Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Sistemas Operativos

Autor:
Leopoldo Taravilse

11 de diciembre de 2013

Índice

1. Procesos

1.1. ¿Qué es un proceso?

Un proceso es un programa en ejecución. El proceso incluye no sólo el código del programa sino también el estado de los registros, la memoria que utiliza el proceso y todos los demás recursos que utiliza durante su ejecución.

Un proceso tiene cinco posibles estados:

- New: El estado del proceso mientras está siendo creado.
- Ready: El estado del proceso que está listo para correr y a la espera de un procesador para correr.
- Running: El estado del proceso que está actualmente corriendo. Puede haber sólo un proceso en estado running al mismo tiempo.
- Waiting: El estado del proceso cuando está esperando un evento como puede ser entrada/salida.
- Terminated: El estado del proceso cuando ya finalizó su ejecución.

Todos los procesos están representados en el sistema operativos por su **Process Control Block (PCB)**. La PCB almacena entre otras cosas

- El estado del proceso.
- El Program Counter asociado a ese proceso, que indica cuál es la próxima instrucción a ejecutar.
- El estado de los registros de la CPU asociados al proceso.
- Información sobre la memoria asociada al proceso.
- Información de entrada/salida. Esto incluye por ejemplo la lista de archivos abiertos o de dispositivos de entrada/salida con los que interactua el proceso.

Cada proceso es unívocamente identificado por su **pid (process id)**.

Cuando un proceso que está ejecutandose en una CPU es desalojado de la misma para dar lugar a que se ejecute otro proceso, se produce lo que se conoce como cambio de contexto. El cambio de contexto consiste, entre otras cosas, en guardar la información del proceso saliente en su PCB, y cargar la del proceso entrante desde su PCB. El cambio de contexto siempre consume tiempo del procesador en el cuál nada util ocurre en ningún proceso, y por lo tanto es deseable que ocurra suficientemente poco, para no desperdiciar tanto tiempo, pero a su vez, suficientemente seguido, como para que ningún proceso se apropie de la CPU por mucho tiempo.

1.2. Creando procesos

Un proceso puede crear otros procesos. Si un proceso A crea un proceso B decimos que A es el proceso padre, y B es el proceso hijo.

En Linux para crear un proceso, el proceso padre debe llamar a la función $fork$, que es una system call. Luego de que la instrucción $fork$ es ejecutada, dos procesos pasan a existir en lugar de uno. La forma de reconocer cuál es el proceso padre y cuál es el proceso hijo es mediante el valor de retorno de $fork$. Si al ejecutarse $fork$ retorna 0, eso quiere decir que el proceso es el hijo, si en cambio retorna un valor distinto de cero, el proceso es el padre y el valor de retorno es el pid del proceso hijo que se acaba de crear.

Cuando un proceso es creado, no necesariamente tiene que seguir ejecutando el código del proceso padre. La función *exec* (que también es una system call), permite a un proceso (generalmente es el hijo quien la invoca) cambiar su código por el de otro programa. Esto permite que un proceso cree otro proceso, para que ejecute el código de otro programa.

También existe la system call *wait*, que le permite a un padre esperar a que termine la ejecución de su proceso hijo. Si el padre tuviese más de un hijo, y se quisiera esperar a que terminen todos ellos, debería hacerse un *wait* por cada uno de sus hijos.

Para terminar un proceso, existe la system call *exit*, que le indica a su padre que el proceso hijo ya ha finalizado su ejecución.

1.3. Comunicación entre procesos

En muchos casos es deseable que los procesos puedan comunicarse entre sí. Vamos a ver dos formas de comunicación entre procesos. Una de estas formas es mediante memoria compartida, mientras que la otra es el intercambio de mensajes entre procesos.

Para que los procesos se comuniquen mediante el uso de memoria compartida, estos deben avisarle al sistema operativo que compartirán parte de la memoria, ya que por defecto ningún proceso puede acceder a memoria que pertenece a otro proceso. Una vez que esto sucede, ambos procesos pueden leer y escribir en una misma región de la memoria, permitiendo que cada proceso pueda leer lo que el otro escribió.

En Linux un espacio de memoria compartida se reserva mediante la system call *shmget*, mientras que se accede con la system call *shmat*. Para dejar de utilizar el espacio de memoria compartida se debe llamar a la system call *shmdt* mientras que para liberar la memoria se utiliza la system call *shmctl*.

Para comunicar dos procesos mediante el intercambio de mensajes, se debe establecer una conexión que permita dos operaciones básicas: send y recieve. Existen a su vez dos tipos de send y dos tipos de receive: bloqueante y no bloqueante. Un send bloqueante espera a que la otra parte reciba el mensaje, mientras que un receive bloqueante espera a que la otra parte reciba el mensaje. En el caso del send y el receive no bloqueantes, simplemente hacen su tarea (enviar o recibir un mensaje) y el proceso sigue con su ejecución normalmente sin esperar nada del otro lado.

La implementación de Unix (y por lo tanto de Linux) del intercambio de mensajes es mediante sockets. Los sockets se pueden ver como extremos en una comunicación. Una vez creados los sockets, se puede escribir y leer como si fuesen un dispositivo de entrada/salida.

2. Threads

2.1. ¿Qué es un thread?

Hasta ahora estudiamos a los procesos como un único hilo de ejecución que se ejecuta en una CPU. Los threads nos permiten separar los procesos en varios hilos de ejecución que pueden correr en simultaneo en distintas CPUs. Un thread es una unidad básica de ejecución que tiene su propio **tid (thread id)**, que sirve para identificarlo de la misma manera que el pid identifica a un proceso, su propio estado del set de registros de la CPU, y su propia área de stack en memoria. Todos los demás recursos como lo son por ejemplo la sección de código o datos en memoria, los archivos abiertos, o los dispositivos de entrada/salida con los que se comunica el thread, con compartidos con todos los threads del mismo proceso.

Algunas de las ventajas del uso de threads son

- Cuando un proceso con un sólo thread está a la espera de una operación bloqueante como puede ser la comunicación con un dispositivo de entrada/salida, un proceso con muchos threads puede tener un sólo thread bloqueado mientras que los demás threads siguen ejecutandose sin tener que esperar.

- Los threads de un mismo proceso comparten su memoria, por lo que es muy útil crear threads en lugar de procesos cuando se quiere compartir memoria y código.
- Mientras que crear un proceso es muy costoso en términos de eficiencia, crear un thread es mucho más económico, entre otras cosas porque no se tiene que reservar memoria o crear una PCB.
- En arquitecturas con múltiples procesadores el uso de threads permite la ejecución de un proceso en más de una CPU, permitiendo así que al correr distintos threads del mismo proceso en distintas CPUs, el proceso se ejecute más rápido que un proceso monothread.

2.2. Pthreads

El standard de threads de POSIX, conocido como pthreads, define una API a la que se deben adaptar las implementaciones de threads para sistemas operativos basados en Unix. El tipo de datos que se utiliza para crear threads es $pthread\_t$ y la forma decrear un thread es con la función $pthread\_create(\&tid, \&attr, startfn, args)$ siendo tid una variable de tipo $pthread\_t$, $attr$ los atributos (en el scope de la materia los ignoramos y los seteamos siempre en NULL), $startfn$ la función donde empieza a correr el thread y $args$ un puntero a void que contiene los argumentos de la función. Además, la función $startfn$ debe ser de tipo puntero a void.

Para esperar a que un thread termine su ejecución, tenemos la función $pthread\_join$, que toma como argumentos el tid del thread y el puntero a void donde queremos guardar el valor de retorno de la función $startfn$ (puede ser NULL si queremos ignorar este valor).

Si bien los threads comparten la memoria del proceso al cual pertenecen, es posible reservar un área de memoria específica para cada thread.

3. Scheduling

3.1. Scheduling

Cuando más de un proceso corre en simultaneo, es necesario decidir cuándo y cuánto tiempo corre cada proceso (o thread). Los algoritmos que determinan cuándo se desaloja un proceso o thread de una determinada CPU, dando lugar a otro proceso o thread para que corra se llaman algoritmos de scheduling. De ahora en más hablaremos de scheduling de procesos, aunque todo lo que digamos es también válido para threads.

Cuando una CPU se libera, es necesario decidir qué proceso empieza a correr en esa CPU. Qué proceso elegimos depende del algoritmo de scheduling.

La política de scheduling debe estar definida en función de optimizar alguna combinación de los siguientes objetivos (algunos de los cuales son contradictorios)

- Fairness (o ecuanimidad): que cada proceso reciba una dosis justa de CPU (para alguna noción de justicia).
- Eficiencia: tratar de que la CPU esté ocupada la mayor cantida de tiempo posible.
- Carga del sistema: intentar minimizar la cantidad de procesos en estado ready.
- Tiempo de respuesta: intentar minimizar al tiempo de respuesta percibido por los usuarios interactivos.
- Latencia: tratar de minimizar el tiempo requerido para que un proceso empiece a dar resultados.
- Tiempo de ejecución: intentar minimizar el tiempo que le toma a cada proceso ejecutar completamente.

- Throughput (o rendimiento): tratar de maximizar la cantidad de procesos listos por unidad de tiempo.
- Liberación de recusos: tratar de que terminen cuanto antes los procesos que tienen reservados más recursos.

Un scheduler puede ser **cooperativo (nonpreemptive)** o **con desalojo (preemptive)**. Si un scheduler es preemptive, además de decidir cuándo le toca ejecutar a un proceso, también debe decidir cuándo ya no le toca ejecutar más a un proceso que está actualmente en ejecución.

3.2. Políticas de Scheduling

Existen varias políticas de scheduling que son muy conocidas. A continuación veremos algunas de ellas.

3.2.1. First Come First Serve (FCFS)

El scheduler First Come First Serve (FCFS) es el más sencillo de todos, aunque sin embargo no es de los más útiles. Este scheduler consiste en simplemente asignarle la CPU a los procesos en el orden en el que van llegando sin preemption.

En un scheduler FCFS las tareas son desalojadas o bien porque termina su ejecución o bien porque solicitan entrada/salida. Podría pasarnos que una tarea que no solicita entrada salida llega primero y ejecuta por un tiempo muy largo, mientras que una segunda tarea tiene que esperar a que la primera termine para solicitar entrada/salida, dejando la CPU inutilizada una vez que esta segunda tarea es desalojada por el uso de entrada/salida.

3.2.2. Shortest Job First (SJF)

El scheduler Shortest Job First le asigna la CPU siempre al proceso cuyo tiempo de ejecución es menor. Esta política es muy buena en muchos casos aunque imposible de llevar a la práctica ya que es imposible saber cuánto tiempo va a llevarle a un determinado proceso ejecutar.

Un scheduler SJF puede ser tanto preemptive como nonpreemptive. En el caso del scheduler SJF con preemption, cuando llega un proceso a la cola de procesos listos, sólo reemplaza al proceso actual si su tiempo de ejecución es menor al tiempo de ejecución restante del proceso actual.

3.2.3. Priority Scheduling

El caso de SJF es un caso de scheduler con prioridades. En un scheduler con prioridades se definen las prioridades de cada proceso y ejecuta el proceso con mayor prioridad en cada paso. En el caso de SJF la prioridad está dada por el tiempo restante de ejecución de un proceso.

Este tipo de schedulers puede dar lugar a lo que se conoce como **starvation (o inanición)**, un escenario donde a un proceso nunca le es asignada una dosis de CPU. Una solución posible a este problema es ir reduciendo la prioridad de cada proceso a medida que va ejecutando, para que eventualmente todos los procesos terminen ejecutandose. Aunque esta solución es parcial, ya que podrían llegar todo el tiempo procesos nuevos con alta prioridad y dejar sin ejecutar a un proceso de baja prioridad.

3.2.4. Round Robin

El scheduler Round Robin es parecido al FCFS pero con preemption. Se le agrega lo que se denomina **quantum**, que es una unidad de tiempo tras el cual el proceso que corre en una CPU es desalojado y es insertado al final de la cola de procesos ready. El quantum suele ser de entre 10 y 100 milisegundos.

En un scheduler Round Robin, si el quantum es muy grande, va a parecerse mucho a un FCFS, en cambio si es muy chico, va a haber mucha pérdida de tiempo en los cambios de contexto, por lo cual es muy importante saber elegir el quantum apropiado.

3.2.5. Scheduling con múltiples procesadores

En el caso de que haya múltiples procesadores, es conveniente que un proceso ejecute siempre en el mismo procesador, para optimizar el uso de la caché. Existen políticas de scheduling que respetan esto a rajatabla (siempre se utiliza el mismo procesador para un mismo proceso), donde se dice que hay afinidad dura, y políticas en las que se dice que hay una afinidad blanda, ya que no siempre un mismo proceso corre en un mismo procesador.

4. Sincronización de procesos

4.1. Condición de carrera

Supongamos que dos procesos están corriendo concurrentemente, y comparten memoria. Supongamos también que, dada una variable v, los procesos quieren hacer $v++$ y $v--$ respectivamente. Las ejecuciones, vistas en operaciones atómicas del procesador, serían

```
1 registro1 = v;
2 registro1 = registro1+1;
3 v = registro1;
```

```
1 registro2 = v;
2 registro2 = registro2-1;
3 v = registro2;
```

Debido a que los procesos no poseen el control sobre el scheduler, podría suceder que el orden en el que se ejecuten estas operaciones sea

```
1 registro1 = v;
2 registro2 = v;
3 registro1 = registro1+1;
4 v = registro1;
5 registro2 = registro2-1;
6 v = registro2;
```

Dando lugar a que si, por ejemplo, v valía 4 en un principio, pase a valer 3 en lugar de seguir valiendo 4 luego de ser incrementada y decrementada. A este problema se lo conoce como condición de carrera.

Esto puede ocurrir siempre que haya contención (varios procesos quieren acceder al mismo recurso) y concurrencia (dos o más procesos se ejecutan en simultaneo, no necesariamente en paralelo).

4.2. Sección crítica

A veces un proceso tiene que ejecutar un bloque de código que implica escribir parte de la memoria compartida o escribir un archivo. Esta parte del código en la que el proceso no puede ejecutarse en simultaneo con otra parte de otro proceso que escriba las mismas variables o archivos se la conoce como sección crítica.

Cuando más de un proceso tiene la sección crítica, sólo uno a la vez puede acceder a la misma al mismo tiempo. La solución al problema de la sección crítica tiene tres requerimientos:

- Exclusión mutua: Si un proceso entra a su sección crítica, ningún otro proceso puede estar en su sección crítica.
- Progreso: Si ningún proceso está ejecutando su sección crítica y algunos procesos quieren entrar en su sección crítica, sólo uno puede hacerlo y la decisión de qué proceso entra en su sección crítica no puede ser postpuesta indefinidamente.
- Espera acotada: Existe un límite de cuántas veces pueden entrar otros procesos a sus secciones críticas antes que un proceso pueda entrar a la suya una vez que el proceso pidió entrar en su sección crítica.

4.2.1. Solución de Peterson al problema de la sección crítica

Una posible solución al problema de la sección crítica cuando hay sólo dos procesos, es la solución de Peterson.

Esta solución utiliza dos variables:

```
int turn;
bool flag[2];
```

turn va a ser un entero que indique a qué proceso le toca correr su sección crítica y flag es un flag que dice si el proceso está listo para correr su sección crítica.

La solución de Peterson para el proceso i (0 o 1, $j = 1 - i$) es la siguiente:

```
do{
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j);
    //sección crítica
    flag[i] = false;
    //sección no crítica
}while(true);
```

Lo que hace este código básicamente es lo siguiente: Antes de entrar a la sección crítica un proceso avisa que quiere entrar seteando su flag en tru, después le cede el turno al otro proceso, y si el otro proceso tiene su flag en true se queda esperando a que termine de correr su sección crítica. De esta manera podemos asegurar los tres requisitos de la solución de la sección crítica:

- Exclusión mutua: No puede ser que dos procesos entren a la vez a la sección crítica. Si un proceso está en su sección crítica tiene su flag en true, luego el otro proceso tiene que tener el turno para no entrar en el while, pero como el proceso i está en la sección crítica, cuando j setea el turno en i entra en el while hasta que i libera el flag. Si los dos quisieran entrar a la vez, ambos flags estarían prendidos y sólo uno de los dos procesos tendría su turno.
- Progreso: Uno de los dos procesos va a tener el turno en todo momento, luego si sólo uno quiere entrar a la sección crítica va a entrar porque el otro va a tener el flag apagado, y si los dos quieran entrar, el primero que le asigne el turno al otro, va a recuperar el turno cuando el otro proceso se lo asigne y va a entrar en la sección crítica, dejando al otro ciclar en el while.
- Espera acotada: Una vez que el proceso entra en el while, cuando el otro proceso libera el flag, ya puede entrar a la sección crítica, aún si el otro proceso vuelve a pedir la sección crítica ya que en ese caso le va a ceder el turno.

4.2.2. Test and Set

Otra solución al problema de la sección crítica requiere un poco de ayuda del hardware. Si contamos con una instrucción **TestAndSet** atómica que nos devuelva el valor que tenía una variable booleana antes de ejecutar la instrucción, y la setea en true, entonces podemos hacer uso de lo que se conoce como **locks**.

La instrucción TestAndSet debe hacer atómicamente (es decir, sin la posibilidad de ser interrumpida en el medio de la instrucción) lo mismo que hace el siguiente código

```
bool TestAndSet(bool *var)
{
    bool res = *var;
    *var = true;
    return res;
}
```

La solución implementando locks con TestAndSet sería la siguiente:

```
bool lock // variable global compartida por todos los procesos
while(true)
{
    while(TestAndSet(&lock));
    //sección crítica
    lock = false;
    // sección no crítica
}
```

Para evitar inanición la variable *lock* debe ser inicializada en false. Además, como TestAndSet es atómica, no puede haber condición de carrera si se ejecutan dos TestAndSets en dos procesos separados ya que uno se ejecutará primero y el otro después.

En los casos de la solución de Peterson y de la implementación de locks con TestAndSet, estamos haciendo un while que cicla permanentemente hasta que le toca el turno al proceso de correr su sección crítica. Esto se llama **busy waiting** y es una **MUY MALA IDEA** ya que consumimos mucho tiempo de la CPU ejecutando el while cuando tranquilamente podríamos poner un sleep en el cuerpo del while. Aunque también debemos ser cuidadosos, si el sleep es por un tiempo muy grande podemos perder eficiencia, y si es por un tiempo muy chico también estamos desperdiciando tiempo del procesador como en el caso de busy waiting.

4.3. Semáforos

Sería de mucha utilidad tener la posibilidad de tener una variable *lock* y pedirle al procesador que continue con la sección crítica sólo cuando esta tome un determinado valor. Afortunadamente, no somos los primeros a los que se nos ocurre esta idea, sino que ya **Dijkstra** (si, en negrita porque es Dijkstra) pensó en este problema e implementó una solución que conocemos como **semáforos**.

Un semáforo es una variable entera, que toma valores no negativos, que es inicializada con un valor elegido por el programador, y que luego de ser inicializada sólo puede ser accedida mediante dos funciones: **wait** y **signal**. Estas funciones son operaciones atómicas y su ejecución es similar a la de los siguientes códigos:

```
wait(S)
{
    while(S<=0);
    S--;
}
```

```
signal(S)
{
    S++;
}
```

Existen dos tipos distintos de semáforos. Uno de ellos son los semáforos que pueden tomar cualquier valor no negativo, mientras que los otros sólo pueden tomar dos valores: 0 y 1. Estos últimos son conocidos como semáforos binarios o **mutex**.

Los mutex pueden ser utilizados por ejemplo para resolver el problema de la sección crítica. Cuando el mutex tiene el valor 1 y un proceso hace un wait, dejandolo en 0, se dice que el proceso toma el mutex. Al hacer un signal de un mutex que está en 0, se dice que el proceso libera el mutex. Para resolver el problema de la sección crítica es necesario tener un mutex inicializado en 1, luego cada proceso que quiere entrar en su sección crítica debe tomar el mutex para entrar, y liberarlo al salir de la sección crítica. De este modo, no puede haber más de un proceso a la vez en la sección crítica.

Para evitar el busy waiting, los semáforos se implementan de la siguiente manera: Además del valor del semáforo, se guarda una lista de procesos en espera. Cada vez que un proceso hace un wait, si tiene que esperar a que el valor del semáforo sea mayor a cero, libera la CPU en la que está corriendo, pasando a la lista de espera asociada al semáforo que pidió. Este proceso no es agregado a la cola de procesos ready sino que, al hacer un signal, el primer proceso en la lista de espera (que funciona como una cola), es pasado a la cola de procesos ready. De esta manera la implementación de los semáforos sería la siguiente:

```
struct semaphore{
    int value;
    process *list;
};
```

```
wait(semaphore *S)
{
    S->value--;
    if(S->value<0)
    {
        S->list.add(thisProcess);
        block();
    }
}
```

```
signal(semaphore *S)
{
    S->value++;
    if(S->value<=0)
    {
        remove process P from S->list;
        wakeUp(P);
    }
}
```

Con esta implementación de semáforos, los valores que toman los mismos pueden ser negativos y se usa este valor, cuando es negativo, para poder determinar cuántos procesos hay en lista de espera. La función *block* bloquea el proceso actual (sacandolo de la CPU donde está corriendo sin encolarlo en la cola de procesos ready) mientras que la función $wakeUp(P)$ encola el proceso P en la cola de procesos ready. Ambas operaciones (*block* y $wakeUp$) son system calls provistas por el sistema operativo.

La cola de procesos en espera por un semáforo no es necesariamente una cola FIFO, sino que puede ser una cola de prioridades o cualquier otro tipo de cola (al igual que las colas de procesos listos para ser cargados en una CPU por el scheduler). Es importante notar que si bien podemos usar cualquier tipo de cola (por ejemplo, una cola LIFO), según la estrategia que utilizemos para encolar y desencolar podríamos generar inanición.

Un uso muy común de los semáforos es cuando hay N copias de un recurso dado R. En este caso un semáforo se inicializa en N, y cada vez que un proceso quiere hacer uso de una copia del recurso R pide el semáforo, liberandolo luego de utilizar el recurso.

Un problema muy común en el uso de semáforos, el cuál abordaremos más en detalle en la próxima sección, es cuando hay un conjunto de procesos que está esperando eventos que sólo pueden ocurrir en otro proceso de ese mismo conjunto. Por ejemplo, dos procesos A y B, ambos están trabados en un wait y cada uno está esperando el signal del otro proceso, como se ve en el código a continuación:

```
A()
{
    semB.wait();
    semA.signal();
}
```

```
B()
{
    semA.wait();
    semB.signal();
}
```

En este caso, A se queda esperando que B libere su semáforo, sin liberar el propio, mientras que B espera que A libere el suyo, sin liberar el semáforo que A está pidiendo. En este caso los dos procesos se van a quedar trabados indefinidamente ya que nadie les va a liberar los semáforos que están pidiendo. Este problema se conoce como **deadlock**.

Veamos algunos ejemplos de problemas que pueden ser resueltos con semáforos, y sus respectivas soluciones.

4.3.1. Rendezvous

Supongamos que tenemos los procesos A y B que quieren ejecutar dos instrucciones cada uno, y queremos que ninguno de los dos ejecute su segunda instrucción antes de que el otro haya ejecutado su primer instrucción, sin poner restricciones sobre el orden en el que ejecuta cada uno su primer instrucción, ni tampoco la segunda. Este problema se conoce como rendezvous y su solución con semáforos es la siguiente:

```
A()
{
    a1;
    semA.signal();
    semB.wait();
    a2;
}
B()
{
    b1;
    semB.signal();
    semB.wait();
    b2;
```

```
}
```

Donde $a1$ y $a2$ son las instrucciones del proceso A, $b1$ y $b2$ son las instrucciones del proceso B, y $semA$ y $semB$ son dos semáforos que empiezan inicializados en cero.

4.3.2. Barrera

Supongamos ahora que tenemos una situación análoga a la del rendezvous, pero con $N > 2$ procesos en lugar de 2. La solución a este problema se conoce como barrera y su implementación es la siguiente:

```
P()
{
    //instrucciones previas al rendezvous
    mutex.wait();
        count++;
        if(count==n)
            barrier.signal();
    mutex.signal();
    barrier.wait();
    barrier.signal();
    //instrucciones posteriores al rendezvous
}
```

Este sería el código de cada uno de los procesos que necesitan esperar a los otros procesos para seguir con su ejecución. En este caso *count* es una variable que cuenta cuántos procesos llegaron al punto del rendezvous, y empieza inicializada en cero, *barrier* es un semáforo que empieza inicializado en cero (la barrera), y *mutex* es un mutex que proteje la variable *count* y que empieza inicializado en 1.

El semáforo al que se le hace un signal inmediatamente después de un wait se lo denomina turnstile (molinete en inglés) ya que sirve para que los procesos vayan pasando de a uno por ese punto.

4.3.3. Barrera reutilizable

A veces queremos reutilizar una barrera, por ejemplo para que dentro de un loop cada ejecución del loop requiera que todos los procesos se encuentren en un punto, ejecuten su sección crítica, y luego sigan ejecutandose hasta la próxima ejecución del loop en la que nuevamente necesitamos de la barrera. Veamos una solución a este problema:

```
P()
{
    while(condicion)
    {
        //instrucciones previas a la sección crítica
        mutex.wait();
            count++;
            if(count==n)
            {
                turnstile2.wait();
                turnstile.signal();
            }
        mutex.signal();
        turnstile.wait();
        turnstile.signal();
        //sección crítica
```

```
        mutex.wait();
            count--;
            if(count==0)
            {
                turnstile.wait();
                turnstile2.signal();
            }
        mutex.signal();
        turnstile2.wait();
        turnstile2.signal();
        //instrucciones posteriores a la sección crítica
    }
}
```

En este caso *turnstile* empieza inicializado en 0 mientras que *turnstile*2 empieza inicializado en 1.

4.3.4. Productor - Consumidor

Supongamos que tenemos dos procesos, uno que produce recursos y otro que los utiliza. Queremos que el productor vaya almacenando recursos en un buffer mientras que el consumidor retire recursos del buffer siempre que haya algún recurso disponible en el buffer para utilizarlo. El acceso al buffer debe ser exclusivo, es decir, no pueden acceder el productor y el consumidor al mismo tiempo. Veamos una solución a este problema:

```
producer()
{
    while(true)
    {
        resource = waitForResource();
        mutex.wait();
            buffer.add(resource);
            items.signal();
        mutex.signal();
    }
}
consumer()
{
    while(true)
    {
        items.wait();
        mutex.wait();
            resource = buffer.get();
        mutex.signal();
        resource.use();
    }
}
```

En este caso, *items* es un semáforo que indica la cantidad de items en el buffer, *mutex* protege el buffer, y resource es una variable local tanto para el productor como para el consumidor. Esta solución funciona también para múltiples productores y múltiples consumidores.

4.3.5. Read-Write Lock

Un problema de sincronización muy común aparece cuando tenemos varios procesos, algunos de los cuales quieren escribir una variable, y otros leerla. Puede haber varias lecturas en simultaneo, pero sólo una escritura a la vez, y nadie puede leer mientras alguien está escribiendo. Una posible solución a este problema es la siguiente:

```
writer()
{
    turnstile.wait();
        roomEmpty.wait();
        //sección crítica
    turnstile.signal();
    roomEmpty.signal();
}
reader()
{
    turnstile.wait();
    turnstile.signal();
    mutex.wait();
        count++;
        if(count==1)
            roomEmpty.wait();
    mutex.signal();
    //sección crítica
    mutex.wait();
        count--;
        if(count==0)
            roomEmpty.signal();
    mutex.signal();
}
```

En este caso tanto *mutex* como *roomEmpty* (que también es un mutex) empiezan inicializados en 1, al igual que *turnstile*. *count* cuenta la cantidad de readers en la sección crítica y empieza inicializado en cero.

4.4. Monitores

Otra alternativa a los semáforos son los monitores. Un monitor es un tipo de datos que contiene variables (normalmente son las variables que se desean proteger de accesos concurrentes) y metodos que operan sobre esas variables, así como un metodo para inicializarlas. Los monitores también tienen lo que se conoce como variables de condición, que son parecidas a los semáforos. Una variable de condición tiene también las operaciones atómicas signal y wait, pero con la diferencia de que sólo se puede despertar de un wait con un signal posterior al wait, y no guardan ningún valor. Esto es, si ocurre un signal mientras ningún proceso está bloqueado por un wait, el signal es ignorado, mientras que si ocurre un signal cuando hay otro proceso esperando por un wait, entonces el proceso que está trabado en wait tiene permiso para seguir ejecutandose.

5. Deadlock

5.1. Condiciones de Coffman

Como vimos anteriormente, la situación en la que en un conjunto de dos o más procesos están todos esperando a que ocurra un evento que sólo puede desencadenar otro proceso de ese conjunto, se la conoce

como deadlock. En esta sección nos centraremos en el caso de deadlock de recursos (ya sea una variable en memoria, un archivo, una dispositivo de entrada/salida, etc.), es decir, cuando un conjunto de procesos está esperando a que se libere un recurso que tiene otro proceso de ese conjunto.

Sólo puede ocurrir deadlock cuando se cumplen las siguientes cuatro condiciones (notar que pueden cumplirse y aún así no haber deadlock):

- Exclusión mutua: Cada copia de cada recurso puede ser asignada a un sólo proceso en un mismo instante de tiempo.
- Hold and Wait: Los procesos que tienen recursos asignados pueden pedir más recursos.
- No preemption: Nadie le puede quitar los recursos a un proceso, sino que los debe liberar por su cuenta.
- Espera circular: Hay dos o más procesos, cada uno de los cuales está esperando por un recurso que tiene el proceso siguiente.

5.2. Algoritmos de prevención de deadlock

5.2.1. Estados seguros

Un estado se dice seguro, si podemos enumerar los procesos que están corriendo actualmente $P_1, P_2, \ldots, P_n$ de modo tal de que P_1 tiene disponibles todos los recursos que necesita terminar, y para todo $i > 1$ se cumple que entre los recursos disponibles y los recursos que tienen tomados los procesos $P_1, \ldots, P_{i-1}$ tiene todos los recursos que necesita para terminar.

Un algoritmo de prevensión de deadlock, que implica conocer cuántos recursos de cada tipo va a necesitar cada proceso de antemano, consiste en sólo asignar recursos a los procesos si la asignación lleva a un estado seguro.

5.2.2. Algoritmo del banquero

El algoritmo del banquero es uno de los algoritmos más famosos de prevención de deadlock. Este algoritmo consiste en asegurarse, cada vez que entra un proceso, de preguntarle cuántos recursos de cada tipo va a necesitar como máximo, y sólo darle lugar a que se ejecute si, asignandole la máxima cantidad de recursos que necesita, se llega a un estado seguro.

5.2.3. Deteción de estados seguros

Hasta ahora venimos hablando de estados seguros pero no dijimos todavía cómo determinar si un estado es seguro. Supongamos que tenemos M recursos distintos, y N procesos. Entonces el algoritmo, que tiene complejidad $O(MN^2)$, consiste en los siguiente pasos:

1. Inicializar un vector booleano de longitud N en false, que indica que ningún proceso terminó.
2. Buscar un proceso que no haya terminado y tal que todos los recursos que necesita para terminar están disponibles. Si no existe dicho proceso ir al paso 4.
3. Marcar al proceso del paso 2 como terminado y liberar todos sus recursos. Volver al paso 2.
4. Si todos los procesos terminaron, el estado es seguro, caso contrario no lo es.

6. Memoria

6.1. Direcciones físicas vs. direcciones lógicas

La memoria puede verse como una tira de bytes consecutivos. Cada uno de estos bytes tiene una dirección (enteros consecutivos empezando desde el cero) mediante la cual se pueden acceder. Esta dirección se denomina dirección física.

Cada vez que ejecutamos un proceso, este accede a memoria para cargar su código, sus variables, y toda la demás información que necesite guardar. Si cada proceso decidiera utilizar direcciones específicas de la memoria, estaríamos en problemas cada vez que querramos cargar dos procesos que utilicen la misma dirección de memoria, ya que por ejemplo, el código de un proceso podría pisar variables del otro. Para resolver este problema tenemos la **Memory Management Unit (MMU)**, que es la unidad que se encarga de manejar la memoria. Cada vez que un proceso quiere acceder a memoria, utiliza lo que se denomina una dirección lógica (o virtual) y es la MMU la que se encarga de transformarla en una dirección física, asignandole una porción de la memoria física al proceso.

Cada proceso tiene su espacio de memoria y no puede acceder a memoria que no le corresponde. De esta manera evitamos que un proceso pueda acceder a memoria de otro proceso.

6.2. Swapping

Si tenemos muchos procesos ejecutandose en paralelo, y cada uno de estos procesos consume mucha memoria, puede pasarnos que no entren todos los procesos en memoria. Para solucionar esto, existe una técnica llamada swapping, que consiste en pasar parte de la memoria a disco, para poder cargar otros fragmentos en memoria.

El problema es que hacer swapping es lento, porque requiere accesos a disco que es mucho más lento que la memoria.

6.3. Asignación dinámica de la memoria

Cuando hacemos swapping, o también cuando cargamos procesos en memoria y cuando los descargamos luego de finalizada su ejecución, vamos ocupando espacios de memoria que no son contiguos, y la memoria libre queda fragmentada. Existen varias estrategias sobre cómo asignar bloques de memoria libre a los fragmentos que tenemos que cargar, algunos de ellos son:

- First fit: En el primer bloque de memoria libre en el que entre el fragmento.
- Best fit: En el bloque de memoria libre más chico en el que entre el fragmento.
- Worst fit: En el bloque de memoria libre más grande.
- Quick Fit: Se mantiene una lista de bloques libres de los tamaños más comunes. Por ejemplo, se mantiene la lista de los bloques de 4KB, de 8KB, de 12KB... Un bloque de 5KB puede ir a la lista de bloques de 4KB. De esta manera se puede hacer por ejemplo first fit en la lista que corresponda.

Existen dos métodos que son los más comunes para almacenar la información sobre los bloques libres y ocupados de la memoria. Uno de ellos son los bitmaps y el otro las listas enlazadas.

Cuando manejamos la memoria con bitmaps (mapas de bits) tenemos un bit por cada unidad de memoria que queremos asignar. Si por ejemplo, asignamos de a 32 bits, entonces $\frac{1}{33}$ de la memoria va a estar ocupada por el bitmap. Cada bit del bitmap es 0 si la unidad de memoria que representa está libre, y 1 si está ocupada. El problema que tiene este enfoque es que es bastante costoso encontrar bloques grandes de memoria. También

hay un trade-off entre el tamaño de la unidad mínima de asignación de memoria (que si es muy grande puede llevar a desperdiciar mucha memoria) y el tamaño del bitmap (que si asigna unidades muy chicas de memoria puede ocupar una porción significativa de la misma).

Manejar la memoria con listas enlazadas implica tener una lista de bloques, libre o reservados, en la que en cada campo de la lista se indica el tamaño del bloque, dónde empieza el bloque en memoria, y si está libre u ocupado. En este caso es fácil asignar memoria ya que si se hace al principio o al final de un bloque (es lo más común) sólo hay que dividir una entrada de la lista en dos, la parte que ocupamos y la parte que queda libre. Liberar memoria es un poco más complicado pero no tanto, sólo hay que tener en cuenta cuatro casos, que son las dos posibilidades para el bloque anterior (libre u ocupado) y las mismas dos posibilidades para el bloque siguiente. En caso de que el bloque anterior y/o el siguiente estén libres, entonces hay que mergear dos o tres entradas de la lista en una sóla entrada correspondiente a un bloque libre.

6.4. Fragmentación

Existen dos tipos de fragmentación de la memoria. Cuando vamos asignando bloques de memoria a los procesos, podría pasarnos que nos quedaran varios bloques libres no contiguos, que alcancen entre todos para satifacer un pedido, pero que ninguno tenga el tamaño necesario para satifacer el pedido en un sólo bloque. Esto se conoce como fragmentación externa. Si en cambio, un proceso pide por ejemplo 2000 bytes, y la memoria se asigna de a 1KB, entonces se le va a asignar 2KB de memoria al proceso, dejando 48 bytes asignados al proceso que quedan inutilizados. Esto se conoce como fragmentación interna.

Una posible solución al problema de la fragmentación es permitir que las direcciones lógicas no mapeen a direcciones físicas contiguas. Una implementación de esta solución es la paginación.

6.5. Paginación

La paginación consiste en dividir la memoria virtual (o lógica) en bloques de un tamaño fijo (generalmente 4KB) llamados páginas, y la memoria física en bloques del mismo tamaño llamados frames.

Cuando hay paginación, la MMU es la que se encarga de asignar páginas a frames, y las direcciones lógicas están compuestas por una parte que representa el número de página, y otra parte que representa el offset dentro de la página. Es por esto que las páginas siempre deben medir una cantidad de bytes que sean potencias de dos, para poder dividir la dirección en bits de número de página y bits de offset.

Con paginación las páginas que no están en memoria son guardadas en disco, si en el momento de pedir una página, esta no se encuentra en memoria, se produce una page fault, que es una excepción del sistema operativo, y la rutina de atención se encarga de cargar la página en memoria.

Para saber si una página está en memoria, y en caso de que esté, en qué frame está, la MMU guarda una tabla de páginas, con una entrada por cada página, en las que se guarda entre otras cosas, un bit que indica si la página está cargada en memoria, y el índice del frame en el que está cargada. Además se guardan bits que indican, por ejemplo, si la página fue modificada luego de ser cargada en memoria, para saber si guardarla en disco al desalojarla o simplemente descartarla y quedarse con la versión que ya había en disco.

Si por ejemplo tuviesemos 4GB de memoria, y páginas de 4KB, tendríamos una tabla de páginas con un millón de entradas. Además, la tabla de páginas es propia de cada proceso, es decir, si tenemos N procesos, tenemos N tablas de páginas, por lo que gran parte de la memoria estaría ocupada por tablas de páginas. Para solucionar este problema, se suele dividir las tablas en dos niveles: un directorio de tablas de páginas que apunta a tablas de páginas y una tabla de páginas por cada entrada del directorio de tablas de páginas. Así, si un proceso usa menos de 4MB de memoria, y las páginas ocupan 4KB, entonces podemos tener un directorio de tablas de páginas con 1024 entradas, una de ellas apuntando a una tabla de páginas con 1024 entradas, y podemos direccionar los 4MB (1024 páginas de 4KB) con tan solo 8KB (4KB del directorio de tablas y 4KB de la única tabla).

Una de las desventajas de paginación es que, al tener que convertir las direcciones virtuales en físicas, direccionar una página puede ser un poco lento, ya que requiere ir a buscar la página a la tabla de páginas, lo que implica un direccionamiento a memoria, y después buscar la página en el frame que corresponda, es decir, otro direccionamiento a memoria. El necesitar de dos direccionamientos a memoria (o tres en el caso de dos niveles de tablas) hace que obtener instrucciones o datos de la memoria sea más lento. La solución que se encontró a este problema es una pequeña caché que se denomina **Translation Lookaside Buffer (TLB)**. La TLB, también conocida como memroia asociativa, contiene el frame en el que se encuentran las páginas más usadas. Al buscar una página en memoria, lo primero que se hace es buscarla en el TLB, y en caso de que se encuentre presente en el mismo, el acceso a la página es mucho más rápido.

6.6. Reemplazo de páginas

Si implementamos paginación, nos ahorramos el problema de decidir en qué bloque de memoria libre cargar una página, ya que todos los bloques son frames del mismo tamaño (el tamaño de las páginas), pero en caso de que no haya suficiente memoria libre, debemos decidir de alguna manera qué páginas desalojar de memoria.

Veremos a continuación algunos algoritmos de reemplazo de páginas. Para todos estos algoritmos existen dos versiones, una donde se desalojan las páginas del mismo proceso, y otro donde se puede desalojar cualquier página de memoria.

Es importante utilizar un buen algoritmo de reemplazo de páginas para evitar el **trashing** (la situación en la que se pierde mucho tiempo swapeando páginas de memoria a disco).

6.6.1. Algoritmo Óptimo

El algoritmo óptimo de reemplazo de páginas consiste en desalojar de la memoria siempre la página que va a ser referenciada más tarde, o alguna página que no vaya a ser nunca más referenciada si existe. Este algoritmo es imposible de implementar pero se usa para comparar con los otros algoritmos a la hora de medir performance.

6.6.2. Not Recently Used

Las tablas de páginas suelen guardar dos bits R y M que indican si la página fue referenciada y/o modificada respectivamente desde el momento en el que se cargó en memoria. Ambos bits son generalmente seteados por hardware en cada acceso a memoria y el bit R es reseteado periódicamente por el sistema operativo. Las páginas se dividen en cuatro categorías según los bits R y M:

1. Ni referenciadas ni modificadas.
2. Modificadas pero no referenciadas (puede pasar cuando el bit R se resetea).
3. Referenciadas pero no modifcadas.
4. Referenciadas y modificadas.

El algoritmo Not Recently Used (NRU) desaloja de memoria una página al azar de la categoría más baja en la que haya al menos una página. La idea es que las páginas que fueron referenciadas recientemente son más propensas a ser referenciadas nuevamente, y para desempatar, es siempre más costoso desalojar una página modificada, ya que hay que cargarla a disco.

6.6.3. First In First Out

El algoritmo FIFO consiste en desalojar siempre la página que fue cargada en memoria primera entre todas las páginas que están cargadas en memoria. Es uno de los más sencillos pero es bastante ineficiente ya que puede ser que una página cargada hace mucho tiempo sea una de las más utilizadas.

6.6.4. Second Chance

Second Chance es parecido a FIFO, pero difiere en que revisa el bit R de la página. Si el bit R de la página más vieja es 0 entonces la desaloja, si en cambio es 1, lo resetea a 0 y la vuelve a encolar como si la página fuese recién cargada en memoria.

6.6.5. Clock

Es muy parecido a Second Chance, pero en vez de guardar una cola, guarda una lista circular, y mueve un puntero a lo largo de la lista. Si una página tiene el bit R en 1 entonces avanza el puntero a la próxima posición de la lista, y cuando encuentra una página con el bit R en 0, entonces la cambia por la nueva página y avanza también el puntero.

6.6.6. Least Recently Used

El algoritmo Least Recently Used desaloja siempre la página que fue usada hace más tiempo. En vez de guardar el momento en el que fue cargada en memoria, como en FIFO, guarda el momento en el que fue referenciada por última vez.

6.7. Segmentación

Tenemos dos problemas que surgen ahora: protección y reubicación. El primero podemos solucionarlo con una tabla de páginas para cada proceso, para que cada proceso pueda acceder sólo a sus páginas, pero todavía nos queda el segundo.

Una solución que existe para estos dos problemas es la segmentación. Un segmento es un espacio de memoria de tamaño variable, que es direccionado mediante un registro que hace referencia al principio del segmento.

A diferencia de paginación, donde las páginas son invisibles para el programador, los segmentos son visibles y el programa debe especificar a qué segmento hace referencia cuando quiere acceder a memoria.

Todavía tenemos con segmentación el problema de la fragmentación y el swapping. Por eso se suelen combinar la paginación con la segmentación.

Intel implementa segmentación en sus procesadores Pentium con una GDT (Global Descriptor Table) global y una LDT (Local Descriptor Table) para cada proceso. Los registros CS (code segment) y DS (data segment) se denominan selector de segmento y son registros de 16 bits que apuntan a una entrada de la GDT o de la LDT (13 bits para la entrada en la tabla, 1 para indicar a en qué tabla está el descriptor de segmento y 2 para permisos). Una vez cargado el descriptor del segmento, que nos dice donde empieza y cuánto mide el segmento, se obtiene por un registro mediante el cual se direcciona, el offset en el segmento y, chequeando previamente que el offset sea válido, se convierte la dirección lógica en dirección lineal (la que luego pasa por paginación).

A diferencia de las páginas los segmentos pueden solaparse, por ejemplo, cuando hay memoria compartida.

6.8. Copy on Write

Como vimos anteriormente, un proceso puede crear otros procesos mediante la system call $fork$. Cuando un proceso crea otro proceso, ambos comparten la memoria que tenía el proceso padre. Existen dos opciones para manejar la memoria en este caso: Una de ellas consiste en copiar toda la memoria y tener dos copias de los mismos datos y el mismo código en memoria desde el principio, y la otra es hacer la copia en algún momento entre que se crea el proceso y se necesita escribir la memoria.

Cuando el sistema operativo decide copiar la memoria al momento en el que uno de los dos proceso decide escribirla, se dice que estamos haciendo Copy on Write.

7. Archivos y Directorios

7.1. Definición de archivo

Para nosotros un archivo es una secuencia de bytes que son almacenados en una unidad de almacenamiento como puede ser por ejemplo un disco rígido. Todos los archivos tienen un nombre, y pueden tener una extensión (una o más letras después del último caracter '.' en el nombre del archivo).

Un archivo generalmente tiene los siguientes atributos:

- Nombre: El nombre del archivo, posiblemente con su extensión.
- Identificador: Un nombre que el usuario no ve, que sirve para identificarlo en el file system (más adelante veremos qué es un file system)
- Tipo: Puede ser ejecutable, archivo de texto plano, una imagen, etc.
- Ubicación: En qué dispositivo y qué directorio dentro del dispositivo se encuentra el archivo.
- Tamaño: El tamaño en bytes o en bloques del archivo.
- Permisos: Quién puede leer, escribir o ejecutar el archivo.
- Fecha, hora y usuario: Sirve para saber cuándo se creó o se modificó el archivo y quién fue el usuario que lo creó o modificó.

Un archivo es una unidad de almacenamiento lógica sobre la cuál se pueden realizar las siguientes operaciones:

- Crear un archivo: Primero se debe buscar espacio en el file system para el archivo, y luego se debe ubicar el archivo en el directorio correspondiente asignandole un nombre.
- Escribir un archivo: Para escribir un archivo se debe invocar una system call del sistema operativo indicando el nombre del archivo y el contenido que se desea escribir. Dado el nombre del archivo, el sistema operativo debe buscar el directorio donde se encuentra para ubicar el archivo en el file system.
- Leer un archivo: Para leer un archivo debemos usar una system call especificandole el nombre del archivo y dónde queremos que cargue el archivo en memoria.
- Borrar un archivo: Hay que buscar el archivo en el directorio correspondiente, liberar el espacio que ocupa el archivo y borrar la entrada correspondiente a ese archivo en el directorio.

7.2. Directorios

Los archivos en un file system están organizados en directorios. Existen varias formas de definir la estructura lógica de un directorio. A continuación veremos algunas de ellas.

7.2.1. Single Level Directory

Una opción no muy conveniente es tener todos los archivos en un sólo directorio. Esto puede traer problemas ya que si todos los archivos están en un mismo directorio, entonces no podemos tener dos archivos con el mismo nombre en el disco. De esta manera, los usuarios tienen que, por ejemplo, cuidarse de no usar nombres de archivos que hayan usado otros usuarios. Definitivamente no es la mejor opción.

7.2.2. Two Level Directory

Otra opción, que tampoco es la mejor, es tener un directorio con un subdirectorio para cada usuario. Esto soluciona el problema de nombrar archivos con un nombre que haya usado otro usuario para otro archivo, pero sigue teniendo el problema que no podemos tener dos archivos con un mismo nombre bajo un mismo usuario. También tiene el problema de que no se pueden compartir archivos salvo que permitamos que un usuario acceda al directorio de otro usuario.

7.2.3. Tree Structured Directory

Los directorios pueden ser vistos también como árboles. Existe un directorio al que llamamos root (o raiz, que es la raiz del árbol), y cada uno de los demás directorio está contenido en otro directorio, formando así un árbol. Con esta estructura, cada directorio puede contener archivos y subdirectorios.

Cada archivo tiene un nombre y un path. El path es el camino en el árbol del directorio raiz hasta el directorio donde se encuentra el archivo. Podemos considerar al path seguido por el nombre del archivo como el nombre real del archivo para el sistema operativo. Por ejemplo, si el archivo se llama hola.txt y su path es /abc/def/ghi, entonces podemos considerar/abc/def/ghi/hola.txt como el nombre del archivo.

Los directorios son considerados un tipo de archivo especial, cada entrada en la tabla del directorio (la que contiene la lista de archivos del directorio) tiene un bit que indica si el archivo que corresponde a esa entrada se trata de un directorio o si es otro tipo de archivo.

7.2.4. Acyclic Graph Directories

Otra opción es mantener un grafo dirigido acíclico (DAG, por sus siglas en inglés) de directorios. Esto se logra agregandole a la versión de árbol de directorios la posibilidad de tener links de un directorio a otro, haciendo que un archivo o directorio pueda estar en dos directorios al mismo tiempo.

Un problema que surge ahora es cómo recorrer los directorios, ya que no nos gustaría recorrer dos veces el mismo directorio. Otro problema es cuándo borrar un archivo o directorio, la solución a esto es que cuando se borra un link no se borre el directorio original.

7.2.5. General Graph Directories

Otra opción que tenemos es agregar links y que el grafo ya no tenga porqué ser acíclico, en este caso hay varias consideraciones que tener en cuenta a la hora de borrar archivos o recorrer el file system.

8. File systems

8.1. ¿Qué es un file system?

Los file systems (o sistemas de archivos) son los encargados de darle forma al contenido de un disco, y de organizar los archivos que hay en el mismo. Los discos están divididos en sectores. El sector 0 de un disco se denomina **Master Boot Record (MBR)** y es la parte del disco que se ejecuta al prender una computadora.

Sobre el final del MBR se encuentra la tabla de particiones. Esta tabla dice dónde empieza y dónde termina cada partición, y además indica qué partición se debe ejecutar desde su primer bloque, llamado boot block.

8.2. Implementación de file systems

Lo más importante a la hora de implementar un file system es saber qué bloques del disco se corresponden con qué archivos. Hay varias formas de implementar esto, veremos a continuación algunas de ellas.

8.2.1. Almacenamiento continuo

Una forma de implementar un file system es almacenando los archivos en bloques continuos. De esta manera sólo es necesario recordar, para cada archivo, en qué bloque comienza el archivo y cuántos bloques ocupa. También logramos así una alta performance ya que sólo hay que buscar un bloque una vez (el primer bloque) y de ahí en más no es necesario buscar más bloques ya que son todos consecutivos.

Lamentablemente esta implementación, que es super eficiente a la hora de leer archivos en el disco, es imposible de implementar en discos de lectura/escritura como son por ejemplo los discos rígidos, ya que cuando el disco se empieza a llenar y empezamos a borrar archivos, empiezan a quedar espacios libres en el medio del disco. Además, una vez creado un archivo, si queremos seguir agrandando el archivo (por ejemplo, un documento de texto al que le queremos agregar dos páginas) no podemos ya que nos vamos a chocar con el archivo siguiente.

Afortunadamente, esta implementación que es eficiente y sencilla de implementar, está presente en algunas unidades de almacenamiento como lo son por ejemplo los CD-ROMs.

8.2.2. Almacenamiento con listas enlazadas

Otra alternativa es guardar los bloques como listas enlazadas. Sólo guardamos en el file system el puntero al primer bloque, y después en cada bloque guardamos un puntero al siguiente bloque del archivo. Una desventaja que tiene esta implementación es que para buscar el n-ésimo bloque hay que recorrer los primeros $n - 1$ bloques.

8.2.3. Tablas FAT

Una alternativa al almacenamiento con listas enlazadas es lo que se conoce como **File Allocation Table (FAT)**, que es una tabla que, para cada bloque guarda el número del siguiente bloque del archivo al que pertenece ese bloque. Para el último bloque del archivo guarda un -1 indicando que terminó el archivo en ese bloque.

Una ventaja que tiene el uso de tablas FAT es que el acceso aleatorio es mucho más eficiente, ya que al estar todo el bloque ocupado con datos, y no tener los punteros al bloque siguiente, calcular el offset en un archivo se vuelve mucho más rápido ya que los tamaños de los bloques son potencias de 2, y siempre es más fácil calcular un offset cuando tenemos que saltear n bloques que contienen una cantidad de datos potencia de 2.

Una desventaja que tiene la FAT es que como toda la tabla tiene que estar en memoria, esta ocupa demasiado espacio en la misma. Por ejemplo, con un disco de 200GB, bloques de 1KB, y entradas de 4 bytes, la tabla ocuparía 800MB de memoria.

8.2.4. i-nodos

Los i-nodos (index node) son estructuras de datos que continen los atributos de un archivo y links a los bloques en los que está almacenado el archivo. La ventaja de los i-nodos sobre las tablas FAT es que los i-nodos (uno por archivo) sólo están presentes en memoria cuando el archivo está cargado en memoria, por lo que podemos tener un disco arbitrariamente grande que no vamos a tener que tener una cantidad proporcional de i-nodos cargados en memoria.

El problema surge cuando hay más bloques que la capacidad de los i-nodos. Por ejemplo, si un archivo tiene 1000 bloques y un i-nodo tiene capacidad para direccionar a 512 bloques, entonces un i-nodo no va a servir en este caso. Para solucionar este problema, algunas implementaciones de i-nodos reservan las últimas entradas para apuntar a bloques que a su vez apuntan a bloques del archivo.

La implementación de UNIX de los i-nodos tiene tres entradas reservadas en cada i-nodo, que son las últimas tres. Las primeras 12 entradas en el i-nodo apuntan a bloques del archivo (en realidad antes de estas 12 entradas hay espacio para atributos del archivo, por lo que estas primeras 12 entradas no son lo primero que hay en el i-nodo). La entrada número 13 apunta a un bloque que tiene 2048 entradas (recordemos que los bloques en UNIX son de 8KB por lo que 2048 entradas representa entradas de 32 bits) que apuntan a bloques del archivo, esto nos suma 16MB a los 96KB que ya podía tener un archivo con las primeras 12 entradas. La entrada 14 apunta a un bloque que a su vez apunta en cada entrada a bloques que apuntan a bloques del archivo, esto nos da la posibilidad de 32GB. A su vez, la entrada 15 permite triple indireccionamiento (bloques que apuntan a bloques que apuntan a bloques que apuntan a bloques del archivo!!!!) lo que permite archivos de hasta 64TB!

8.3. Implementando directorios

El árbol de directorios en UNIX se implementa con un i-nodo especial que apunta al root directory, y en cada directorio hay un bloque que tiene una lista de pares (nombre de archivo,i-nodo), donde el archivo puede ser a su vez un subdirectorio, que le indica cuál es el i-nodo correspondiente a cada uno de los archivos en ese directorio (los archivos, nuevamente, pueden ser directorios). En algunos casos, cuando los directorios tienen muchos archivos, en lugar de guardar los nombres de los archivos con sus i-nodos correspondientes uno por uno, se utiliza una tabla de hash que hashea los nombres de los archivos y nos da los pares de (nombre de archivo,i-nodo) que matchean ese hash.

Generalmente los i-nodos se encuentran después del boot block y del superblock (un bloque que contiene información sobre el file system como por ejemplo qué file system es y cuántos i-nodos tiene). Después de todos los i-nodos vienen los bloques del disco.

8.4. Atributos

Cada archivo tiene sus atributos, que en el caso de los i-nodos, se guardan en los primeros bytes de cada i-nodo. Algunos de estos atributos son:

- Permisos.
- Tamaño.
- Propietario(s).

- Fechas de creación, modificación y acceso.
- Flags.
- CRC (bytes que sirven para verificar que el archivo no esté dañado).

8.5. Journaling

Para evitar escrituras constantes en el disco podríamos tener una caché (una copia de bloques de disco en memoria) que se maneje de modo similar a las páginas. Un problema que puede tener esto es por ejemplo que se corte la energía eléctrica antes de que se bajen los datos de memoria a disco.

Una alternativa a este problema es lo que se conoce como journaling. Algunos file systems tienen un log o journal (de ahí el nombre journaling) que es un registro de cambios que hay que hacer en el disco. Eso se graba en un buffer circular en disco, y es mucho más rápido escribir en este buffer que en bloques aleatorios ya que la escritura es secuencial. Estos cambios se van actualizando en el disco, y si por ejemplo, se llena el buffer, se actualizan todos los cambios para tener lugar nuevamente en el buffer. Cuando el sistema se levanta luego de un apagado inesperado estos cambios se realizan en el disco.

8.6. Bloques libres

Un problema importante a resolver por el file system es cómo mantener la lista de bloques libres. Hay dos approachs a este problema que son los más comunes. Uno es mantener un bitmap con cada bit en 0 si el bloque está libre y en 1 si el bloque está ocupado. Por ejemplo con bloques de 8KB, y un disco de 500GB, necesitamos alrededor de 1000 bloques para almacenar el bitmap.

La otra posibilidad es mantener una lista de bloques libres, en los mismos bloques libres. Guardamos un puntero al primer bloque libre, y en ese bloque libre guardamos una lista de bloques libres, reservando una entrada para el próximo bloque donde continúa la lista. La ventaja de este approach es que los bloques libres se almacenan en bloques libres, y si tenemos menos bloques libres (el disco casi lleno) entonces la lista de bloques libres es más chica y entra en los pocos bloques libres que tenemos.

Una optimización a esta última implementación es guardar un par que tenga un bloque libre, y la cantidad de bloques libres a partir de ese bloque. Esta optimización, sin embargo, se vuelve ineficiente cuando el disco se empieza a fragmentar demasiado.

8.7. Network file systems

Existen file systems para sistemas distribuidos, en donde un file system reside en un servidor y varios clientes pueden acceder a los archivos del file system. Un ejemplo de un network file system es NFS. NFS permite montar directorios de un file system remoto en un file system local para permitir el acceso a los archivos como si fuesen parte del file system local. Pese a ser muy sencillo, NFS no escala bien.

9. Entrada/Salida

9.1. Drivers

Los dispositivos de entrada salida necesitan una forma de comunicarse con la CPU. Por ejemplo, un mouse necesita saber decirle al procesador que el usuario hizo un click o un desplazamiento hacia la derecha. Para esta comunicación existen módulos de software muy específicos llamados drivers. Existen varias formas en las que los drivers pueden comunicarse con los dispositivos para saber cuándo los dispositivos terminaron

de hacer algo (por ejemplo, cuando el usuario movio el mouse, cuando el disco terminó de escribir un archivo, o cuándo una impresora terminó de imprimir un documento). Veremos a continuación tres de estas formas.

9.1.1. Polling

El driver le pregunta periódicamente al dispositivo si hay novedades. Es una de las formas más sencillas de comunicar un dispositivo con un driver, aunque es poco eficiente porque consume mucha CPU. En algunos casos muy puntuales puede ser una buena idea ya que si bien consume mucha CPU, es más eficiente que lo que consumen las interrupciones con los cambios de contexto.

9.1.2. Interrupciones

El dispositivo le avisa al controlador de interrupciones que terminó de hacer su tarea. Este a su vez, decide cuándo atender la interrupción y avisarle al driver que tiene trabajo que hacer. Cuando interviene la CPU estamos ante una E/S programada (Programmed I/O).

9.1.3. DMA

Se requiere de un componente de Hardware, el controlador de DMA. Este se encarga de comunicarse con los dispositivos y le avisa a la CPU por medio de una interrupción cuando ya terminó de realizar la entrada/salida.

9.2. Subsistema de E/S

El manejador de entrada/salida le provee al programador una API sencilla para comunicarse con los dispositivos. Sin embargo, hay cosas que no se le pueden ocultar a los procesos. Estos deben saber por ejemplo si tienen acceso exclusivo a un dispositivo. Esta responsabilidad es compartida entre el manejador de E/S y los drivers.

Una de las tareas del sistema operativo es ocultar las particularidades de cada dispositivo brindando una API genérica para acceder a todos los dispositivos del mismo tipo. Los dispositivos pueden ser de lectura, escritura o ambas, pueden ser dedicados o compartidos, y pueden tener distintas velocidades de respuesta.

Existen dos tipos de dispositivos de entrada/salida.

9.2.1. Char devices

Los char devices son dispositivos que transmiten la información byte a byte. Estos dispositivos no soportan acceso aleatorio y no utilizan caché. Algunos ejemplos son mouses, teclados, etc.

9.2.2. Block devices

Los block devices son dispositivos en los que la información se transmite por bloques. Estos dispositivos soportan acceso aleatorio y generalmente utilizan un buffer (caché).

9.3. Discos

Una de las claves para obtener un buen rendimiento de E/S es manejar apropiadamente el disco. El disco tiene una cabeza que se mueve y moverlo consume tiempo. Los pedidos de acceso a disco llegan constantemente y es importante atenderlos de modo tal de minimizar estos movimientos pero sin generar inanición.

El tiempo necesario para que el disco rote y la cabeza quede sobre el sector deseado se llama latencia rotacional, pero lo más importante, es el tiempo de búsqueda (seek time), que es el tiempo necesario para que la cabeza se ubique sobre el cilindro que tiene el sector buscado.

Existen varias políticas de scheduling de acceso a disco, algunas de las cuales optimizan el seek time para hacer los accesos a disco más rápidos.

9.3.1. First Come First Serve

Esta política es la más simple, y consiste en atender los pedidos en el orden en el que van llegando. Si bien es la política más simple porque no requiere nada más que guardar los pedidos en una cola FIFO, suele ser muy ineficiente.

9.3.2. Shortest Seek Time First

La política SSTF consiste en atender siempre el pedido que involucre el menor seek time. Esta política es un poco más eficiente que FCFS pero puede generar inanición. Al igual que FCFS no es una política muy buena y está lejos de ser óptima.

9.3.3. SCAN

El algoritmo SCAN (o algoritmo del ascensor) recorre el disco de una punta a la otra ida y vuelta, atendiendo los pedidos en orden. Si llega un pedido por el cilindro por el que acaba de pasar la cabeza tiene que esperar a que llegue al final y vuelva. Igualmente es mucho más eficiente que los anteriores porque optimiza un poco más que FCFS el uso de la cabeza y no genera inanición.

9.3.4. Circular SCAN

Circular SCAN (o C-SCAN) funciona como SCAN, con la diferencia de que recorre el disco de una punta a la otra, y cuando llega al extremo final vuelve al extremo inicial sin atender recorridos en el camino.

9.3.5. Look

El algoritmo Look funciona como SCAN pero con la diferencia de que si ve que no tiene pedidos en lo que le queda por recorrer del disco, empieza el camino hacia el otro lado atendiendo pedidos sin ir hasta el fondo y volver. También existe la versión Circular Look (o C-Look).

En la vida real no se utiliza ninguno de estos algoritmos sino que se agregan prioridades.

9.4. Gestión del Disco

Al formatear los discos se ponen unos códigos en cada sector del disco que sirven a la controladora para detectar y corregir errores. Fucionan como un prefijo y un postfijo de la parte donde van los datos en cada sector. Si al leer el sector, estos códigos no tienen el valor que deberían el sector está dañado.

Los discos suelen tener una sección en ROM que carga a memoria algunos sectores del disco y los comienza a ejecutar. Este programa no es parte del sistema operativo sino que es un programa que carga el sistema operativo.

Los bloques dañados se manejan a veces por software y el file system es el responsable de tenerlos anotados, por ejemplo en FAT.

Los discos SCSI vienen con sectores extra para reemplazar a los defectuosos. La controladora es la encargada de reemplazar un sector defectuoso por uno de los sectores extra actualizando una tabla interna. Para no interferir con el scheduler de E/S los discos suelen traer sectores extra en todos los cilindros.

9.5. Backup

Hay formas de proteger la información, algunas más seguras y otras menos seguras. La política MSSVR (mirá si se va a romper!) suele ser una de las menos efectivas. Hay otras formas de proteger la información, como por ejemplo, haciendo backup.

Se suele hacer en cintas y hay tres tipos de backups que son los más usados:

- Copias totales: Se copia toda la información del disco.
- Copias incrementales: Se copian todos los archivos modificados desde la última copia total o incremental.
- Copias diferenciales: Se copian todos los archivos modificados desde la última copia total.

9.6. RAID

A veces hacer backup no alcanza, se puede romper un disco justo antes de hacer el backup y perdemos toda la información desde el último backup. Para evitar ese problema se suele utilizar RAID (Redundant Array of Inexpensive Disks). La idea básicamente es copiar la información en más de un disco para que si se me rompe uno siga teniendo el otro. Tiene algunas ventajas como que puedo hacer lecturas en simultaneo de los dos discos pero es muy costoso escribir en dos discos a la vez. Además, tener dos copias de toda la información puede requerir demasiado espacio en disco. Existen varios tipos de RAID que balancean todas estas ventajas y desventajas:

- RAID 0 (stripping): No aporta redundancia. Consiste en tener la mitad de la información en un disco y la mitad en otro. Permite escrituras en simultaneo si los discos están en diferentes controladoras.
- RAID 1 (mirroring): Se mantienen dos copias del mismo disco. Es costoso pero soporta la caida de un disco.
- RAID 0+1: Es un RAID 1 de dos RAID 0. El RAID 1 provee redundancia y el RAID 0 rendimiento. Es costoso al igual que RAID 1.
- RAID 1+0: Es un RAID 0 de dos RAID 1. Es mejor que RAID 0+1 ya que si se cae un disco en un RAID 0+1 todo el RAID 0 queda inutilizable, en cambio si se cae un disco en un RAID 1+0 sólo ese disco queda inutilizable. Si se caen dos discos en un RAID 0+1 la probabilidad de que todos los discos queden inutilizables es $\frac{2}{3}$ mientras que en un RAID 1+0 es $\frac{1}{3}$. Es igual de costoso que RAID 0+1.
- RAID 2: Requiere 3 discos de paridad por cada 4 de datos. Es más lento que RAID 1. Cada bloque lógico se distribuye entre todos los discos.
- RAID 3: Requiere 1 disco de paridad por cada 4 de datos. Es más eficiente que RAID 2. Al igual que en RAID 2, cada bloque lógico se distribuye entre todos los discos.
- RAID 4: Es parecido a RAID 3 pero hace el stripping a nivel de bloque.
- RAID 5: En vez de tener un disco de paridad, distribuye la paridad entre todos los discos. Esto hace que no haya un cuello de botella en el disco de paridad.
- RAID 6: Es como RAID 5 pero utilizando dos bloques de paridad. Los dos bloques de paridad usan paridades distintas, uno de ellos la paridad común y corriente y el otro algún código de correción de errores como por ejemplo el código de Reed-Solomon.

RAID no proteje contra cosas como borrar accidentalmente un archivo, por eso se suele combinar RAID con backups.

9.7. Spooling

Hay dispositivos que requieren acceso dedicado, como por ejemplo las impresoras. Si mandamos a imprimir dos documentos no nos gustaría que se impriman las líneas de los dos documentos alternadamente sino que queremos imprimir un documento y después imprimir el otro. Tampoco queremos que el proceso que mandó a imprimir se bloquee esperando que terminen todas las demás impresiones. Para eso podemos poner todos los trabajos en una cola y un proceso se encarga de ir desencolándolos. El sistema operativo nunca se entera que está haciendo spooling, los usuarios sí, y pueden acceder a la cola, por ejemplo, para cancelar la impresión de un documento.

10. Sistemas distribuidos

10.1. ¿Qué son los sistemas distribuidos?

Existen tres conceptos que son similares pero no iguales:

- Cómputo simultáneo: Un scheduler asigna un quantum a cada proceso. Los procesos se ejecutan simultaneamente.
- Cómputo paralelo: Más de una CPU en una misma máquina.
- Cómputo distribuido: Varios procesadores que no comparten memoria, clock, etc.

Hay cuatro razones importantes por las que los sistemas distribuidos pueden ser una buena opción:

- Compartir recursos: A veces es muy útil poder compartir los recursos entre varias máquinas (desde archivos en un disco hasta una impresora).
- Velocidad de cómputo: En sistemas distribuidos podemos aprovechar el uso de procesadores en máquinas distintas para mejorar la velocidad de cómputo.
- Confiabilidad: Un sistema distribuido puede soportar la falla de una de sus máquinas permitiendo que otras máquinas se ocupen de las tareas de las que estaba encargada la máquina que falló.
- Comunicación: La comunicación entre procesos en distintas máquinas de un sistema distribuido puede ser muy útil.

La taxonomía de Flynn divide a los sistemas en SISD, SIMD, MISD, MIMD (Single/Multiple Instruction Single/Multiple Data).

Los sistemas distribuidos comparten el scheduler y un canal de comunicaciones, pero no la memoria ni el clock.

Es importante la manera en la que los procesos se comunican entre sí en un sistema distribuido. La forma en la que se estructuran los procesos para que cooperen entre sí se llama arquitectura de software. El qué comparten y qué no (memoria, scheduler, clock, etc) se llama arquitectura de hardware.

10.2. Arquitecturas de Hardware

Existen varias arquitecturas de hardware, a continuación analizaremos algunas de ellas.

- Symetric Multi Processing (SMP): consiste en varios procesadores compartiendo la memoria. Los procesadores suelen ser iguales.

- Multicore: Más de un procesador, compartiendo caché L2.
- NUMA: Cada procesador tiene su memoria local, aunque puede, de manera más lenta, acceder a la memoria de los otros procesadores.
- Redes: Es el nombre que se le suele dar cuando son un conjunto de computadoras independientes. Hasta hace un tiempo se utilizaba el término NOW (Network of Workstations).
- Clusters: Un conjunto de computadoras conectadas por una red de alta velocidad que comparten el scheduler.

Un grid es un conjunto de clusters, cada uno bajo un dominio administrativo distinto, generalmente alejados geográficamente.

Cloud computing es el término que se utiliza para clusters donde uno puede alquilar poder de cómputo.

10.3. Arquitecturas de Software

Telnet es un protocolo (y un programa que utiliza el protocolo) para conectarse a un equipo remotamente. Con telnet los recursos necesarios para cierta parte del procesamiento están en un equipo remoto. Telnet existe desde 1969.

RPC (Remote Procedure Call) es un mecanismo que permite hacer llamadas a subrutinas que se encuentran en otro equipo remotamente.

Tanto Telnet como RPC implican una máquina activa que procesa y una máquina pasiva que manda a procesar. Estas arquitecturas se suelen denominar cliente/servidor. El servidor es un componente que da servicios cuando el cliente se lo pide. El programa principal hace de cliente de los servicios que va necesitando para completar su tarea.

La conjetura de Brewer dice que en un entorno distribuido sólo se pueden garantizar dos de las siguientes tres: consistencia, disponibilidead y tolerancia a fallos.

10.4. Locks en sistemas distribuidos

En sistemas distribuidos no hay TestAndSet atómicos, entonces, ¿cómo implementamos los locks?. Una de las soluciones más sencillas es designar un nodo para que se encargue de administrar los recursos, y crear procesos en ese nodo (proxies) que representen a los procesos remotos, teniendo que pedirle los procesos remotos a los proxies que negocien los recursos por ellos.

Este enfoque centralizado tiene varios problemas. Uno de ellos es el cuello de botella. Otro problema es que ante la falla del servidor se cae todo el sistema.