita crear datos del proceso padre. En su lugar, comparte los datos entre el proceso padre y el proceso hijo (incluyendo el stack), ahorrando el costo de duplicar las tablas de páginas. Notemos que, al estar compartiendo las mismas tablas de páginas, si alguno de los procesos intentara manipular los datos, el otro proceso notaría los cambios inmediatamente. `vfork` está diseñado para la situación donde un proceso hijo, justo después de ser creado, inmediatamente ejecuta un `execve` para cargar un nuevo programa. Además, el kernel garantiza que el proceso padre es bloqueado hasta que el proceso hijo llame a `exit` o a `execve`. Originalmente, `fork` copiaba (innecesariamente) todo el espacio de direcciones y no utilizaba la técnica de *copy-on-write*, por lo que se necesitaba otro mecanismo que permita crear procesos de manera más eficiente. Hoy en día, se recomienda evitar el uso de `vfork`.

La otra system call que implementa Linux para la creación de procesos es `clone`. Esta system call crea threads y nos da la posibilidad de seleccionar con precisión aquellos elementos que se desean compartir entre el proceso padre y el hijo, y cuáles se desean copiar.

2.2.3. Terminación de Procesos

Un proceso termina cuando ejecuta su última instrucción y le pide al sistema operativo que lo borre llamando a la system call `exit`. Todos los recursos del proceso, incluyendo memoria física, virtual, archivos abiertos y buffers de E/S son liberados y reclamados por el sistema operativo. Sin embargo, la entrada en la tabla de procesos **debe permanecer** hasta que el proceso padre llame a `wait`, ya que esta tabla contiene el estado de terminación del proceso.

Un proceso puede causar la terminación de otro proceso mediante una syscall apropiada (`kill`).

Naturalmente, solo un proceso con la autorización necesaria puede terminar a otro proceso (generalmente, un proceso padre puede terminar a sus procesos hijos). Normalmente, esto se hacer por alguno de estos motivos:

- El proceso hijo excedió el uso de algunos recursos reservados.
- La tarea asignada al proceso hijo ya no es necesaria.
- El padre está terminando, y el sistema operativo no permite que un hijo continúe si su padre termina.

Un proceso que ha terminado, pero cuyo padre no ha llamado a `wait`, se conoce como proceso **zombie**. Todos los procesos pasan a este estado cuando terminan, pero solo durante poco tiempo. Una vez que su proceso padre llama a `wait`, el pid del proceso zombie y su entrada en la tabla de procesos son liberados.

En caso de que el proceso padre termine sin llamar a `wait`, sus procesos hijos quedan **huérfanos**. En los sistemas UNIX, tradicionalmente se le asignaba a `init` como el nuevo padre de los procesos huérfanos. `init` periódicamente invoca a `wait`, permitiendo que los procesos huérfanos puedan liberar su pid y su entrada en la tabla de procesos. En la mayoría de los sistemas Linux modernos, también se permite que otros procesos puedan heredar procesos huérfanos y manejar su terminación.

2.3. Threads

2.3.1. Motivación y Uso

En los sistemas operativos tradicionales, cada proceso tiene un espacio de direcciones y un único thread de control. De hecho, esto es prácticamente la definición con la que estuvimos trabajando sobre procesos. En cualquier caso, en muchas situaciones, es deseable tener múltiples threads de control en el mismo espacio de direcciones que ejecuten concurrentemente, como si se trataran de procesos diferentes, salvo por el espacio de direcciones compartido.

Ahora con los threads podemos agregar un nuevo elemento: la habilidad de compartir un espacio de direcciones y todos sus datos entre entidades paralelas. Esta habilidad es esencial para ciertas aplicaciones, donde no funcionaría utilizar múltiples procesos.

Otra ventaja que trae la incorporación de threads es que son menos pesados que los procesos. En consecuencia, son más fáciles y rápidos de crear y destruir que los procesos. En muchos sistemas, crear un thread es 10-100 veces más rápido que crear un proceso.

Una tercera razón para tener threads es también un argumento de performance. Los threads no son más eficientes con respecto a los procesos cuando son threads de CPU, pero cuando se tiene una cantidad sustancial de cómputo y de E/S, tener threads permite que estas actividades se superpongan, volviendo más rápida a la aplicación. Además, los threads son útiles en sistemas con múltiples CPUs, donde es posible la ejecución paralela (y no solo la concurrente).

La manera más sencilla de entender por qué los threads terminan siendo útiles es mediante un ejemplo concreto. Consideremos el caso de un procesador de texto. Usualmente, los procesadores de texto muestran en pantalla el documento que está siendo creado formateado exactamente como se vería si lo imprimiéramos en papel.

Supongamos que estamos escribiendo un libro en un solo documento sobre este procesador de texto. Consideremos ahora qué pasa si borramos una oración de la página 1 de un libro de 800 páginas. Luego de pasar un rato revisando que no haya ningún otro error en la página, nos movemos sobre el documento hasta la página 600. El procesador de texto ahora se ve forzado a reformatear todo el libro hasta la página 600 en el acto. Esto podría llevar a un delay sustancial hasta que podamos ver la página 600.

Los threads pueden ayudar en esta situación. Supongamos que el procesador de texto está escrito en un programa con dos threads. Un thread interactúa con nosotros y el otro se encarga de reformatear el documento. A penas se borra una oración de la página 1, el thread interactivo le dice al thread de fondo

que se encargue de reformatear el documento. Mientras tanto, el thread interactivo continua escuchando al teclado y mouse para responder a los comando simples, mientras que el thread de fondo termina de reformatear el documento por completo. De esta manera, cuando saltamos a la página 600, todo el libro ya habría sido completamente reformateado, permitiendo mostrar inmediatamente esta página.

Debe quedar claro que tener dos procesos separados no funcionaría en este caso, dado que ambos deben operar sobre el mismo documento. Al tener dos threads en lugar de procesos, comparten memoria y tienen acceso al mismo documento que está siendo editado. Con dos procesos, esto sería imposible (sin utilizar algún mecanismo de IPC).

2.3.2. Modelo de Threads

Ahora que sabemos la utilidad de los threads y cómo pueden ser utilizados, investiguemos esta idea un poco más de cerca. El modelo de procesos junta en una única entidad (el proceso) dos conceptos que, en principio, pueden pensarse como independientes: agrupar recursos del sistema y ejecutar en la CPU. A veces, resulta más útil separar estos conceptos en dos entidades distintas y es de lo que trata el modelo de threads.

Una manera de ver a un proceso es la manera en que agrupan recursos relacionados. Un proceso tiene un espacio de direcciones que contiene un programa y datos, al igual que otros recursos. Estos recursos pueden incluir archivos abiertos, procesos hijos, información de auditoría, entre otros. Al agrupar toda esta información en un proceso, se puede manejar más fácilmente.

El otro concepto que tiene un proceso es su thread de ejecución, que usualmente lo llamamos **thread**. El thread tiene asociado un program counter que mantiene un registro de la siguiente instrucción a ejecutar. También tiene registros, que almacenan las variables con las que está trabajando. También tiene un stack, que contiene la historia de ejecución, con un stack frame asociado a cada procedimiento que fue invocado pero que todavía no finalizó. Si bien un thread debe ejecutar dentro de un proceso, el thread y el proceso son conceptos diferentes y pueden ser tratados por separado. Los procesos agrupan recursos; los threads son las entidades que son asignadas y desalojadas de la CPU para ejecutar.

Lo que agregan los threads al modelo de procesos es permitir que múltiples ejecuciones independientes en gran medida tomen lugar en el mismo ambiente de proceso. Los distintos threads en un proceso no son tan independientes entre sí como lo son los distintos procesos. Todos los threads en un mismo proceso comparten exactamente el mismo espacio de direcciones, lo que significa que también comparten las mismas variables globales (segmentos de datos y de heap), pero cada thread tiene su propio stack (variables automáticas).

Items compartidos entre threads de un proceso	Items privados por thread
Espacio de direcciones	Program counter
Variables globales	Registros
Archivos Abiertos	Stack
Procesos hijos	Estado del thread
Alarmas pendientes	
Señales y handlers	
Información de auditoría	

Figura 2.5: Items compartidos por los threads de un proceso e items privados por thread.

Notemos que, como cada thread puede acceder a todo el espacio de direcciones de su proceso, un thread puede leer, escribir e incluso borrar el stack de otro thread (del mismo proceso). **No hay ningún tipo de protección entre threads**. Esto se debe a que

1. Es imposible proteger los datos entre threads de un mismo proceso.

2. No debería ser necesario.

A diferencia de los procesos, que pueden llegar a ser incluso de usuarios distintos y podrían ser hostiles entre sí, todos los threads de un proceso son propiedad de un mismo usuario, que presumiblemente ha creado múltiples threads para que puedan cooperar. Además de compartir un espacio de direcciones, los threads pueden compartir el mismo conjunto de archivos abiertos, procesos hijos, etc. En la Fig. 2.5 podemos ver en la primera columna los elementos que son propiedad de un proceso (y no de sus threads), y en la segunda columna aquellos elementos que son propiedad de cada thread (a pesar de que, en principio, no estén protegidos).

Una consecuencia de este esquema es que, por ejemplo, si un thread abre un archivo, éste es visible a todos los otros threads en el proceso y todos ellos pueden escribir y leer del mismo.

Al igual que un proceso tradicional, un thread puede estar en uno de varios estados: *running, waiting, ready o terminated* (con el mismo significado que tenían anteriormente). Un thread en estado *running* tiene reservada la CPU para sí y actualmente está activo. En cambio, un thread en estado *waiting* está esperando a que ocurra un cierto evento para poder ser elegible para continuar con su ejecución. Un thread en estado *ready* es elegible para ejecutar y lo hará a penas éste sea elegido por el scheduler. Las transiciones entre estados son exactamente las mismas que teníamos en el modelo de procesos.

Es importante notar que cada thread tiene su **propio stack**. Cada stack de un thread contiene un stack frame por cada procedimiento invocado que todavía no ha terminado de ejecutar. Estos stack frames contienen las variables locales de los procedimientos y las direcciones de retorno. Cada thread, generalmente, va a llamar a distintos procedimientos y, en consecuencia, van a tener una historia de ejecución distinta. Es por este motivo que **cada thread debe tener su propio stack**.

Cuando estamos en un sistema con *multithreading*, los procesos usualmente inician con un único thread. Este thread inicial tiene la capacidad de crear nuevos threads llamando a un procedimiento *thread_ create*, que devuelve el identificador del thread creado. El thread creado va a correr automáticamente en el espacio de direcciones del thread que lo creó. En general, no hay una relación jerárquica entre threads, siendo todos los threads de un proceso iguales.

Cuando un thread termina su trabajo, debe llamar a un procedimiento *thread_ exit*. Este procedimiento se encarga de destruir el thread y que deje de ser elegible por el scheduler. También es posible que un thread se quede esperando a que termine un thread específico (identificado mediante su *tid*) al llamar a un procedimiento *thread_ join*. Este procedimiento bloquea al thread invocador hasta que el thread especificado haya terminado.

Otro procedimiento común de threads es *thread_ yield*, que permite que un thread voluntariamente ceda su tiempo de CPU a otro thread. Este procedimiento es importante, dado que no hay ninguna interrupción de reloj que fuerce la multiprogramación entre threads de un mismo proceso.

2.3.3. POSIX Threads

Para hacer posible escribir código portable, la IEEE ha definido un estándar para threads. El paquete que define se conoce como **Pthreads**. Recordemos que la mayoría de los sistemas UNIX son POSIX compatible, por lo que dan soporte a este paquete. Algunas de las funciones más importantes son las siguientes:

- **Pthread_create**. Crea un nuevo thread dentro del proceso.
- **Pthread_exit**. Destruye al thread invocador.
- **Pthread_join**. Espera a que termine un thread específico.
- **Pthread_yield**. Libera la CPU para que otro thread pueda correr.
- **Pthread_attr_init**. Crea e inicializa la estructura de atributos de un thread.
- **Pthread_attr_destroy**. Borra la estructura de atributos de un thread.

El estándar POSIX.1 requiere que todos los threads compartan un rango de atributos, es decir, se tienen atributos por proceso que son compartidos por todos los threads asociados a ese proceso. Además, establece ciertos atributos que deben ser privados para cada thread.

Items compartidos entre threads de un proceso	Items privados por thread
process id	thread id
parent process id	afinidad de CPU
process group id y session id	máscara de señales
controlling terminal	Stack
descriptores de archivos abiertos	variable errno
signal dispositions	capabilities
directorio actual	signal stack
nice value (parámetro del scheduler)	

Figura 2.6: Items compartidos por los threads de un proceso e items privados por thread según Pthreads.

En resumen, todos los threads en Pthreads tienen ciertas propiedades. Cada uno tiene un identificador (**tid**), un conjunto de registros (que incluye al program counter), un conjunto de atributos, que son almacenados en la estructura de atributos. Los atributos incluyen el tamaño del stack, parámetros de scheduling y otros elementos que un thread necesite usar. Se garantiza que el *tid* es único solo dentro de un proceso: en todas las funciones de pthreads que acepten un tid como parámetros, ese identificador por definición hace referencia a un thread dentro del mismo proceso que el thread invocador.

Podemos crear un thread usando *pthread_ create*. Esta llamada devuelve el *tid* del thread creado. Cuando un thread ha terminado su trabajo, puede pasar a estado *terminated* llamando a *pthread_ exit*. Esta llamada detiene al thread y libera su stack.

Normalmente, un thread necesita esperar a que otro thread realice cierta tarea y termine antes de poder ejecutar. El thread que necesita esperar puede llamar a *pthread_ join* para esperar por un thread específico. Para indicar cuál de todos los threads se debe esperar, se pasa por parámetro su *tid*.

A veces un thread puede continuar ejecutando, pero decide liberar la CPU para que otro thread pueda continuar. Esto se logra llamando a *pthread_ yield*. En principio, no hay una system call equivalente para procesos, justamente porque la asunción es que los procesos compiten ferozmente por los recursos del sistema y cada uno quiere obtener todo el tiempo de CPU que sea posible. Sin embargo, como los threads de un proceso trabajan en conjunto, a veces es deseable liberar la CPU para darle oportunidad a otro thread para ejecutar.

Los dos procedimientos siguientes trabajan con los atributos. *pthread_ attr_ init* crea la estructura de atributos asociada con un thread y la inicializa con sus valores por defecto. Estos valores pueden ser modificados en la estructura de atributos. Finalmente, *pthread_ attr_ destroy* borra la estructura de atributos, liberando memoria. No afecta a los threads asociados a esta estructura.

2.3.4. Implementando Threads

Hay dos lugares principales para implementar threads: user space y el kernel. La elección no está tan clara, y es posible una implementación híbrida donde tenemos threads de usuario y threads del kernel que se mapean entre sí. A continuación, vamos a describir estos métodos, junto con sus ventajas y desventajas.

El primer método consiste en colocar todo el paquete de threads en user space. El kernel no sabe de la existencia de los threads. Para el kernel, solo hay procesos single-threaded. La primera ventaja es que el paquete de threads puede ser implementado en un sistema operativo que no soporte threads. De esta

manera, los threads son implementados por una biblioteca de usuario.

Todas estas implementaciones tienen la misma estructura general, como se muestra en la Fig. 2.7(a). Los threads corren encima de un run-time system, que es un conjunto de procedimientos que manejan threads, que incluyen: *pthread_ create*, *pthread_ exit*, etc.

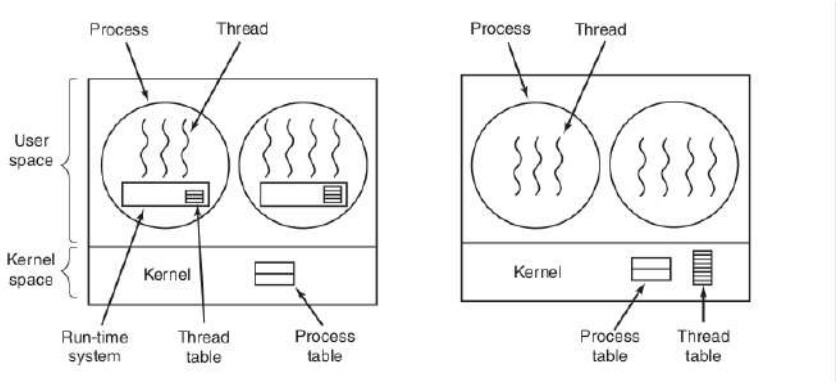

Figura 2.7: (a) Un paquete de threads de nivel de usuario. (b) Un paquete de threads manejado por el kernel.

Implementación en User-Space

Cuando un thread es administrado en user space, cada proceso necesita su propia **tabla de threads** privada para mantener un registro de los threads en ese proceso. Esta tabla es análoga a la tabla de procesos del kernel, solo que mantiene la información de las propiedades por thread y no las propiedades por proceso, como el program counter, stack pointer, registros, estado, etc.

La tabla de threads es administrada por el run-time system. Cuando un thread pasa del estado *ready* o a *waiting*, la información necesaria para restaurarlo es guardada en la tabla de threads, exactamente de la misma manera que el kernel se guarda la información de los procesos en su tabla de procesos.

Cuando un thread hace algo que lo bloquea localmente, por ejemplo, se queda esperando a otro thread en su proceso, llama a uno de los procedimientos del run-time system. Este procedimiento revisa si el thread debe pasar al estado *waiting*. Si lo debe hacer, se guarda los registros del thread en la tabla de threads, y busca en la tabla un thread en estado *ready* para ponerlo a ejecutar, restaura sus registros y lo pone a ejecutar. Intercambiar los threads de esta manera es al menos un orden de magnitud más rápido que llamar al kernel para que realice la misma tarea, siendo un gran argumento a favor de los paquetes de threads de usuario.

Cuando un thread terminó de correr por el momento, por ejemplo, cuando llama a *thread_ yield* puede guardar la información del thread en la tabla de threads por su cuenta. Además, puede llamar al scheduler de threads para elegir otro thread a ejecutar. El procedimiento que guarda el estado del thread y el scheduler **son procedimientos locales**, por lo que invocarlos es mucho más eficiente que hacer una system call. Entre otras cosas, no se necesita levantar una excepción, ni hacer un cambio de contexto, ni flushear la cache, etc. Esto hace que esta manera de trabajar con threads sea muy eficiente.

Otra ventaja que tienen los threads de usuario es que permiten que cada proceso tenga su propio algoritmo de scheduling personalizado. Además, escalan mejor, ya que los threads del kernel necesitan invariablemente usar espacio en la memoria del kernel para guardar tablas y el stack, que puede llegar a ser un problema si se tiene una gran cantidad de threads.

A pesar de todas estas ventajas, los paquetes de threads de usuario tienen algunos problemas considerables. El primero de ellos es el problema de cómo se implementan las system calls bloqueantes. Supongamos que un thread quiere leer las teclas presionadas del teclado. Hacer que el thread realice la system call es inaceptable, ya que detendría a todos los demás threads del proceso. Uno de los principales objetivos de tener threads es justamente permitir que cada uno pueda hacer llamadas bloqueantes, pero prevenir que un thread que está esperando bloquee a todos los demás.

Otro problema con los paquetes de threads de usuario es que si un thread comienza a ejecutar, ningún otro thread en ese proceso podrá correr salvo que el primero decida voluntariamente liberar la CPU. Dentro de un proceso, no hay interrupciones de clock, haciendo imposible implementar un scheduling con desalojo entre los threads de un proceso.

Implementando Threads en el Kernel

Ahora, consideremos el caso en el que el kernel conoce y administra los threads. No se necesita de un run-time system en cada proceso, sino que solo se necesita uno solo (ver Fig. 2.7(b)). Tampoco hay una tabla de threads por cada proceso; en cambio, el kernel mantiene una tabla de threads única que mantiene un registro de todos los threads en el sistema. Cuando un thread quiere crear otro thread o destruir uno existente, hace una llamada al kernel, quien se encarga de la creación o destrucción actualizando esta tabla.

La tabla de threads del kernel almacena los registros, el estado y otra información de cada thread. La información es la misma que teníamos en el thread de usuario, pero ahora se almacena en espacio del kernel. Es decir, en el espacio del kernel no solo se almacena la tabla de proceso, sino que ahora también se guarda la tabla de threads.

Todas las llamadas que podrían bloquear a un thread son implementadas como system calls, a un costo considerablemente mayor que todos los procedimientos que teníamos en el modelo anterior. Cuando un thread se bloquea, el kernel puede correr otro thread del mismo proceso o un thread de otro proceso. Notemos la diferencia con los threads de usuario, donde el run-time system solo podía elegir threads de un mismo proceso.

Debido al relativamente alto costo de crear y destruir threads en el kernel, algunos sistemas deciden reciclar los threads. Esto es, cuando un thread es terminado, se lo marca como *no-ejecutable*, pero las estructuras de datos del kernel no son modificadas. Más adelante, cuando un nuevo thread debe ser creado, se reactiva el thread viejo, ahorrándonos cierto overhead.

Cuando ocurre una system call bloqueante, los threads del kernel pueden revisar fácilmente si el proceso tiene algún otro thread en estado *ready*, pudiendo elegir uno de ellos mientras se espera a que termine la system call bloqueante. Esto resuelve el problema que teníamos en los threads de usuarios, que no podían (en principio) pasarle el control a otro thread del mismo proceso.

Implementación Híbrida

Otra manera de implementar threads consiste en usar threads del kernel y luego mapearlos a threads de usuario. Cuando se usa esta estrategia, es posible configurar la cantidad de threads del kernel que se utilizarán y la cantidad de threads de usuario por cada thread del kernel. De esta manera, el kernel solo es consciente de la existencia de los threads del kernel y solo considera éstos al momento de hacer el scheduling. Algunos de los threads del kernel tendrán encima múltiples threads de usuario. Estos threads de usuario son creados, destruidos y participan del scheduling tal cual lo haría un thread de usuario en el primer modelo que vimos.

Para evitar que un thread de usuario bloquee a todos los demás threads dentro de un mismo thread del kernel, se puede utilizar una técnica conocida como **scheduler activation**. La idea es que cuando el kernel sepa que un thread se ha bloqueado (porque ejecutó una system call bloqueante), el kernel notifica al run-time system del proceso, pasándole como parámetros el *tid* del thread en cuestión y una descripción del evento que ocurrió. Para que esta notificación ocurra, el kernel activa al run-time system en una dirección inicial conocida, similar a una señal en UNIX. Este mecanismo se conoce como **upcall**.

Una vez activado, el run-time system puede elegir un nuevo proceso, típicamente marcando al thread actual como bloqueado y tomando otro thread en estado *ready*. Más tarde, cuando el kernel se entere que el thread original puede continuar con su ejecución, hace otra upcall al run-time system para informárselo.

Capítulo 3

Interprocess Communication

En este capítulo, vamos a introducir los principales métodos de intercomunicación entre procesos, bajo el estándar POSIX.1. Permitir la intercomunicación entre procesos trae una serie de ventajas. Por un lado, es posible que varias aplicaciones puedan estar interesadas en la misma pieza de información. Utilizando los métodos de IPC, es posible proveer un ambiente que permita el acceso concurrente a esta información.

Además, podemos hacer que varios procesos cooperen para resolver una tarea particular, mejorando en performance, a partir de dividirla en subtareas más chicas y que cada subtarea sea ejecutada en paralelo. Notemos la diferencia entre *concurrencia* y *paralelismo*. Un sistema concurrente permite que varias tareas tengan cierto progreso en su ejecución, mientras que un sistema paralelo permite que se realice más de una tarea de forma simultánea. Por lo tanto, podemos tener concurrencia sin paralelismo, pero no viceversa. Luego, este tipo de mejora solo se puede obtener si la computadora tiene múltiples procesadores.

Por último, nos permite tener un diseño modular, dividiendo las funciones del sistema en diferentes procesos, mejorando la seguridad del sistema. Por ejemplo, originalmente los browsers corrían como un solo proceso que atendía varias pestañas, por lo que si alguna de éstas se colgaba, se colgaba todo el navegador. Al aislar las distintas pestañas en procesos distintos conseguimos que, en caso de que se cuelgue un proceso, los demás pueden seguir funcionando. Incluso, es posible ejecutar algunos procesos en un **sandbox**, limitando el acceso a disco y a la red, y minimizando los efectos de cualquier vulnerabilidad en el sistema.

3.1. Introducción

Los procesos cooperativos necesitan de un mecanismo de comunicación, conocido como *interprocess communication* (**IPC**), que les permita intercambiar datos. Tradicionalmente, existen dos modelos de comunicación entre procesos: memoria compartida y pasaje de mensajes. En el modelo de **memoria compartida**, lo que hace es establecer una región de memoria para que sea compartida por procesos cooperativos. En el modelo de **pasaje de mensajes**, la comunicación se realiza a partir del intercambio de mensajes entre procesos cooperativos. Ambos modelos se contrastan en la Fig 3.1.

Los mecanismos de pasaje de mensajes son útiles para intercambiar pequeñas cantidades de datos, ya que no es necesario controlar el acceso concurrente a la cola de mensajes. Además, es más fácil de implementar en sistemas distribuidos, comparado con implementar memoria compartida. Por otro lado, la memoria compartida puede ser más rápida que los mecanismos de pasaje de mensajes. Los mecanismos de pasaje de mensaje requieren de la intervención del kernel en cada mensaje que se envía. En cambio, en los modelos de memoria compartida, solo se necesita la intervención del kernel para establecer la región de memoria compartida. Una vez esté establecida, no se requiere de la intervención del kernel.

En el modelo de programación tradicional de UNIX, tenemos múltiples procesos corriendo en un sistema. Recordemos que cada proceso cuenta con su propio espacio de direcciones. Si queremos que los

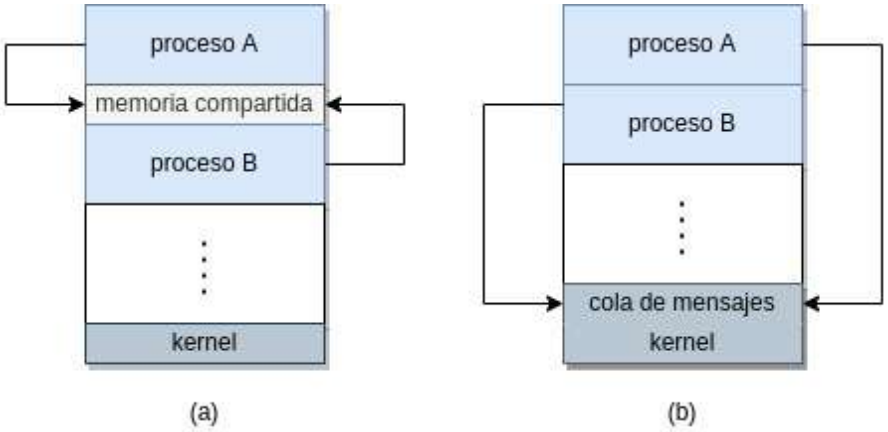

Figura 3.1: Modelos de comunicación. (a) Memoria Compartida. (b) Pasaje de Mensajes.

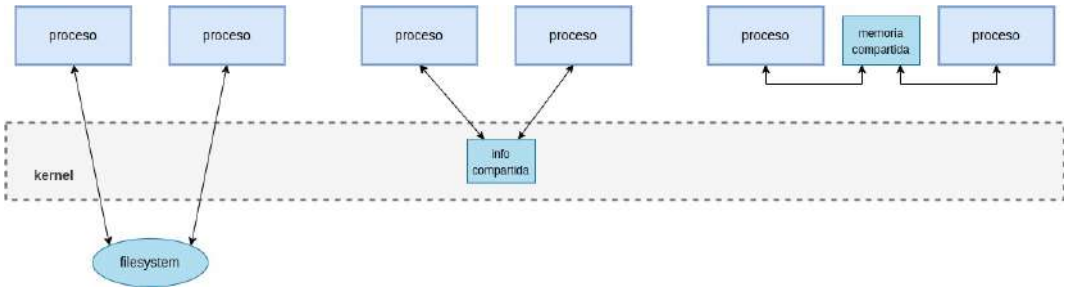

Figura 3.2: Tres maneras de compartir información entre procesos de UNIX.

procesos cooperen entre sí, existen varias maneras de compartir información. En la Fig. 3.2 se muestra un resumen de éstas.

1. Los dos procesos de la izquierda están compartiendo información que reside en un archivo del filesystem. Para acceder a estos datos, cada proceso debe pasar por medio del kernel, llamando a las system calls adecuadas (`read, write`, etc.).

2. Los dos procesos del medio están compartiendo información que reside dentro del kernel. Un caso de ejemplo son los pipes. Nuevamente, cada operación de lecto-escritura involucra hacer una system call.

3. Los dos procesos de la derecha están compartiendo información que reside en una región de memoria compartida. Una vez establecida la región de memoria compartida en cada proceso, éstos pueden acceder a los datos sin tener que involucrar al kernel.

3.1.1. Persistencia de Objetos IPC

Una de las características importantes de cualquier canal de comunicación es la persistencia de sus datos. Podemos clasificar a los distintos objetos destinados a la intercomunicación entre procesos dependiendo de cuánto tiempo permanece en existencia.

- Un objeto IPC es proceso-persistente si continúa en existencia hasta que el último proceso que mantiene al objeto abierto lo cierra. Ejemplos de objetos proceso-persistentes son los pipes y FIFOs.

- Un objeto IPC es kernel-persistente si continúa en existencia hasta que se reinicie el kernel o hasta que el objeto sea explícitamente borrado. Un ejemplo de esto es la memoria compartida.

- Un objeto IPC es filesystem-persistente si continúa en existencia hasta que el objeto sea explícitamente borrado. El objeto retiene su valor incluso si el kernel es reiniciado. Si la memoria compartida es implementada usando archivos mapeados a memoria, entonces es filesystem-persistente.

Hay que tener cuidado cuando pensamos en la persistencia de los datos en un objeto IPC, porque no siempre resulta evidente. Por ejemplo, si bien los datos de un pipe son mantenidos dentro del kernel, los pipes son proceso-persistentes y no kernel-persistentes. Esto quiere decir que, luego de que el último proceso que tiene ese pipe abierto para lectura lo cierra, el kernel descarta todos los datos y borra el pipe. Similarmente, incluso en el caso de los FIFOs que tienen nombres dentro del filesystem, también son proceso-persistentes porque todos los datos en un FIFO son descartados luego de que el último proceso que tenga a ese FIFO abierto lo cierre.

3.2. Paradigma open-read-write-close

Antes de hablar de los distintos objetos IPC, es necesario que se tenga una idea de cómo se representan y se acceden los archivos y dispositivos en un sistema UNIX. Las primitivas de entrada y salida de UNIX siguen un paradigma conocido como **open-read-write-close**. Antes de que un proceso de usuario pueda realizar operaciones de E/S, se debe llamar a `open` para especificar el archivo a utilizar (*pathname*) y el modo de acceso (*flags*).

```
int open(const char *pathname, int flags);
```

La llamada a open devuelve un número entero (**file descriptor**) que el proceso va a utilizar cada vez que quiera realizar operaciones de E/S sobre el archivo. Una vez abierto un objeto, el proceso realiza una o más llamadas a `read` y `write` para transferir datos. `Read` transfiere datos desde el archivo al espacio de direcciones del proceso; `write` transfiere datos desde el espacio de direcciones al archivo. Tanto `read` como `write` toman tres parámetros que especifican el file descriptor a utilizar, la dirección del buffer y el número de bytes a transferir.

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Una vez completadas todas las operaciones de transferencia, el proceso llama a `close` para informarle al sistema operativo que ha terminado de usar el objeto.

```
int close(int fd);
```

Cuando un proceso termina, el sistema operativo automáticamente cierta todos los descriptores que quedaron abiertos.

Los **file descriptors** son números enteros que actúan como índices de una tabla de archivos abiertos. Cada proceso en UNIX viene con su propia tabla (en su PCB) al momento de ser creado. Los file descriptors son usados por el kernel para referenciar a los archivos abiertos que tiene cada proceso. Cada entrada de esta tabla apunta a un archivo específico (ver Fig. 3.3).

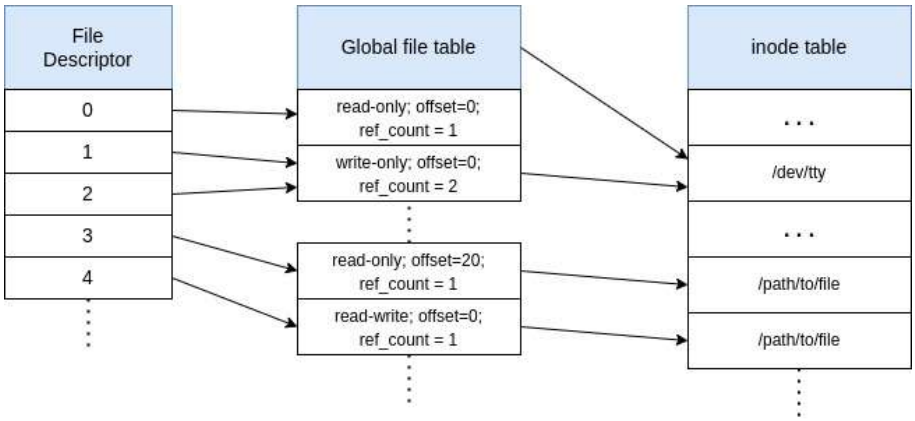

Figura 3.3: Esquema general sobre referencias de file descriptors.

La mayoría de los procesos esperan tener abiertos 3 file descriptors (las entradas 0, 1 y 2 de esta tabla). Estos file descriptors se corresponden con:

- 0 = entrada estándar.
- 1 = salida estándar.
- 2 = error estándar.

3.2.1. Efectos de fork, exec y exit sobre objetos IPC

Recordemos que UNIX utiliza las system calls `fork` y `execve` para iniciar nuevas aplicaciones. Es un procedimiento de dos pasos. En el primero, `fork` crea una copia separada del proceso que actualmente está corriendo. En el segundo, la nueva copia reemplaza su espacio de direcciones con la aplicación deseada. Cuando un programa llama a `fork`, la copia recién creada hereda todos los accesos a todos los archivos abiertos (esto incluye a los pipes, memory mappings, sockets, etc.). Cuando un programa llama a `execve`, la nueva aplicación retiene el acceso a todos los sockets abiertos; sin embargo, los memory mappings son deshechos.

Pipes, FIFOs y Sockets.

- *fork*. El proceso hijo obtiene copias de todos los file descriptors abiertos del padre.
- *exec*. Todos los file descriptors abiertos permanecen abiertos, salvo que se especifique lo contrario con algún flag.
- *exit*. Todos los file descriptors abiertos son cerrados; todos los datos son removidos en el último cierre.

mmap memory mappings.

- *fork*. Los mapeos a memoria en el padre son heredados y mantenidos en el proceso hijo.
- *exec*. Los mapeos a memoria son desmapeados.
- *exit*. Los mapeos a memoria son desmapeados.

Internamente, el sistema operativo mantiene un contador de referencias asociado a cada objeto, de manera tal que sabe cuántos procesos tienen acceso al mismo. Cuando se termina de usar un objeto, se debe llamar a `close`. Internamente, el sistema operativo decrementa el contador de referencias del objeto y, en caso de llegar a cero, lo destruye.

3.3. IPC con Pasaje de Mensajes

El pasaje de mensajes provee un mecanismo que permite a los procesos comunicarse sin tener que compartir el mismo espacio de direccionamiento. Es especialmente útil en los sistemas distribuidos, donde los procesos pueden encontrarse en distintas computadoras.

3.3.1. Ordinary pipes

Un pipe ordinario, también llamado **pipe anónimo**, es un archivo temporal y anónimo que reside en memoria. Actúa como buffer para leer y escribir de manera secuencial. Esto es, un `write` a un pipe o un FIFO siempre agrega los datos al final, y un `read` siempre devuelve lo que está al principio. Si se intenta usar la system call `lseek` para avanzar sobre el archivo, se levanta un error.

Permiten que dos procesos puedan comunicarse de la forma estándar productor-consumidor: el productor escribe de un lado del pipe (write end) y el consumidor lee del otro lado (read end). Como resultado, los pipes ordinarios son half-duplex (o unidireccionales). Su principal limitante es que los pipes ordinarios no tienen nombre y, por tanto, solo pueden usarse entre procesos relacionados (normalmente, padre-hijo).

Los pipes son creados a partir de llamar a la función

```
int pipe(int fd[2])
```

Esta función devuelve dos file descriptors: `fd[0]` (read end) y `fd[1]` (write end). Ambos archivos ya se encuentran abiertos para lectura y escritura, respectivamente. Los pipes en UNIX son un tipo especial de archivo, y pueden ser accedidos usando las system calls `read` y `write` ordinarias.

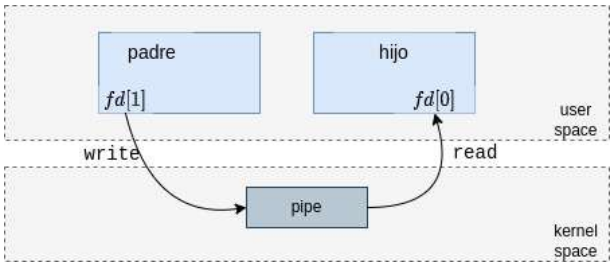

Figura 3.4: Pipe entre dos procesos.

Un pipe ordinario no puede ser accedido por fuera del proceso que lo creó. Típicamente, se utilizan para intercomunicar procesos relacionados de la siguiente manera. Un proceso padre crea un pipe y lo usa para comunicarse con un proceso hijo que crea vía `fork`. Recordemos que un proceso hijo hereda los archivos abiertos de su padre.

Luego, el proceso padre cierra el read end del pipe, mientras que el proceso hijo cierra el write end. Esto nos provee un canal unidireccional de datos entre los dos procesos, como podemos ver en la Fig. 3.4 (las flechas representan el flujo de datos).

También podemos crear pipes desde la consola usando el caracter "|". Este comando permite comunicar dos procesos al redireccionar la salida de uno como entrada del otro utilizando un pipe como medio. Por ejemplo, si ingresamos por consola un comando como

```
who | sort | lp
```

el shell crea los tres procesos con dos pipes entre ellos. Además, redirecciona el read end de cada pipe a la entrada estándar de cada proceso y el write end de cada pipe a la salida estándar de cada proceso (según corresponda). Mostramos esto en la Fig. 3.5. Para redireccionar la entrada o salida estándar de

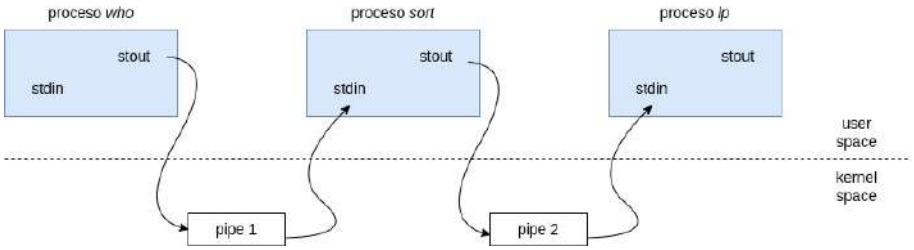

Figura 3.5: Pipes entre tres procesos en un pipeline del shell.

un proceso, se utiliza la system call

```
int dup2(int oldfd, int newfd)
```

Esta system call crea una copia del file descriptor `oldfd`, usando el número del file descriptor especificado en `newfd`. Si el file descriptor `newfd` estaba abierto, se lo cierra antes de ser reusado. De esta manera, tomando a la entrada o salida estándar como `newfd`, podemos redireccionarlas al archivo que queramos.

Todos los pipes mostrados hasta ahora son half-duplex (unidireccionales), proveyendo un canal de datos en un solo sentido. Cuando se desea tener un canal de datos bidireccional, debemos crear dos pipes y usar uno para cada dirección. En particular, la secuencia de pasos a tomar es la siguiente:

1. crear pipe1 (fd1[0], fd1[1]) y pipe2 (fd2[0] y fd2[1]),

2. fork,
3. el proceso padre cierra el read end del pipe1 (fd1[0]),
4. el proceso padre cierra el write end del pipe2 (fd2[1]),
5. el proceso hijo cierra el write end del pipe1 (fd1[1]),
6. el proceso hijo cierra el read end del pipe2 (fd2[0]).

El hecho de que se necesite de la relación de padre-hijo para utilizar estos pipes quiere decir que este mecanismo solo sirve para la comunicación entre procesos en la misma máquina. Una vez que los procesos hayan finalizado su comunicación o hayan terminado, este tipo de pipes anónimos dejan de existir.

Ejemplo de Uso

En la Fig. 3.6 se muestra cómo podemos usar pipes.

```
#include "unpipc.h"

void client(int, int), server(int, int);

int main(int argc, char**argv)
{
    int pipe1[2], pipe2[2];
    pid_t childpid;

    pipe(pipe1);
    pipe(pipe2);
    if((childpid = fork()) == 0)
    {
        close(pipe1[1]);
        close(pipe2[0]);

        server(pipe1[0], pipe2[1]);

        exit(0);
    }
    close(pipe1[0]);
    close(pipe2[1]);

    client(pipe2[0], pipe1[1]);

    waitpid(childppid, NULL, 0);
    exit(0);
}
```

Figura 3.6: Función *main* para cliente-servidor usando dos pipes.

```c
#include "unpipc.h"

void client(int readfd, int writefd)
{
    size_t len;
    ssize_t n;
    char buff[MAXLINE];
    fgets(buff, MAXLINE, stfin);
    len = strlen(buff);
    if(buff[len-1] == '\n'){len--;}

    write(writefd, buff, len);
    while((n = read(readfd, buff, MAXLINE))>0)
    {
        write(STDOUT_FILENO, buff, n);
    }
}
```

Figura 3.7: Función *client* para cliente-servidor usando dos pipes (o FIFOs).

```c
#include "unpipc.h"

void server(int readfd, int writefd)
{
    int fd;
    ssize_t n;
    char buff[MAXLINE + 1];

    if((n = read(readfd, buff, MAXLINE)) == 0)
    {
        err_quit("EOF while reading pathname");
    }

    buff[n] = '\0'M
    if((fd = open(buff, O_RDONLY)) < 0)
    {
        snprintf(buff + n, sizeof(buff) - n, ": can't open, %s\n", strerror(errno));
        n = strlen(buff);
        write(writefd, buff, n);
    } else
    {
        while((n = read(fd, buff, MAXLINE))  > 0){
            write(writefd, buff, n);
        }
        close(fd);
    }

}
```

Figura 3.8: Función *server* para cliente-servidor usando dos pipes (o FIFOs).

3.3.2. Named pipes

Como estuvimos viendo, los pipes anónimos no tienen nombre, y su mayor desventaja es que tan solo pueden ser usados entre procesos que tengan un padre en común. Dos procesos no relacionados no pueden usar un pipe anónimo para intercambiar datos.

Los named pipes, también llamados FIFOs, proveen una herramienta mucho más poderosa de comunicación. Un FIFO es un archivo especial (un named pipe) similar a un pipe, salvo porque debe ser accedido como parte del filesystem. Puede ser abierto por múltiples procesos para lectura y escritura. Cuando los procesos están intercambiando datos a través de un FIFO, el kernel pasa todos los datos internamente sin escribirla al filesystem. Por lo tanto, un archivo especial FIFO realmente no tiene contenido en el filesystem; la entrada en el filesystem solo sirve como punto de referencia para que los procesos puedan acceder al FIFO por medio de un nombre en el filesystem.

Esto permite que procesos no relacionados accedan a un mismo FIFO para intercambiar datos. Además, los named pipes siguen existiendo luego de que los procesos que se están comunicando hayan terminado (aunque los datos que pudieron quedar sin leer se pierden).

Podemos crear un archivo especial FIFO mediante la función

```
int mkfifo(const char *pathname, mode_t mode);
```

El pathname es el nombre del FIFO que queremos crear (un pathname ordinario de UNIX). El modo especifica los permisos del archivo, similar al segundo parámetro de la system call open. Una vez creado el FIFO, éste puede ser manipulado con las system calls tradicionales sobre archivos: `open`, `read`, `write` y `close`.

El kernel administra exactamente un pipe object por cada archivo especial FIFO que es abierto por al menos un proceso. El FIFO debe ser abierto en los dos extremos antes de que se puedan pasar datos. Normalmente, abrir un FIFO es bloquea al proceso hasta que el otro extremo también sea abierto.

Una vez creado este FIFO, debe ser abierto para lectura o escritura, usando la system call open. Un FIFO debe ser abierto en modo solo-lectura o en modo solo-escritura. No debe ser abierto en modo lectura-escritura, porque un FIFO es half-duplex.

Ejemplo de Uso

En la Fig. 3.9 se muestra un cómo podemos usar FIFOs.

```c
#include "unpipc.h"

#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"

void client(int, int), server(int, int);

int main(int argc, char**argv)
{
    int readfd, writefd;
    pid_t childpid;

    if((mkfifo(FIFO1, FILE_MODE) < 0) && (errno != EEXIST)){
        err_sys("can't create %s", FIFO1);
    }
    if((mkfifo(FIFO2, FILE_MODE) < 0) && (errno != EEXIST)){
        unlink(FIFO1);
        err_sys("can't create %s", FIFO2);
    }
    if((childpid = fork()) == 0)
    {
        readfd = open(FIFO1, O_RDONLY, 0);
        writefd = open(FIFO2, O_WRONLY, 0);

        server(readfd, writefd);
        exit(0);
    }
    writefd = open(FIFO1, O_WRONLY, 0);
    readfd = open(FIFO2, O_RDONLY, 0);

    client(readfd, writefd);

    waitpid(childppid, NULL, 0);
    close(readfd);
    close(writefd);
    unlink(FIFO1);
    unlink(FIFO2);
    exit(0);
}
```

Figura 3.9: Función *main* para cliente-servidor usando dos FIFOs.

3.3.3. Reglas Adicionales

Notemos algunas reglas adicionales que afectan a la lecturas y escrituras sobre pipes o FIFOs.

- Si pedimos leer más datos de los que actualmente están disponibles en el pipe o FIFO, tan solo se devuelven los datos disponibles. Al momento de utilizar pipes o FIFOs, debemos tener en cuenta la posibilidad de que se nos devuelva menos bytes de los que habíamos pedido leer.

- Si el número de bytes a escribir es menor o igual a PIPE_BUF[1], se garantiza que la escritura será atómica. Esto significa que si dos procesos intentan escribir sobre un mismo pipe o FIFO al mismo tiempo o bien todos los datos del primer proceso son escritos, seguido de todos los datos del segundo proceso, o viceversa. El sistema no va a intercalar los datos de ambos procesos. Sin embargo, si el número de bytes a escribir es mayor a PIPE_BUF, no se puede garantizar que la

[1]POSIX.1 establece que PIPE_BUF debe ser al menos 512 bytes. En Linux, es 4096 bytes.

operación sea atómica.

3.4. IPC con Memoria Compartida

La memoria compartida es la forma más rápida de IPC disponible. El kernel no se involucra de ningún modo en el pasaje de datos entre estos procesos. Con esto queremos decir que no es necesario ejecutar ninguna system call para intercambiar los datos. Evidentemente, el kernel debe establece los mapeos de memoria que le permiten a los procesos compartir memoria, y luego administrar esta memoria con el paso del tiempo (page faults, etc.). Lo que normalmente se requiere, sin embargo, es una forma explícita de sincronización entre procesos.

Para ver por qué la memoria compartida es considerada la forma más rápida de IPC, consideremos los pasos típicos involucrados en un programa que copia archivos en el ejemplo cliente-servidor utilizando como canal de IPC pipes o FIFOs.

1. El servidor lee el archivo de entrada. El archivo de datos es leído por el kernel a su memoria y luego es copiado del kernel al proceso.

2. El servidor escribe estos datos en un mensaje, usando un pipe o un FIFO, teniendo que copiar los datos del proceso al kernel.

3. El cliente lee los datos del canal de intercomunicación, teniendo nuevamente que copiar los datos desde el kernel al proceso.

4. Finalmente, los datos son copiados desde el buffer del cliente al archivo de salida (la salida estándar).

Un total de cuatro copias de los datos son requeridas. Adicionalmente, estas cuatro copias se realizan entre el kernel y un proceso, siendo estas copias costosas (relativo a una copia de datos entre el mismo kernel o entre un mismo proceso).

El problema de estos mecanismos de comunicación es que, para que dos procesos puedan intercambiar información, es necesario pasar a través del kernel. Los mecanismos de memoria compartida evitan pasar por el kernel al permitir que dos o más procesos compartan una región de memoria.

Los procesos pueden intercambiar información a partir de leer y escribir sobre esta región compartida. Una vez que se establece esta región, los procesos son responsables de asegurarse de que no se estén pisando entre sí— y no es responsabilidad del sistema operativo garantizar un uso adecuado de esta memoria. Es importante notar que la región de memoria debe estar mapeada en ambos espacios de direcciones (ver Fig. 3.10).

Ahora, si utilizamos memoria compartida, los pasos para el ejemplo cliente-servidor son los siguientes:

1. El servidor obtiene acceso al objeto de memoria compartida (por ejemplo, con un semáforo).

2. El servidor lee el archivo de entrada al objeto de memoria compartida. El segundo parámetro de la system call read, que se corresponde con la dirección del buffer de datos, apunta al objeto de memoria compartida.

3. Cuando se completa la lectura, el servidor notifica al cliente, usando el semáforo.

4. El cliente escribe los datos desde el objeto de memoria compartida al archivo de salida.

De esta forma, los datos tan solo son copiados dos veces— desde el archivo de entrada a la memoria compartida y desde la memoria compartida al archivo de salida.

3.4.1. mmap, munmap y msync

La comunicación entre procesos usando memoria compartida requiere que los procesos que quieran comunicarse puedan establecer una región de memoria compartida. La memoria compartida en POSIX se organiza usando **memory-mapped files**, que asocian la región de memoria compartida con un archivo

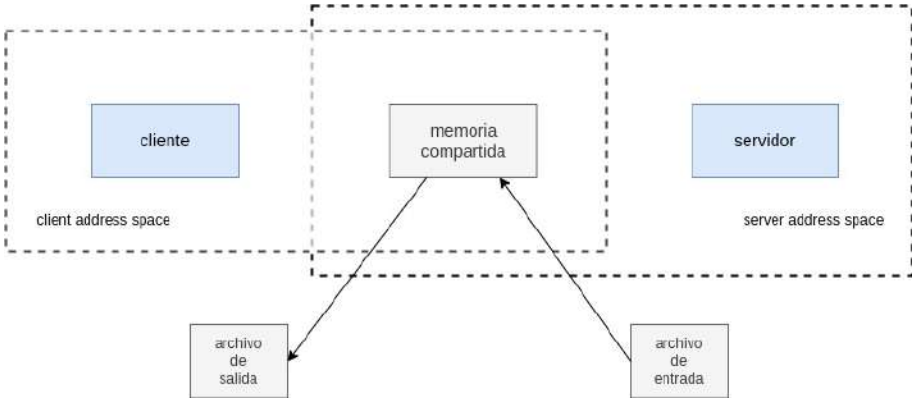

Figura 3.10: Copiando archivo de datos desde un servidor a un cliente usando memoria compartida.

(o un objeto POSIX de memoria compartida). Para esto, se utiliza la función

```
void *mmap(void *addr, size_t len, int prot, int flags, int fs, off_t offset)
```

- *addr* puede especificar la dirección de inicio dentro del proceso donde se mapeará el objeto identificado por el file descriptor *fd*. Normalmente, se completa con un *null pointer*, indicándole al kernel que elija una dirección "cualquiera". En cualquier caso, el valor de retorno de la función es la dirección inicial donde el file descriptor ha sido mapeado.
- *len* es el número de bytes a mapear dentro del espacio de direcciones del proceso, empezando a *offset* bytes del principio del archivo. Típicamente, *offset* es 0.
- La protección de la región de memoria mapeada es especificada por el parámetro *prot*. Normalmente, se tiene acceso de lectura-escritura.
- Los *flags* son especificados por las constantes mostradas en la Fig. 3.11. Si se especifica que la región es privada, quiere decir que las modificaciones sobre los datos mapeados realizados por el proceso invocador serán visibles solo para éste y no afectará al objeto en sí (ya sea un file object o un shared memory object). Si se especifica que la región es compartida, quiere decir que las modificaciones que se realicen sobre ésta serán visibles a todos los procesos que estén compartiendo este objeto, y estos cambios sí modificarán al objeto en sí.

flags	Descripción
`MAP_SHARED`	Los cambios son guardados
`MAP_PRIVATE`	Los cambios son privados

Figura 3.11: Parámetro *flags* para `mmap`.

Una manera de compartir memoria entre un proceso padre y un proceso hijo es llamar a `mmap` especificando que el objeto será compartido antes de llamar a fork. POSIX.1 garantiza que los mapeos a memoria en el padre son retenidos por el hijo. Además, los cambios realizados por el padre serán visibles por el hijo, y viceversa. Luego de que `mmap` retorne, el archivo *fd* utilizado como parámetro puede ser cerrado. Esto no tiene efectos sobre el mapeo establecido.

Para remover un mapeo del espacio de direcciones del proceso, llamamos a

```
int munmap(void *addr, size_t len)
```

- *addr* es la dirección que había retornado `mmap`
- *len* es el tamaño de la región mapeada.

- Si la región fue mapeada como privada, los cambios realizados son descartados.

Típicamente, el algoritmo de administración de memoria del kernel mantiene al archivo original (en disco) sincronizado con la región de memoria mapeada en memoria (siempre que ésta no haya sido marcada como privada). Esto es, si modificamos una ubicación en memoria que está mapeada en memoria a un archivo, entonces, en algún tiempo futuro, el kernel actualizará el archivo correspondiente.

En algunos casos, nos interesa poder asegurarnos que el archivo en disco se corresponda con lo que tenemos en la región de memoria mapeada, y para ello llamamos a `msync`.

```
int msync(void *addr, size_t len, int flags)
```

3.4.2. Usos de mmap

La función `mmap` no solo nos sirve para compartir memoria entre procesos. Su función es más general, siendo la de mapear un archivo dentro del espacio de direcciones de un proceso. Esto nos puede servir en varios escenarios distintos.

Si tenemos un archivo (regular) abierto, podemos mapearlo a nuestro espacio de direcciones llamando a `mmap`— esta técnica se conoce como **memory-mapped I/O**. Utilizar archivos mapeados a memoria nos ahorra tener que estar pensando en E/S, pasándole la responsabilidad de hacer los cambios en disco al kernel. De esta manera, nos evitamos tener que llamar a `read`, `write` o `lseek` explícitamente, simplificando nuestro código. (Notemos que hay algunos archivos que no pueden mapearse a memoria, por ejemplo, los sockets. Para que pueda ser mapeado a memoria, es necesario que los file descriptors puedan ser accedidos usando read y write.)

Otro uso de `mmap` es provee **memoria compartida** entre procesos no relacionados. En este caso, podemos copiar en un archivo los contenidos iniciales de la memoria que se quiera compartir y, luego, compartir este archivo con otros procesos haciendo un mapeo a memoria (`MAP_SHARED`) de este archivo. De esta manera, cualquier cambio que realicen procesos sobre el archivo mapeado a memoria son visibles a todos los demás procesos. En este caso, en lugar de mapear un archivo regular, nos conviene mapear un objeto POSIX de memoria compartida para intercomunicar dos procesos no relacionados.

Para crear un objeto POSIX de memoria compartida, usamos la system call `shm_open`. Esta system call crea un archivo temporal (el objeto POSIX de memoria compartida) en un file system que reside en memoria virtual, típicamente montado bajo */dev/shm*. Inicialmente, este archivo tiene cero bytes. Para poder establecer el tamaño en bytes del objeto, usamos la función `ftruncate`. Una vez definido el tamaño del objeto, lo podemos mapear a memoria usando `mmap`.

Este mecanismo de memoria compartida funciona correctamente, y forma parte del estándar POSIX.1. La desventaja que tenemos es que es necesario crear un archivo en el filesystem, llamar a `shm_open`, y luego inicializar el archivo con ceros con `ftruncate`.

Cuando el propósito de llamar a `mmap` es proveer una pieza de memoria que será compartida mediante `fork`, podemos simplificar este escenario, dependiendo de la implementación.

1. La mayoría de sistemas operativos *proveen anonymous memory mapping*, que nos evita completamente tener que crear o abrir un archivo. En cambio, podemos especificar con un flag que el mapeo será anónimo (no va a tener un archivo asociado) y completar *fd* con -1.

2. Una alternativa a la memoria anónima consiste en utilizar archivos especiales como */dev/zero*. Este dispositivo devuelve bytes de 0s cuando es leído, y cualquier cosa que se escriba sobre este dispositivo es descartada.

Ejemplos de Uso

A continuación, se muestra cómo podemos usar mmap para compartir memoria con memoria anónima (Fig. 3.12), con un archivo (Fig. 3.13) y usando un objeto de memoria compartida POSIX (Fig. 3.14).

```c
#include "unpipc.h"

struct shared {
    sem_t mutex;
    int count;
} shared;

int main(int argc, char **argv)
{
    if(argc != 2){
        err_quit("usage: incr3_map_anon <#loops>");
    }

    int nloop = atoi(argv[1]);
    struct shared * ptr = mmap(NULL, sizeof(struct shared),
                               PROT_READ | PROT_WRITE,
                               MAP_SHARED | MAP_ANON, -1, 0);

    sem_init(&ptr->mutex, 1, 1);
    setbuf(stdout, NULL);
    if(fork() == 0){
        for(int i = 0; i < nloop; i++){
            sem_wait(&ptr->mutex);
            printf("child: %d\n", ptr->count++);
            sem_post(&ptr->mutex);
        }
        exit(0);
    }

    for(int i = 0; i < nloop; i++){
        sem_wait(&ptr>mutex);
        printf("parent: %d\n", ptr->count++);
        sem_post(&ptr->mutex);
    }
    exit(0);
}
```

Figura 3.12: Contador y semáforo en memoria (anónima) compartida

```c
#include "unpipc.h"

struct shared {
    sem_t mutex;
    int count;
} shared;

int main(int argc, char **argv)
{
    if(argc != 3){
        err_quit("usage: incr3 <pathname> <#loops>");
    }
    int nloop = atoi(argv[2]);

    int fd = open(argv[1], O_RDWR | O_CREAT, FILE_MODE);
    write(fd, &shared, sizeof(struct shared));
    struct shared *ptr = mmap(NULL, sizeof(struct shared),
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);

    sem_init(&ptr->mutex, 1, 1);
    setbuf(stdout, NULL);
    if(fork() == 0){
        for(int i = 0; i < nloop; i++){
            sem_wait(&ptr->mutex);
            printf("child: %d\n", ptr->count++);
            sem_post(&ptr->mutex);
        }
        exit(0);
    }

    for(int i = 0; i < nloop; i++){
        sem_wait(&ptr>mutex);
        printf("parent: %d\n", ptr->count++);
        sem_post(&ptr->mutex);
    }
    exit(0);
}
```

Figura 3.13: Contador y semáforo en memoria compartida (memory mapped file).

```c
#include "unpipc.h"

int main(int argc, char **argv)
{
    int flags = O_RDWR | O_CREAT;
    while((int c = getopt(argc, argv, "e")) != -1){
        switch(c){
            case 'e':
                flags |= O_EXCL;
                break;
        }
    }
    if (optind != argc - 2){
        err_quit("usage: shmcreate [ -e ] <name> <length>");
    }
    off_t length = atoi(argv[optind + 1]);
    int fd = shm_open(argv[optind + 1], flags, FILE_MODE);
    ftruncate(fd, length);
    char *ptr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    exit(0);
}
```

Figura 3.14: Crear un objeto de memoria compartida POSIX del tamaño especificado.

3.5. Sockets

En un sistema UNIX, la abstracción central en la comunicación entre procesos distribuidos son los **sockets**. Un par de procesos pueden comunicarse a través de una red utilizando un par de sockets, donde cada proceso tiene su propio socket. Podemos pensar a los sockets como una generalización del mecanismo de acceso a archivos en UNIX, proveyendo un endpoint (interfaz) para la comunicación entre procesos.

En general, la comunicación entre procesos basada en sockets utiliza una arquitectura cliente-servidor:

- El **servidor** espera por pedidos de clientes escuchando por un puerto específico. Una vez ser recibe un pedido, el servidor acepta la conexión del cliente para completar la conexión. Los servidores que implementan servicios específicos escuchan en un puerto bien conocido— todos los puertos debajo del 1024 son considerados puertos bien conocidos y se utilizan para implementar servicios estándar como SSH, FTP, HTTP, etc.

- Cuando un proceso **cliente** inicia un pedido de conexión, el sistema operativo le asigna un puerto con algún número mayor a 1024. La dirección IP permite que el paquete llegue a la computadora correcta, mientras que el puerto permite que se lo entregue al proceso correcto. Es importante notar que las conexiones deben ser únicas, es decir, si tenemos otro proceso que quiere hacerle un pedido al mismo servidor, se le asignará un puerto mayor a 1024 y que no esté siendo usado por algún otro proceso.

Siempre que tenga sentido, queremos que los sockets se comporten exactamente igual que los archivos en UNIX, de manera tal que puedan ser accedidos con las operaciones tradicionales open-read-write-close. Sin embargo, podemos ver que la interacción entre procesos y los protocolos de red es más compleja que la interacción entre procesos y archivos convencionales. En particular, la interfaz del protocolo debe permitir crear el código de un servidor que espera una conexión de manera pasiva al igual que el código de un cliente que establece conexiones de manera activa. Por este motivo, se necesita una interfaz más compleja para el manejo de sockets, agregando varias system calls y library calls nuevas.

En el caso de sockets en UNIX, se intentó construir un mecanismo general que pueda adaptarse

a muchos protocolos (y no solo a TCP/IP). Esta generalidad hace posible que el sistema operativo incluya software de otros protocolos, permitiendo que una aplicación pueda usar uno o más protocolos al mismo tiempo. Como consecuencia, la aplicación debe especificarle al sistema operativo cómo se deben interpretar los datos, resultando en una interfaz aún más compleja.

3.5.1. Interfaz de Sockets

Creando un Socket

Al igual que con los archivos tradicionales, le podemos pedir al sistema operativo que cree un socket cuando sea necesario. En particular, la system call `socket` crea un endpoint para la comunicación bajo demanda, y retorna un *socket descriptor* que hace referencia a ese endpoint.

```
int socket(int domain, int type, int protocol)
```

- *domain* especifica el dominio de la comunicación, esto es, la familia de protocolos que serán utilizados con el socket. Esto es, especifica cómo deben interpretarse las direcciones cuando son suministradas. Las familias soportadas incluyen TCP/IP internet (PF_INET), Xerox Corporation PUP internet (PF_PUP), Apple Computer Incorporated Apple Talk network (PF_APPLETALK), UNIX file system (PF_UNIX), entre otros.

- *type* especifica la semántica de la comunicación. Posibles tipos incluyen una comunicación secuencial, confiable, bidireccional, orientada a conexión (SOCK_STREAM), una comunicación basada en datagramas (SOCK_DGRAM), etc.

- *protocol* especifica un protocolo particular a ser usado con el socket. Este parámetro sirve para acomodar posibles combinaciones inválidas, al igual que soportar múltiples protocolos con el mismo tipo de servicio dentro de la misma familia.

Para hacer posible utilizar primitivas como `read` y `write` tanto con sockets, el sistema operativo reserva sockets descriptors y file descriptors desde un mismo conjunto de números enteros, y se asegura que no haya un socket con el mismo descriptor que un archivo tradicional.

La razón por la que diferenciamos a los *file descriptors* de los *socket descriptors* es que, en el primer caso, el sistema operativo hace un *binding* entre el file descriptor y un archivo específico al momento de abrir el archivo; sin embargo, los sockets se crean sin tener que hacer un *binding* con una conexión específica. En su lugar, tenemos varias system calls que nos permiten, dinámicamente, asociar una conexión con el socket abierto. Recordemos que una conexión está determinada por una dirección local, una dirección externa y un protocolo. Aclarado esto, dejamos de hacer la distinción.

Especificar una Dirección local

Dijimos que, inicialmente, un socket es creado sin estar asociado a ninguna dirección de local. En muchos casos, las aplicaciones no les interesa cuál es la dirección local que utilizarán y suelen dejar que el protocolo de software elija en su nombre una dirección local adecuada. Sin embargo, en el caso de un servidor, la situación es distinta. Los servidores que operan en un puerto *bien conocido* deben ser capaces de especificar ese puerto al sistema. Para esto, una vez que se haya creado el socket, el servidor utiliza la system call `bind`, estableciendo una dirección local para el socket (en el caso de TCP/IP, puerto bien conocido + dirección IP local). `bind` puede fallar si, por ejemplo, el puerto elegido ya está en uso por otro programa o si se pide una dirección IP inválida.

```
int bind(int sockfd,
        const struct sockaddr *localaddr,
        socklen_t addrlen);
```

- *sockfd* es el file descriptor del socket que queremos hacer el binding.

- *localaddr* es una estructura que especifica la dirección local a la que debe asociarse el socket.

- *addrlen* es un número entero que especifica la longitud de la dirección medida en bytes.

En lugar de dar la dirección como una simple secuencia de bytes, se utiliza la estructura específica para direcciones (ver Fig. 3.15). El valor de *address family* determina el formato del resto de la estructura. Por ejemplo, en el caso de la familia de protocolos TCP/IP, se incluye el número de puerto (bytes 0-1) y la dirección IP (bytes 2-5).

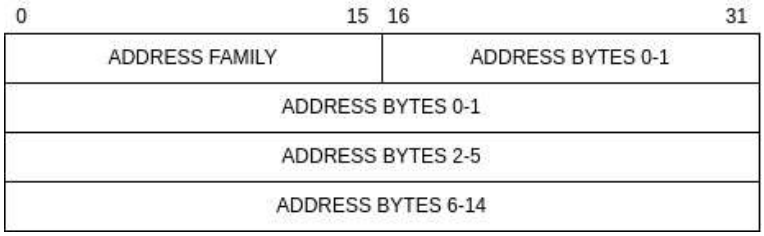

Figura 3.15: Estructura *sockaddr*.

Conectar Socket a un destino

Ya vimos cómo crear un socket y cómo asignarle una dirección local. Nos falta ver cómo hacemos para asignar una dirección externa. En este punto, decimos que el socket se encuentra *desconectado*. La manera de establecer una conexión con un destino permanente es por medio de la system call `connect`, pasando al estado *conectado*. Antes de poder transferir datos (para un servicio de comunicación confiable), las aplicaciones deben llamar a *connect* y, de esta manera, establecer una conexión con el servidor. Si queremos utilizar un servicio sin conexión de comunicación basado en datagramas, no es necesario llamar a *connect*; aún así, resulta conveniente llamar a *connect* porque, de esta manera, tan solo tenemos que especificar la dirección de destino una sola vez.

```
int connect(int sockfd,
        const struct sockaddr *extaddr,
        socklen_t addrlen);
```

- *sockfd* es el file descriptor del socket con el que nos queremos conectar.
- *extaddr* es la dirección a la que nos queremos conectar.
- *addrlen* es la longitud de la dirección.

Los sockets nos permiten comunicar dos procesos en computadoras distintas. Un socket se identifica con una dirección IP concatenada con un número de puerto (y un protocolo de transporte). Las direcciones IP nos permiten identificar las computadoras involucradas en la comunicación (cada computadora tiene su propia dirección IP). Los puertos nos permiten identificar el proceso que participa en una comunicación dentro de una computadora. Más específicamente, un puerto puede ser visto como un objeto donde los procesos pueden dejar y retirar mensajes, y es responsabilidad del sistema operativo administrar estos puertos para permitir la comunicación entre procesos. Los sockets también se pueden usar de forma local (utilizando la dirección IP localhost 127.0.0.1). Esto permite utilizar la misma interfaz para hablar tanto con una máquina externa como para trabajar internamente.

Enviando Datos

Una vez que la aplicación haya establecido la conexión, puede usar el socket para transmitir datos. En total, existen cinco posibles funciones que nos permiten enviar datos por un socket: *write*, *writev*, *send*, *sendto*, *sendmsg*. Típicamente, usamos *write* para enviar datos o, equivalentemente, *send* con *flags* = 0. Los *flags* en la system call *send* nos permiten hacer cosas como: enviar mensajes urgentes más rápidamente (*out-of-band*), controlar el uso de tablas de ruteo, etc.

```
ssize_t send(int sockfd, const void *buf,
            size_t len, int flags);
```

Recibiendo Datos

También tenemos las cinco system calls correspondientes para recibir datos a través de un socket: *read, redv, recv, recvfrom, recvmsg*. Nuevamente, suele ser más conveniente usar *read* o, equivalentemente, *recv* con *flags* = 0. Los *flags* en *recv* nos permiten, por ejemplo, hacer un look-ahead a algún mensaje siguiente, sin tener que remover los mensajes intermedios del socket.

```
ssize_t recv(int sockfd, void *buf,
             size_t len, int flags);
```

Especificando Longitud de Cola

Consideremos el siguiente escenario. Tenemos un servidor que crea un socket, hacer un binding con un puerto bien conocido y espera la entrada de algún cliente. Si el servidor utiliza un servicio de comunicación orientado a conexión, o si computar una respuesta para el cliente lleva un tiempo considerable, es posible que otro cliente intente establecer una nueva conexión con el servidor antes de que éste pueda responder al primer pedido. Para evitar rechazar o descartar estos pedidos entrantes, el servidor debe indicarle al protocolo de red que desea encolar estos pedidos hasta que tenga tiempo para procesarlos. La system call `listen` nos permite que un servidor pueda preparar un socket para conexiones entrantes.

```
int listen(int sockfd, int qlength);
```

En términos de protocolo de red, la system call `listen` pone al socket en un modo pasivo, listo para aceptar conexiones entrantes y, en caso de llegar múltiples pedidos, el protocolo de red encole hasta *qlength* pedidos entrantes— superado este límite, los empieza a descartar (dependiendo del protocolo de red, el cliente puede recibir un error, se puede intentar retransmitir o puede ser ignorado). Notemos que *listen* solo aplica para sockets que hayan seleccionado el servicio orientado a conexión.

Aceptar Conexión

Hasta ahora, vimos cómo un servidor puede utilizar las system call *socket*, *bind* y *listen* para crear, hacer un binding y especificar la longitud de la cola de un socket y pasar a modo pasivo, respectivamente. Nos falta ver cómo hace el servidor para terminar de establecer la conexión con un cliente particular, en cuanto se reciba un pedido de conexión por parte del mismo. Para ello, se utiliza la system call `accept`.

```
int accept(int sockfd,
        struct sockaddr *extaddr,
        socklen_t *addrlen);
```

La llamada a *accept* bloquea al servidor hasta que algún pedido de conexión sea recibido. Luego, se extrae el primer pedido de conexión de la cola de conexiones pendientes. El sistema completa los datos recibidos del cliente en las variables apuntadas por *extaddr* y *addrlen*. Finalmente, el sistema crea un nuevo socket que está conectado al cliente, y retorna el file descriptor del nuevo socket. El socket recién creado no está en modo *listening*. El socket original no se ve afectado por esta llamada, por lo que el servidor lo puede reutilizar para seguir aceptando pedidos.

Un servidor puede manejar pedidos de manera iterativa o concurrente. Bajo una estrategia **iterativa**, hay un único proceso servidor que se encarga de atender al cliente. Una vez finalizada la comunicación, el servidor cierra el socket creado. Luego, llama a *accept* para atender el siguiente pedido, repitiendo el ciclo. Bajo una estrategia **concurrente**, tenemos múltiples procesos que se encargan de atender los pedidos de clientes. La idea es la siguiente. Tenemos un proceso maestro, que se encarga de llamar a *accept*. Cada vez que retorne de la llamada, el servidor maestro crea un proceso hijo para que se encargue de atender la conexión. Inmediatamente después de hacer el *fork*, el servidor maestro cierra su copia del socket creado, para luego llamar a *accept* y repetir el ciclo. Mientras tanto, el proceso hijo heredó una copia del socket creado, por lo que puede atender la conexión. Cuando termina la comunicación, cierra el socket y termina.

El diseño concurrente para servidores puede parecer confuso, porque tenemos múltiples procesos usando el mismo puerto local. La clave está en entender que, en el caso de TCP/IP, la conexión es identificada por el par de endpoints (y no solo por el puerto + IP del servidor). Luego, no importa

cuántos procesos utilizan el número de puerto local, siempre que tengan direcciones de destino distintas. En el caso del servidor concurrente, tan solo hay un proceso por cliente y un proceso adicional para aceptar las conexiones en modo *listening*. Como cada proceso tiene una dirección externa distinta, cuando llega un paquete TCP, será enviado al socket que pertenece a esa conexión. Si no existe tal socket, el paquete será enviado al socket en estado *listening*, que tan solo responderá a paquetes TCP que pidan establecer una conexión nueva.

3.5.2. Implementación de primitivas

Existen diferentes opciones de diseño para implementar cada primitiva, que determinan si la comunicación entre procesos es sincrónica o asincrónica:

- **Sincrónica**. La idea es que nos quedamos esperando a que la system call sea completada extremo a extremo.
  - Un `send` sincrónico no devuelve el control hasta que el mensaje no fue leído por el proceso del otro lado. Es decir, no basta con que el mensaje llegue a la máquina destino, sino que es necesario que el proceso correspondiente lea el mensaje (el proceso podría estar bloqueado, estar ready, etc.) y, mientras tanto, el emisor se queda bloqueado.
  - Un `receive` sincrónico bloquea al proceso hasta recibir el mensaje.
- **Asincrónica**. La idea es que no vamos a esperar a que la system call sea completada extremo a extremo.
  - Cuando hacemos `send` asincrónico, el proceso emisor entrega un mensaje al sistema operativo y continúa su operación, para que (eventualmente) sea entregado al destinatario. Para saber si el mensaje llegó a su destino, el sistema operativo puede avisarle mediante una interrupción o usando algún mecanismo de E/S no bloqueante.
  - Cuando hacemos `receive` asincrónico, siempre retiramos un mensaje del puerto; si no hay ningún mensaje, se devuelve un null.

  Podemos ver que la comunicación asincrónica es más flexible, pero requiere de una mayor lógica de programación.

Capítulo 4

Scheduling de Procesos

Cuando una computadora es multiprogramada, casi todos los recursos deben ser reservados para su uso, siendo la CPU uno de los recursos principales. Frecuentemente, se tienen múltiples procesos o threads compitiendo por la CPU al mismo tiempo. Esta situación se da cuando dos o más procesos están simultáneamente en estado *ready*. En ese caso, se debe elegir cuál de los dos o más procesos será el siguiente en correr. La parte del sistema operativo que se encarga de realizar esta elección es el **scheduler**. El scheduling es central al diseño de un sistema operativo y tiene un impacto fenomenal sobre el rendimiento del sistema.

4.1. Conceptos básicos

Recordemos que en un sistema con un único núcleo de procesamiento solo puede haber un proceso corriendo en cada momento. Los demás procesos deberán esperar hasta que el procesador esté libre, y pueda ser asignado nuevamente a otro proceso. El objetivo de la multiprogramación es siempre tener algún proceso corriendo, y así maximizar la utilización de la CPU.

Un proceso es ejecutado hasta que deba esperar, típicamente, por alguna E/S. En un sistema simple, la CPU simplemente se quedaría sin hacer nada (idle), desperdiciando todo este tiempo de espera. Con la multiprogramación, se busca usar este tiempo de espera de forma productiva. Para ello, se mantienen varios procesos en memoria al mismo tiempo, de manera tal que cuando un proceso tenga que esperar por algún evento, el sistema operativo puede desalojar al proceso de la CPU y colocar otro proceso que estaba en memoria. De esta manera, conseguimos aumentar la utilización de la CPU.

Cada vez que la CPU no está siendo utilizada, se debe seleccionar uno de los procesos en estado *ready* para colocarlo en la CPU. El algoritmo de selección es realizado por el **scheduler**, que selecciona un proceso de todos los procesos en memoria que están listos para ser ejecutados, y reserva a la CPU para ese proceso.

4.1.1. Comportamiento de un Proceso

Casi todos los procesos alternan entre ráfagas de cómputo con ráfagas de E/S. Normalmente, la CPU corre por un tiempo sin detenerse, para luego realizar E/S (bloqueante)— por ejemplo, para leer o escribir un archivo. Cuando se completa la system call, el proceso vuelve a correr hasta que necesite hacer otra E/S. En este contexto, cuando hablamos de hacer E/S, nos estamos refiriendo a cualquier evento que fuerce a un proceso a pasa al estado *waiting*.

Lo importante es notar que algunos procesos van a pasar la mayor parte de su tiempo computando, mientras que otros procesos van a pasar la mayor parte de su tiempo esperando por E/S. Los primeros son conocidos como **procesos de CPU** (o CPU-bound) y los últimos como **procesos de E/S** (o I/O-bound). Los procesos de CPU, típicamente, tienen largas ráfagas de CPU e infrecuentes esperas por E/S, mientras que los procesos de E/S cortas ráfagas de CPU y, en consecuencia, frecuentes esperas por E/S.

Notemos que un proceso es de E/S si no realiza mucho cómputo entre cada espera por E/S, y esto **no depende del tiempo de espera por la E/S en sí**. Esta clasificación se basa en el tiempo de cómputo entre cada E/S, y no por el tiempo de espera en cada E/S particular.

La idea que nos debemos llevar es que si un proceso de E/S quiere correr, deberíamos darle una oportunidad rápido para que pueda hacer su pedido a disco y que el disco se mantenga ocupado.

4.1.2. ¿Cuándo interviene el scheduler?

Estuvimos diciendo que los schedulers deben tomar decisiones sobre cuál es el siguiente proceso que le toca correr. La pregunta que nos surge es ¿cuándo se deben tomar estas decisiones? Resulta que hay una variedad de situaciones en donde se podrían llegar a tomar estas decisiones.

1. Cuando un proceso es aceptado (pasa de *new* a *ready*), se debe tomar la decisión de si debe continuar el proceso padre o si debe continuar el proceso hijo.

2. Cuando un proceso pasa de estado *running* al estado *waiting*— por ejemplo, como resultado de una llamada a E/S, una invocación de `wait` sobre un proceso, un semáforo o algún otro evento.

3. Cuando un proceso llama a `exit`. Ese proceso no puede continuar corriendo, por lo que algún otro proceso debe ser seleccionado para ejecutar.

4. Cuando un proceso pasa del estado *running* a *ready*, por ejemplo, cuando ocurre una interrupción. Si el clock del hardware provee interrupciones periódicas (por ejemplo, 60 veces por segundo), se puede tomar una decisión de scheduling en cada interrupción de reloj. También es posible que haya ingresado un proceso de mayor prioridad que haya forzado el desalojo de este proceso.

5. Cuando un proceso pasa del estado *waiting* al estado *ready*, por ejemplo, luego de completar una E/S. Cuando se completa una E/S, todos los procesos que estaban esperando por ese resultado pasan al estado *ready*. Es decisión del scheduler si poner a ejecutar algunos de estos procesos, continuar la ejecución del proceso actual o poner a ejecutar otro proceso.

Para las situaciones 2 y 3, no hay opciones en términos de scheduling: un nuevo proceso debe ser seleccionado para su ejecución. Sin embargo, para las situaciones 1, 4 y 5, podemos dejar que el proceso siga corriendo en la CPU o podemos desalojarlo. Esta decisión va a depender de si el scheduler es preemptive o nonpreemptive.

Dependiendo de la manera en que manejan las interrupciones de reloj, los algoritmos de scheduling se dividen en dos categorías. Un algoritmo de scheduling **nonpreemptive** (o cooperativo) selecciona un proceso para ejecutar y le permite correr hasta que se bloquea o hasta que libere la CPU de forma voluntaria (por ejemplo, reduciendo su prioridad por medio de la system call `sched_yield`). Luego de una interrupción de reloj, el proceso que estaba corriendo antes de la interrupción es reanudado— salvo que esté disponible algún proceso de mayor prioridad.

Cuando el scheduling toma lugar solo bajo las situaciones (1,) 2 y 3, decimos que el esquema de scheduling es **nonpreemptive**. Bajo un scheduling nonpreemptive, una vez que la CPU ha sido reservada para un proceso, el proceso retiene la CPU hasta que termina o hasta que pasa al estado *waiting* (por ejemplo, cuando hace E/S). Es decir, los procesos voluntariamente entregan el control al scheduler de nuevo, mediante alguna system call apropiada.

En cambio, un algoritmo de scheduling **preemptive** (o con desalojo) selecciona un proceso y lo deja correr por un máximo de tiempo preestablecido— conocido como **quantum**. Si todavía sigue corriendo al final de este intervalo de tiempo, es suspendido (pasa a estado *ready*) y el scheduler selecciona otro proceso para correr.

Los schedulers con desalojo se cuelgan de la interrupción del reloj para decidir si el proceso actual debe seguir ejecutándose o si le toca a otro. Esto hace que un proceso no tenga garantías sobre la continuidad de su ejecución: puede ser desalojado en cualquier momento. Esto podría ser un problema en un sistema operativo *real-time*, donde no podemos posponer indefinidamente la ejecución de los procesos. En cambio, en un ambiente con usuarios interactivos, poder desalojar un proceso es esencial para evitar

que un proceso retenga indefinidamente la CPU, denegando el servicio a los demás procesos.

Por último, en los sistemas batch, donde no tenemos usuarios impacientes esperando en sus terminales por una respuesta rápida, utilizar algoritmos de scheduling nonpreemptive (o preemptive con largos períodos para cada proceso) suele ser aceptable. Esta estrategia evita algunos *switches* entre procesos (en principio) innecesarios, reduciendo la cantidad de ciclos de CPU desperdiciados a causa de los cambios de contexto.

4.2. Objetivos del Scheduling

Diferentes algoritmos de scheduling tienen distintas propiedades, y elegir un algoritmo particular podría favorecer una clase de procesos sobre otros. Al elegir qué algoritmo utilizar en una situación particular, debemos considerar las propiedades de los algoritmos. Muchos de estos objetivos son antagónicos, por lo que no se pueden satisfacer a la vez. En definitiva, cada política de scheduling va a buscar maximizar una función objetivo, tratando de impactar lo menos posible en el resto. Algunos objetivos incluyen los siguientes:

- **Ecuanimidad** (o *fairness*). Cada proceso reciba una cantidad justa de CPU. No hay una única definición de esa ecuanimidad.
- **Eficiencia** (o Utilización de CPU). Queremos que la CPU esté ocupada la mayor cantidad de tiempo posible.
- **Carga del sistema**. Queremos minimizar la cantidad de procesos que están esperando por la CPU. Podríamos pensar que todo proceso, en algún momento, va a hacer un pedido de E/S y se va a bloquear hasta que ese pedido esté. La idea es que le damos a cada proceso su cuota de CPU para que la mayor parte de procesos posibles se queden bloqueados esperando E/S (pasando al estado *waiting*).
- **Tiempo de respuesta** (o Response time según [TB18]). Buscamos minimizar el tiempo de respuesta percibido por los usuarios interactivos. Hay una diferencia enorme entre los procesos que dialogan con un usuario (procesos interactivos) y los que no (procesos batch), porque hay una sensación de tiempo de respuesta. Ante esta percepción queremos favorecer a esos procesos para que se perciba un sistema rápido.
- **Latencia** (o Response time según [SGG18]). Lo que queremos es minimizar el tiempo requerido para que un proceso empiece a dar resultados. Queremos que se genere algo que aporte valor más rápido posible.
- **Tiempo de ejecución** (o Turnaround time). Queremos que los procesos les tome ejecutarse punta a punta el menor tiempo posible.
- **Rendimiento** (o throughput). Queremos maximizar el número de procesos completados por unidad de tiempo.
- **Waiting time**. Los algoritmos de scheduling no afectan la cantidad de tiempo que necesita un proceso para ejecutar o hacer E/S. Solo afecta a la cantidad de tiempo que un proceso debe esperar en la *ready queue*. El *waiting time* es la suma de los períodos de espera en la *ready queue*.
- **Liberación de recursos**. Hacer que terminen cuanto antes los procesos que tiene reservados más recursos. La idea es que si un recurso está reservado por un proceso, otros procesos que lo necesiten se quedarían bloqueados esperando por ese recurso. Entonces, queremos que estos recursos sean liberados lo antes posible para que los otros procesos puedan continuar con su ejecución.
- **Proporcionalidad**. Cumplir con las expectativas de un usuario interactivo. Normalmente, tenemos una idea de cuánto tiempo deberían tardar las cosas en realizarse. Si hacemos un pedido complejo que debería tardar mucho tiempo en resolverse, estamos dispuestos a esperar mucho tiempo en obtener un resultado. Por otro lado, si un pedido es simple, esperamos que se resuelva rápidamente.
- **Predictibilidad**. En los sistemas real-time, perder ocasionalmente un deadline no es fatal. Sin

embargo, que un proceso corra de forma errática es un problema. Si se trata de un proceso de audio, la calidad de solido puede deteriorarse rápidamente. Para evitar este problema, el scheduling de procesos debe ser altamente predecible y regular.

Objetivos Generales

En cualquier sistema, la propiedad de **fairness** es importante. Esta propiedad nos dice que procesos comparables deben tener un servicio comparable. Darle a un proceso mucho más tiempo de CPU que un proceso equivalente no es justo. En caso de tener distintas categorías de procesos, procesos en distintas categorías pueden ser tratados distinto y se sigue considerando justo.

Otro objetivo general es el de mantener a todas las partes del sistema ocupadas cuando sea posible, objetivo conocido como **balance**. Si la CPU y todos los dispositivos de E/S pueden mantenerse ocupados todo el tiempo, significa que tenemos un balance perfecto entre procesos de CPU y procesos de E/S.

Sistemas Batch

Cuando se administran grandes centros de cómputo que corren muchos procesos batch, típicamente se miran a estas tres métricas para saber qué tan bien está funcionando el sistema: throughput, turnaround time y utilización de CPU.

- El **throughput** es el número de trabajos por hora que el sistema completa. Claramente, completar 50 trabajos es mejor que completar 40 trabajos.
- El **turnaround time** es la media estadística del tiempo desde que una proceso es creada hasta que es completada. Nos dice cuánto tiempo, en el caso promedio, se debe esperar para obtener un resultado.

  Notemos que el throughput y el turnaround time pueden llegar a ser conflictivos. Por ejemplo, si en un sistema tenemos muchas procesos que necesitan poco tiempo para completarse y otras que necesitan mucho tiempo para completarse, un scheduler que maximice throughput elegiría siempre las procesos cortas. En consecuencia, el turnaround time sería infinito por culpa de inanición en las procesos largas.
- La **utilización de CPU** suele ser usada como métrica en sistemas batch. A pesar de esto, en realidad no es una buena medida. Por otro lado, sí nos puede servir como indicador de que necesitamos agregar más poder de cómputo. Una métrica que también nos sirve como indicador de que necesitamos agregar más poder de cómputo es la **carga del sistema** (cantidad de procesos activos en estado *ready*).

Sistemas Interactivos

Para sistemas interactivos, normalmente nos van a interesar otro tipo de objetivos. El más importante es minimizar el **tiempo de respuesta**, esto es, el intervalo de tiempo que hay entre que enviamos un comando por la terminal hasta que obtenemos el resultado. En una computadora personal, donde hay varios procesos daemons corriendo en el fondo, intentar abrir un programa o un archivo debería tomar precedencia sobre un proceso de fondo. Atender a todos los pedidos interactivos primero es percibido como un buen servicio.

Sistemas Real-Time

Los sistemas de tiempo real tienen propiedades distintas a los sistemas interactivos. Se caracterizan por tener deadlines que deben (o al menos deberían) cumplirse. Por lo tanto, la necesidad principal en un sistema real-time es poder cumplir con todos estos deadlines.

4.3. Algoritmos de Scheduling

Es momento de pasar de los problemas generales que afectan al scheduling a los algoritmos específicos que se utilizan. En esta sección, vamos a ver algunos de los algoritmos de scheduling más conocidos.

4.3.1. FCFS Scheduling

El algoritmo más simple es el first-come, first-served (**FCFS**). Con este algoritmo, los procesos son asignados a la CPU en el orden en que van llegando a la *ready queue*. Básicamente, tenemos una única cola de procesos esperando por la CPU. Cuando el primer proceso entra al sistema, comienza inmediatamente a ejecutar y se le permite correr tanto tiempo como desee. A medida que van entrando nuevos procesos, se los va colocando al final de la cola. Cuando el proceso que está corriendo se bloquea, le toca su turno al primer proceso en la cola. Cuando un proceso bloqueado pasa al estado *ready*, se lo coloca al final de la cola.

Para este algoritmo, una lista enlazada mantiene un registro de todos los procesos en estado *ready* (en realidad, es una lista de las PCBs de los procesos). Elegir un proceso para correr solo requiere remover el primero de la cola. Notemos que este algoritmo es *nonpreemptive*: una vez que la CPU ha sido reservada para un proceso, ese proceso retiene la CPU hasta que termine de ejecutar o hasta que haga E/S. Los procesos no son desalojados solo por estar mucho tiempo ejecutando.

El problema de este algoritmo es que supone que todos los procesos son iguales. Si llega un mega-proceso que requiere de mucha CPU, todos los procesos posteriores tendrán que esperar a que termine este proceso grande, empeorando los tiempos de espera promedio. Esto empeora aún más en el caso de que tengamos procesos que hacen mucha E/S y un solo proceso que hace un uso intensivo de la CPU.

- Cada vez que le toca ejecutar al proceso de CPU, se va a quedar ejecutando por mucho tiempo en la CPU. Durante este tiempo, todos los otros procesos terminarán su E/S y se moverán a la *ready queue*, esperando su turno por la CPU. Mientras que estos procesos esperan, los dispositivos de E/S no tienen ningún trabajo que hacer.

- Eventualmente, el proceso de CPU termina su uso de la CPU y se mueve a un dispositivo de E/S. Todos los procesos de E/S, que tienen ráfagas de CPU cortas, ejecutan rápidamente y se van a las colas de E/S. En este punto, la CPU no tiene nada que hacer hasta que se termine de realizar la E/S del proceso de CPU.

Este problema se lo conoce como **efecto convoy**[1], ya que todos los otros procesos se amontonan esperando a que el proceso grande deje de usar la CPU, resultando en una menor utilización de la CPU.

4.3.2. Round-Robin Scheduling

Uno de los algoritmos más viejos, simples y justos es el algoritmo Round Robin (**RR**). A cada proceso se le asigna un intervalo de tiempo (típicamente de 10-100 milisegundos), conocido como **quantum**, durante el cual se le permite al proceso correr. Si el proceso todavía está corriendo al final del quantum, la CPU es desalojada y asignada a otro proceso. Claramente, si el proceso se ha bloqueado o ha finalizado antes de que termine su quantum asignado, la CPU es asignada a otro proceso en ese momento. Notemos que es un algoritmo *preemptive*.

Round robin es fácil de implementar. Todo lo que tiene que hacer el scheduler es tratar a la *ready queue* como una cola circular. Cuando el proceso termina de usar su quantum, se lo coloca al final de la lista. Es un algoritmo ecuánime, que consigue resolver la inanición de procesos, pero va en contra de todos los demás objetivos.

Cuando usamos este mecanismo, aparece una pregunta: ¿cuánto tiempo debería durar el quantum? Cambiar de un proceso a otro requiere de cierto tiempo para hacer toda la administración asociada a los cambios de contexto— guardar y cargar registros, mapeos de memoria, actualizar tablas, flushear la memoria cache, etc. Si es muy largo, la política RR sería equivalente a una FCFS y los procesos interactivos podrían estar esperando demasiado tiempo. Si es muy corto, aplicar esta estrategia resultaría en un gran número de context switches.

4.3.3. SJF Scheduling

Otra política de scheduling es la llamada Shortest Job First (**SJF**). Este algoritmo asume que se conocen de antemano los tiempos de ejecución de cada proceso— factible en un sistema batch. Cuando

[1] Escolta de un conjunto de vehículos.

la CPU esté disponible, se le asigna aquel proceso con el proceso de menor duración (en caso de empate, se utiliza FCFS). De esta manera, se producen resultados más rápido y se minimiza el tiempo de espera promedio (siempre que todos los procesos estén disponibles en todo momento). Notemos que es un algoritmo *nonpreemptive*.

SRTN

Una versión *preemptive* de este algoritmo es conocida como Shortest Remaining Time Next (**SRTN**). Con este algoritmo, el scheduler siempre elige al proceso cuyo tiempo restante de ejecución es el de menor duración. Aquí, nuevamente, el tiempo de ejecución debe ser conocido. Cuando un proceso llega, su tiempo total es comparado con el tiempo restante del proceso actual. Si el nuevo proceso tiene una menor duración, el proceso actual es suspendido y el nuevo proceso es asignado a la CPU. Esto permite que procesos de poca duración tengan un buen servicio.

SPN

En un sistema interactivo, no podemos conocer de antemano los tiempos de ejecución de cada proceso. Aún así, es posible adaptar este algoritmo para que funcione en un sistema interactivo utilizando heurísticas que nos den un resultado aproximado. Generalmente, los procesos interactivos siguiente el siguiente patrón: se espera por un comando, se ejecuta el comando, se espera por un comando, etc. En lugar de considerar el tiempo de ejecución de un proceso completo, podemos considerar el tiempo de ejecución de cada ráfaga de CPU. De esta manera, seleccionamos aquel proceso con la ráfaga de CPU de menor duración. Esta variante es conocida como Shortest Process Next (**SPN**).

Típicamente, podemos estimar la siguiente ráfaga de CPU con el promedio exponencial de las duración de ráfagas anteriores. Podemos definir el promedio exponencial con la siguiente fórmula. Sea t_i la duración de la i-ésima ráfaga de CPU, y sea τ_{i+1} nuestro valor predicho para la siguiente ráfaga. Entonces, para $0 \leq \alpha \leq 1$ definimos:

$$\tau_{i+1} = \alpha t_i + (1 - \alpha)\tau_i.$$

El valor de t_i contiene la información más reciente, mientras que τ_i almacena la historia pasada. El parámetro α nos permite controlar el peso relativo entre el dato reciente y la historia pasada sobre nuestra predicción. El valor inicial de τ_0 puede ser una constante o un promedio de todo el sistema.

4.3.4. Priority Scheduling

Hasta ahora, estuvimos haciendo una asunción implícita sobre la importancia de los procesos: todos los procesos son igual de importantes. Sin embargo, esto no tiene por qué ser así, por ejemplo, podríamos querer darle prioridad a los procesos de interactivos sobre los procesos de fondo. La necesidad de considerar factores externos como parte del algoritmo de scheduling nos lleva al **scheduling basado en prioridades** (priority scheduling). La idea es sencilla: a cada proceso le asignamos una prioridad, y el scheduler selecciona al proceso con la mayor prioridad (en caso de empate, se puede usar FCFS o Round-Robin).

Las prioridades pueden ser definidas de forma interna o externa.

- Las prioridades **internas** se definen usando alguna medida para computar la prioridad de un proceso. Por ejemplo, límites de tiempo, memoria requerida, el número de archivos abiertos o la tasa entre la longitud promedio de las ráfagas de E/S y las ráfagas de CPU.
- Las prioridades **externas** son establecidas por un criterio por fuera del sistema operativo, como la importancia del proceso, el tipo de proceso (batch, interactivo, real-time), el monto pagado por usar la computadora, etc.

Los algoritmos de scheduling basados en prioridades pueden ser preemptive o nonpreemptive. Cuando un proceso llega a la *ready queue*, su prioridad se compara con la prioridad del proceso que actualmente está corriendo. Un algoritmo preemptive va a poner a correr el nuevo proceso en la CPU si su prioridad es mayor que la del proceso que actualmente está corriendo. En cambio, si el algoritmo es nonpreemptive, simplemente pondrá al proceso en la cabeza de la *ready queue*.

Un problema importante con los algoritmos de scheduling basados en prioridades es que un proceso podría quedarse esperando indefinidamente en la cola de procesos listos. En un sistema con mucha carga, es posible que continuamente sigan llegando procesos con mayor prioridad, postergando por indefinidamente la ejecución de procesos con baja prioridad. Este escenario se conoce como **inanición** (starvation). Es importante notar que **todos los esquema con prioridades fijas conducen indefectiblemente a escenarios de inanición**.

Para resolver este problema, podemos utilizar esquemas de **prioridades variables**[2]. La idea es que los procesos vayan ganando prioridad de forma incremental a medida que esperan en la *ready queue*. Eventualmente, incluso un proceso que inicialmente tenía la menor prioridad posible llegaría a tener la mayor prioridad posible en el sistema y será ejecutado. Esta solución se la conoce como **envejecimiento** (aging).

Multiple Queues Scheduling

Uno de los scheduler basados en prioridades más tempranos estuvo en el sistema operativo CTSS, que corría en la IBM 7094. Los cambios de procesos eran demasiado lentos porque la 7094 solo podía tener un proceso en memoria. Cada cambio significaba bajar el proceso actual a disco y traerse uno nuevo del disco. Se necesitaba una manera de reducir el impacto del alto costo de un cambio de proceso.

Podemos pensar en otorgarle largos intervalos de tiempo a aquellos procesos de CPU una vez cada tanto, en lugar de muchos intervalos de tiempo cortos, y así reducir la cantidad de cambios de procesos. Sin embargo, ya vimos que darle a todos los procesos un gran quantum significa una pérdida en los tiempos de respuesta. Su solución consistía en agregar clases de prioridades. Procesos con la prioridad más alta corren 1 quantum. Procesos con la siguiente prioridad corren por 2 quanta.[3] Procesos con la siguiente prioridad corren por 4 quantums, etc. Cuando un proceso no le alcanza su cuota de CPU, es pasado a la siguiente cola, disminuyendo su prioridad a cambio de un mayor tiempo de CPU en su próximo turno (ver Fig. 4.1).

Como ejemplo, consideremos un proceso que necesita ejecutar por un total de 100 quantums para completar su tarea. Inicialmente, se le daría 1 quantum y, pasado este tiempo, se lo desalojaría. La próxima vez se le asignarían 2 quantums antes de ser desalojado. En sus siguientes turnos se le asignarían 4, 8, 16, 32 y 64 quantums. Tan solo se necesitarían 7 swaps en lugar de los 100 que requeriría un algoritmo RR. Además, a medida que el proceso va bajando en la escala de prioridades, éste será asignado a la CPU con cada vez menos frecuencia, permitiendo que procesos cortos, interactivos puedan ejecutar rápidamente.

Tener múltiple colas también nos permite que cada utilizar un algoritmo particular en cada cola, dándonos una mayor flexibilidad. Este esquema deja a los procesos de E/S y a los procesos interactivos (que típicamente se caracterizan por tener ráfagas de CPU cortas) en la cola de mayor prioridad. Para prevenir la inanición, un proceso que espere por demasiado tiempo en una cola de baja prioridad podría gradualmente ser movido a una cola de mayor prioridad. Además, cuando llega un proceso nuevo, podemos colocarlo en alguna de las colas directamente, dependiendo del tipo de proceso. Por ejemplo, podemos colocar a los procesos interactivos en la cola de máxima prioridad, mientras que colocamos a los procesos batch en una cola con baja prioridad. Otra heurística consiste en que cuando un proceso termina de hacer E/S, lo colocamos en la cola de máxima prioridad, porque se supone que va a volver a hacerse interactivo.

4.4. Multiprocessor Scheduling

Hasta el momento, estuvimos discutiendo los problemas generales de scheduling que aparecen en un sistema monoprocesador. En un sistema monoprocesador, el scheduling es unidimensional. La única pregunta que se debe responder es *¿Cuál es el siguiente proceso a ejecutar?*. Cuando tenemos múltiples CPUs, el scheduling se vuelve bidimensional. Ahora, no solo tenemos que decidir cuál es el próximo proceso a ejecutar, sino que también debemos decidir en qué CPU se pondrá a ejecutar. Esta dimensión

[2] En [TB18] también se hace una distinción entre prioridades estáticas y prioridades dinámicas. Se entiende por prioridades dinámicas a aquellas que se calculan en tiempo de ejecución . Por ejemplo, el algoritmo de SPN sería de prioridades dinámicas, mientras que SJF sería de prioridades estáticas.

[3] Latinismo para "quantums".

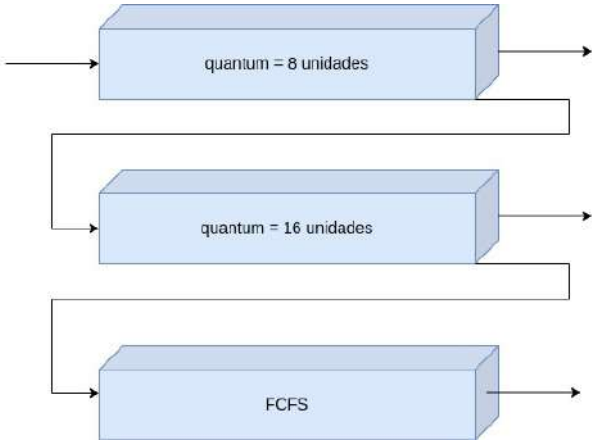

Figura 4.1: Colas Multinivel.

adicional aumenta considerablemente la complejidad de los algoritmos de scheduling. Empezamos a tener consideraciones adicionales como el *balanceo de carga* y la *afinidad del procesador*.

Una única estructura compartida

La forma más simple de scheduling en un sistema multiprocesador consiste en tener una única estructura de datos común a todos los procesadores del sistema, donde se mantendrán todos los procesos en estado *ready* que podrían ser seleccionados para ejecutar cualquier CPU. Normalmente, esta estructura va a contener múltiples colas para procesos de distintas clases de prioridades (ver Fig. 4.2(a)). En el

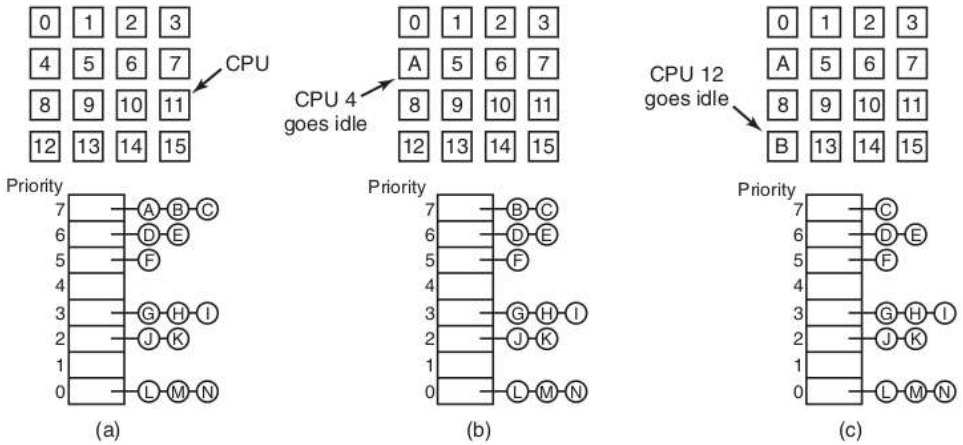

Figura 4.2: Usando una única estructura de datos para scheduling en multiprocesador.

ejemplo tenemos 16 CPUs inicialmente ocupadas, y un conjunto de 14 procesos que están esperando para ejecutar. La primera CPU en ser liberada es la número 4, que selecciona al proceso de mayor prioridad. Como se puede ver en la Fig. 4.2(b), el proceso de mayor prioridad es el A. Luego, la CPU número 12 se libera y elige al proceso B, como se muestra en la Fig. 4.2(c).

Tener una sola estructura de datos común a todas las CPUs permite utilizar la técnica de *time-sharing* fácilmente, y obtenemos un resultado comparable al que teníamos en un sistema monoprocesador. Además, nos provee un mecanismo de balanceo de carga automático, dado que no es posible tener una CPU desocupada y varias CPUs sobrecargadas al mismo tiempo.

Una desventaja que tiene esta estrategia es la contención sobre la estructura de datos. Todos los procesadores necesitan acceder continuamente a esta estructura. Como cada procesador necesita tomar posesión del lock que protege a la estructura compartida, a medida que aumentamos la cantidad de CPUs, aumenta la contención sobre la estructura de procesos *ready*. Esto, probablemente, resulte en en

un cuello de botella.

Otro factor que debemos tomar en cuenta al momento de hacer el scheduling es que no siempre va a ser equivalente ejecutar un proceso en una CPU que en otra. Consideremos la siguiente situación. Cuando tenemos un proceso A que ha estado corriendo por un rato largo sobre una CPU K, la cache de la CPU K estará poblada con un montón de bloques de memoria asociados al proceso A. Si el proceso A, luego de ser desalojado, rápidamente vuelve a estar en el estado *ready*, vamos a obtener una mejor performance si le asignamos la CPU K sobre cualquier otra CPU. Esto se debe a que la cache de K posiblemente continúe manteniendo algunos bloques de A. Tener bloques de cache pre-cargados incrementa la tasa de cache hits, mejorando la performance. Lo mismo vale para la TLB. Esto se conoce como **afinidad del procesador**.

Una estructura privada por cada CPU

Algunos sistemas multiprocesador toman en cuenta este efecto y lo utilizan para lograr una mejor performance. Intentando llevar esta idea a un extremo, podemos pensar en un algoritmo de scheduling donde cada procesador tenga su propia estructura privada de procesos en estado *ready*. La idea básica es hacer un esfuerzo considerable para que un mismo proceso corra siempre en la misma CPU.

Una manera de crear esta afinidad es utilizar un **algoritmo de scheduling de dos niveles**. Cuando se crea un proceso, se le asigna una CPU— por ejemplo, aquella con menor carga en ese momento. Esta asignación de procesos a CPUs es el primer nivel del algoritmo. Como resultado, cada CPU obtiene una colección privada de procesos.

El scheduling de procesos en sí se realiza en un segundo nivel del algoritmo. Se hace en cada CPU de manera independiente, utilizando alguno de los algoritmos de scheduling ya conocidos. Al mantener cada proceso en una misma CPU, se maximiza la afinidad con el procesador. Esto no solo mejora la afinidad con la cache, sino que también reduce al mínimo la contención sobre las estructuras de datos.

Sin embargo, sigue habiendo problemas con esta estrategia, ya que se pueden generar **desbalanceos de carga** sobre los distintos procesadores. Es importante mantener la carga balanceada entre los procesadores para maximizar los beneficios de tener más de un procesador. De otro modo, uno o más procesadores podrían quedarse sin nada que hacer, mientras que otros procesadores podrían tener una alta carga. Los algoritmos de **balanceo de carga** intentan mantener la carga de trabajo distribuida de forma pareja entre todos los procesadores del sistema SMP. Notemos que esto solo tiene sentido cuando los procesadores tienen colas de procesos *ready* privadas.

Hay dos estrategias generales para el balanceo de carga: push migration y pull migration. Estas estrategias no son mutuamente excluyentes y, de hecho, suelen ser implementadas en paralelo.

- Con **push migration**, una tarea específica del kernel revisa de forma periódica la carga de cada procesador y, si encuentra un desbalance sobre la carga de los procesadores, distribuye los procesos pendientes a procesadores con poca carga.
- Con **pull migration**, un procesador se da cuenta de que tiene poca carga y, entonces, se trae tareas pendientes de otros procesadores.

La afinidad del procesador puede tomar varias formas. Cuando un sistema operativo tiene un política de intentar mantener un proceso corriendo en un mismo procesador, pero no lo garantiza, decimos que hay una **afinidad blanda** (soft affinity). En este caso, el sistema operativo intentará mantener un proceso en un procesador, pero es posible que el proceso migre ente dos procesos durante un balanceo de carga. En contraste, algunos sistemas tienen system calls que dan soporte a una **afinidad dura** (hard affinity), permitiendo que un proceso especifique un subconjunto de procesadores en los cuales puede correr. Muchos sistemas proveen ambos tipos de afinidades.

Supongamos que tenemos dos procesadores A y B, y tenemos un proceso P que estaba corriendo en A. Mientras P estuvo corriendo en A, los datos accedidos más recientemente por P poblaron la memoria cache del procesador A. Como resultado, los accesos sucesivos a memoria del proceso van a ser principalmente cache hits. Eventualmente, el scheduler lo desaloja y P termina en la *ready queue*. De repente, se libera el procesador B, ¿nos conviene traerlo a B, o es preferible esperar a que se libere A

para tener una mayor eficiencia con la cache?

En caso de que el scheduler decida migrar el proceso P al procesador B, los contenidos de la memoria cache del procesador A deben ser invalidados, y el cache para el procesador B debe volver a llenarse. Luego, si la política de scheduling decide migrar un proceso a otro procesador, debido al alto costo de invalidar y repoblar caches, es posible que el proceso P termine tardando más en completarse que si hubiese ejecutado en el mismo procesador que antes.

Podemos ver que el balanceo de cargas puede contrarrestar los beneficios de mantener la afinidad con el procesador, ya que termina moviendo procesos de un procesador con el que tienen afinidad a otro con el que no tienen necesariamente afinidad. En otras palabras, hay una tensión natural entre mantener las cargas de los procesadores balanceadas y mantener la afinidad con el procesador. La mayoría de sistemas operativos con soporte SMP intentan evitar tener que migrar procesos de un procesador a otro y, en su lugar, intentan mantener al proceso corriendo en el mismo procesador, aunque se tarde un poco más en obtenerlo.

NUMA awareness

La arquitectura de la memoria principal de un sistema (UMA / NUMA) también puede afectar a la afinidad al procesador. En una arquitectura NUMA, donde cada procesador tiene su propia memoria local, cada CPU tiene un acceso más rápido a su memoria local que a la memoria de otra CPU, a pesar de que el sistema de interconexión permita que se comparta un mismo espacio de direcciones físicas. Si el scheduler del sistema operativo y los algoritmos de administración de memoria son **NUMA-aware** y trabajan en conjunto, entonces un proceso que ha sido asignado a una CPU puede reservar la memoria más cercana a la CPU donde ejecuta, haciendo que los accesos a memoria sean lo más rápido posible.

4.5. Scheduling en Sistemas Real-Time

Los sistemas de tiempo real son aquellos en donde las tareas tienen fechas de finalización (deadlines) estrictas que hay que cumplir. Esto no significa que las tareas sean rápidas, sino que deben ser predecibles. En general, se usan en entornos críticos, por lo que si un deadline no se cumple, suele suceder algo muy malo.

Podemos distinguir entre sistemas soft real-time y los hard real-time. Los sistemas **soft real-time** no proveen garantías sobre cuándo un proceso crítico será puesto a ejecutar. Solo garantiza que a ese proceso se le dará preferencia sobre procesos no críticos. Los sistemas **hard real-time** tienen requerimientos más estrictos. Una tarea debe ser servida antes de su deadline; atenderla luego de que se haya pasado su deadline es lo mismo que no haberla atendido.

La característica más importante de un sistema operativo real-time es que pueda responder inmediatamente a un proceso real-time a penas este proceso requiera de la CPU. Como resultado, el scheduler para un sistema real-time debe dar soporte a un algoritmo basado en prioridades con desalojo. No-temos que proveer un scheduler con desalojo basado en prioridades solo garantiza una funcionalidad soft real-time. Los sistemas hard real-time deben dar mayores garantías sobre las tareas real-time, para asegurarnos que serán atendidas de acuerdo con sus requerimientos deadline, y hacer cumplir estas garantías requiere de características adicionales del scheduling. En lo que sigue de esta sección, vamos a ver algoritmos de scheduling apropiados para sistemas hard real-time.

La política de Earliest-Deadline-first (EDF) asigna prioridades de forma dinámica según el deadline del proceso. Cuanto más cerca esté el deadline, mayor será su prioridad. Bajo la política EDF, cuando un proceso pasa al estado *ready*, debe anunciar cuál es su deadline al sistema. Luego, la prioridad del proceso podría tener que ser ajustada para reflejar el deadline anunciado.

La motivación de usar EDF es que es teóricamente óptimo, es decir, debería poder organizar la ejecución de los procesos de manera tal que cada proceso pueda cumplir con su deadline y la utilización de la CPU termine siendo del 100 %, siempre que esto sea posible. En la práctica, debido a los tiempos de context switching entre procesos y la atención de interrupciones, es imposible lograr este nivel de utilización de la CPU.

Lo que es inusual sobre esta forma de scheduling es que un proceso debe anunciar sus requerimientos de deadline al scheduler. Entonces, usando una técnica conocida como algoritmo de **admission-control**, el scheduler puede aceptar el proceso, garantizando que el proceso será completado en tiempo, o rechazarlo como imposible si no puede garantizar que la tarea será atendida antes del deadline.

4.6. Thread Scheduling

Antiguamente, todos los procesos tenían un único hilo de ejecución. En este tipo de sistemas, el scheduler opera a nivel de proceso. Hoy en día, todos los sistemas operativos modernos soportan procesos multithreaded y, en estos sistemas, el scheduler opera a nivel de thread y no a nivel de proceso.

Cuando hablamos de scheduling sobre threads, es importante distinguir el caso de threads de usuario y el caso de threads del kernel. Si el threading es realizado por una biblioteca dentro del user-space y el kernel desconoce la existencia de los threads, entonces el scheduling ocurre a nivel de procesos. Cuando el threading es realizado por el kernel, la situación es distinta. Aquí, el kernel es consciente de todos los threads y puede seleccionar específicamente a un thread dentro de un proceso. Ambas situaciones son completamente distintas, por lo que las analizaremos por separado.

User Threads Scheduling

Primero consideremos el caso de threads de usuario manejados por una biblioteca en el user-space. Como el kernel no es consciente de la existencia de los threads, opera como siempre lo hace: elige un proceso A y le asigna la CPU por un quantum. Luego, el run-time system decide cuál de todos los threads de A se debe poner a ejecutar, digamos A_1. Este thread continuará ejecutando hasta que se bloquee, ceda la CPU a otro thread de A o se le acabe el quantum. Si A_1 utiliza todo el quantum del proceso A, el kernel seleccionará a otro proceso para ejecutar.

Cuando le toque nuevamente al proceso A, el thread A_1 continuará con su ejecución. Si este thread actúa de manera egoísta, continuará consumiendo todo el tiempo asignado al proceso A, mientras que el resto de los threads de A nunca obtendrán tiempo de CPU para poder ejecutar. Sin embargo, los demás procesos del sistema no se verán afectados por las decisiones egoístas de A_1. Obtendrán lo que el scheduler del sistema operativo considere apropiado, sin importar lo que suceda dentro del proceso A.

Normalmente, los threads son cooperativos, por lo que la situación descrita anteriormente no suele ser un problema en la práctica. Típicamente, cada thread de A va a correr por un corto intervalo de tiempo, cediendo la CPU devuelta al scheduler de threads para que pueda ejecutar otro thread de A. Esto podría llevar a la secuencia $A_1, A_2, A_3, A_1, A_2, A_3$, antes de que el kernel cambie el proceso A por el proceso B. Esta situación se muestra en la Fig. 4.3(a).

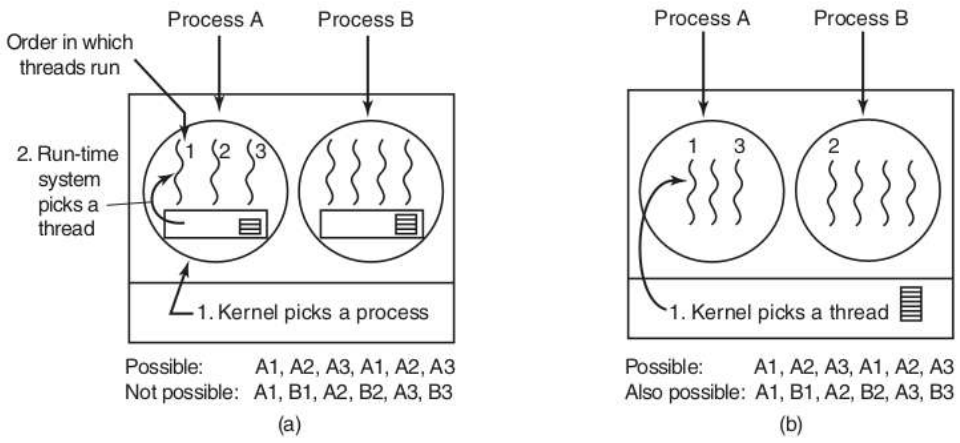

Figura 4.3: (a) Posible scheduling de threads de usuario en un proceso con un quantum de 50-msec y threads que corren por 5 msec por ráfaga de CPU. (b) Posible scheduling de threads del kernel con las mismas características que en (a).

El algoritmo de scheduling utilizado por el run-time system puede ser cualquiera de los ya conocidos. En la práctica, se suelen utilizar round-robin o un scheduling basado en prioridades. La única diferencia es la ausencia de las interrupciones de reloj para desalojar un thread que haya ejecutado por mucho tiempo.

Kernel Threads Scheduling

Ahora consideremos la situación con threads del kernel. Aquí, el kernel es consciente de la existencia de los threads, por lo que el scheduler selecciona un thread particular para ejecutar en lugar de un proceso. Al thread seleccionado le asigna un quantum, y es desalojado forzosamente si excede este intervalo de tiempo.

Si tenemos un quantum de 50-msec y threads que se bloquean luego de 5 msec, el orden de ejecución de threads para un intervalo de 30 msec podría ser: $A_1, B_1, A_2, B_2, A_3, B_3$. Notemos que esto no sería posible usando threads de usuario— al menos con estos parámetros. Esta situación se muestra en la Fig. 4.3(b).

Diferencias

Una diferencia importante entre threads de usuario y threads del kernel es la performance. Hacer un cambio de threads con threads de usuario tan solo nos cuesta un par de instrucciones. Con threads del kernel se requiere un cambio de contexto completo, cambiar los mapeos a memoria e invalidar la cache. Esto hace que cambiar threads del kernel sea varias veces más costoso que cambiar threads de usuario. Por otro lado, los threads del kernel pueden bloquearse en una E/S sin obligarnos a tener que suspender a todo el proceso. Recordemos que si usamos threads de usuario nos vemos obligados a bloquear todo el proceso, dado que el kernel no es consciente de la existencia de los threads.

Otra diferencia es que, cuando utilizamos threads del kernel, el scheduler sabe que si está ejecutando un thread en el proceso A, es más costoso pasar a ejecutar un thread del proceso B que pasar a ejecutar otro thread del proceso A. Esta información puede ser usada para tomar la decisión de scheduling. Por ejemplo, dados dos threads con la misma prioridad, pero uno perteneciente al mismo proceso que el thread que acaba de bloquearse y el otro pertenece a un proceso diferente, se podría preferir seleccionar al primero de manera preferencial sobre el segundo.

Otro factor importante es que los threads de usuario puede utilizar un scheduler de threads específico a la aplicación. El run-time system conoce lo que pueden hacer todos los threads, y puede aprovechar esta información para tomar mejores decisiones.

4.7. Linux Scheduler

A continuación, vamos dar un vistazo general al mecanismo de scheduling de Linux. Para comenzar, Linux implementa threads del kernel, por lo que el scheduling es basado en threads, y no en procesos. Se distinguen tres clases de threads, en concordancia con el estándar POSIX.1b:

- Real-time FIFO (`SCHED_FIFO`).
- Real-time round robin (`SCHED_RR`).
- Timesharing (`SCHED_OTHER`).

Los threads real-time FIFO son los de mayor prioridad y no pueden ser desalojados, salvo por un nuevo thread real-time FIFO de mayor prioridad. Los threads real-time FIFO son iguales a estos, excepto por que tienen un quantum asociado y, en consecuencia, pueden ser desalojados durante la atención de una interrupción de reloj.

Ninguna de estas clases son, de hecho, real-time. Ni siquiera podemos especificar deadlines y menos tener garantías sobre éstas. Estas clases simplemente nos permiten establecer ciertos procesos con mayor prioridad que los threads en la clase estándar.

Para obtener y configurar la política de scheduling de un thread, la API POSIX especifica las

siguientes dos funciones:

```
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

El primer parámetro para ambas funciones es un puntero al conjunto de atributos del thread. El segundo parámetro es (1) un puntero a una variable entera donde se retornará la política actual de scheduling o (2) un valor entero (`SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER`).

Internamente, los threads real-time son representados con niveles de prioridad desde 0 a 99, siendo 0 el nivel de mayor prioridad y 99 el menor; mientras que los threads tradicionales son representados con niveles de prioridad desde 100 a 139. El scheduling de los threads tradicionales se da en un segundo nivel, y se utiliza un algoritmo distinto— conocido como Completely Fair Scheduler (**CFS**). De esta manera, los threads tradicionales no compiten por recursos con los threads real-time, sino que estos últimos siempre tienen prioridad sobre los primeros.

4.7.1. CFS

El scheduler CFS es un algoritmo de scheduling basado en prioridades. CFS no asigna prioridades estáticamente a lo threads, sino que va registrando el tiempo que cada thread ha estado corriendo— con una granularidad de nanosegundos —y, en base a éste, se calcula dinámicamente un valor conocido como **virtual run time** que se utiliza como prioridad en el algoritmo de scheduling. Este valor se almacena en una variable `vruntime` para cada thread. Luego, el scheduler simplemente selecciona la thread con menor `vruntime`. Además, una thread de mayor prioridad que está disponible para ejecutar puede desalojar a una thread de menor prioridad. Es decir, CFS es un algoritmo de scheduling **preemptive**.

Al igual que en la mayoría de sistemas UNIX, Linux asocia un **nice value** a cada thread. El valor por defecto es 0, pero esto puede ser cambiado utilizando la system call `nice` (si se tienen los permisos adecuados). El rango de posibles **nice values** es -20 a 19. Este valor es la prioridad estática del thread, y afecta en el cálculo del `vruntime`. En particular, para un proceso de menor prioridad estática, el tiempo virtual pasa más rápido; mientras que, para un proceso de mayor prioridad estática, el tiempo pasa más lento. Es decir, un thread de menor prioridad estática (mayor *nice value*) perderá más rápidamente la CPU y será devuelto a la cola de procesos listos antes que un thread de mayor prioridad.

La estructura de datos que se utiliza como *ready queue* es un *red-black tree*. Los threads son ordenados en el árbol basándose en su `vruntime`. De esta manera, podemos consultar el siguiente nodo a ejecutar en tiempo constante, y también podemos insertar un nodo en la estructura en tiempo logarítmico (en función de la cantidad de threads). (Recordemos que el nodo con menor clave siempre es el de más a la izquierda.)

Los threads que están en estado *waiting*, esperando por operaciones de E/S o algún otro evento del kernel, se colocan en otra estructura de datos— conocida como *wait queue*. Cada evento que podría bloquear a un thread tiene una *wait queue* propia asociada. Esta estructura incluye un puntero a una lista enlazada de threads y un spin lock. El spin lock es necesario para controlar el acceso concurrente a la *wait queue*.

Balanceo de Carga y Migración de threads

El scheduler CFS también realiza un balanceo de carga usando una técnica sofisticada que, no solo ecualiza las cargas entre núcleos de procesamiento, sino que también minimiza la migración de threads y es NUMA-aware.

CFS define la carga de cada thread como la combinación de la prioridad del thread y su tasa promedio de utilización de CPU. Usando esta métrica, es posible balancear la carga simplemente asegurándose que todas las CPUs tengan aproximadamente la misma carga total (suma de las cargas de los threads activos en las colas privadas de cada CPU).

Como vimos anteriormente, migrar un thread puede resultar negativo para la performance, por ejemplo, debido a que se invalida la cache o, en sistemas NUMA, por incurrir en tiempos de acceso a memoria más largos. Para resolver este problema, Linux identifica un sistema jerárquico de **dominios**

de scheduling.

Un dominio de scheduling es un conjunto de núcleos de procesamiento que pueden ser balanceados entre sí. Cuanta más distancia haya entre dos dominios en la jerarquía, mayor será el costo de migrar un thread de un dominio a otro. Es decir, los núcleos en cada dominio son agrupados de acuerdo a cómo comparten recursos en el sistema.

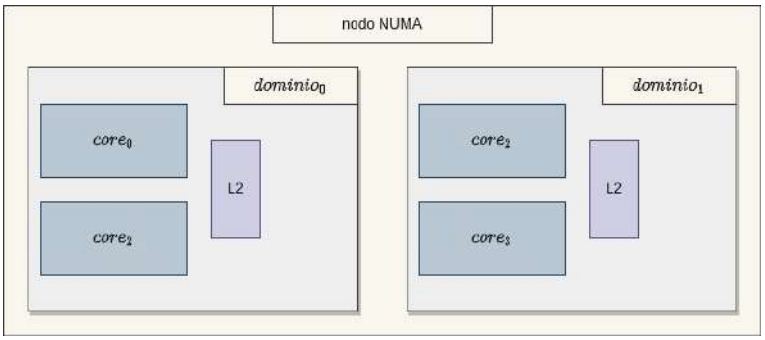

Figura 4.4: Balanceo de carga NUMA-aware con scheduler CFS de Linux.

Por ejemplo, si bien cada core puede tener su propia cache de nivel 1 (L1), es posible que dos cores compartan una cache de nivel 2 (L2) y, por tanto, los podemos organizar dentro de un mismo dominio. Si tenemos dos pares de cores, y cada par tiene su propia cache de nivel 2, entonces tendremos dos dominios distintos. De la misma manera, estos dos dominios pueden compartir una misma cache de nivel 3 (L3) y, en consecuencia, pueden ser organizados en un dominio a nivel de procesador (también conocido como nodo NUMA).Esta idea se muestra en la Fig. 4.4. Esto se puede seguir extendiendo un nivel más, en un sistema NUMA, donde tenemos un dominio que combina varios nodos NUMA cercanos.

Para saber a con qué CPU un thread tiene afinidad, se mantiene una *affinity mask* asociada a cada thread (cada thread tiene un atributo propio en la tabla de threads). Esta máscara se configura para poder asegurar que un thread solo pueda ser ejecutado en un subconjunto seleccionado de CPUs. Se puede configurar llamando a la system call `sched_setaffinity` y consultar llamando a la system call `sched_getaffinity`.

La estrategia general detrás de CFS es balancear la carga dentro de dominios, empezando desde el nivel más bajo de la jerarquía. Usando como ejemplo la Fig. 4.4, inicialmente un thread podría migrar tan solo entre cores del mismo dominio. El balanceo de carga en el siguiente nivel ocurría entre $dominio_0$ y $dominio_1$.

Capítulo 5

Sincronización entre procesos

5.1. Introducción

Los procesos (y threads) frecuentemente necesitan comunicarse entre sí. Para lograr la comunicación entre procesos (o threads), primero se deben resolver una serie de problemas intrínsecos a la comunicación entre procesos. Un primer problema es cómo hacer para que un proceso pueda compartir información con otro proceso. Para esto, se debe poder tener un medio compartido que permita la comunicación entre procesos. Además, es necesario tener un mecanismo para que los procesos puedan ejecutar de manera concurrente y no de manera secuencial. En capítulos anteriores, ya vimos cómo el scheduler opera para permitir la ejecución concurrente o paralela de procesos, y también vimos distintos mecanismos para compartir información (pasaje de mensajes, memoria compartida, threads). También, dijimos que se necesitan mecanismos de sincronización para operar sobre datos compartidos. En particular, estos mecanismos de sincronización deben permitirnos evitar que dos o más procesos terminen metiéndose en el camino del otro y, al mismo tiempo, lograr una secuenciación adecuada entre dependencias existentes. Por ejemplo, si un proceso A produce datos y un proceso B los consume, es necesario que A primero produzca los datos para que luego B los pueda consumir.

Compartir información entre threads es una tarea trivial, donde que los threads de un mismo proceso comparten un mismo espacio de direcciones. Sin embargo, la ejecución concurrente puede contribuir a problemas que involucran la integridad de datos compartidos, independientemente si estamos trabajando con threads o procesos. Asegurar una secuenciación adecuada de procesos o threads, en ambos casos, también es un desafío. A continuación, vamos a discutir problemas y soluciones inherentes a la programación concurrente, que aplican tanto para el caso de threads como procesos— aunque normalmente vamos a estar hablando de procesos. La ubicación de la memoria compartida (kernel o file system) tampoco afecta la naturaleza de la comunicación o de los problemas que aparecen.

Notemos que todos los mecanismos estudiados en este capítulo asumen que los procesos tienen una memoria común. En un sistema distribuido, donde los procesos no comparten una misma memoria, es necesario otro tipo de herramienta de sincronización (por ejemplo, basada en sockets).

5.1.1. Race Conditions

Para introducirnos en los problemas asociados a la ejecución concurrente, consideremos el siguiente ejemplo. Imaginemos que debemos administrar un fondo de donaciones donde se sortean premios entre aquellos que participan de la donación. Cada vez que se realiza una compra, debemos incrementar el número de **ticket** y manejar el **fondo** acumulado.

Si estuviéramos en un ambiente de ejecución secuencial, el código para este programa en C, junto con su traducción a assembler para una arquitectura de registro acumulador, sería similar al que se muestra en la Fig. 5.1.

Ahora, supongamos que el sorteo está siendo un éxito y se nos pide aumentar la cantidad de procesos

```
int donar(int donacion){          load fondo
    fondo += donacion;            add donacion
                                  store fondo
    ticket++;                     load ticket
                                  add 1
    return ticket;                store ticket
}                                 return Registro
```

Figura 5.1: Fondo de donaciones (ejecución secuencial).

concurrentes que se encargan de atender la compra de tickets. Esto es, la ejecución del programa dejará de ser secuencial, y empezaremos a tener varios procesos ejecutando el mismo código de manera concurrente.

En particular, vamos a contar con **dos procesos** P_1 y P_2 para atender la compra de tickets, y las variables **ticket** y **fondo** se compartirán entre ambos procesos. Estamos suponiendo que tenemos un único registro acumulador, un único procesador y que el scheduler va alternando entre ambos procesos. Dadas estas condiciones, supongamos que se nos presenta la siguiente situación:

- En P_1 se quiere donar 10$.
- En P_2 se quiere donar 20$.
- Se parte con un fondo acumulado de 100$.
- El ticket actual es el número 5.

El resultado esperado, luego de terminar ambas donaciones, es que haya un fondo de 130$ y que se hayan repartido los tickets 6 y 7 a cada proceso correspondiente. No nos va a importar quién recibe qué ticket, sino que tan solo nos va a importar que cada donante reciba un ticket distinto.

Como la ejecución de los procesos es concurrente, existen múltiples scheduling posibles para la ejecución de un mismo programa. En particular, consideremos el scheduling mostrado en la Fig. 5.2. Las líneas de separación en la figura representan el momento en el que el scheduler pone a funcionar el otro proceso.

El proceso P_1 lee el valor de la variable *fondo* y la carga en su registro acumulador. Luego, suma 10 al valor en su registro. Justo después, el scheduler decide desalojar a P_1 y asignarle la CPU a P_2. P_2 lee el valor de la variable *fondo* y la carga en su registro. Notemos que en este punto, P_1 no llegó a actualizar el valor de *fondo*, sino que tan solo actualizó el valor de su registro. Por lo tanto, P_2 lee el valor inicial de *fondo*. Luego, P_2 le suma 20 al valor en su registro, y es scheduler lo desaloja.

P_1 vuelve a ser seleccionado para ejecutar, y continúa con la ejecución del programa actualizando el valor de la variable *fondo* con el valor almacenado en su registro acumulador (110$). El scheduler vuelve a desalojar a P_1 y le asigna la CPU a P_2. P_2 continúa la ejecución del programa donde lo dejó, y actualiza el valor de *fondo* con el valor en su registro (120$). En este punto, P_2 terminó pisando el valor escrito por P_1, por lo que los 10$ que había donado P_1 se pierden. P_2 lee la variable *ticket* y guarda su valor en su registro. Luego, suma 1 al valor de su registro, para ser desalojado por el scheduler.

Nuevamente, P_1 retoma su ejecución, carga el valor de *ticket* en su registro, suma 1 al valor del registro y actualiza el valor de la variable *ticket* con el valor de su registro (6). Notemos que, en este punto, P_2 todavía no actualizó la variable *ticket*, por lo que P_1 termina leyendo el valor inicial de la variable. P_1 es desalojado, y vuelve a ejecutar P_2. Ahora, P_2 escribe sobre la variable *ticket* el valor de su registro (6), para ser desalojado nuevamente. Finalmente, P_1 retorna de la función, el scheduler asigna la CPU a P_2, y P_2 retorna.

Claramente, el resultado obtenido no coincide con el resultado esperado: se perdieron 10$ y se imprimió dos veces el mismo ticket. El problema que tuvimos es que P_1 y P_2 accedieron al mismo tiempo a las variables compartidas y modificaron sus valores, en lugar de esperar a que el otro proceso terminara su trabajo con las variables compartidas. Situaciones como ésta, donde dos o más procesos están leyendo o escribiendo datos compartidos y el resultado final depende de quién ejecuta en qué orden, son llamadas

P_1 : donar(10)	P_2 : donar(20)	Registro de P_1	Registro de P_2	**fondo**	**ticket**
load fondo		100		100	5
add 10		110		100	5
	load fondo	110	100	100	5
	add 20	110	120	100	5
store fondo		110	120	110	5
	store fondo	110	120	120!!	5
	load ticket	110	5	120	5
	add 1	110	6	120	5
load ticket		5	6	120	5
add 1		6	6	120	5
store ticket		6	6	120	6
	store ticket	6	6	120	6!!
return Registro		6	6	120	6
	return Registro	6	6	120	6

Figura 5.2: Un scheduling desfavorable.

condiciones de carrera (o race conditions).

Notemos que no es culpa del scheduler que hayamos obtenido un resultado incorrecto. Un programa concurrente debe garantizar que toda ejecución sea **equivalente** a alguna ejecución secuencial de los mismos procesos— incluso ante un scheduling adversario que busca la peor combinación posible. Para que un programa paralelo sea considerado correcto, éste tiene que estar libre de condiciones de carrera.

¿Cómo podemos resolver este problema? Lo que podemos hacer es forzar a que, en todo momento, tan solo pueda hacer un proceso accediendo a los datos compartidos. Dicho de otro modo, buscamos un mecanismos que nos permita garantizar lograr una exclusión mutua, esto es, una manera de asegurarnos que si un proceso está usando una variable o archivo compartido, el otro proceso será excluido de hacer lo mismo. La elección de operaciones primitivas adecuadas para conseguir la exclusión mutua es un problema central de diseño en cualquier sistema operativo, y es un tema que examinaremos en gran detalle en las próximas secciones.

5.2. Sección Crítica y Exclusión Mutua

Típicamente, podemos separar el código de un programa paralelo en dos partes. Una parte está destinada a realizar cómputos internos que no involucran variables compartidas y, por tanto, no generan condiciones de carrera (la llamamos *rem*). Por otro lado, tenemos la sección de código que involucra variables compartidas o archivos compartidos, que generan condiciones de carreras. Esa sección del programa, donde la memoria compartida es accedida, la llamamos **sección crítica**.

El problema de la sección crítica consiste en diseñar un protocolo que se pueda usar para sincronizar la ejecución de los procesos, evitando que dos procesos accedan al mismo tiempo a la sección crítica. Si bien este requerimiento evita las condiciones de carrera, no es suficiente para que los procesos puedan cooperar correctamente y de manera eficiente. Por lo menos, necesitamos garantizar las siguientes propiedades:

- **Exclusión Mutua**. Solo puede haber un proceso está ejecutando en la sección critica a la vez.

- **Progreso**. Ningún proceso corriendo por fuera de la sección crítica puede bloquear a ningún otro proceso.
- **Espera acotada** (o *Freedom from Starvation*). Cada proceso que intenta entrar a su sección crítica, eventualmente, lo consigue. Notemos que no nos alcanza con garantizar que siempre haya procesos entrando y saliendo de la sección crítica— propiedad conocida como *Freedom from Deadlock*. Es necesario garantizar que, en todo momento, el tiempo de espera para cualquier proceso esté acotado (finitamente).

En un sentido abstracto, el comportamiento que queremos es el presentado en la Fig. 5.3. Aquí, un proceso A intenta entrar a la sección crítica en tiempo T_1. Un poco después, en tiempo T_2, el proceso B intenta entrar a la sección crítica y falla, porque otro proceso ya está en la sección crítica.

Consecuentemente, B deja de progresar en la ejecución de su código hasta tiempo T_3, cuando A deja la sección crítica. En ese momento, B entra inmediatamente a la sección crítica de su código. Eventualmente, B libera la sección crítica y volvemos a la situación inicial.

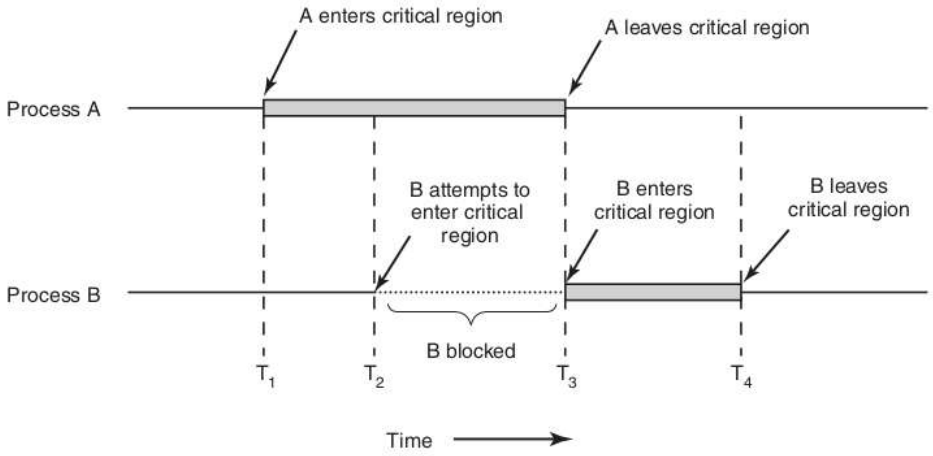

Figura 5.3: Exclusión mutua usando sección crítica.

Dada la descripción del problema, podemos empezar a pensar en posibles solución. A continuación, vamos a examinar varias propuestas para lograr la exclusión mutua.

Deshabilitar Interrupciones

En un sistema monoprocesador, los procesos concurrentes no pueden ejecutar en paralelo; solo pueden ejecutar de manera concurrente. En este contexto, la solución más simple al problema de la exclusión mutua consiste en deshabilitar todas las interrupciones justo después de entrar a la sección crítica y volver a habilitarlas justo después de salir de ella.

Como solo es posible desalojar un proceso solo puede ser desalojado como resultado de una interrupción, una vez que se hayan deshabilitado las interrupciones, podemos examinar y actualizar la memoria compartida sin miedo a que otro proceso intervenga.

Claramente, esta solución no es una solución general. Por un lado, no es una buena idea permitir que cualquier proceso de usuario pueda deshabilitar interrupciones. En caso de permitirlo, estos podrían simplemente nunca volver a activarlas. Además, al deshabilitar interrupciones, no solo se bloquea temporalmente a aquellos procesos que acceden a la sección crítica, sino que también a todos los demás procesos que no participan de la comunicación. Conceptualmente, no podemos definir distintas secciones críticas, sino que tan solo podemos definir una gran sección crítica. A pesar de todo esto, en un sistema monoprocesador, todavía es una alternativa conveniente para que el kernel (y no los procesos de usuario) pueda actualizar variables o listas compartidas entre todos los procesos.

Notemos que esta estrategia no funciona en un sistema multiprocesador. En este tipo de sistemas,

deshabilitar interrupciones no garantiza la exclusión mutua. Recordemos que cada CPU tiene asignado su propio proceso, por lo que los demás procesos continuarán corriendo y podrán acceder a la memoria compartida, pudiendo interferir con las operaciones de la primera CPU. Consecuentemente, queda claro que se necesitan esquemas más sofisticados.

Solución de Peterson

En 1981, G. L. Peterson descubrió un método sencillo para lograr exclusión mutua mediante una solución de software. Este algoritmo consiste de dos procedimientos: `enter_region` y `leave_region`.

```
1  #define FALSE 0
2  #define TRUE 1
3
4  int turn;
5  int interested[2];
6
7  void enter_region(int process)
8  {
9      interested[process] = TRUE;
10     turn = process;
11     int other = 1-process;
12     while(turn == process && interested[other] == TRUE)
13     {
14     // BUSY WAITING
15     }
16 }
17
18 void leave_region(int process)
19 {
20         interested[process] = FALSE;
21 }
22
23 int main(){
24         enter_region(pid);
25         critical_region();
26         leave_region(pid);
27     rem();
28 }
```

Figura 5.4: Solución de Peterson para la exclusión mutua

Antes de entrar a la sección crítica, cada proceso debe llamar a `enter_region` con su propio número de proceso como parámetro. En el ejemplo, contamos con dos procesos: P_0 ($pid = 0$) y P_1 ($pid = 1$). Esta llamada hará el proceso se quede esperando, si es necesario, hasta que sea seguro entrar a la sección crítica. Al salir de la sección crítica, el proceso debe llamar a `leave_region` para indicar que ha terminado y permitir que el otro pueda entrar en el futuro.

Veamos cómo funciona la solución. Inicialmente, ningún proceso está en su sección crítica. Consideremos el caso en que un proceso llama a `enter_region` y, inmediatamente después de que retorne de esta llamada, el otro proceso llama a `enter_region`.

- Supongamos que el proceso P_0 es el primero en llamar a `enter_region`. Esto lo hace indicando su interés al poner en su posición del arreglo y poniendo en la variable *turn* un 0. Como el proceso P_1 no está interesado, `enter_region` retorna inmediatamente.

- Si P_1 hace una llamada a `enter_region` inmediatamente después de que P_0 retorna de su llamada a `enter_region`, P_1 se quedará allí esperando a que P_0 salga de la sección crítica. En particular, cuando la variable compartida `interested`[0] pase a valer `FALSE`. Esto solo puede si P_0 llama a

`leave_region`, indicando que salió de la sección crítica.

Ahora consideremos el caso donde ambos procesos llaman a `enter_region` casi simultáneamente.

- Ambos almacenan su número de proceso en *turn*. Quien sea el primero en actualizar esta variable, será el que termina accediendo primero a la sección crítica. Supongamos que P_1 escribe último, por lo que *turn* pasa a valer 1.
- Cuando ambos procesos llegan al `while`, P_0 entra inmediatamente a su sección crítica, mientras que P_1 debe esperar. Esto porque la condición

  ```
  turn == process
  ```

  evalúa a *verdadero* para P_1, mientras que evalúa a *falso* para P_0.[1]

Un problema que tiene esta solución es que no está garantizado su correctitud en sistemas con arquitecturas modernas, donde los compiladores y procesadores podrían reordenar las instrucciones de lectura y escritura que no tengan dependencias entre sí. En particular, no hay ninguna dependencia entre la asignación de *turn* y la asignación de *interested*. Si el compilador o el procesador decide invertir estas dos asignaciones en ambos procesos, es posible que ambos procesos terminen accediendo al mismo tiempo a la sección crítica (ver Fig. 5.5).

P_0 : `enter_region`(0)	P_1 : `enter_region`(1)	interested[0]	interested[1]	turn
turn = 0		false	false	0
	turn = 1	false	false	1
	interested[1] = true	false	true	1
	turn == 1 && interested[0]	false	true	1
	`critical_region`	false	true	1
interested[0] = true		true	true	1
turn == 0 && interested[1]		true	true	1
`critical_region`		true	true	1

Figura 5.5: Efectos de reordenar instrucciones sobre la solución de Peterson.

Claramente, necesitamos una solución que funcione en sistemas con arquitecturas modernas.

Locks: TAS y CAS

Como tercer intento, consideremos tener una variable compartida (*lock*), inicializada en 0. Cuando un proceso intenta entrar a la sección crítica, éste primero revisa el estado del lock.

- Si el lock está en 0, significa que no hay ningún proceso actualmente en la sección crítica. Por lo tanto, el proceso pone en lock en 1 y entra a la sección crítica.
- Si el lock está en 1, significa que hay otro proceso en la sección crítica. Por lo tanto, el proceso debe esperar hasta que el lock pase a valer 0.

Desafortunadamente, esta idea contiene exactamente la misma falla que el ejemplo que vimos. Supongamos que un proceso intenta leer el lock y mira que es un 0. Antes de que pueda poner un 1, otro proceso es asignado a la CPU, y pone el lock en 1. Cuando el primer proceso le toma correr nuevamente, también verá el lock en 1, y los dos proceso estarán en la sección crítica al mismo tiempo. Para resolver este problema, necesitamos de la ayuda del hardware.

[1] Esta implementación está sacada de [TB18]. En [SGG18] hay otra implementación, pero es equivalente. La diferencia está en que se asigna *other* a *turn*, en lugar de *process*, y se compara *turn* con *other* en el *while*, en lugar de comparar con *process*. Estas diferencias no tienen ningún efecto sobre el comportamiento de las operaciones.

Muchos sistemas modernos proveen instrucciones especiales de hardware que nos permiten testear y modificar el contenido de una variable o intercambiar el contenido de dos variables de forma **atómica**. Que una instrucción sea atómica quiere decir que funciona como una unidad ininterrumpible e indivisible—ningún otro proceso puede acceder a la variable hasta que la instrucción sea completada.

En lugar de discutir una instrucción específica para una máquina específica, por ejemplo XCHG en arquitecturas Intel IA-32, vamos a abstraer los conceptos principales detrás de este tipo de instrucciones. Las dos instrucciones que nos van a interesar son *TestAndSet* y *CompareAndSwap*. Las instrucciones `TestAndSet` (**TAS**) y `CompareAndSwap` (CAS) pueden ser definidas como se muestra en la Fig. 5.6.

- TAS toma un parámetro: `reg`. A la variable apuntada por `reg` se le asigna el valor 1 en la variable, y se devuelve el valor original de la variable apuntada por `reg`.
- CAS toma tres parámetros: `reg`, `esperado` y `nuevo_valor`. A la variable referenciada por `reg`, se le asigna el `nuevo_valor` si y solo si el valor de `reg` es igual al `esperado`. En cualquier caso, CAS siempre devuelve el valor original de la variable apuntada por `reg`.

La característica más importante de estas instrucciones es que son ejecutadas de forma atómica. Luego, si dos TAS fueran ejecutadas en paralelo, cada una en un núcleo distinto, está garantizado que serán ejecutadas de forma secuencial (en algún orden arbitrario). Para garantizar la atomicidad de estas instrucciones especiales, se necesita ayuda de un hardware específico.

TAS lee una palabra de memoria y la escribe en un registro (lo mismo ocurre con CAS). Por supuesto, se necesitan dos ciclos de bus (como mínimo) para realizar la lectura y la escritura. Si no hacemos nada, durante esos ciclos de bus podría venir otra CPU, acceder al bus de memoria y acceder a la variable compartida. Para evitar esta situación, TAS debe primero bloquear el bus de memoria, hacer los dos accesos a memoria y desbloquear el bus. Típicamente, para bloquear el bus se utiliza una línea especial del bus, dedicada a poder bloquearlo hasta que ambos ciclos sean completados. Siempre y cuando se mantenga levantada esta línea especial, ninguna otra CPU tendrá acceso al bus de memoria. Claramente, este tipo de instrucciones solo puede ser implementadas en sistemas con el hardware y los protocolos necesarios. Sin embargo, la mayoría de sistemas modernos tienen estas características, por lo que no es un problema.

```
int CAS(int *reg, int esperado, int nuevo_valor)
{
    int resultado = *reg;
    if (*reg == esperado){
        *reg = nuevo_valor;
    }
    return resultado;
}
```

```
bool TAS(bool *reg)
{
    bool resultado = *reg;
    *reg = true;
    return resultado;
}
```

Figura 5.6: Pseudocódigo de TAS y CAS. Es importante notar que esto es simplemente un pseudocódigo que busca facilitar la comprensión del funcionamiento de estas instrucciones especiales. Estas operaciones forman parte de la ISA, y no se implementan en un lenguaje de alto nivel.

Es importante notar que bloquear el acceso al bus de memoria es muy distinto a deshabilitar las interrupciones. Deshabilitar las interrupciones no evita que un segundo procesador pueda acceder a una palabra durante ese tiempo. De hecho, deshabilitar interrupciones en un procesador no afecta de ningún modo a todos los demás procesadores. La única forma de evitar que otro procesador acceda a la memoria hasta que se termine de procesar la instrucción es bloqueando el bus de memoria.

Ahora, veamos cómo podemos usar estas instrucciones para implementar *locks* correctamente (ver Fig. 5.7 y Fig. 5.8). Antes de entrar a la sección crítica, un proceso debe llamar a *enter_ region*, haciendo **busy waiting** (o espera activa) hasta que el lock esté libre; luego obtiene el lock y retorna.

Recordemos que TAS asigna 1 a la variable y devuelve el **valor anterior**. Esto quiere decir que si nos devuelve 0, la sección crítica estaba libre y que conseguimos el acceso único a la sección crítica (poniendo el 1 en la variable). Tenemos la certeza de que nuestro proceso fue el que puso el 1 porque

la operación se realizó de manera atómica indivisible. Si nos devuelve 1, nos toca esperar porque otro proceso nos ganó de mano. Lo mismo ocurre con CAS.

Cuando salga de la sección crítica, el proceso debe llamar *leave_ region*, que libera el lock al asignarle un valor de 0. Una vez fuera de la sección crítica, podemos ejecutar el resto del código (libre de race conditions).

```
bool lock = 0; // Compartida.
void enter_region(){
    while(TAS(&lock) != 0){/*Busy waiting*/}
}
void leave_region(){
    lock = 0;
}
void main()
{
    enter_region();
    critical_region();
    leave_region();
    rem();
}
```

Figura 5.7: Implementando Exclusión Mutua con TAS. Nuevamente, vale aclarar que estas implementaciones son pseudocódigo. Una implementación real usando TAS o CAS debe hacerse en un lenguaje de ensamblador, y no en un lenguaje de alto nivel.

```
void enter_region(){
    while(CAS(&lock, 0, 1) != 0)){/*Busy waiting*/}
}
```

Figura 5.8: Implementando Exclusión Mutua con CAS. El resto de las funciones son las mismas que en el caso de la implementación con TAS.

El término **busy waiting** (o spin waiting), hace referencia a que un proceso no haga progreso hasta que obtenga permiso para entrar a su sección crítica, sino que se la pase ejecutando un código que revisa continuamente el estado del lock.

El uso de instrucciones especiales para garantizar la exclusión mutua tiene una serie de ventajas. Es aplicable a cualquier número de procesos, tanto en sistemas monoprocesador como en sistemas multiprocesador, y funciona en arquitecturas modernas. Además, puede ser usado para dar soporte a múltiples secciones críticas: cada sección crítica puede ser definida con sus propias variables (sin afectar al resto de procesos).

5.3. Sincronización con Objetos

Las soluciones basadas en utilizar instrucciones de hardware presentadas en el apartado anterior, si bien son correctas, son complicadas y generalmente no las instrucciones TAS y CAS no son accesibles para programar aplicaciones. En su lugar, los sistemas operativos suelen venir con herramientas de software de más alto nivel para resolver el problema de la sección crítica. Una de las herramientas más sencillas son las **variables atómicas**, que proveen operaciones atómicas en tipos de datos básicos como enteros o booleanos. Las variables atómicas suelen ser utilizadas para actualizar datos compartidos como contadores o generadores de secuencias.

La mayoría de sistemas que soportan variables atómicas, proveen tipos de datos especiales al igual que funciones primitivas para acceder y manipular variables atómicas. En la Fig. 5.9 se muestran algunos ejemplos. Estas operaciones son implementadas usando las operaciones primitivas de hardware como TAS o CAS. En general, se conoce como **primitivas de sincronización** a aquellos objetos cuyos métodos exportados son las propias instrucciones de sincronización.

```
private bool reg;
atomic int getAndInc(){
    int old_value = reg;
    reg++;
    return old_value;
}
atomic int getAndAdd (int value){
    int old_value = reg;
    reg = reg + value;
    return old_value;
}
atomic T CAS(T expected, T new_value){
  T old_value = reg;
  if(expected == old_value){
       reg = new_value;
  }
    return old_value;
}
```

```
atomic bool get(){return state;}

atomic bool set(bool new_state){
    state = new_state;
}
atomic bool getAndSet(bool new_state){
    bool temp = state;
    state = new_state;
    return temp;
}
atomic bool testAndSet(){
    return getAndSet(true);
}
```

Figura 5.9: Especificación de primitivas para Variables atómicas. Nuevamente, esto es simplemente pseudocódigo. Implementaciones reales de estas primitivas se deben hacer en lenguaje ensamblador, utilizando instrucciones especiales como TAS o CAS.

En base a estas primitivas de sincronización, es posible construir herramientas más robustas directamente en lenguajes de alto nivel, evitando tener que recurrir a las instrucciones de máquina.

5.3.1. Mutex Locks

La herramienta más sencilla es el **mutex** (o mutex lock). Utilizamos un mutex lock para proteger el acceso a una sección crítica, y así poder eliminar las condiciones de carrera. Un proceso debe llamar a `lock` antes de poder entrar a una sección crítica, y debe llamar a `unlock` al salir. Existen múltiples maneras de implementar un mutex: hay implementaciones que hacen busy waiting, otras que son bloqueantes, para threads del kernel, para threads de usuario, etc. A continuación, vamos a presentar algunas implementaciones y vamos a discutir las ventajas y desventajas de cada una.

TASLock

Cuando escribimos programas secuenciales, usualmente es aceptable ignorar los detalles arquitecturales del sistema. Desafortunadamente, la programación paralela todavía no ha llegado a ese punto, por lo que es crucial entender cómo la arquitectura afecta a la performance.

La operación `testAndSet` fue la principal instrucción de sincronización provista por muchas arquitecturas multiprocesador tempranas. Recordemos que esta instrucción opera sobre una variable que contiene un valor binario, *true* o *false*. La instrucción TAS guarda un *true* en la variable de manera atómica, y devuelve el valor anterior de la variable. Recordemos que, para lograr la atomicidad en un sistema multiprocesador, la instrucción TAS lockea al bus compartido, previniendo que otras CPUs puedan acceder al bus, luego hacer los dos accesos a memoria, y libera el bus. Como el proceso se queda loopeando esperando a que se libere el lock, a este tipo de mutex se lo conoce como **spin lock**.

A primera vista, esta instrucción parece ideal para implementar un spin lock. El lock está libre cuando la variable está en *false*, y está ocupado cuando está en *true*. Luego, la operación `lock` consiste en aplicar repetidas veces `testAndSet` hasta que devuelva *false*, es decir, hasta que el lock esté libre. La operación de `unlock` simplemente escribe *false* en la variable. La clase TASLock presentada en la Fig. 5.10 muestra un algoritmo de lockeo basado en la instrucción `testAndSet`.

```
atomic<bool> ocupado;

void create(){
    ocupado.set(false);
}
void lock(){
    while(ocupado.testAndSet()){/*Spinning*/}
}
void unlock(){
    ocupado.set(false);
}
```

Figura 5.10: La clase TASLock.

TTASLock

Ahora consideremos una segunda implementación la alternativa, ilustrada en la Fig. 5.11. En lugar de realizar los `testAndSet` directamente, el proceso lee repetidamente el estado del lock hasta que esté libre, es decir, hasta que `get` devuelva *false*. Recién ahí, el proceso realiza el `testAndSet`. Esta técnica se conoce como *test-and-test-and-set* y este tipo de lock se llama TTASLock.

```
atomic<bool> ocupado;

void create(){
    ocupado.set(false);
}
void lock(){
    while(true){
        while(ocupado.get()){/*Local Spinning*/}
        if (!ocupado.testAndSet()){
            return;
        }
    }
}
void unlock(){
    ocupado.set(false);
}
```

Figura 5.11: La clase TTASLock.

Claramente, ambos algoritmos son equivalentes desde el punto de vista de la correctitud: cada uno garantiza la exclusión mutua. A primera vista, podríamos pensar que ambos algoritmos son igualmente eficientes. Sin embargo, en un sistema multiprocesador real, hay una gran diferencia en performance.

Comparativa: TASLock vs TTASLock

Al analizar el tiempo que N procesos tardan en ejecutar una sección crítica, siempre se producen resultados similares a los mostrados en la Fig. 5.12. Cada punto en el gráfico representa la misma cantidad de trabajo, por lo que en ausencia de efectos de contención, todas las curvas deberían ser planas. La diferencia es notable: TASLock tiene un bajo rendimiento y, aunque TTASLock funciona sustancialmente mejor, vemos que está lejos de ser lineal.

Estas diferencias pueden ser explicadas en términos de las arquitecturas modernas multi-procesador. Recordemos un poco cómo funcionan los sistemas SMP. Una arquitectura multiprocesador SMP se comunica mediante un bus compartido. Tanto los procesadores como el controlador de memoria pueden broadcastear en el bus, pero solo un procesador (o memoria) puede broadcastear al mismo tiempo. Todos los procesadores (y memoria) pueden escuchar.

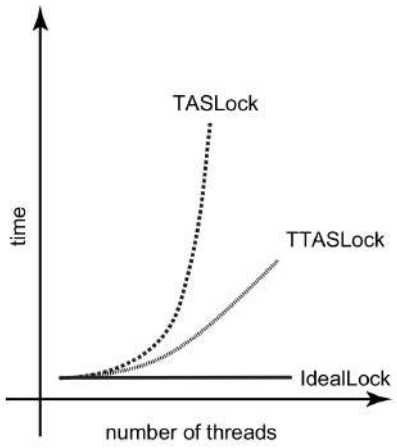

Figura 5.12: Esquema del rendimiento de un TASLock, un TTASLock y un lock ideal libre de overhead.

Cada procesador tiene su propia *cache*. Cuando un procesador lee una dirección en memoria, primero verifica si la dirección y su contenido están presentes en la cache. Si está presente, entonces tenemos un *cache hit* y podemos cargar el valor inmediatamente. Sino, tenemos un *cache miss* y debemos buscar los datos en la cache de algún otro procesador o, en el peor caso, en memoria.

Para intentar leer el valor de la cache de otro procesador, se broadcastea la dirección de la varieble por el bus compartido, y los demás procesadores *snoopean* esta dirección de memoria. Si alguno de los procesadores tiene esa dirección en su cache, entonces responde broadcasteando la dirección y su valor. Si ningún procesador tiene esa dirección, entonces hay que esperar a que la memoria responda con el valor de la variable. Recordemos que un acceso a memoria típicamente requiere de varios órdenes más ciclos de máquina que los accesos a cache, por lo que un buen uso de la cache es fundamental para el rendimiento en una arquitectura multi-procesador.

TASLock. Ahora, consideremos cómo es que el algoritmo TASLock funciona en una arquitectura con bus compartido. La primer llamada a `testAndSet` genera un broadcast en el bus compartido, ya que el procesador que la intenta leer todavía no la tiene en su cache. Como todos los procesos deben usar el bus para comunicarse con la memoria, estas llamadas a TAS retrasan a todos los demás procesos— incluso a aquellos que no están esperando por el lock.

A primera vista, podría parecer que, pasado este primer broadcast, la presencia de caches debería eliminar el problema de la contención sobre el bus, pero no lo hace. Una vez que la CPU que ha leído el estado del lock, se traer una copia a su cache. Siempre que no haya otra CPU intentando obtener el lock, el procesador puede manejarse con su copia en la cache. Cuando la CPU que está en posesión del lock escriba un 0 para liberarlo, el protocolo de cache automáticamente invalida todas las copias en las demás caches. Por lo tanto, el procesador que estaba haciendo continuamente TAS se ve obligado a hacer un segundo broadcast para poder leer el valor actualizado.

En la práctica, las palabras que rodean al lock son necesitadas por la CPU que está en posesión del lock. Como la instrucción TAS hace una escritura, se necesita acceso exclusivo al bloque de cache que contiene al lock. Por lo tanto, cada TAS que se ejecute invalida al bloque en la cache de la CPU que está en posesión del lock— a pesar de que el estado del lock en realidad no cambió.

Tan pronto como el poseedor del lock acceda a una palabra adyacente al lock, el bloque de cache será traído a su CPU. Consecuentemente, el bloque entero de cache que contiene al lock hace el viaje de ida y de vuelta constantemente entre el poseedor del lock y el que está haciendo busy waiting, generando incluso más tráfico en el bus del que lecturas individuales del lock habrían generado. Además, en caso de tener muchos procesos haciendo busy waiting, es posible que estos procesadores terminen monopolizando el uso del bus, impidiendo que el procesador que tenía el lock lo libere. Esto explica por qué TASLock tiene un rendimiento tan pobre.

TTASLock. Ahora, consideremos el comportamiento del algoritmo TTASLock. Cuando un proceso lee el estado del lock por primera vez sufre un cache miss, teniendo que esperar hasta que el valor sea cargado en su cache. Mientras que el lock no sea liberado, todos los procesos haciendo busy waiting vuelven a leer el valor del lock, pero siempre caen en su cache. Como leer de la cache no produce tráfico por el bus compartido, no se ralentiza a los accesos a memoria de los demás procesos. Además, si la CPU que tiene el lock no modifica las variables que están en el mismo bloque de cache que el lock, cada CPU puede tener una copia del bloque de cache en el modo *shared*, eliminando todas las transferencias de cache blocks.

Sin embargo, la situación se deteriora cuando el lock es liberado. La CPU que está en posesión del lock lo libera escribiendo *false* sobre la variable. Esto invalida inmediatamente las copias cacheadas de todos los procesadores que estaban haciendo busy waiting. Cada uno sufre un cache miss, vuelve a leer el valor y todos ellos llaman a `testAndSet` casi simultáneamente.

El primero en tener éxito invalida a todos los demás, que deben volver a leer el valor, causando una tormenta de tráfico sobre el bus compartido. Eventualmente, los procesos se normalizan y vuelven a hacer **local spinning**. Esta noción de *local spinning*, donde los procesos leen repetidamente los valores de la cache en lugar de usar el bus, es un principio crítico para el diseño de spin locks eficientes.

Busy waiting: consideraciones adicionales

La técnica de busy waiting funciona bajo la asunción de que, eventualmente, el proceso que está en la sección crítica, eventualmente, sale. Claramente, si algún proceso hace trampa y nunca sale de la sección crítica, la exclusión mutua falla. Las secciones críticas solo funcionan si los procesos que participan de la comunicación cooperan entre sí.

En un sistema multiprocesador con scheduler preemptive, puede ocurrir que un proceso que obtenga un spin lock justo antes de que se acabe su quantum. Si, en ese momento, hubieran procesos ejecutando en paralelo esperando por ese lock, se desperdicia una gran cantidad de ciclos de CPU hasta que ese proceso vuelva a ejecutar y libere el lock. Es más, en caso de que tengamos un sistema con al menos dos clases de prioridades, ni siquiera podemos garantizar que todo proceso que entre a la sección crítica, eventualmente, salga.

Consideremos el siguiente escenario. El proceso P_1 llama exitosamente a `lock` y entra a la sección crítica. P_1 es desalojado por la llegada de P_2, que es un proceso con mayor prioridad. Si P_2 ahora intenta usar el mismo recurso que P_1, y se le denegará el acceso debido al mecanismo de exclusión mutua. Luego, se pondrá a hacer busy waiting. Sin embargo, como P_1 tiene menor prioridad que P_2, nunca será seleccionado por el scheduler, por lo que nunca va a poder salir de la sección crítica y P_2 se quedará haciendo busy waiting indefinidamente. Es decir, si el scheduler no puede garantizar *starvation freedom* sobre la CPU, entonces es posible que el sistema entre en *deadlock*. Este tipo de situaciones, donde un proceso de menor prioridad termina afectando el progreso de un proceso de mayor prioridad, se conocen como **inversión de prioridades**.

Para resolver este problema, algunos sistemas utilizan la técnica de **smart scheduling**. La idea es que cuando un proceso adquiere un spin lock, se setea un flag asociado al proceso que muestra que este thread actualmente tiene un spin lock. Cuando se libera el lock, se limpia el flag. De esta manera, el scheduler se puede dar cuenta cuándo un proceso tiene un spin lock y cuándo no. Con esta información, el scheduler puede decidir darle un poco más de tiempo al proceso para que complete su sección crítica y que libere el lock, evitando desperdiciar ciclos de CPU de los procesos paralelos. En un scheduler non-preemptive, esto no es un problema.

Mutex para threads

Cuando trabajamos con threads de usuario, que tan solo cooperan con threads de su mismo proceso, es necesario adaptar el comportamiento de `lock`. Esto se debe a que un thread de usuario (en principio) no puede ser desalojado por el scheduler de threads, ya que no hay un clock que detenga a los threads que estuvieron ejecutando por demasiado tiempo. Consecuentemente, un thread que intente adquirir un lock haciendo busy waiting se quedará loopeando para siempre y nunca obtendrá el lock, justamente porque nunca permite que otro thread (del mismo proceso) ejecute y libere el lock. Es por este motivo que es necesario adaptar el comportamiento de `lock`. Cuando un thread de usuario falla en adquirir un lock,

debe llamar a `thread_yield` para cederle la CPU a otro thread del mismo proceso. Consecuentemente, no hay (demasiado) busy waiting.

```
atomic<bool> ocupado;
void lock(){
    while(ocupado.testAndSet()){ thread_yield(); }
    }
```

Figura 5.13: Mutex para threads de usuario.

Como `thread_yield` es tan solo una llamada al scheduler de threads en user space, es muy eficiente ya que no se necesita intervención del kernel.

5.3.2. Sleep y Wakeup

Todas las soluciones que estuvimos viendo hasta ahora tienen una característica común: hacen busy waiting. Sabemos que, durante el período en el que un proceso está en su sección crítica, ningún otro proceso puede entrar a su sección crítica; si quisieran entrar, tendrían que esperar hasta que se libere la sección crítica. Durante todo este período de espera, estos procesos no realizan ninguna tarea productiva. De hecho, si tenemos múltiples procesos haciendo busy waiting, estos procesos pueden terminar monopolizando el bus de memoria, afectando negativamente a los demás procesos del sistema (incluso a aquellos que no están involucrados en la comunicación).

El problema fundamental es: ¿qué podemos hacer cuando estamos esperando por cierto evento que podría ocurrir en un intervalo de tiempo arbitrariamente largo? Si la única herramienta de sincronización que tenemos es un mutex, tan solo podemos revisar continuamente el estado del evento. Suponiendo que podemos conocer el estado del evento mirando una variable compartida *shared*, lo único que podemos hacer es quedarnos loopeando esperando por el evento. Cada vez que accedemos a la variable compartida, primero debemos obtener el lock y liberarlo al finalizar. Esto se muestra en la Fig. 5.14.

```
while(true){
    mutex.lock();
    if(shared.condition()){
        shared.critical_region();
     }
    mutex.unlock();
}
```

Figura 5.14: Esperando por evento con mutex.

Si el evento en cuestión ocurre muy de vez en cuando, esta solución desperdicia una enorme cantidad de tiempo de CPU. Por lo tanto, queda claro que necesitamos alguna otra herramienta de sincronización que nos permita, de alguna manera, sincronizar procesos sin tener que hacer (demasiado) busy waiting.

El primer gran paso en lidiar con los problemas de procesos concurrentes vino en 1965 con el trabajo de Dijkstra [Dij65]. El principio fundamental de su trabajo es el siguiente: dos o más procesos pueden cooperar mediante señales simples. Un proceso puede ser forzado a esperar en cierto lugar específico hasta que haya recibido una señal específica. Luego, necesitamos otra herramienta de sincronización que le permita a un proceso *dormir* hasta que otro proceso lo *despierte*, por medio de una señal. *(Cuando usamos el término señal, no se está haciendo referencia al mecanismo de señales de UNIX.)*

Problema de Productor-Consumidor

Para poder razonar sobre las distintas herramientas de sincronización que permiten garantizar la exclusión mutua entre proceso, Dijkstra planteó una serie de problemas de sincronización. Uno de los problemas clásicos que desarrolló fue el llamado problema del *Productor-Consumidor* (con buffer acotado).

Este es uno de los problemas más frecuentes en programas paralelos, por lo que vale la pena detenernos a analizar las soluciones propuestas.

El enunciado general del problema es el siguiente. Tenemos dos procesos que son llamados *productor* y *consumidor*. El **productor** es un proceso cíclico y, cada vez que hace un ciclo, produce una cierta porción de información. El **consumidor** también es un proceso cíclico y, cada vez que hace un ciclo, puede procesar la siguiente porción de información ya generada, siempre que haya alguna disponible.

La relación productor-consumidor implica un canal de comunicación en un solo sentido entre ambos procesos, por donde la información puede ser transmitida. Los procesos son conectados mediante un buffer (FIFO) acotado con una capacidad de hasta `MAX` elementos. El sistema debe evitar que las operaciones sobre el buffer se superpongan. Esto es, solo puede haber un agente que accediendo al buffer al mismo tiempo (hay exclusión mutua entre productor y consumidor). Además, hay que asegurarnos de que el productor no intente agregar elementos en el buffer si está lleno, y que el consumidor no intente remover elementos del buffer si está vacío.

Como mencionamos anteriormente, lo que buscamos es tener algún mecanismo basado en señales para hacer "dormir", en caso de que sea necesario, y también se debe poder "despertar" a alguno de estos procesos dormidos. Para ello, vamos a utilizar dos system calls: `sleep` y `wakeup`. La system call `sleep` suspende al proceso que la invoca, mientras que la system call `wakeup` despierta a algún proceso dormido. Para esto, la system call `wakeup` toma un único parámetro que puede ser una dirección de memoria usada para asociar cada `sleep` con cada `wakeup` o, alternativamente, el identificador del proceso que se quiere despertar.

El comportamiento que buscamos es el siguiente. Si el productor quiere agregar un item y el buffer está lleno, el productor se va a dormir. Similarmente, si el consumidor quiere remover un item del buffer y ve que el buffer está vacío, se va a dormir. Además, el productor se debe encargar de despertar al consumidor (en caso de que esté durmiendo) y el consumidor también debe encargarse de despertar al productor (en caso de que esté durmiendo).

Para mantener un registro de la cantidad de items en el buffer, necesitamos una variable compartida `nitems`. Si el número máximo de items que puede tener el buffer es `MAX`, el productor debe primero revisar si `nitems` es `MAX`. Si lo es, el productor se va a dormir; sino, el productor *agrega un item* e incrementa `nitems`. El código del consumidor es similar. Primero, revisa si `nitems` es 0. Si lo es, se va a dormir; sino, remueve un item y decrementa el contador. Cada uno de los procesos también debe revisar si el otro debe ser despertado y, en caso de que así sea, lo despierta. El código de tanto productor como consumidor está en la Fig. 5.15;

```
int nitems = 0; // Compartida
int buffer[]; // Compartida
void consumer(void *){
    int item;
    while(TRUE){
        if(nitems == 0){/*¿vacio?*/
            sleep();
        }
        item = get_item(buffer);
        if(--nitems == MAX - 1){/*¿lleno?*/
            wakeup(producer);
        }
        consume_item(item);
    }
}
```

```
void producer(void *){
    int item;
    while(TRUE){
        item = produce_item();
        if(nitems == MAX){ /*¿lleno?*/
            sleep();
        }
        insert_item(item, buffer);
        if(++nitems == 1){ /*¿vacio?*/
            wakeup(consumer);
        }
    }
}
```

Figura 5.15: Solución al Problema del Consumidor-Productor con condiciones de carrera.

Esta estrategia suena simple, pero lleva a ciertos tipos de race conditions. Supongamos que nos encontramos en la situación mostrada en la Fig. 5.16.

1. Inicialmente, el buffer está vacío y el consumidor acaba de leer **nitems** para ver si es 0.

2. En ese instante, el scheduler decide desalojar al consumidor temporalmente y empezar a ejecutar al productor.

3. El productor inserta un *item* en el buffer, incrementa **nitems** y se da cuenta que ahora vale 1. Esto quiere decir que, justo antes de insertar el buffer, éste estaba vacío. Por lo tanto, el productor supone que el consumidor estaba durmiendo y lo despierta. Desafortunadamente, el consumidor todavía no se fue a dormir, por lo que la señal se pierde.

4. Cuando le vuelva a tocar al consumidor, se fijará si el valor que había leído de **nitems** es 0. Como sí había leído un 0, se va a dormir.

5. Más tarde o temprano, el productor llenará el buffer y también se irá a dormir. Ambos procesos se quedarán dormidos por siempre. Este escenario donde uno o más procesos se quedan esperando por una condición que ya es verdadera se conoce como *The Lost Wake-Up Problem*.

La esencia del problema aquí es que la señal que envía `wakeup` solo funciona si hay algún otro esperándola del otro lado. Si pudiéramos, de alguna manera, la señal perdure en el tiempo, el esquema funcionaría correctamente.

Consumidor	Productor	**nitems**
	insert_item(item, buffer)	0
nitems == 0		0
	++nitems == 1	1
	wakeup()	1
sleep()		1

Figura 5.16: Un scheduling desfavorable.

Semáforos

Ésta era la situación en 1965, cuando Dijkstra sugirió usar una variable entera para contar la cantidad de *wakeups*, y acumularlos para un uso futuro. Un semáforo s es una variable entera que, salvo por su inicialización, es accedida únicamente a través de dos primitivas de sincronización:`wait` y `signal`. La primitiva `signal` nos permite transmitir una señal, mientras que la primitiva `wait` nos permite recibir una señal previamente transmitida. Si la señal correspondiente todavía no ha sido transmitida, el proceso es bloqueado hasta que la transmisión ocurra.

Para implementar esto, podemos ver al semáforo como una variable que tiene un valor entero sobre el cual se definen tres operaciones:

1. Un semáforo puede ser inicializado en un valor entero no negativo. Cuando un semáforo solo puede valor 0 ó 1, lo llamamos **semáforo binario**. Los semáforos binarios pueden utilizarse para garantizar la exclusión mutua, por lo que suelen utilizarse como *mutex* (bloqueante).

2. La operación de `wait` decrementa el valor del semáforo. Si el valor se vuelve negativo, entonces el proceso ejecutando el `wait` será bloqueado. Caso contrario, el proceso continuará con su ejecución.

3. La operación de `signal` incrementa el valor del semáforo. Si el valor resultante es menor o igual a cero, entonces algún proceso bloqueado por `wait` es desbloqueado. Si el scheduler es preemptive, al momento de despertar el proceso bloqueado, se revisa si su prioridad es mayor al que actualmente está ejecutando. Si lo es, se lo desaloja y se le asigna la CPU al proceso recién despertado— en el contexto de Linux, esto se conoce como **wake-up preemption**.

Más allá de estas tres operaciones, no hay ninguna forma de observar ni manipular a los semáforos. Es importante notar que todas las modificaciones sobre el valor del semáforo en las operaciones `wait`

y `signal` deben ejecutarse de forma atómica. En particular, no pueden haber dos procesos ejecutando simultáneamente un `wait` y un `signal` sobre el mismo semáforo.

En un sistema mono-procesador, el sistema operativo deshabilitan temporalmente todas las interrupciones mientras que se revisa el estado del semáforo, lo actualiza y pone a dormir al proceso (en caso de que sea necesario), para luego reactivar las interrupciones. Como todas estas acciones toman tan solo unas pocas instrucciones, no hay grandes daños al deshabilitar las interrupciones. En un sistema multiprocesador, cada semáforo debe ser protegido por un spin lock. Esta atomicidad es absolutamente esencial para resolver los problemas de sincronización y evitar race conditions.

Es importante entender que usar un spin lock para prevenir que varias CPUs accedan al mismo tiempo al semáforo es muy distinto a que el productor o el consumidor haga busy waiting sobre el buffer. La operación que se realice sobre el semáforo tardará unos pocos microsegundos, mientras que un productor o un consumidor podría tener que esperar por una cantidad arbitraria de tiempo.

```
1  struct semaphore {
2          int value;
3          struct process *queue;
4  };
5      void wait(semaphore &s)
6      {
7          s.value--;
8          if (s.value < 0) {
9              agregar este proceso a s.queue;
10             sleep();
11         }
12     }
13     void signal(semaphore &s)
14     {
15         s.value++;
16         if (s.value <= 0) {
17             remover un proceso P de s.queue;
18             wakeup(P);
19         }
20     }
```

Figura 5.17: Especificación de las Primitivas de Semáforos.

Para empezar, el semáforo es inicializado con un valor no negativo. Este valor representa al número de procesos que pueden hacer un `wait` y continuar inmediatamente con su ejecución. Cuando el valor llega a cero, el siguiente proceso en hacer un `wait` será bloqueado. Cada llamado subsiguiente a `wait` hará que el valor del semáforo sea cada vez más negativo. El valor negativo de un semáforo representa a la cantidad de procesos esperando en el semáforo a ser desbloqueados. Cada `signal` desbloquea uno de los procesos que estaban esperando cuando el valor del semáforo es negativo. Cuando tenemos múltiples procesos esperando en un semáforo, normalmente se selecciona uno cualquiera de manera arbitraria. En algunos sistemas, compatibles con el estándar POSIX 1003.1b, permiten establecer prioridades a los distintos procesos para luego poder seleccionar al de mayor prioridad.

Cuando se bloquea un proceso, se lo coloca en una *waiting queue* asociada al semáforo (y el proceso en cuestión pasa del estado *running* a *waiting*). Luego, el control es transferido al scheduler, y éste selecciona el siguiente proceso a ejecutar. Cuando se despierta un proceso, se lo saca de la *waiting queue* y se lo coloca en la *ready queue* correspondiente (y el proceso pasa de *waiting* a *ready*).

Para asociar a cada proceso con una *waiting queue* particular, es posible agregar un campo en la PCB del proceso que almacene un puntero a una *waiting queue* particular.

Volviendo al problema del Productor-Consumidor, tenemos que lograr la exclusión mutua sobre los accesos al buffer, garantizar que el productor no sobrepase la capacidad del buffer y que el consumidor no intente sacar un elemento cuando el buffer esté vacío.

- Para garantizar que los procesos no se superpongan al momento de acceder al buffer, vamos a utilizar un semáforo binario al que llamaremos `mutex`, que nos va a garantizar la exclusión mutua sobre la sección crítica. El productor realiza un `wait` antes de agregar el ítem producido al buffer y un `signal` después, y así se evita que el consumidor acceda al buffer durante la operación de agregado. De la misma manera, el consumidor realiza un `wait` antes de retirar un ítem del buffer y un `signal` después.

- Para garantizar que el consumidor no intente retirar ítems cuando el buffer esté vacío, vamos a utilizar un semáforo que represente la cantidad de elementos que actualmente están en el buffer. Cada vez que un productor quiera agregar un ítem, hacemos un `signal`. Cada vez que un consumidor quiera sacar un ítem, hacemos un `wait`. Luego, en caso de que un consumidor intente sacar un ítem cuando el buffer está vacío (`full_slots` = 0), éste será bloqueado al momento de ejecutar el `wait` hasta que se el productor haga un `signal`.

- Para garantizar que el productor no intente agregar ítems cuando el buffer está lleno, vamos a utilizar un semáforo que represente la cantidad de espacios disponibles que actualmente nos quedan en el buffer. Inicialmente, el buffer cuenta con `MAX empty_slots`, porque todavía no hay ningún elemento en el buffer. Cada vez que un productor quiere agregar un ítem, hacemos un `wait`. Cada vez que un consumidor quiere sacar un ítem, hacemos un `signal`. Luego, en caso de que un productor intente agregar un ítem cuando el buffer esté lleno (`empty_slots` = 0), éste será bloqueado al momento de ejecutar el `wait`, hasta que el consumidor haga un `signal`.

```
1  semaphore mutex = 1; /* Controla el acceso a la seccion critica */
2  semaphore full_slots = 0; /* Cuenta la cantidad de slots ocupados*/
3  semaphore empty_slots = MAX; /* Cuenta la cantidad de slots vacios*/
4
5  void producer(void *)
6  {
7      int item;
8      while(TRUE){
9          item = produce_item();
10         wait(empty_slots); /* Espero a que el buffer no este lleno */
11         wait(mutex)
12         insert_item(item, buffer);
13         nitems++;
14         signal(mutex);
15         signal(full_slots);
16     }
17
18 }
19 void consumer(void *)
20 {
21
22     wait(full_slots); //* Espero a que el buffer no este vacio */
23     wait(mutex)
24     item = get_item(buffer);
25     signal(mutex);
26     signal(empty_slots);
27     consume_item(item);
28 }
```

Figura 5.18: Productor-Consumidor con semáforos.

En este ejemplo, hemos usado semáforos en dos sentidos distintos. El semáforo `mutex` es usado para garantizar la exclusión mutua, es decir, para garantizar que tan solo un proceso al mismo tiempo ingrese a la sección crítica y no es específico al problema. Siempre que tengamos una sección crítica, vamos a tener que controlar el acceso a ésta. El segundo uso de semáforos es para la sincronización específica al problema. Los semáforos `full_slots` y `empty_slots` son necesarios para garantizar que ciertos eventos

ocurran en cierto orden específico. En particular, nos permiten asegurar que un productor se detenga cuando el buffer está lleno y que el consumidor se detenga cuando está vacío.

Spinning vs Blocking

En general, cualquier protocolo de exclusión mutua nos presenta la siguiente pregunta: ¿qué hacemos cuando no podemos obtener el lock? Hay dos alternativas. Si nos quedamos intentando, el lock se conoce como *spin lock*, y verificar continuamente el estado del lock se le llama hacer *busy waiting* (o *spinning*). La alternativa es bloquear al proceso y pedirle al sistema operativo que seleccione otro proceso para ejecutar, que a veces se llama hacer *blocking*.

Evidentemente, hacer busy waiting solo tiene sentido en sistemas multi-procesador. Esto se debe a que no hay ninguna otra CPU que pueda liberar el lock. Si un thread intenta obtener un lock y falla, se quedará bloqueado hasta que se le acabe su quantum, y durante todo ese tiempo no le dará la oportunidad al thread poseedor del lock de liberarlo. Recordemos que en los sistemas operativos modernos los quantums oscilan entre 10 y 100 milisegundos, mientras que el tiempo necesario para un cambio de contexto suele ser inferior a 10 microsegundos; por tanto, queda claro que hacer busy waiting hasta el final del quantum no es una buena idea.

Hacer busy waiting gasta ciclos de CPU directamente. Hacer blocking también gasta ciclos de CPU, ya que el estado actual del proceso debe ser guardado, se debe obtener el lock que protege a la lista de procesos ready, se debe seleccionar el próximo proceso a ejecutar, se debe cargar su estado y se debe poner a ejecutar. Además, la cache de la CPU tendrá que ser repoblada, por lo que muchos cache misses costosos ocurrirán a medida que el proceso ejecute. Eventualmente, se debe traer devuelta al proceso original, teniendo que pagar nuevamente todos los costos de un cambio de contexto. Estos ciclos de CPU gastados en hacer los dos cambios de contexto más todos los cache misses son desperdiciados.

Considerando lo anteriormente mencionado, hacer busy waiting solo tiene sentido si esperamos que el lock va a estar disponible en un corto período de tiempo. Como regla general, vamos a decir que nos conviene usar spin locks cuando el tiempo que se ocupa en la sección crítica es mayor a dos cambios de contexto.

Futex Locks

Con el aumento del paralelismo, poder sincronizar y lockear de manera eficiente es muy importante para la performance. Como dijimos anteriormente, los spin locks son rápidos si la espera es corta (menor a dos cambios de contexto), pero gastan ciclos de CPU de manera innecesaria si la espera es larga. Si hay mucha contención, los tiempos de espera se vuelven más largos, por lo que es esperable que resulte más eficiente hacer blocking. Desafortunadamente, hacer blocking tiene el problema opuesto: funciona bien bajo alta contención, pero continuamente cambiar estar realizando cambios de contexto es demasiado costoso si tenemos poca contención. Además, predecir el tiempo de espera no es una tarea sencilla.

Una solución interesante que intenta combinar ambas técnicas es conocida como **futex** (*fast user space mutex*). Un futex es una característica de Linux que implementa un locking básico (similar a un mutex) pero que evitar llamar al kernel salvo que sea necesario. En la mayoría de programas que utilizan mutex, la contención suele ser baja. Por lo tanto, realizar un cambio de contexto suele ser costoso, y es mejor evitar llamar al kernel cuando no es necesario.

Un futex consiste de dos partes: un servicio del kernel y una biblioteca de usuario. El servicio del kernel provee una *waiting queue* donde se almacenan los procesos bloqueados en el lock. Estos procesos no participarán del algoritmo de selección del scheduler, salvo que el kernel explícitamente los desbloquee. Para que un proceso sea colocado en una de estas *waiting queues*, es necesario realizar una costosa system call— que queremos evitar siempre que sea posible. En ausencia de contención, un futex funciona completamente en user space.

Supongamos que el lock inicialmente está libre (lock = 0). Luego, un thread adquiere el lock al realizar un TAS atómico, para saber si el lock estaba libre u ocupado. Si estaba libre, la llamada fue exitosa y el thread obtuvo el lock. Sin embargo, en caso de que el lock ya estaba ocupado, nuestro thread debe esperar. En ese caso, la biblioteca de futex no hace spinning, sino que utiliza una system call para bloquear el proceso.

También es posible hacer busy waiting durante un período corto de tiempo (menor a 2 cambios de contexto) y, pasado este umbral, pasar a hacer blocking. Podemos mejorar esta solución al definir un umbral dinámico, que dependa del historial observado de los mutex en los que espera cada proceso. Los mejores resultados se obtienen cuando el sistema mantiene un registro de los últimos tiempos de busy waiting observados y asume que el siguiente va a ser similar a los anteriores. De esta manera, si el sistema ve que el proceso ha hecho busy waiting por más de 2 cambios de contexto en los últimos intentos, entonces, la próxima vez que falle en obtener el lock directamente, se hace blocking directamente.

Mutex Reentrante

Hay situaciones donde resulta conveniente permitir que un proceso pueda llamar a `lock` múltiples veces, antes de liberar el mutex. Claramente, si un proceso llama dos veces a `lock` en cualquiera de las implementaciones que vimos hasta ahora, el proceso caerá en deadlock. Cuando un mutex permite que proceso pueda llamar repetidas veces a `lock`, sin caer en deadlock, se lo conoce como **mutex reentrante** (o recursivo). El escenario típico donde usaríamos un lock recursivo es cuando una función que utiliza un mutex llama recursivamente a otra que también utiliza el mismo mutex. En general, si se llamó n veces a `lock`, para liberar un mutex reentrante es necesario llamar n veces a `unlock`. En la Fig. 5.19 y en la Fig. 5.20 se muestran posibles implementaciones de mutex reentrante (o recursivo) que hacen busy waiting (uno spinning y otro local spinning).

```
1  #define NONE -1          /* pid > 0 */
2  #define ERROR -1
3
4  int calls;
5  atomic<int> owner;       /* lock */
6
7  void create() {
8      owner.set(NONE);
9      calls = 0;
10 }
11 void lock() {
12     pid_t pid = getpid();
13     while(owner.get() != pid) {
14         while (owner.compareAndSwap(NONE, pid) != pid) {/*spinning*/}
15     }
16     calls++;
17 }
18 void unlock () {
19     pid_t pid = getpid();
20     if(calls == 0 || owner.get() != pid){
21         exit(ERROR);
22     }
23     if(--calls == 0){owner.set(NONE);}
24 }
```

Figura 5.19: (spinning) mutex reentrante.

```
1  void lock() {
2      pid_t pid = getpid();
3      while(owner.get() != pid){
4          while(owner.get() != NONE){/*local spinning*/}
5          owner.compareAndSwap(NONE, pid);
6      }
7      calls++;
8  }
```

Figura 5.20: (local spinning) mutex reentrante.

Mutexes en Pthreads

La biblioteca Pthreads provee herramientas para la sincronización de threads. El mecanismo básico utiliza una variable **mutex**, que puede estar libre u ocupada, para resguardar cada sección crítica. Un thread que desea entrar a la sección crítica primero intenta acceder obtener el lock asociado al mutex. Las llamadas más importantes relacionadas a mutex se muestran en la Fig. 5.21. Como es esperable, los mutex pueden crearse o destruirse. Para intentar obtener el lock, tenemos dos opciones. `lock` permite obtener el lock y, en caso de fallar, bloquea al proceso. La segunda opción que tenemos es `trylock`, que también permite obtener el lock. La diferencia está en que no bloquea en caso de falla, sino que devuelve un código de error. Esto permite hacer busy waiting si es necesario (pero no permite hacer *local spinning*). Finalmente, podemos desbloquear el mutex y, si hay al menos un proceso esperando por el mutex, despertar a exactamente un proceso.

```
1  /* Crea un mutex */
2  int pthread_mutex_init(pthread_mutex_t *mutex,
3                         pthread_mutexattr_t *attr);
4
5  /* Destruye un mutex existente. */
6  int pthread_mutex_destroy(pthread_mutex_t *mutex);
7
8  /* Obtiene un mutex o bloquea. */
9  int pthread_mutex_lock(pthread_mutex_t *mutex);
10
11 /* Obtiene un mutex o falla. */
12 int pthread_mutex_trylock(pthread_mutex_t *mutex);
13
14 /* Libera el mutex. */
15 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Figura 5.21: Algunas de las llamadas de Pthreads relacionadas con mutexes.

Además de los mutexes, Pthreads ofrece un segundo mecanismo de sincronización: las **variables de condición**. Los mutexes son útiles para permitir o bloquear el acceso a una sección crítica. Las variables de condición nos sirven para cuando queremos que los procesos sean bloqueados si no se cumple cierta condición.

Como ejemplo, consideremos nuevamente el escenario del productor-consumidor. Un proceso pone items en un buffer compartido y otro proceso consume los items. Si el productor descubre que no hay más espacio disponible en el buffer, tiene que bloquearse hasta que se haya liberado espacio. Los mutexes permiten revisar el estado del buffer atómicamente, pero no permiten que otro proceso le avise que cambió el estado del buffer. Las variables de condición permiten que otro proceso, el consumidor, le avise al productor que ahora sí hay espacio disponible.

Las llamadas más importantes relacionadas con las variables de condición se muestran en la Fig. 5.22. Podemos destruir y crear variables de condición. Las operaciones principales sobre las variables de condición son `wait` y `signal`. El primero bloquea al proceso invocador hasta que algún otro proceso haga un `signal` sobre la misma variable de condición.

El proceso bloqueado suele estar esperando a que otro proceso realice cierto trabajo, libere un recurso o realice alguna otra actividad. Solo cuando se cumpla dicha condición, el proceso bloqueado va a tener permitido continuar. Las variables de condición permiten esta espera bloqueante se realice de manera atómica. La llamada `broadcast` es usada cuando hay múltiples procesos esperando por la misma condición y que, cuando ésta se satisfaga, queremos despertar a todos los procesos bloqueados.

Las variables de condición y los mutexes son usados en conjunto. El patrón típico es que un proceso obtenga un lock asociado a un mutex, para luego esperar sobre cierta variable de condición. Eventualmente, otro proceso hará un `signal` sobre la variable y el proceso bloqueado podrá continuar. Notemos que, para que este patrón funcione correctamente, hacer un `wait` sobre una variable de condición debe desbloquea atómicamente el lock asociado al mutex de la sección crítica. Por ese motivo, esta llamada

```c
/* Crea una variable de condición.*/
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t attr);

/* Destruye una variable de condición existente. */
int pthread_cond_destroy(pthread_cond_t *cond);

/* Despierta otro proceso. */
int pthread_cond_signal(pthread_cond_t *cond);

/* Bloquea proceso hasta recibir signal. */
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);

/* Despierta a todos los procesos. */
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Figura 5.22: Algunas de las llamadas de Pthreads relacionadas con variables de condición.

debe tomar como parámetro el mutex de la sección crítica— si se llama a `wait` sin estar en posesión del mutex pasado por parámetro, el comportamiento de esta llamada no está definido.

También es importante notar que las variables de condición (a diferencia de los semáforos) no tienen memoria. Si se envía un signal a una variable de condición por la que ningún thread está esperando, la señal se pierde. Por este motivo, hay que tener cuidado al momento de usar variables de condición.

5.3.3. Monitores

Si bien los semáforos y mutexes son herramientas que permiten la sincronización eficiente de procesos, usarlas de forma incorrecta puede resultar en graves errores difíciles de detectar. Una estrategia para lidiar con este tipo de errores consiste en incorporar herramientas de sincronización de más alto nivel que sean más fáciles de utilizar. Una de estas herramientas son los **monitores**.

Un monitor es un nuevo tipo de variable, cuyas instancias constan de una colección de procedimientos, variables y estructuras de datos agrupadas en un módulo o paquete especial (por ejemplo, una clase). Los procesos pueden llamar a los procedimientos de un monitor cuando lo deseen, sin tener que hacer locking ni blocking explícitamente. Sin embargo, no tienen permitido acceder directamente a las estructuras internas de un monitor desde procedimientos declarados por fuera del mismo. La sintaxis del tipo monitor se muestra en la Fig. 5.23.

Los monitores tienen una propiedad importante que los hacen útiles para lograr la exclusión mutua: solo un proceso puede estar activo dentro del monitor al mismo tiempo. Es decir, todos los procedimientos en un monitor conforman una gran sección crítica, de manera tal que ningún par de procesos podrán acceder al mismo tiempo a esta sección crítica.

Los monitores son construcciones a nivel de lenguaje de programación, por lo que el compilador sabe que son especiales y puede manejar las llamadas a procedimientos de monitor de manera diferente al resto de procedimientos. El compilador es quien se encarga de implementar la exclusión mutua sobre la entrada a un monitor, y típicamente utiliza un mutex o un semáforo binario. Como el compilador se encarga de garantizar la exclusión mutua, es mucho menos probable que algo salga mal.

Típicamente, el compilador se encarga de agregar instrucciones al inicio de cada procedimiento. De esta manera, cuando un proceso llama a un procedimiento de un monitor, las primeras instrucciones del procedimiento se fijarán si hay algún otro proceso actualmente activo dentro del monitor. Si es así, el proceso invocador será bloqueado hasta que el otro proceso haya salido del monitor. Caso contrario, puede acceder al monitor.

Claramente, la definición que dimos hasta ahora del tipo monitor no es una herramienta de sincroni-

```
monitor m
{
    int count; /* Declaracion de variables compartidas */
    function Proc1(...) {

    }
    function Proc2(...) {

    }
    // ...
    function Procn(...) {

    }
    initialize(...){
        count = 0;
        ...
    }
}
```

Figura 5.23: Un monitor.

zación completa. Tan solo podemos garantizar la exclusión mutua, pero no tenemos forma de definir un esquema de sincronización particular a un problema. Pensando en el problema del productor-consumidor, es relativamente sencillo mover todas las verificaciones sobre el estado del buffer a los procedimientos de un monitor. Sin embargo, ¿cómo puede un productor bloquearse cuando se encuentra con que el buffer está lleno? Necesitamos definir mecanismos de sincronización adicionales: las **variables de condición** (o condition variables).

La idea es introducir el nuevo tipo *condition*, junto con dos operaciones: `wait` y `signal`. Cuando un proceso ejecutando un procedimiento de un monitor descubre que no puede continuar— por ejemplo, si el buffer está lleno—, debe hacer un `wait` sobre una variable de condición, digamos, *full*. Esta acción hace que el proceso invocador sea bloqueado hasta que algún otro proceso invoque a `signal` sobre la misma variable de condición— que reanuda a exactamente un proceso bloqueado. Cuando un proceso es bloqueado dentro del monitor, se permite que otro proceso acceda al monitor sin ser bloqueado.

Ahora, supongamos que un proceso P llama a `signal`, y existe un proceso bloqueado Q asociado a la misma variable de condición que debería ser despertado. Para evitar tener dos procesos activos dentro del monitor, necesitamos una regla para saber qué hacer luego de un signal. Para este problema, se han planteado varias posibilidades.

Una primer opción, propuesta por Hoare, conocida como **signal and wait**, consiste en permitir que el proceso Q recién despertado ejecute, suspendiendo a P hasta que Q salga del monitor. Una segunda opción, propuesta por Hansen, consisten en poner como restricción que solo se pueda llamar a `signal` como última instrucción dentro de un procedimiento de monitor. De esta manera, se evita tener que decidir qué hacer con P y Q, ya que P sale del monitor inmediatamente después de hacer el `signal`. Por último, tenemos la regla conocida como **signal and continue**, donde P continúa ejecutando el procedimiento hasta terminar y salir del monitor, para que recién en ese momento Q sea reanudado.

En general, la propuesta de Hansen es conceptualmente más sencilla y más fácil de implementar, por lo que vamos a considerar que siempre que tan solo se puede hacer un `signal` como última instrucción de un procedimiento de monitor.

Notemos que las variables de condición no son contadores ni acumulan señales para un uso posterior —a diferencia de los semáforos. Luego, si se hace un `signal` sobre una variable de condición y, en ese instante, no hay ningún proceso esperando por la señal, la señal se pierde. En ese sentido, pareciera que las operaciones de `wait` y `signal` sobre variables de condición son muy similares a las system calls `sleep` y `wakeup`. Por lo tanto, podríamos pensar que vamos a tener las mismas race conditions que teníamos con estas system calls. Sin embargo, hay una diferencia crucial: `sleep` y `wakeup` fallaban porque un proceso podía irse a dormir mientras que, al mismo tiempo, otro proceso intentaba despertarlo. Sin embargo,

esta situación no puede darse usando monitores, ya que las variables de condición se operan dentro de procedimientos del monitor, dentro de los cuales está asegurada la exclusión mutua.

En la Fig. 5.24 y en la Fig. 5.25 se muestra una solución al problema del Productor-Consumidor usando monitores.

```
monitor buffer()
{
    condition full, empty; /* compartidas */
    int nitems;            /* compartidas */
    int buffer[];          /* compartidas */

    void initialize(){
        nitems = 0;
    }
    void insert(int item){
        if( nitems == MAX ){
            wait(full);
        }
        insert_item(buffer, item);
        if ( ++nitems == 1 ){signal(empty);}
    }
    void remove(int &item){
        if( nitems == 0){
            wait(empty);
        }
        item = get_item(buffer);
        if( --nitems == MAX - 1 ){signal(full);}
    }
}
```

Figura 5.24: Solución al Problema del Consumidor-Productor usando monitores. (Monitor)

```
void producer(void *){
    int item;
    while(TRUE){
        item = produce_item();
        buffer.insert(item);
    }
}
void consumer(void *){
    int item;
    while(TRUE){
        buffer.remove(item);
        consume_item(item);
    }
}
```

Figura 5.25: Solución al Problema del Consumidor-Productor usando monitores. (Código para consumidor y productor)

5.4. Liveness

Una consecuencia de usar herramientas de sincronización para coordinar el acceso a secciones críticas es la posibilidad de que un proceso intentando entrar a su sección crítica espere para siempre. Las propiedades de **liveness** hacen referencia al conjunto de propiedades que un sistema debe satisfacer para asegurar que los procesos hagan progreso en su ejecución. Un proceso que se queda esperando de manera indefinida bajo estas circunstancias es un ejemplo de *liveness failure*. Los esfuerzos en proveer

exclusión mutua usando herramientas como los spin locks y semáforos pueden llevar a este tipo de fallas en la programación paralela. En esta sección, vamos a explorar algunas situaciones que pueden llevar a liveness failures.

Deadlock

Al utilizar semáforos que pueden suspender la ejecución de procesos, es posible que terminemos en una situación donde dos o más procesos se quedan esperando indefinidamente por un evento que puede ser causado solo por uno de los procesos que están esperando. Cuando se llega a un estado como éste, estos procesos se dicen que están en deadlock. Decimos que un conjunto de procesos están en **deadlock** cuando *cada proceso en el conjunto está esperando por un evento que solo puede ser causado por otro proceso en el conjunto.*

Para mostrar esto, consideremos un sistema que consiste de dos procesos, P_0 y P_1, cada uno accediendo a dos semáforos, S y Q, que inicialmente valen 1:

P_0	P_1
`wait`(S)	`wait`(Q)
`wait`(Q)	`wait`(S)
.	.
.	.
.	.
`signal`(S)	`signal`(Q)
`signal`(Q)	`signal`(S)

Supongamos que P_0 ejecuta `wait`(S) y luego P_1 ejecuta `wait`(Q). Cuando P_0 ejecuta `wait`(Q), debe esperar a que P_1 ejecute `signal`(Q). Similarmente, cuando P_1 ejecuta `wait`(S), debe esperar a que P_0 ejecute `signal`(S). Como estas operaciones de `signal` no pueden ser ejecutadas, P_0 y P_1 están en deadlock.

Coffman, en su trabajo de 1971 [CES71], postula una serie de condiciones necesarias para la existencia de deadlocks. Se nos dice que para que se pueda dar una situación de deadlock se deben cumplir las siguientes cuatro condiciones al mismo tiempo:

1. **Mutual exclusion**. Un recurso tiene que ser de asignación exclusiva, es decir, un solo proceso puede usar el recurso al mismo tiempo. Si algún otro proceso pide por este recurso, el proceso debe ser retrasado hasta que el recurso sea liberado.

2. **Hold and Wait**. Un proceso debe estar en posesión de al menos un recurso y debe estar esperando para obtener recursos adicionales que actualmente están en posesión de otros procesos.

3. **No preemption**. Los recursos no pueden ser desalojados, esto es, un recurso solo puede ser liberado voluntariamente por el proceso que lo tiene.

4. **Circular wait**. Debe existir un conjunto de procesos tal que cada uno está esperando por un recurso que está en posesión del siguiente, formando un ciclo de espera circular.

Los deadlocks se pueden describir de manera más precisa en términos de un grafo dirigido llamado **system resource-allocation graph**. Este grafo consiste de un conjunto de vértices V y un conjunto de aristas E. El conjunto de vértices V se particiona en dos tipos diferentes de nodos: $P = \{P_1, P_2, \ldots, P_n\}$, el conjunto formado por todos los procesos activos en el sistema, y $R = \{R_1, R_2, \ldots, R_m\}$, el conjunto formado por todos los tipos de recursos en el sistema.

Un arco que va del proceso P_i al tipo de recurso R_j significa que el proceso P_i ha pedido por una instancia del tipo de recurso R_j y está actualmente esperando por ese recurso. Un arco que va del tipo

de recurso R_j al proceso P_i significa que una instancia del tipo de recurso R_j ha sido reservado para P_i.

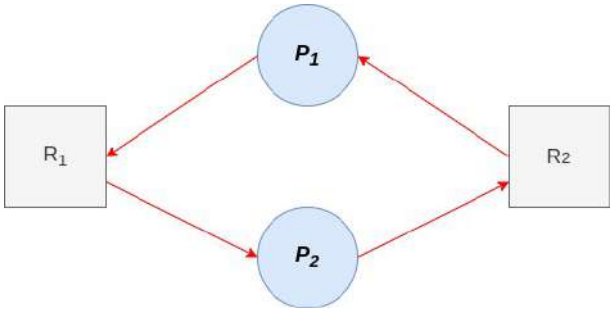

Figura 5.26: Resource allocation graph en Deadlock.

Dada la definición de este grafo, se puede demostrar que, si el grafo no contiene ningún ciclo, entonces ningún proceso en el sistema está en deadlock. Si el grafo contiene algún ciclo, entonces es posible que haya algún deadlock:

- Si el ciclo involucra un conjunto de tipos de recursos, donde cada uno tiene una sola instancia, entonces ha ocurrido un deadlock. Cada proceso involucrado en el ciclo está en deadlock. En este caso, un ciclo en el grafo es tanto condición necesaria como suficiente para la existencia de deadlocks.
- Si cada tipo de recurso tiene varias instancias, entonces el ciclo no necesariamente implica que ha ocurrido un deadlock. En este caso, un ciclo en el grafo es condición necesaria pero no suficiente para la existencia de deadlock.

Los deadlocks pueden ser evitados al asegurar que una de las cuatro condiciones necesarias para la existencia de deadlock no ocurra. De las cuatro condiciones, eliminar la espera circular es la única que se puede realizar en la práctica. Para ello, el sistema operativo utiliza algoritmos para evitar otorgar recursos que pueden llevar a un estado inseguro, donde es posible caer en deadlock. También, hay algoritmos de detección de deadlocks para evaluar procesos y recursos en un sistema, y determinar si un conjunto de procesos están en deadlock. En caso de que haya ocurrido un deadlock, un sistema puede intentar recuperarse al terminar uno de los procesos en la cola circular o desalojar recursos que hayan sido asignados a un proceso que entró en deadlock.

Livelock

Otro tipo de liveness failure son los **livelocks**. Son similares a los deadlocks; ambos impiden que dos o más procesos progresen, pero por distintos motivos. Los livelocks ocurren cuando un proceso intenta efectuar continuamente una acción que fracasa. Los procesos no están suspendidos, pero no están haciendo ningún progreso en su ejecución.

Típicamente, sucede cuando hay varios procesos reintentando operaciones fallidas de manera simultánea. Luego, generalmente puede ser evitado a partir de tener a cada proceso reintentar la operación fallida en tiempos aleatorios. Un ejemplo de esto son las colisiones de paquetes en las redes Ethernet. En lugar de intentar retransmitir un paquete inmediatamente después de que haya ocurrido una colisión, se hace un *exponential backoff* por un período aleatorio de tiempo antes de intentar retransmitir.

Starvation

Otra de las liveness failures más conocidas es la **inanición** (o starvation) de procesos. Ocurre cuando un proceso particular es bloqueado de manera indefinida. Por ejemplo, puede suceder cuando cierto proceso tiene una baja prioridad fija y el sistema está cargado con muchos procesos de alta prioridad, evitando que el proceso de baja prioridad consiga ser seleccionado por el scheduler para ejecutar.

Inversión de Prioridades

Otro tipo de liveness failure es el problema de inversión de prioridades. Anteriormente, vimos cómo este tipo de problemas pueden aparecer cuando usamos spin locks. Sin embargo, también pueden ocurrir cuando usamos semáforos o futex locks.

Un desafío de scheduling surge cuando un proceso de alta prioridad necesita leer o modificar datos del kernel que actualmente están siendo accedidos por un proceso de menor prioridad— o una cadena de procesos de menor prioridad. Como los datos del kernel típicamente están protegidos, el proceso de alta prioridad debe esperar a que el proceso de menor prioridad termine lo que está haciendo. La situación se vuelve aún más complicada si el proceso de menor prioridad es desalojado en favor de otro proceso con una prioridad intermedia.

En forma de ejemplo, asumamos que tenemos tres procesos— L, M, H— cuyas prioridades siguen el orden $L < M < H$. Asumamos que el proceso H necesita un semáforo S que actualmente está siendo accedido por el proceso L. Normalmente, el proceso H esperaría a que L termine de usar el recurso S. Sin embargo, supongamos que el proceso M pasa a estar disponible para ejecutar, por lo que termina desalojando al proceso L. Indirectamente, un proceso con una menor prioridad— el proceso M— termina afectando al tiempo que tiene que esperar el proceso H para que L libere el recurso S.

Este problema de liveness se conoce como **inversión de prioridades**, y solo puede ocurrir en sistemas con más de dos prioridades. Típicamente, la inversión de prioridades se evita implementando un protocolo de herencia de prioridades. De acuerdo con este protocolo, todos los procesos que están accediendo a recursos que necesitan procesos de mayor prioridad heredan su prioridad. Una vez hayan hecho lo que tenían que hacer con el recurso en cuestión, sus prioridades vuelven a sus valores originales.

La inversión de prioridades puede ser más que una inconveniencia de scheduling. En sistemas con restricciones de tiempo ajustadas— como los sistemas de tiempo real— la inversión de prioridades puede causar que un proceso tarde más de lo que debería en completar una tarea. Cuando esto ocurre, otras fallas pueden suceder en cascada, resultando en una falla total del sistema.

Consideremos caso del el Sojourner rover (un robot para realizar experimentos en Marte). Poco después de que el Sojourner comenzara a operar, empezó a experimentar frecuentes reseteos del sistema. Cada reseteo reinicializaba todo el hardware y software, incluyendo las comunicaciones. El problema era una inversión de prioridades, donde una tarea de alta prioridad estaba tardando más de lo esperado para completar su trabajo. La tarea estaba siendo forzada a esperar por un recurso compartido que estaba en posesión de una tarea de menor prioridad que, a su vez, estaba siendo desalojada por múltiples tareas de prioridad intermedia. Eventualmente, otra tarea se daría cuenta del problema y forzaría un reinicio de la máquina.

5.5. Razonamiento y problemas clásicos

Vamos a analizar algunos problemas clásicos del mundo paralelo. Por un lado, estos son problemas que aparecen una y otra vez, de varias formas distintas, cuando estamos desarrollando algún algoritmo paralelo. Además, nos van a servir de excusa para entender cómo se razona cuando tenemos un sistema paralelo. Conocer los problemas clásicos y sus soluciones no debe ser excusa para no pensar en las particularidades y el contexto de cada problema a resolver.

La pregunta que nos surge es, ¿cómo podemos garantizar que el programa funciona correctamente y cumple con lo pedido? En materias anteriores, teníamos los conceptos de *pre-condición* y *post-condición* que nos ayudaban a formalizar el correcto funcionamiento de nuestros programas. Decíamos que un programa era correcto si terminaba en una cantidad finita de pasos y cumplía con la post-condición, partiendo desde un punto en el que se cumplía la pre-condición.

La correctitud de programas paralelos, por su propia naturaleza, es más compleja que la de su contraparte en programas secuenciales, y su análisis requiere de un conjunto diferente de herramientas— incluso para el propósito de un razonamiento informal. El problema es que las herramientas que teníamos para pensar sobre la correctitud de programas secuenciales se nos quedaron chicas: en los programas paralelos no tenemos una única ejecución posible, sino que muchas.

Necesitamos una nueva herramienta conceptual y una nueva definición de correctitud para programas paralelos. Con este propósito, vamos introducir un modelo formal para la computación asíncrona, el *modelo de autómata de E/S*. Este es un modelo general que nos va a servir para describir casi cualquier tipo de sistema asíncrono concurrente. Nos provee una forma precisa para describir y razonar acerca de las distintas componentes del sistema que interactúan entre sí y que operan a velocidades relativas arbitrarias. Además, las acciones pueden realizarse al mismo tiempo, en un orden impredecible.

5.5.1. Modelo de Sistema Asíncrono

Un autómata de E/S modela una componente del sistema distribuido que puede interactuar con otras componentes del sistema. Es un tipo simple de máquina de estados donde las transiciones están asociadas con *acciones*. Las acciones son clasificadas como de input, output o interna. Las acciones de input y output son usadas para la comunicación con el ambiente del autómata, mientras que las acciones internas solo son visibles para el autómata. Las acciones de input se asumen que no están bajo el control del autómata— solo nos llegan del exterior— mientras que el autómata especifica qué acciones de output e internas deberían realizarse.

Un ejemplo de un típico autómata de E/S es un proceso en un sistema asíncrono distribuido. La interfaz de un autómata de proceso típico con su ambiente es ilustrado en la Fig. 5.27. El autómata P_i es dibujado como un círculo, donde las flechas entrantes etiquetadas representan acciones de input y las flechas salientes etiquetadas representan acciones de output. Las acciones internas no son mostradas. El autómata representado recibe inputs en forma de $init(v)_i$ del mundo exterior, buscando representar la llegada de una entrada con valor v. El autómata transmite outputs en la forma $decide(v)_i$, buscando representar una decisión sobre v. Para poder llegar a una decisión, el proceso P_i podría querer comunicarse con alguno de los otros procesos usando un mensaje. La interfaz para el mensaje consiste de acciones de output en la forma de $send(m)_{i,j}$, que representa al proceso P_i enviando un mensaje con contenido m al proceso P_j, y las acciones de input de la forma $receive(m)_{j,i}$ que representa al proceso P_i recibiendo un mensaje con el contenido m enviado por el proceso P_j. Cuando el autómata realiza cualquiera de las acciones indicadas (o cualquier acción interna), también podría cambiar de estado.

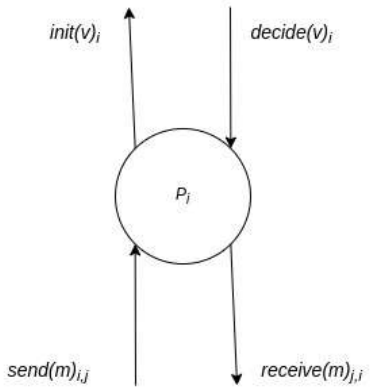

Figura 5.27: Un autómata de E/S de proceso.

Formalmente, lo primero que se especifica para un autómata de E/S es su *signature*, que es simplemente la descripción de su acciones de input, output e internas. Vamos a asumir que tenemos un conjunto universal de acciones. Una signature S es una tripla consistente de tres conjuntos disjuntos de acciones: el conjunto de acciones de input $in(S)$, el conjunto de acciones de output $out(S)$ y el conjunto de acciones internas $int(S)$. Definimos a las acciones externas $ext(S)$ como $in(S) \cup out(S)$, a las acciones localmente controladas $local(S)$ como $out(S) \cup int(S)$, y a todas las acciones de S como $acts(S)$. La *external signature* (también llamada interfaz externa) $extsig(S)$ se define como la signature $(in(S), out(S), \emptyset)$.

Un autómata A consiste de cinco componentes:

- $sig(A)$ una signature.
- $states(A)$ un conjunto de estados (no necesariamente finito).

- $start(A)$ un subconjunto no vacío de $states(A)$ conocido como los estados iniciales.
- $trans(A)$ una relación estado-transición, donde $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$; esta relación debe tener la propiedad de que para cada estado s y para cada acción de input π, hay una transición $(s, \pi, s') \in trans(A)$.
- $tasks(A)$ una partición de tareas que es una relación de equivalencia sobre $local(sig(A))$, teniendo a lo sumo una cantidad contable de clases de equivalencia.

Vamos a usar $acts(A)$ como una abreviación de $acts(sig(A))$ y, similarmente, $in(A)$, etc.

Llamamos a un elemento (s, π, s') de $trans(A)$ una *transición* o paso de A. La transición (s, π, s') es llamada transición de input, de output, etc., dependiendo de si la acción π es una acción de input, de output, etc.

Si para un estado s y una acción π particular A tiene alguna transición de la forma (s, π, s'), decimos que π está **disponible** en s. Vamos a asumir que cada acción de input está disponible en cada estado. Esta asunción significa que el autómata no es capaz de bloquear a las acciones de input en ningún momento. Decimos que el estado s es *quiescente*[2] si tan solo están disponibles las acciones de input en el estado s.

La partición de tareas $tasks(A)$ debería ser pensada como una descripción abstracta de tareas dentro del autómata. Esta partición es usada para definir condiciones de fairness en la ejecución del autómata—condiciones que nos dicen que el autómata debe continuar, durante su ejecución, dando turnos de manera justa a sus tareas. Esto es útil para modelar un componente del sistema que realiza más de una tarea. Nos vamos a referir a las clases de partición de tareas simplemente como tareas.

En general, la relación de transición es descrita en un estilo de precondición-efecto. La idea es agrupar todas las transiciones que involucran un tipo particular de acción en una sola pieza de código. Este código especifica las condiciones bajo las cuales la acción está disponible, como un predicado en el pre-estado s (**precondición**). Luego describe los cambios a ocurrir como resultado de la acción, en forma de un programa simple que es aplicado a s para producir s' (**efecto**).

Ahora vamos a describir cómo un autómata A ejecuta. Un fragmento de ejecución de A es una secuencia $s_0, \pi_1, s_1, \pi_2, \ldots$ de alternancia entre estados y acciones de A tal que $(s_i, \pi_{k+1}, s_{k+1})$ es una transición de A para cada $i \geq 0$. Notemos que si la secuencia es finita, debe terminar con un estado (no puede terminar con una acción). Un fragmento de ejecución que comienza con un estado inicial es llamado una **ejecución**. Notamos al conjunto de ejecuciones de A como $execs(A)$. Un estado se dice alcanzable en A si es el estado final de una ejecución finita de A.

Correctitud. Vamos a decir que un programa paralelo es *correcto* si para cualquier ejecución posible del programa cumplen con una serie de condiciones de sincronización. Estas condiciones varían dependiendo de las necesidades de cada problema, y habrá que definirlas de manera formal y precisa.

Algunas veces nos interesa observar solo el comportamiento externo del autómata. Luego, el *trace* de una ejecución α de A, denotado por $trace(\alpha)$, es la subsecuencia de α que consiste de todas las acciones externas. Decimos que β es un *trace* de A si β es un *trace* de alguna ejecución de A. Notamos al conjunto de traces de A como $traces(A)$.

5.5.2. Fairness

En los sistemas paralelos, usualmente solo nos interesan aquellas ejecuciones donde todos los componentes obtengan turnos de manera justa para realizar su trabajo. Recordemos que cada autómata viene equipado con una partición de sus acciones controladas; cada clase de equivalencia en la partición representa alguna tarea que el autómata debe realizar. Nuestra noción de **fairness** es que cada tarea obtenga infinitas oportunidades para realizar una de sus acciones.

Formalmente, un fragmento de ejecución α de un autómata A se dice *fair* si las siguientes condiciones

[2] Que está quieto, pudiendo tener movimiento propio.

valen para cada clase C de $tasks(A)$:

1. Si α es finito, entonces C no está disponible en el estado final de α.

2. Si α es infinito, entonces α contiene o bien infinitos eventos de C o bien infinitas ocurrencias de estados en donde C no está disponible.

Usamos el término *evento* para denotar la ocurrencia de una acción en una secuencia, por ejemplo, una ejecución o un trace.

Fariness. Podemos entender a la definición de fairness como que infinitamente seguido[3], cada tarea (i.e., clase de equivalencia) C recibe un turno. Cuando sea que esto ocurra, o bien la acción de C es realizada o ninguna acción de C podría haber sido realizada ya que ninguna acción estaba disponible.

Denotamos al conjunto de ejecuciones justas de A como $fairexecs(A)$. Decimos que β es un *fair trace* de A si β es el trace de una ejecución justa de A, y denotamos al conjunto de fair traces de A como $fairtraces(A)$.

5.5.3. Tipos de Propiedades

Un autómata puede ser visto como una caja negra. Solo vemos los traces de las ejecuciones del autómata. Algunas de las propiedades a ser demostradas sobre el autómata son naturalmente formuladas como propiedades de sus traces (o de sus fair traces).

Formalmente, una *trace property* P consiste de lo siguiente:

- $sig(P)$, una signature que no contiene acciones internas.
- $traces(P)$, un conjunto (finito o infinito) de secuencias de acciones en $acts(sig(P))$.

Esto es, una trace property especifica tanto una interfaz externa como un conjunto de secuencias observadas en esa interfaz. Cuando decimos que un autómata A satisface una trace property P puede significar dos cosas diferentes:

- $extsig(A) = sig(P)$ y $traces(A) \subseteq traces(P)$
- $extsig(A) = sig(P)$ y $fairtraces(A) \subseteq traces(P)$

En cualquier caso, la idea intuitiva es que todo comportamiento externo que es producido por A es permitido por la propiedad P. Notemos que no necesitamos la inclusión opuesta— que todo trace de P pueda ser exhibido por A.

Podemos clasificar a las condiciones de sincronización en dos familias de propiedades: las propiedades de safety y las de liveness.

Propiedades de Safety. Las propiedades de *safety* son aquellas que nos dicen que cierta cosa *mala* nunca sucede. Por ejemplo, una luz de tráfico nunca se pone en verde en todas las direcciones, incluso si hay una falla en el suministro de energía. Los contraejemplos para este tipo de propiedades son **finitos**, es decir, se pueden dar en una cantidad finita de pasos.

Formalmente, decimos que una trace property P es una *trace safety property* si P satisface las siguientes condiciones.

1. $traces(P)$ es no vacío. Nada malo puede pasar antes de que ningún evento ocurra, esto es, cuando todavía no empezamos a ejecutar.

2. $traces(P)$ es *cerrado por prefijo*, esto es, si $\beta \in traces(P)$ y β' es un prefijo finito de β, entonces

[3] Un evento que ocurre ocasionalmente, pero que nunca para de ocurrir ocasionalmente, se le dice que ocurre infinitamente seguido (*infinitely often*). Si hubiera algún límite sobre la cantidad de veces que puede ocurrir, entonces no sería un evento que ocurre infinitamente seguido.

$\beta' \in traces(P)$. Es decir, si nada malo sucede en un trace, entonces nada malo ocurre en cualquier prefijo del trace.

3. $traces(P)$ es *cerrado por límite*, esto es, si $\beta_1, \beta_2, \ldots$ es una secuencia infinita de secuencias finitas en $traces(P)$ y, para cada i, β_i es un prefijo de β_{i+1}, entonces la única secuencia β que es el límite de β_i bajo el pedido de sucesivas extensiones está también en $traces(P)$. Es decir, si algo malo pasa en un trace infinito, entonces ocurre como resultado de algún evento particular que se da en un prefijo finito del trace.

Propiedades de Liveness. Las propiedades de *liveness* nos dicen que una cosa *buena* particular va a ocurrir en algún momento. Por ejemplo, una luz roja de tráfico eventualmente, se pone en verde. Los contraejemplos para este tipo de propiedades son **infinitos**, es decir, se dan en una cantidad infinita de pasos.

Formalmente, decimos que una trace property P es una propiedad de liveness si para toda secuencia finita sobre $acts(P)$ tiene alguna extensión en $traces(P)$.

Las definiciones que se pueden hacer sobre las propiedades de liveness y safety para ejecuciones son análogas a las de traces.

En el contexto de esta materia, no vamos a demostrar la correctitud del programa, sino que nos alcanza con poder argumentar la correctitud del mismo. Para hacer demostraciones formales, se utilizan este tipo de autómatas y se deben formalizar las propiedades de nuestros programas en alguna lógica (típicamente, temporal). Una lógica temporal consiste de un lenguaje lógico que contiene símbolos para nociones temporales como *eventualmente* y *siempre*, sumado a un conjunto de reglas de demostración para describir y verificar propiedades de las ejecuciones.

Otro método para demostrar una propiedad de liveness, conocido como el método de funciones de progreso, está especialmente diseñado para demostrar que cierto objetivo es eventualmente alcanzado. Este método involucra definir una función de progreso de estados del autómata a un conjunto bien fundado y mostrar que ciertas acciones garantizan que el valor de esta función continúe decrementándose hasta llegar al objetivo. También existen herramientas como Model Checkers y Theorem Provers para verificar si el programa es correcto.

5.5.4. Problema clásico: Turnos

Tenemos N procesos $P_0, P_2, \ldots, P_{N-1}$ ejecutando concurrentemente, y cada uno ejecuta cierta tarea. Supongamos que la tarea para cada proceso i consiste en imprimir `soy el proceso i;`. Entonces, lo que buscamos es que se impriman en el orden:

```
soy el proceso 0;
soy el proceso 1;
...
soy el proceso N-1;
```

En ese sentido, el problema de Turnos se reduce a forzar el orden en el que los procesos ejecutan. Una solución a este problema, utilizando semáforos, se muestra en la Fig. 5.28.

Podemos formalizar la condición de correctitud de la siguiente manera: si las tareas s_i ejecutan en el orden:

$$TURNOS = s_0, s_1, \ldots, s_{N-1},$$

entonces el programa es correcto. Esta propiedad es de tipo *safety*, porque los contraejemplos son finitos. Si encontramos una secuencia finita de pasos en donde dos procesos se ejecutan fuera de orden, entonces habremos demostrado que el programa no es correcto. La idea para argumentar que el programa de la Fig. 5.28 es correcto consiste en que la única forma en la que se pudo ejecutar s_i es que s_{i-1} haya enviado un `signal`, y esto solo puede ocurrir si s_{i-1}, $s_{i-2}, \ldots, s_0$ ya se ejecutaron.

```
semaphore sem[N+1]

proc init() // Inicializacion de semaforos.
{
    for (i = 0; i < N+1; i++){
        sem[i] = 0;
    }
    for (i = 0; i < N; i++){
        spawn P(i); // Lanzamos los N procesos
    }
    sem[0].signal();
}

proc P(i) { // Codigo para cada proceso

    sem[i].wait(); // Espera su turno.
    print("soy el proceso "  + i); // Hace lo que tiene que hacer
    sem[i+1].signal(); // Le avisa al proximo.
}
```

Figura 5.28: Turnos: Implementación con semáforos.

5.5.5. Problema Clásico: Rendezvous (o Barrera de Sincronización)

Tenemos N procesos $P_0, P_1, \ldots, P_{N-1}$ que tienen que ejecutar dos secciones de código: $a(i)$, $b(i)$. La propiedad que queremos que se cumpla es que todos los procesos ejecuten primero la sección de código $a(i)$ y, una vez que todos terminaron de ejecutar esa parte, recién ahí se ejecuten las secciones $b(j)$.

No se pretende que se ejecuten en orden los $a(0), a(1), \ldots, a(N-1)$, y luego los $b(0), b(1), \ldots, b(n-1)$. Eso sería restringir por demás el paralelismo: no hay que imponer ningún orden entre los a's, ni entre los b's. Lo que buscamos es que tanto los a's por un lado y los b's por otro lado, se ejecuten de la forma más paralela posible, pero los b's no pueden ejecutar hasta que terminen todos los a's. En algún sentido, hay una barrera entre todos los a's y todos los b's.

Podríamos pensar que una forma correcta de formalizar esta propiedad sería la siguiente:

$$BARRERA = \text{Ningún } b \text{ se ejecuta antes de que terminen todos los } a's.$$

Luego, proponemos el código mostrado en la Fig. 5.29 que satisface esta propiedad. Cada proceso P_i

```
atomic<int> cant = 0; // Procesos que terminaron a.
semaphore barrera = 0; // Barrera esta baja.

proc P(i) {
    a(i);
    if (cant.getAndInc() < N-1){ // Se puede ejecutar b?
        barrera.wait();  // No, esperar.
    }
    else{
        barrera.signal(); // Si, entrar y avisar.
    }
    b(i); // Seccion critica
}
```

Figura 5.29: Rendezvous: Implementación incorrecta.

ejecuta $a(i)$, para luego preguntar si el resto de los procesos ya ejecutaron su parte a. Si no se terminaron de ejecutar todos los a's, el proceso P_i se bloquea en el `wait`. Si los otros $N-1$ procesos ya habían terminado de ejecutar su parte a, el proceso P_i hace un `signal` y ejecuta su parte b.

Intuitivamente, podemos ver que la propiedad que definimos anteriormente se cumple en este programa. Sin embargo, esta solución es incorrecta. Lo que está haciendo el programa es mandar a dormir a los primeros $N-1$ procesos en llegar a la barrera, y el proceso restante es el único que envía un `signal`. Este `signal` solo despierta a uno de los $N-1$ procesos dormidos, y solo este proceso termina ejecutando su parte b. Luego, nos quedaron $N-2$ procesos bloqueados en el `wait` de la barrera que nunca llegaron a despertarse.

Recordemos que este programa cumplía con la propiedad que habíamos definido. El problema que estamos teniendo es que nos olvidamos de pedir que todos los b's logren ejecutarse en algún momento, es decir, nos faltó alguna propiedad de *liveness*.

Para describir esta propiedad de que todos los procesos ejecuten cierta porción de código, vamos a recurrir a un modelo para razonar sobre programas paralelos. El sistema es modelado como una colección de procesos y variables compartidas. Cada proceso i es un tipo de máquina de estados, con un conjunto $states_i$ de estados y un subconjunto $start_i$ de estados iniciales. Cada proceso i tiene acciones etiquetadas, que describen las actividades en las que participa. Estas acciones se clasifican en acciones de input, output o internas. Distinguimos entre dos tipos diferentes de acciones internas: aquellas que involucran memoria compartida y aquellas que involucran estrictamente cómputo local.

Hay una relación *trans* de transiciones para todo el sistema, que es un conjunto de triplas (s, π, s'), donde s y s' son estados del autómata, esto es, combinaciones de estados para todos los procesos y valores en todas las variables compartidas, y donde π es la etiqueta de alguna acción.

El problema de la exclusión mutua involucra la asignación de un solo, indivisible, recurso entre n usuarios $U_1, U_2, \ldots, U_n$ que no puede ser usado por dos usuarios al mismo tiempo. Un usuario con acceso al recurso es modelado como si estuviera en una *critical section*, que es simplemente un subconjunto designado de sus estados. Cuando un usuario no está involucrado de ninguna manera con el recurso, decimos que está en la *remainder section*. Para poder ganar acceso a su sección crítica, un usuario ejecuta un *trying protocol* y, luego de que termina de usar el recurso, ejecuta un *exit protocol*. Este procedimiento puede ser repetido, de manera tal que cada usuario siga un ciclo, moviéndose de su *remainder section* a su *trying section*, luego a su *critical section*, luego a su *exit section*, y luego devuelta a su *remainder section*. Este ciclo se muestra en la Fig. 5.30.

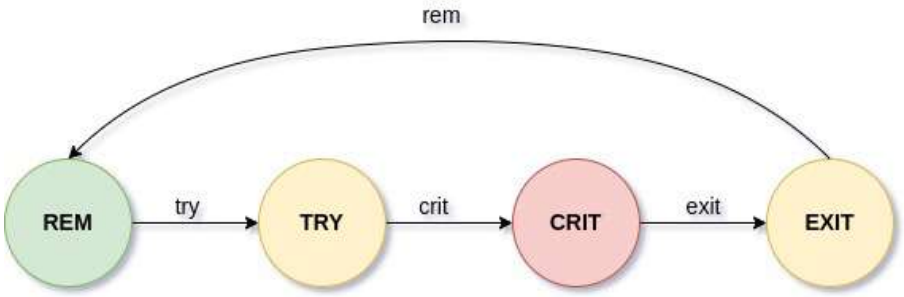

Figura 5.30: Modelo de Programas Paralelos (N. Lynch).

Tenemos a un programa que está ejecutando un código que no forma parte de la sección crítica (REM), luego comienza a intentar entrar a la sección crítica (TRY), logra entrar (CRIT), eventualmente consigue salir (EXIT), y finalmente vuelve a el resto del programa (REM).

Lo que propone el modelo de N. Lynch es que a cada estado de ejecución del programa se le puede asignar unívocamente uno de estos estados. Es decir, en todo paso de la ejecución del programa, estamos dentro de alguno de estos cuatro estados. Notamos de la siguiente manera:

- **Estado de Procesos**: $\sigma : [0 \ldots N-1] \mapsto \{REM, TRY, CRIT, EXIT\}$. (Cada proceso está en alguno de los cuatro estados).
- **Transición**: $\sigma \xrightarrow{\pi} \sigma', \pi \in \{rem, try, crit, exit\}$.

- **Ejecución**: $\tau = \tau_0 \xrightarrow{\pi_0} \tau_1 \xrightarrow{\pi_1} \tau_2 \ldots$. Es la secuencia de estados que recorre cada proceso. $\tau_k(i)$ indica el estado del i-ésimo proceso cuando está haciendo su k-ésimo paso (donde un estado puede estar incluido en REM, TRY, CRIT o EXIT).

Entonces, empezamos a ver cómo podríamos formalizar esas propiedades. La propiedad que queremos que se cumpla, conocida como WAIT-FREEDOM, sería algo como: *todo proceso que intenta acceder a la sección crítica, en algún momento, lo logra.* La intuición sería que el sistema está libre de procesos que esperan para siempre.

Formalmente, para toda ejecución, para todo paso dentro de esa ejecución, para todo proceso, si en el paso k de la ejecución τ del proceso i vemos que está intentando acceder a la sección crítica, entonces va a existir un paso posterior de la ejecución τ del proceso i en el que se haya accedido a la sección crítica.

$$\textit{WAIT-FREEDOM} \equiv (\forall \tau, k, i)(\, \tau_k(i) = TRY \Rightarrow (\exists k' > k)(\, \tau'_k(i) = CRIT\,)).$$

Ahora que tenemos bien definida la propiedad que debemos cumplir para que el programa sea correcto, podemos modificar la solución anterior para que cumpla con Wait-Freedom. En la Fig. 5.31 se muestra una solución correcta del problema.

```
1  atomic<int> cant = 0; // Procesos que terminaron a.
2  semaphore barrera = 0; // Barrera esta baja.
3
4  proc P(i) {
5      a(i);
6      if (cant.getAndInc() < N-1){ // Se puede ejecutar b?
7          barrera.wait();  // No, esperar.
8      }
9      barrera.signal(); // Despertar a uno
10     b(i); // Seccion critica
11 }
```

Figura 5.31: Rendezvous: Implementación correcta.

Trabajando con este modelo, podemos formalizar otras propiedades. Para notar estas propiedades, vamos a utilizar la lógica temporal LTL, donde

- $\Box(P)$. Significa que P vale para todos los pasos de la ejecución.
- $\Diamond(P)$. Significa hay algún paso posterior en el que vale P.
- **Excl** (exclusión mutua). Para toda ejecución τ y para todo estado τ_k, no puede haber más de un proceso i tal que $\tau_k(i) = CRIT$.

$$Excl \equiv \Box \#CRIT \leq 1.$$

- **Lock-Freedom**. Para toda ejecución τ y estado τ_k, si en τ_k hay un proceso i en TRY y ningún i' en CRIT, entonces existe $j > k$ tal que en el estado τ_j algún proceso i' está en CRIT. Lo que nos está diciendo esta propiedad es que si tengo algún proceso intentando entrar a la sección crítica y ésta está libre, en algún momento posterior va a ser ocupada— no necesariamente por el que estaba intentando entrar.

$$\text{Lock-Freedom} \equiv \Box(\#TRY \geq 1 \wedge \#CRIT = 0 \Longrightarrow \Diamond \#CRIT > 0).$$

- **Starvation-Freedom**. Si durante todos los pasos de ejecución de todo proceso se cumple OUT, es decir, que cada vez que se entra a la sección crítica, eventualmente se sale; entonces vale que para todos los pasos de ejecución de todo proceso se cumple IN, es decir, siempre sucede que si un proceso intenta entrar a la sección crítica, eventualmente lo logra. "Si todos se portan bien, todo va a salir bien". Podemos ver que es una condición más fuerte que pedir solo LOCK-FREEDOM,

ya que no solo nos pide que la sección crítica se ocupe, sino que se ocupe por todo proceso que intente entrar.

$$\text{Starvation-Freedom} \equiv \forall i \Box OUT(i) \Rightarrow \forall i \Box IN(i).$$

- **Wait-Freedom**. Cada vez que un proceso quiere entrar a la sección crítica, en algún momento futuro lo logra (sin importar cómo se comporten el resto de procesos). Podemos ver que es más fuerte que pedir Starvation-Freedom, porque no estamos pidiendo que el antecedente valga para garantizar que el consecuente valga.

$$\text{Wait-Freedom} \equiv \forall i \Box IN(i).$$

Un proceso i cumple con $IN(i)$ si cada vez que intenta entrar a la sección crítica, en algún momento futuro, lo logra; y cumple $OUT(i)$ si cada vez que está dentro de la sección crítica, en algún momento futuro, sale.

$$IN(i) \equiv i \in TRY \Rightarrow \Diamond i \in CRIT.$$

$$OUT(i) \equiv i \in TRY \Rightarrow \Diamond i \in REM.$$

Wait-free. Otra propiedad de liveness es la propiedad de *wait-free*. Decimos que un método es *wait-free* si garantiza que, para cada llamado, su ejecución termina en un número finito de pasos. Si el número de pasos es acotado (que posiblemente dependa de la cantidad de procesos), la propiedad se conoce como *bounded wait-free*. Un método wait-free cuya eficiencia no dependa en el número de procesos activos se conoce como *population-oblivious*.

Un método *wait-free* es atractivo porque garantiza que cada proceso que le toca su turno hace progreso. Sin embargo, estos algoritmos suelen ser ineficientes y, algunas veces, nos va a convenir utilizar algoritmos con garantías menos restrictivas.

Lock-free . Un método es *lock-free* si garantiza que, infinitamente seguido, finaliza en un número finito de pasos. Claramente, cualquier método que sea *wait-free* es también *lock-free*, pero no viceversa. Los algoritmos *lock-free* admiten la posibilidad de que algún proceso sufra de inanición. En la práctica, hay muchas situaciones donde es posible que haya inanición, pero es tan poco probable que un algoritmo rápido *lock-free* podría resultar más eficiente que un algoritmo más lento *wait-free*.

5.5.6. Problema clásico: Lectores / escritores

Supongamos que tenemos una base de datos compartida entre varios procesos concurrentes, a la que podemos acceder usando las funciones dadas `write` y `read`. Algunos de estos procesos podrían querer solo leer la base de datos, mientras que otros podrían querer actualizarla. Distinguimos entre estos dos tipos de procesos a partir de referirnos a los primeros como **lectores** y a los últimos como **escritores**. Queremos que los lectores puedan leer en simultáneo con otros lectores, pero que los escritores necesiten acceso exclusivo sobre la base para poder escribir. Esta propiedad se conoce como Single-Writer/Multiple Readers (SWMR). Concretamente, queremos que se cumplan las siguientes condiciones:

- Cualquier número de lectores puede estar en la sección crítica de manera simultánea.
- Los escritores deben tener acceso exclusivo a la sección crítica.

En otras palabras, un escritor no puede entrar a la sección crítica mientras que cualquier otro proceso (escritor o lector) esté allí. Mientras haya algún escritor en la sección critica, no puede entrar ningún otro proceso. En primera instancia, buscamos una solución que nos garantice el cumplimiento de estas restricciones, que permita el acceso a la base de datos a escritores y lectores, y que sea libre de deadlocks.

Vamos a necesitar las siguientes variables compartidas para resolver el problema.

```
1  int readers = 0
2  semaphore mutex = Semaphore(1)
3  semaphore roomEmpty = Semaphore(1)
```

El contador `readers` mantiene un registro de cuántos lectores hay actualmente en la sección crítica—`mutex` protege el contador compartido. `roomEmpty` vale 1 si no hay ningún proceso (lector o escritor) en la sección crítica, y 0 en el caso contrario.

El código para escritores es simple. Si la sección crítica está vacía, un escritor puede entrar, pero entrar tiene el efecto de excluir a todos los demás procesos.

```
1    roomEmpty.wait()
2    write()
3    roomEmpty.signal()
```

Notemos que cuando el escritor sale de la sección crítica, ésta queda vacía. Esto se debe a que ningún otro proceso pudo haber entrado mientras el escritor estaba en la sección crítica.

Para el caso de los lectores, necesitamos mantener un registro de la cantidad de lectores que actualmente están en la sección crítica. De este modo, vamos a poder distinguir el caso en el que entra el primer lector y el caso en el que sale el último lector.

El primer lector que llega tiene que esperar por `roomEmpty`.

- Si la sección crítica está vacía, entonces el lector procede y, al mismo tiempo, restringe la entrada de nuevos escritores. Los lectores subsiguiente pueden entrar, porque ninguno de ellos intenta esperar por `roomEmpty`.
- Si ya había un escritor en la sección crítica, entonces el primer lector espera por `roomEmpty`. Como el lector mantiene el mutex, cualquier lector subsiguiente se quedará esperando por el `mutex`.

El código luego de la sección crítica es similar. El último lector en salir de la sección crítica *apaga las luces*—esto es, hace un `signal` en `roomEmpty`, permitiendo que si había algún escritor esperando, éste pueda entrar.

```
1  mutex.wait()
2      readers += 1
3      if readers == 1:        # el primer lector bloquea CRIT
4          roomEmpty.wait()    # para uso exclusivo de lectores
5  mutex.signal()
6
7  read()
8
9  mutex.wait()
10     readers -= 1
11     if readers == 0:
12         roomEmpty.signal()  # el ultimo lector desbloquea CRIT
13 mutex.signal()
```

Para pensar sobre la correctitud del programa, nos va a servir enunciar una serie de propiedad sobre cómo se comporta el programa. Podemos ver que las siguientes condiciones se cumplen.

- Solo un lector puede estar esperando por `roomEmpty`, mientras que varios escritores pueden estar esperando en este semáforo.
- Cuando un lector hace un `signal` en `roomEmpty` la sección crítica está vacía.

Patrones similares a este código aparecen con frecuencia: el primer proceso en una sección bloquea un semáforo y el último en salir lo desbloquea. De hecho, es tan común que nos conviene darle un nombre y encapsular este comportamiento en un nuevo objeto.

El nombre del patrón es **Lightswitch**, por la analogía de cuando la primera persona en una sala prende la luz (bloquea al mutex) y la última la apaga (desbloquea al mutex). En la Fig. 5.32 podemos

ver la definición de la clase.

```
class Lightswitch:
    initialize():
        counter = 0
        mutex = Semaphore(1)

    lock(semaphore):
        mutex.wait()
            counter += 1
            if counter == 1:
                semaphore.wait()
        mutex.signal()

    unlock(semaphore):
        mutex.wait()
            counter -= 1
            if counter == 0:
                semaphore.signal()
        mutex.signal()
```

Figura 5.32: Definición de la clase Lightswitch.

`lock` toma un semáforo como parámetro, el cual será administrado por el LightSwitch. Si el semáforo está bloqueado, el proceso invocador se bloquea en `semaphore` y todos los procesos subsiguientes se bloquearán en `mutex`. Cuando el semáforo es desbloqueado, el primer proceso que estaba esperando lo vuelve a bloquear y todos los procesos que estaban esperando proceden.

`unlock` no tiene ningún efecto hasta que cada proceso que haya llamado a `lock` también llame a `unlock`. Cuando el último proceso llama a `unlock`, desbloquea al semáforo.

Usando esta nueva clase, podemos reescribir el código de lectores de una manera más simple:

```
readSwitch.lock(roomEmpty)
read()
readSwitch.unlock(roomEmpty)
```

Si bien este código resuelve el problema que habíamos planteado, es posible que los escritores sufran de inanición, es decir, no se cumple con wait-freedom. Supongamos que llegan de manera continua los suficientes lectores como para que `readers` nunca valga 0. Entonces, programáticamente nunca vamos a hacer el `roomEmpty.signal`. Esta situación no es un deadlock, porque algunos procesos todavía están progresando, pero no es una situación deseable.

Ahora veamos cómo podemos extender esta solución para que, cada vez que llegue un escritor, los lectores existentes puedan terminar, pero ningún lector adicional pueda entrar. La idea es agregar un molinete para los lectores y permitir que los escritores puedan bloquearlo. Los escritores tienen que pasar por el mismo molinete, donde deberían revisar el valor de `roomEmpty`. Si un escritor logra pasa por el molinete, pero todavía no consigue entrar a la sección crítica, se fuerza a los lectores a esperar en el molinete. Luego, una vez que salga el último lector de la sección crítica, tenemos garantizado que al menos un escritor va a entrar a la sección crítica. Una vez que termine de escribir, recién ahí liberará el molinete para que cualquiera de los lectores que se quedaron esperando en el molinete puedan proceder a competir por el acceso a la sección crítica.

```
readSwitch = Lightswitch()
roomEmpty = Sempahore(1)
turnstile = Semaphore(1)
```

`readSwitch` mantiene un registro de cuántos lectores hay en la sección crítica; bloquea a `roomEmpty` cuando el primer lector entra y lo desbloquea cuando el último lector sale. `turnstile` es un molinete para los lectores y un mutex para los escritores.

Si un escritor llega cuando hay lectores en la sección crítica, se va a bloquear al hacer `wait` sobre `roomEmpty`, lo cual implica que el molinete permanecerá bloqueado hasta que el escritor consiga entrar a la sección crítica. De esta manera, los nuevos lectores no podrán acceder a la sección crítica mientras que haya algún escritor esperando por `roomEmpty`.

```
proc writer(){
    turnstile.wait()
        roomEmpty.wait()
            write()
    turnstile.signal()

    roomEmpty.signal()
}
```

```
proc reader(){
    turnstile.wait()
    turnstile.signal()

    readSwitch.lock(roomEmpty)
        read()
    readSwitch.unlock(roomEmpty)
}
```

Cuando el último lector sale de la sección crítica, hace un `signal` en `roomEmpty`, desbloqueando al escritor que estaba esperando. El escritor inmediatamente entra a su sección crítica, ya que ninguno de los lectores puede pasar el molinete. Notemos que, en este punto, no puede haber ningún lector esperando por `roomEmpty`.

Cuando el escritor sale de la sección crítica hace un `signal` en `turnstile`, que desbloquea a algún proceso que estaba esperando en el molinete— un escritor o un lector. De esta manera, nos garantizamos de que al menos un escritor pueda proceder con su ejecución. De esta manera, evitamos que haya inanición por parte tanto de escritores como de lectores. Lectores y escritores compiten en igualdad de condiciones por el acceso al molinete, cosa que no pasaba con la sala.

5.5.7. Problema Clásico: Filósofos que cenan

El problema de los Filósofos que cenan fue propuesto por Dijkstra en 1965. Tenemos una mesa con cinco platos, cinco tenedores y un gran plato de fideos. Cinco filósofos, que representan procesos cooperativos, se sientan en la mesa y ejecutan el siguiente código en loop:

```
1 proc Filosofo(i) {
2     while(TRUE){
3         pensar();               // REM
4         tomar_tenedores(i)      // TRY
5         comer();                // CRIT
6         soltar_tenedores(i);    // EXIT
7     }
8 }
```

Los tenedores representan los recursos que los procesos necesitan obtener de manera exclusiva para hacer progreso. El problema es que los filósofos necesitan de dos tenedores para comer, por lo que un filósofo hambriento podría tener que esperar a que un vecino que suelte su tenedor.

Asumimos que los filósofos tienen una variable local i que identifica a cada filósofo con un número en el rango $[0, 4]$. Similarmente, los tenedores están numerados del 0 al 4, de manera tal que el filósofo i tiene el tenedor i a su derecha y el tenedor $i + 1 \mod 5$ a su izquierda (ver Fig. 5.33).

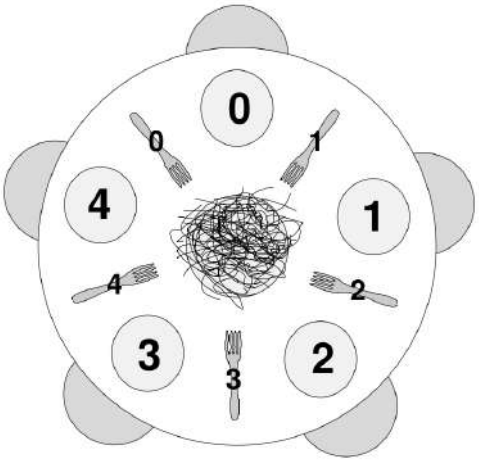

Figura 5.33: Filósofos que cenan.

Asumiendo que los filósofos saben cómo `comer` y `pensar`, lo que buscamos es escribir una versión de `tomar_tenedores` y `soltar_tenedores` que satisfaga las siguientes restricciones:

- Los tenedores son de uso exclusivo.
- Debe ser imposible que ocurra un deadlock.
- Debe ser imposible que un filósofo sufra inanición esperando por un tenedor.
- Debe ser posible que más de un filósofo coma al mismo tiempo. Esta propiedad nos impide usar un mutex para controlar que los `wait` y los `signal` se ejecuten sin ser interrumpidos, porque hacer esto impide que haya más de un filósofo comiendo a la vez. Además, esa solución iría en contra de buscar la mayor concurrencia posible.

Como queremos asegurar el acceso exclusivo a los tenedores, es natural usar una lista de semáforos, uno por cada tenedor. Inicialmente, todos los tenedores están disponibles. Luego, un primer intento para resolver el problema sería algo así:

```
1  def izq(i): return i
2  def der(i): return (i+1) % 5
3
4  tenedores = [Semaphore(1) for i in range(5)]
5
6  def tomar_tenedores(i):
7      tenedores[der(i)].wait()
8      tenedores[izq(i)].wait()
9
10 def soltar_tenedores(i):
11     tenedores[der(i)].signal()
12     tenedores[izq(i)].signal()
```

Si bien los tenedores son de uso exclusivo y puede haber más de un filósofo comiendo simultáneamente, el resto de las condiciones no se satisfacen. El problema que estamos teniendo es que la mesa es circular. Como resultado, cada filósofo puede tomar un tenedor y luego quedarse esperando para siempre por el otro tenedor, generando un deadlock (e inanición).

Veamos una forma simple para evitar estas situaciones de deadlock. Si tan solo permitimos que cuatro filósofos estén en la mesa al mismo tiempo, entonces es imposible que se genere un deadlock. En el peor caso, cada filósofo toma un tenedor. Veamos por qué vale esto.

Supongamos que se genera un deadlock. Supongamos que ningún filósofo tiene dos tenedores. Si hubiera algún filósofo con dos tenedores, éste estaría comiendo, por lo que todos los filósofos actualmente deben tener a lo sumo 1 tenedor. Como tan solo tenemos 4 filósofos, pero tenemos 5 tenedores, necesariamente debe haber al menos un tenedor libre con dos filósofos vecinos. Cualquiera de estos vecinos puede tomar el tenedor restante y comer. Luego, concluimos que no es posible caer en deadlock.

Podemos controlar la cantidad de filósofos en la mesa inicializando un semáforo `footman` con el valor 4.

```
1  footman = Semaphore(4)
2
3  def tomar_tenedores(i):
4      footman.wait()
5      tenedores[der(i)].wait()
6      tenedores[izq(i)].wait()
7
8  def soltar_tenedores(i):
9      tenedores[der(i)].signal()
10     tenedores[izq(i)].signal()
11     footman.signal()
```

Esta solución muestra que al controlar el número de filósofos, podemos evitar situaciones de deadlock. También garantiza que ningún filósofo sufre inanición. Imaginemos que hay un filósofo sentado en la mesa y sus vecinos están comiendo. El filósofo está bloqueado esperando por su tenedor derecho. Eventualmente, su vecino derecho terminará de comer y soltará sus tenedores. Como solo puede haber un único filósofo esperando por cada tenedor, necesariamente lo obtendrá inmediatamente después de que se ejecute el `signal`. Similarmente, se puede argumentar que no sufrirá de inanición esperando por su tenedor izquierdo. Es importante notar que la solución es simétrica, es decir, todos los filósofos primero intentan obtener el tenedor derecho y, luego, el izquierdo.

Otra forma de evitar situaciones de deadlock consiste en cambiar el orden en el que los filósofos toman los tenedores. Hasta ahora, todos los filósofos siempre toman primero el tenedor de la derecha. La idea es que si hay al menos un filósofo que tome siempre primero el tenedor de la derecha y otro filósofo que tome siempre primero el tenedor de la izquierda, entonces no es posible caer en deadlock.

La demostración se prueba por el absurdo. Primero, asumimos que es posible caer en deadlock. Luego, elegimos uno de los filósofos que supuestamente cayó en deadlock. Si es zurdo, entonces debió haber tomado su tenedor izquierdo y debería estar esperando por su tenedor derecho. Por lo tanto, su vecino derecho tuvo que haber tomado su tenedor izquierdo y estar esperando por su tenedor derecho; en otras palabras, también es zurdo. Repitiendo el mismo argumento, podemos demostrar que los filósofos son todos zurdos, lo cual contradice nuestra hipótesis. El caso en el que el filósofo sea diestro es análogo, y podemos demostrar que todos son diestros. En cualquier caso, obtenemos una contradicción; por lo tanto, no es posible caer en deadlock.

5.5.8. Problema clásico: Barbero

Una barbería tiene dos habitaciones: una sala de espera con $N - 1$ sillas, y la sala del barbero que contiene una única silla para cortar el pelo. Si no hay ningún cliente para ser atendido, el barbero se va a dormir. Cuando entra un cliente, pueden pasar tres cosas:

- Si un cliente entra a la barbería y todas las sillas están ocupadas, entonces el cliente se va de la barbería.
- Si el barbero está ocupado, pero hay sillas disponibles, entonces el cliente se sienta en una de las sillas libres.
- Si el barbero está dormido, el cliente despierta al barbero.

Para hacer el problema más concreto, tenemos la siguiente información.

1. Los clientes deberían invocar una función `getHairCut`.
2. Si un cliente llega cuando la tiendo está llena, hacemos un `return`.
3. El barbero debería invocar `cutHair`.
4. Cuando el barbero llama a `cutHair` debería haber exactamente un proceso llamando a `getHairCut` concurrentemente.

Se nos pide escribir un programa que coordine al barbero y a los clientes, garantizando estas restricciones.

```
customers = AtomicInteger(0)

customer = Semaphore(0)
barber = Semaphore(0)

customerDone = Semaphore(0)
barberDonde = Semaphore(0)
```

`customers` es un entero atómico que cuenta la cantidad de clientes que actualmente están en la tienda. El `barber` espera por `customer` hasta que un cliente entra en la tienda, y luego el cliente espera por el `barber` hasta que el barbero haga un `signal` para indicar que puede pasar. Luego del corte de pelo, el cliente hace un `signal` en `customerDone` y espera por `barberDone`.

Si hay N clientes en la tienda, cualquier cliente que llega inmediatamente retorna. Caso contrario, cada cliente hace un `signal` en `customer` y espera por el .

```
def Cliente():
    if customers.get() == N:     # Se fija si hay lugar.
        return                   # Si no hay, se va.
    customers.getAndInc()

    customer.signal()            # Le avisa al barbero que entró.
    barber.wait();               # Espera por el barbero.

        getHairCut();

    customerDone.signal()        # Para cumplir con
    barberDone.wait()            # condicion 4.

    customers.getAndDecrement() # Sale de la barberia.
```

Cada vez que un cliente hace un `signal`, el barbero potencialmente se despierta, y responde haciendo pasar al siguiente con un `signal` en `barber`. Si otro cliente llega mientras el barbero está ocupado, entonces en la siguiente iteración el barbero pasará por el semáforo sin tener que dormir.

```
def Barbero ():
    customer.wait() # Espera por un cliente
    barber.signal() # Le avisa al cliente que puede pasar.

    cutHair()

    customerDone.wait() # Solo puede haber un cliente
    barberDone.signal() # cortandose el pelo a la vez
```

Usando `customerDone` y `barberDone` garantizamos que se termine de cortar el corte de pelo (para ambas partes) antes de que ningún otro cliente entre a la sección crítica.

5.6. Resultados Teóricos sobre Primitivas de Sincronización

5.6.1. Registros read-write

A nivel de hardware, los procesos se comunican leyendo y escribiendo memoria compartida. Una buena forma de entender la comunicación entre procesos es abstraernos de las primitivas de hardware, y pensar a la comunicación como si estuviera pasando a través de objetos compartidos concurrentes.

Un registros *read-write* es un objeto que encapsula un valor que puede ser observado por una función de `read` y puede ser modificado por una función de `write`. En los sistemas multi-procesador, es esperable que los llamados a estas funciones se superpongan todo el tiempo, por lo que necesitamos especificar qué es lo que las llamadas concurrentes significan.

Una estrategia sería depender de la exclusión mutua: proteger cada registro con un mutex adquirido por cada `read` y cada `write`. Desafortunadamente, no podemos usar exclusión mutua en este punto, ya que todas las implementaciones de exclusión mutua que vimos utilizan estos registros read-write.

Un registro RW (Read-Write) se puede analizar en tres dimensiones: su dominio, procesos por operación y su característica en caso de solapamiento. Un registro puede tener un dominio binario (0 y 1) o puede implementar un rango de M valores— decimos que el registro es M-valuado. También es importante especificar cuántos lectores y escritores se espera tener. No es de sorprenderse que es más fácil implementar un registro que tan solo soporta un solo lector y un solo escritor que registros que soportan múltiples lectores y escritores. Para abreviar, usamos SRSW para significar *Single-Reader, Single-Writer*, MRSW para significar *Multiple-Readers, Single-Writer* y MRMW para significar *Multiple-Readers, Multiple-Writers*. Por último, nos queda ver qué pasa cuando las escrituras se solapan con lecturas.

Cuando `read` y `write` no se solapan, esperamos que para cualquier registro read-write, al llamar a `read` siempre devuelve el **último** valor escrito. Si `read` y `write` se solapan, podemos categorizar a los registros read-write en tres casos:

- **Safe**: `read` devuelve fruta, esto es, no se tiene ninguna certeza sobre el valor que puede devolver.
- **Regular**: `read` devuelve algún valor escrito, no necesariamente consistente con una serialización.
- **Atomic**: `read` devuelve un valor consistente con una serialización. Esto quiere decir que si una llamada a un método precede a la llamada de otro (sin solaparse), entonces la primera llamada debe hacer efecto antes de que lo haga la última. Por otro lado, si los métodos se superponen, su orden es ambiguo, por lo que podemos ordenarlos libremente.

Analicemos el ejemplo de la Fig. 5.34. R^i es la i-ésima lectura y $W(v)$ es la escritura del valor v. El tiempo fluye de izquierda a derecha. No importa si el registro es *safe, regular o atomic*, R^1 debe devolver 0, el último valor escrito.

- Si el registro es de tipo *safe* entonces como R^2 y R^3 son concurrentes con $W(1)$, pueden devolver cualquier valor posible dentro del rango del registro. (Si el registro es multivaluado con rango $[0, M]$, cualquier valor entre 0 y M sería válido).
- Si el registro es *regular*, R^2 y R^3 cada lectura podría devolver 0 o 1, independientemente del valor que haya devuelto la otra lectura.
- Si el registro es atómico entonces:
  - si R^2 devuelve 1, R^3 también devuelve 1.
  - si R^2 devuelve 0, R^3 podría devolver 0 o 1.

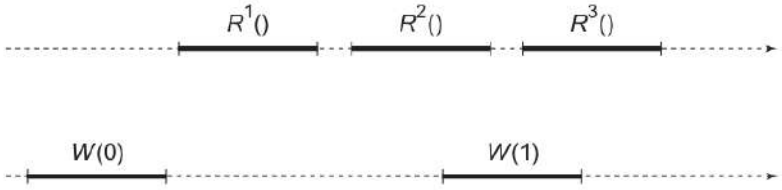

Figura 5.34: Una ejecución de un registro SRSW.

La pregunta que nos podemos hacer es si todo tipo de estructura de datos que puede ser implementada usando los registros más poderosos también puede ser implementada usando los registros más débiles (ver Fig. 5.35).

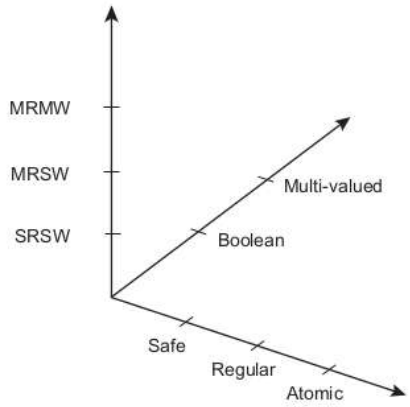

Figura 5.35: El espacio tri-dimensional de implementaciones basadas en registros read-write.

A partir de una serie de construcciones, es posible mostrar que podemos construir un registro atómico MRMW *wait-free* a partir de registros safe SRSW booleanos. Por lo tanto, podemos implementar cualquier algoritmo que utilice registros atómicos en cualquier arquitectura que tan solo soporte registros safe. El algoritmo de Peterson es un ejemplo de cómo podemos implementar registros atómicos MRMW multi-valuados a partir de registros safe SRSW booleanos (también hay otras soluciones posibles, como el algoritmo de Dekker, etc.).

5.6.2. Número de Consenso

Cuando se diseña un nuevo multi-procesador, es necesario determinar qué tipo de instrucciones atómicas habría que incluir. Soportar a todas las posibles instrucciones sería complicado e ineficiente, pero soportar las equivocadas podría hacer difícil o incluso imposible resolver importantes problemas de sincronización.

Buscamos identificar un conjunto de operaciones primitivas de sincronización lo suficientemente poderoso como para resolver problemas de sincronización que suelen aparecer en la práctica. (Evidentemente, podríamos querer soportar otras operaciones no esenciales por conveniencia.) Con este objetivo, necesitamos de alguna manera para evaluar el poder de varias primitivas de sincronización: qué problemas de sincronización pueden resolver, y qué tan eficientemente lo pueden hacer— de esto se trata el problema del consenso.

El problema del *consenso* es un problema abstracto que tiene consecuencias enormes para cualquier cosa, desde diseñar algoritmos hasta arquitecturas de hardware. Un *objeto de consenso* provee un solo método `decidir`. Cada proceso llama a `decidir` con cierta entrada v, con valor 0 o 1, a lo sumo una vez. El método retornará un valor que cumpla con las siguientes condiciones:

- *Consistencia.* Todos los procesos deciden el mismo valor.

- *Validez*. El valor de decisión común tuvo que ser ingresado por algún proceso.

Para llegar a un consenso, cada proceso hace un movimiento hasta que decide un valor. En este contexto, un movimiento sería una llamada a un método de un objeto compartido.

Nos interesan las soluciones wait-free al problema del consenso. A cualquier clase que implemente *consenso* de manera wait-free se dice que es un *consensus protocol*. Recordemos que una implementación de un objeto concurrente es wait-free si todo llamado a cualquier método termina en un número finito de pasos, para cualquier ejecución posible. Queremos saber si una clase particular de objetos es lo suficientemente poderosa como para resolver el problema del consenso.

Decimos que una clase C *resuelve* el problema del consenso para n procesos si existe un protocolo de consenso usando cualquier número de objetos de la clase C y cualquier número de registros atómicos. Llamamos al *consensus number* de la clase C es el mayor n para el cual la clase resuelve el problema del consenso para n proceso.

Notemos que si podemos implementar un objeto de la clase C a partir de uno o más objetos de la clase D y la clase C tiene número de consenso n, entonces la clase D tiene número de consenso de al menos n.

Si analizamos individualmente cada primitiva de sincronización, se puede llegar a los siguientes resultados:

- Los Registros read-write atómicos tienen número de consenso = 1. Esto nos dice que, si queremos implementar una estructura de datos concurrente que sea lock-free en sistemas multi-procesador, necesitamos que nuestro hardware provea operaciones primitivas de sincronización distintas que `read` y `write`.
- Las Colas, pilas tienen número de consenso = 2.
- Las primitivas TAS, getAndSet, getAndIncrement, getAndAdd tienen número de consenso = 2.
- La primitiva compareAndSwap tiene tiene número de consenso infinito.

Concluimos que la primitiva más poderosa, que permite implementar al resto de primitivas mediante un algoritmo wait-free, es compareAndSwap.

Capítulo 6

Preguntas de Final

6.1. Procesos

- ¿Qué es una system call? ¿Para qué se usan? ¿Cómo funcionan? Explicar en detalle el funcionamiento una system call en particular.
- ¿Las system calls son universales a todos los sistemas operativos o son dependientes de cada sistema?
- ¿Para qué sirve la system call fork? ¿Qué debilidades tiene? Comparar con `vfork` y la creación de threads.
- Diferencias entre system calls para crear procesos entre Linux y Windows.
- ¿Cómo funcionan los estados de un proceso? Ready, bloqueado, running.
- Explicar las transiciones de cada estado a cada estado (en particular, de waiting a ready).
- Hablar de la tabla de procesos.
- ¿Qué estructura se debe mantener en memoria para poder tener procesos? *(Explicar PCB)*
- ¿Qué es un proceso, un thread y en qué se diferencian?
- ¿En qué momento se ejecuta un thread? *(quantum del proceso)*
- ¿Qué debería agregar a la PCB para manejar los threads? *(Hablar de qué recursos son compartidos y cuáles no lo son)*
- ¿Qué pasaría si los threads compartieran el stack?
- Qué tendría que ver en un sistema para que piense que va a andar mejor agregando:
  1. más procesadores. *(carga del sistema o utilización de CPU)*
  2. más memoria. *(PFF — ver Cap. de Memoria)*

6.2. IPC

- Hablar de strace y ptrace.

6.3. Scheduling

- Describir los objetivos que pueden tener las políticas de scheduling (fairness, carga del sistema, etc.).
- ¿Qué objetivo prioriza SJF y por qué no se usa en la práctica?
- ¿Cómo funciona el scheduling con múltiples colas?
- ¿Hay algún problema con que las prioridades fueran fijas?
- Hablar sobre la afinidad de un procesador. ¿Qué información extra tenemos que tener en la PCB para mantener afinidad en un sistema multicore?
- Explicar el problema de inversión de prioridades.

6.4. Sincronización

- ¿Para qué necesitamos sincronización entre procesos? ¿Qué soluciones nos ofrece el HW? Explicar el caso para monoprocesador y para multiprocesador. *(instrucciones atómicas y deshabilitar interrupciones)*
- ¿Cómo nos afecta si el scheduler es preemptive o non-preemptive en la implementación de un semáforo?
- Evaluar si están bien o mal utilizadas en los siguientes ejemplos las primitivas de sincronización:
  - Usar TASLock (spinlock) para acceder a disco.
  - Usar semáforos para incrementar un contador.

```
semaphore s;
proc p(){
    s.wait();
    cont++;
    s.signal();
}
```

  - Usar un contador atómico para un recurso que tienen que poder acceder 3 procesos a la vez.

```
proc p(){
    atomic int x;
    if(x.get() <= 3){
        x.inc();
        // CRIT
        x.dec();
    }
}
```

  - usar spinlock para un recurso que tienen que poder acceder 3 procesos a la vez.
- Diferencia entre spin lock y semáforos (hablar de TTAS). ¿En qué contexto preferimos uno sobre el otro y por qué?.
- ¿Cómo implementamos una sección crítica con spin locks?
- Explicar el problema clásico de lectores y escritores. Explicar cómo podemos evitar inanición del escritor.

6.5. Memoria

- Se nos muestra un árbol de procesos donde cada proceso tiene una serie de page frames asignados. Explicar las siguientes situaciones:
  - ¿Por qué todos los procesos de un sistema compartirían una página? *(páginas del kernel o bibliotecas compartidas)*
  - ¿Por qué dos procesos específicos podrían compartir una página? *(hablar de fork y copy-on-write)*
- ¿Para qué sirve la paginación de la memoria? ¿Qué ventajas tiene sobre utilizar direcciones físicas? Hablar sobre el tamaño de las páginas y cómo se relaciona con el tamaño de los bloques en disco. (*hablar de fragmentación interna y fragmentación externa*)
- ¿Qué es un page fault y cómo se resuelve?
- ¿Por qué puede pasar que tengamos muchos procesos en *waiting*, y cómo podría tratar de arreglarlo si no pudiese agregar memoria?

6.6. E/S

- Hablar de RAID (para qué sirve). Explicar la diferencia entre RAID 4 y RAID 5. ¿Cuál es mejor y por qué?
- Explicar los distintos algoritmos básicos de scheduling de disco. Explicar cómo se comportan en HDD y en SDD.
- ¿Qué son los drivers y qué permiten?
- Explicar diferencias entre un disco magnético y un SSD. ¿Qué problemas tiene un SDD? Hablar de write amplification y borrado.
- ¿Cómo hace un disco para saber si un bloque está libre / si puede ser borrado? Explicar cómo podemos indicarle a un SDD que un bloque está libre (y que puede borrado). (*comando TRIM*)
- Explicar cómo se puede hacer una recuperación de datos, después de haber borrado un archivo.

6.7. FS

- ¿Qué es un file descriptor? Nombrar 3 system calls que los afecten.
- ¿Cuándo se revisan los permisos de acceso sobre un archivo? Explicar por qué el file descriptor se crea cuando hacemos un `open` y no se vuelven a revisar los permisos.
- ¿Qué es un FS y para qué sirve?
- ¿Cuándo es adecuado reservar espacio en disco de manera secuencial? ¿Qué beneficios nos trae? *(CD-ROM, ISO-9660)*
- ¿Cuál FS nos conviene utilizar para un sistema embebido: FAT o inodos?
- ¿Cuál FS nos conviene utilizar para implementar **UNDO** (deshacer la última escritura a disco)? ¿Cómo se implementaría en FAT y en inodos?
- ¿Cuál FS nos conviene utilizar para implementar un **backup total**? ¿Cómo se implementaría en FAT y en inodos?
- ¿Cuál FS nos conviene utilizar para implementar un **backup incremental**? ¿Cómo se implementaría en FAT y en inodos?

- ¿Cuál FS nos conviene utilizar para implementar **snapshot**? (*Diferenciar el caso en que queramos tomar una snapshot mientras el sistema está corriendo*) ¿Cómo se implementaría en FAT y en inodos?
- Explicar las diferencias entre FAT e inodos. Ventajas y desventajas de cada uno.
- ¿FAT implementa algún tipo de seguridad?
- Explicar qué es journaling.

ext2

- Describir ext2.
- ¿Qué mantiene un inodo? ¿Cómo es la estructura de directorios?
- ¿Para qué sirven los block groups y los clusters? Motivación para agrupar páginas en bloques, y bloques en grupos de bloques.
- ¿Cuáles son las estructuras más importantes de ext2? Explicar cada una (en particular, hablar del superbloque).
- Explicar cómo se manejan los bloques libres del disco.
- ¿Qué pasa cuando se inicia el sistema luego de haber sido apagado de forma abrupta? Explicar cómo hace el sistema para darse cuenta de si hubo algún error (cuando no hace journaling) y cómo lo arregla. (*inconsistencias entre contadores y bitmaps, entre entradas de inodos y bitmaps, entre entradas de directorios y bitmaps*)
- Explicar las diferencias (ventajas y desventajas) entre soft links y hard links. ¿Por qué no podemos usar un hard link para referenciar inodos de otro FS, incluso si está basado en inodos?
- Explicar cómo se crean y borran archivos con las estructuras del FS (incluido cómo se modifica el block group). Explicar el caso de borrado en hard links.
- Explicar qué ocurre cuando se borra un archivo en ext3 (y diferencias con ext2).

6.8. Distribuidos

- ¿Cómo podemos mantener una base de datos distribuida sincronizada?
- ¿Qué es un sistema distribuido? ¿Qué significa que un sistema sea completamente distribuido? ¿Qué beneficios ofrecen? ¿Qué problemas nos puede traer?
- Explicar los distintos algoritmos de commits que se utilizan para actualizar una copia del archivo que está en varios nodos (2PC y 3PC). ¿Cuál es la diferencia entre 2PC y 3PC? Explicar la diferencia entre weak/strong termination.
- ¿Qué hace un nodo si se cae después de que todos digan ”sí” al coordinador en 2PC?
- Explicar el problema bizantino.
- Explicar token ring ¿Qué problemas tiene? ¿Qué se hace en caso de que se pierda el token? ¿Cómo podemos mejorarlo?
- Explicar la diferencia entre grid y cluster.

File Systems Distribuidos

- ¿Qué es un file system distribuido? Explicar la interfaz VFS.

- Hablar de las limitaciones en DFS.
- ¿Cómo podría hacer para poder tener 2 discos distribuidos, y que los dos contengan la misma información? Y en caso de que se caiga la conexión, ¿cómo hacemos?
- ¿NFS es un file system completamente distribuido?
- Proponer una manera para mantener sincronizados N servidores NFS. Explicar cómo saben los nodos a qué servidor pedirle los datos.
- Si un nodo se cae, ¿cómo hacemos para que se entere después de las transacciones que no tiene?

6.9. Seguridad

- ¿Cuáles son las tres categorías principales de permiso frente a un archivo? (*owner, group, universo, ACL, capability*)
- Explicar cómo son las ACLs en UNIX.
- Explicar SETUID. Explicar otro método (distinto a setuid y setguid) para ejecutar con permisos de otro usuario.
- Explicar cómo funcionan los canarios.
- ¿Para qué sirve el bit NX (No eXecute) de la tabla de páginas?
- Explicar buffer overflow y mecanismos para prevenirlo. *(canario, páginas no ejecutables, randomización de direcciones)* ¿En qué parte de la memoria se buscan hacer los buffer overflow? (*En el stack, pero también se puede hacer en el heap.*
- ¿Cómo se genera el canario? ¿En tiempo de compilación o de ejecución?
- Explicar el ataque *Return to libc.*
- Dar un ejemplo de un error de seguridad ocasionado por race condition. *(cambiar puntero de symbolic links)*
- Explicar las diferencias entre MAC y DAC.
- ¿Qué es una firma digital? Explicar cómo es el mecanismo de autenticación.
- ¿Qué es una función de hash? ¿Cómo se usa en el log-in de un SO? ¿Qué problemas tiene usar un hash para autenticar usuarios? ¿Cómo se puede mejorar la seguridad? ¿Qué es un SALT? ¿Cómo podemos hacer que calcular la función de hash sea más costoso? ¿Qué otros usos tiene la función de hash en seguridad?
- ¿Qué es una clave pública/privada? ¿Cómo podemos distribuir una clave pública?
- ¿Por qué es más seguro utilizar un esquema de clave pública/privada para una conexión por ssh, comparado con usuario/contraseña?
- ¿Cómo podemos asegurarnos que un programa es confiable? ¿Qué pasa si nos modifican tanto el hash como el archivo? Explicar cómo podemos brindar de manera segura las actualizaciones de un software. (*(integridad, autenticación, canal seguro*)
- Explicar las diferencias entre HTTP y HTTPS. Explicar cómo se podría realizar un ataque cuando se realiza una actualización de un programa por HTTP (suponiendo que la actualización está firmada digitalmente) *pegándole a un endpoint http.*
- ¿Se puede considerar Deadlock un problema de seguridad?

- ¿Qué problemas de seguridad hay asociados a los file descriptors? ¿Cómo lo resuelve SELinux?

6.10. Virtualización

- ¿Qué es la virtualización y los contenedores? ¿Cómo se implementan?

Capítulo 7

Modelo de Final

- Qué tendría que ver en un sistema para que piense que va a andar mejor agregando:
  1. más procesadores.
  2. más memoria.
- ¿Cómo funcionan los estados de un proceso? Ready, bloqueado, running.
- Evaluar las siguientes implementaciones para sincronización:
  - usar TASLock para acceder a disco.
  - usar semáforos para incrementar un contador.
  - usar spinlock para un recurso que tienen que poder acceder 3 procesos a la vez.
- Se nos muestra un árbol de procesos donde cada proceso tiene una serie de page frames asignados. Explicar las siguientes situaciones:
  - ¿Por qué todos los procesos de un sistema compartirían una página? *(páginas del kernel o bibliotecas compartidas)*
  - ¿Por qué 2 procesos específicos podrían compartir una página? *(hablar de fork y copy-on-write)*
- ¿Cuál FS nos conviene utilizar para implementar UNDO (deshacer solo la última escritura a disco)? ¿Cómo se implementaría en FAT y en inodos?
- Explicar buffer overflow y mecanismos para prevenirlo. *(canario, páginas no ejecutables, randomización de direcciones)*

Bibliografía

[CES71] E. G. COFFMAN, M. J. ELPHICK y A. SHOSHANI. «System Deadlocks». En: *Computing Surveys* 3.2 (jun. de 1971), págs. 67-78. DOI: 10.1145/356586.356588.

[Com00] Douglas E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. 4.ª ed. Vol. 1. Prentice Hall, 2000.

[Dij65] Edgar W. Dijkstra. «Cooperating sequential processes.» En: (1965). URL: https://homepage.cs.uiowa.edu/~ghosh/skip.pdf.

[Dow] Allen B. Downey. *The Little Book of Semaphores*. URL: https://greenteapress.com/wp/semaphores/.

[HS08] Maurice Herlihy y Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[LB98] Bil Lewis y Daniel J. Berg. *Multithreaded Programming with Pthreads*. un Microsystems Press, 1998.

[Lyn97] Nancy A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1997.

[Mau08] Wolfgang Mauerer. *Professional Linux kernel architecture*. Wiley Pub., 2008.

[SGG18] Abraham Silberschatz, Peter B. Galvin y Greg Gagne. *Operating system concepts*. 10th. Wiley, 2018.

[Sta11] William Stallings. *Operating systems internlas and design principles*. Pretince Hall, 2011.

[Sta16] William Stallings. *Computer Organization and Architecture: Designing for performance*. Pearson, 2016.

[Ste99] W. R Stevens. *Unix network programming*. Vol. 2. Prentice Hall PTR, 1999.

[TB18] Andrew Tanenbaum y Herbert T. Boschung. *Modern Operating Systems*. 4th. Pearson, 2018.