SO: Notas Primer Parcial

Galileo Cappella

2c 2022

1 Inter Process Communication

1.1 Signals

- `int kill(pid_t pid, int sig)`
- `void* signal(int signum, void *handler)`
- `pid_t fork(void)`
- `int execv(const char *pathname, char *const argv[])`
- `pid_t wait(int *wstatus)`
- `pid_t waitpid(pid_t pid, int *wstatus, int options)`
- `pid_t waitpid(pid_t pid, int *wstatus, int options)`
- `uint sleep(uint seconds)`

1.2 Pipes

- `int pipe(int fildes[2])`
- `int close(int fd)`
- `ssize_t write(int fd, const void *buf, size_t count)`
- `ssize_t read(int fd, const void *buf, size_t count)`

1.3 Sockets

- **Server** `socket()` → `bind()` → `listen()` → `accept()`
- **Client** `socket()` → `skip` → `skip` → `connect()`
- `int socket(int domain, int type, int protocol)`, crea un nuevo socket.
- `int bind(int fd, sockaddr* a, socklen_t len)`, asigna una dirección (nombre o IP y puerto) al socket.
- `int listen(int fd, int backlog)`, setea al socket como pasivo que recibirá conexiones. Se maneja una cola para poder recibir varias conexiones.
- `int accept(int fd, sockaddr* a, socklen_t* len)`, extrae de la cola una solicitud de conexión y establece la comunicación entre sockets. Se bloquea en caso de no existir conexioens pendientes. Devuelve un nuevo fd para conexión.

- `int connect(int fd, sockaddr* a, socklen_t* len)`, se conecta a un socket remoto que debe estar escuchando.
- `ssize_t send(int s, void *buf, size_t len, int flags)`
- `ssize_t recv(int s, void *buf, size_t len, int flags)`

2 Scheduling

2.1 Métodos

- **First In, First Out (FIFO)** o **First Come, First Served (FCFS)**.
- **Shortest Job First (SJF)**.
- **Shortest Remaining Time First (SRTF)** = SJF + tiempos de llegada y desalojo.
- **Priority Scheduling (PS)**.
- **Round Robin (RR)**, single/multiple queue.
- **Multilevel Queue Scheduling (MQS)**.
- **Multilevel Feedback Queue Scheduling (MFQS)**, los procesos van llegando a la cola más baja, si no termina en el tiempo dado se mueve a la siguiente cola. Se corre en órden, $\text{queue}_1 \to \text{queue}_2 \to \ldots \to \text{queue}_n \to \text{queue}_1$.
- **Earliest Deadline First (EDF)** para *Real Time (RT)* systems.
- **Linux CFS**, usa un árbol rojo-negro ordenado por *vruntime*, y prioriza la de menor valor.

2.2 Misc.

- **Diagrama de Gantt** muestra el tiempo que ocupan los procesos.
- **CPU burst (ráfaga)**, tiempo de CPU que necesita un proceso.
- El scheduler toma decisiones cuando:
  1. Un proceso cambia de estado running a waiting.
  2. Un proceso cambia de estado running a ready.
  3. Un proceso cambia de estado waiting a ready.
  4. Un proceso termina.

  Sólo considerar 1 y 4 lo hace **nonpreemtive**, y las 4 es **preemptive**.
- El **Dispatcher** da el control al proceso seleccionado luego de:
  1. Context switch.
  2. Cambiar a modo usuario.
  3. Saltar a la ubicación correcta en el programa.
- **Turnaround** = tiempo de finalización - tiempo de llegada.
- **Waiting Time** = turn around - CPU burst.
- Determinar la longitud del siguiente **CPU Burst**: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$ con t_n longitud real, τ_n longitud estimada, $0 \leq \alpha \leq 1$ usualmente $\frac{1}{2}$

3 Sync

- **Race condition** es cuando el resultado depende del orden en que se ejecuten las cosas.
- **Sección crítica** es un cacho de código que sólo hay un proceso ejecutándolo a la vez, y cualquiera que quiera ejecutarlo espera a que se libere.
- Una operación **Atómica** es indivisible a nivel CPU.
- **TestAndSet (TAS)** es una instrucción atómica que cambia el valor de una variable y devuelve el valor que tenía. `while(testAndSet(x)) {}` hace **busy waiting** hasta poder usarla.
- Un **Semáforo** es una variable entera que sólo se la puede manipular con dos operaciones: `s.wait()` que espera hasta que se mande una signal, y `s.signal(n)` que despierta a n procesos esperando en wait. Nos ahorra del busy waiting.
- El **Mutex** es un tipo de semáforo binario.
- El **Spin lock (o TASLock)** hace busy waiting `(TASLock) mtx.lock()`, y luego libera con `mtx.unlock()`. Puede tener menos overhead que un semáforo.
- El **Local spinning (o TTASLock)** tiene menos overhead, hace busy waiting
```
lock() {
  while (true) {
    while(mtx.get()) {};
    if (!mtx.testAndSet()) return;
  }
}.
```
- **Registros Read-Modify-Write** atómicos: `int getAndInc()`, `int getAndAdd(int)`, `T compareAndSw`
`u, T v) { if (u == reg) reg = v }`
- **Cola atómica**:
```
enqueue(T item) { mtx.lock(); q.push(item); mtx.unlock(); }
bool dequeue(T *pitem) {   mtx.lock();
  bool success = !q.empty();
  if (success) pitem = q.pop();
  else pitem = null;
  mtx.unlock();
  return success;
}
```
- **Mutex recursivo** se salva de deadlock por recursión:
```
void create() { owner.set(-1); calls = 0; }
void lock() {
  if (owner.get() != self)
    while(owner.compareAndSwap(-1, self) != self) {}
  calls++;
}
void unlock() { if (- -calls == 0) owner.set(-1); }
```
- Las **Condiciones de Coffman** para un deadlock, todas son necesarias:

utual exclusion Un recurso está en un modo no-compartido.

Hold and wait Un proceso tiene al menos un recurso y está pidiendo más recursos que están siendo tenidos por otros procesos.

No preemtion Un recurso sólo puede ser liberado voluntariamente por el proceso que lo tiene.

Circular wait Hay un ciclo de esperas.

4 Memory Management

4.1 Remoción

- **First In, First Out (FIFO)**
- **Least Recently Used (LRU)**
- **Segunda Oportunidad**, si fue referenciada le doy otra oportunidad
- **Not Recently Used**, primero desalojo las que no fueron referenciadas ni modificadas, luego las referenciadas, y luego las modificadas.

4.2 Allocation

- **First fit**
- **Best fit**, agarro el bloque más chico posible.
- **Worst fit**, agarro el bloque más grande posible.
- **Quick fit**, se usan listas de bloques de tamaño fijo.

5 Ejemplos

5.1 IPC

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdbool.h>
#include <time.h>
#include <sys/wait.h>
int pipes[N][2], pidHijos[N], hijo;
void crearPipes() {
  for (int i = 0; i < N; ++i) pipe(pipes[i]);
}
void cerrarPipesParaPadre() {
  for (int i = 0; i < N; ++i) close(pipes[i][1]);
}
void cerrarPipesParaHijo() {
  for (int i = 0; i < N; ++i)
    if (i != hijo) {
      close(pipes[i][0]);
      close(pipes[i][1]);
    }
  close(pipes[hijo][0]);
}
void handler() {
  printf("Gane!\n");
  exit(EXIT_SUCCESS);
}
void ejecutarHijo() {
  signal(SIGTERM, handler);
  srand(getpid());
  cerrarPipesParaHijo();
  while (1) {
    int numero = rand() % 100;
    write(pipes[hijo][1], &numero, sizeof(numero));
    sleep(1);
```

```
  }
}
void crearHijos() {
  for (hijo = 0; hijo < N; hijo++) {
    pidHijos[hijo] = fork();
    if (pidHijos[hijo] == 0)
      ejecutarHijo();
  }
}
void jugarConHijos() {
  int numeroSecreto = rand() % 50, hijosVivos = N, i = 0;
  while (hijosVivos > 1) {
    if (pidHijos[i] != 0) {
      int numero;
      read(pipes[i][0], &numero, sizeof(numero));
      if(numero > numeroSecreto) {
        hijosVivos--;
        kill(pidHijos[i], SIGKILL);
        waitpid(pidHijos[i], NULL, 0);
        pidHijos[i] = 0;
      }
    }
    i = (i+1) % N;
  }
  while(pidHijos[i] == 0) {
    i = (i+1) % N;
  }
  kill(pidHijos[i], SIGTERM);
}
int main(void) {
  srand(time(NULL));
  crearPipes();
  crearHijos();
  cerrarPipesParaPadre();
  jugarConHijos();
  return 0;
}
```

5.2 Sched

t	Proceso	ejecución (t)	Burst restante (t)
0-1ms	P4	1ms	2ms
1-2ms	P4	1ms	1ms
	P3	0ms	8ms
2-3ms	P4	1ms	0ms
	P3	0ms	8ms
	P1	0ms	6ms
3-4ms	P3	0ms	8ms
	P1	1ms	5ms
4-5ms	P3	0ms	8ms
	P1	0ms	5ms
	P5	1ms	4ms
5-7ms	P3	0ms	8ms
	P1	0ms	5ms
	P5	0ms	4ms
	P2	2ms	0ms
7-10ms	P3	0ms	8ms
	P1	0ms	5ms
	P5	4ms	0ms
10-15ms	P3	0ms	8ms
	P1	5ms	0ms
15-23ms	P3	8ms	0ms

Proceso	Llegada	CPU Burst	Finalización	Turnaround	Waiting
P1	2ms	6ms	15ms	15ms - 2ms = 13ms	15ms - 6ms = 7ms
P2	5ms	2ms	7ms	7ms - 5ms = 2ms	2ms - 2ms = 0ms
P3	1ms	8ms	23ms	23ms - 1ms = 22ms	22ms - 8ms = 14ms
P4	0ms	3ms	3ms	3ms - 0ms = 3ms	3ms - 3ms = 0ms
P5	4ms	4ms	10ms	10ms - 4ms = 6ms	6ms - 4ms = 2ms
Promedio	-	-	-	9.2ms	4.6ms

5.3 Sync

```
//*************************************************************
//S: Barrier
int n = 0, N;
semaphore mtx(1), step[2] = {0, 0};
void wait(void) {
  mtx.wait();
  if (++n == N) step[0].signal(N); //A: Dejo pasar a N
  mtx.signal();
  step[0].wait();
  mtx.wait();
  if (--n == 0) step[1].signal(N);
  mtx.signal();
  step[1].wait();
}
//*************************************************************
//S: Programa
SHARED:
  semaphore permisoLobos(0), permisoCabras(0),
            mtx(1);
  barrier barrera(4);
LOCAL:
  bool remero = false;
```

```
void lobo() { //NOTA: Las cabras son similares
  mtx.wait();
  if (++lobos == 4) {
    lobos = 0;
    permisoLobos.signal(4);
    remero = true;
  } else if (lobos == 2 && cabras => 2) {
    lobos = 0;
    cabras -= 2;
    permisoLobos.signal(2);
    permisoCabras.signal(2);
    remero = true;
  } else mtx.signal();
  permisoLobos.wait();
  abordar();
  barrera.wait();
  if (remero) {
    remar(); remero = false;
    mtx.signal();
  }
}
```