Ingeniería Informática. Curso 3º.
Sistemas Operativos
Examen Final. TEORIA.
4 de Septiembre de 2009

1. **[PROCESOS]**

a) Considerar el siguiente código:

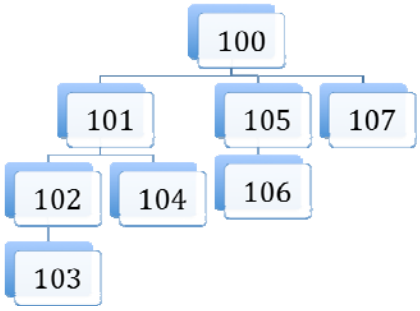
```
void main() {
  int j=10; pid_t pid;

  for (int i=0; i < 3; i++) {
     printf("i=%d, j=%d\n", i, j);
     pid = fork()
     if ((pid %2) == 1)j--;
  }
}
```

Dibujar el árbol de procesos resultante e indicar qué mensajes imprime cada proceso. Asumir que la **prioridad es para el proceso hijo**. El pid del proceso inicial es 100, y los demás van obteniendo valores de pid consecutivos. Asumir, además, que la función *printf* imprime por pantalla de manera inmediata (sin usar buffers intermedios).

Solución:

100: i=0, j=10
101: i=1, j=10
102: i=2, j=10
101: i=2, j=10
100: i=1, j=9
105: i=2, j=9
100: i=2, j=8



b) Explica cómo un proceso puede convertirse en *zombie*

Solución:
un proceso zombie o "defunct" (muerto) es un proceso que ha completado su ejecución pero aún tiene una entrada en la tabla de procesos, permitiendo al proceso que le ha creado (proceso "padre") leer el estado de su salida mediante una llamada wait(). Mientras tal lectura no ocurra el proceso permanecerá "zombie".

2. **[SINCRONIZACION Y COMUNICACIÓN]**

a) Explica el funcionamiento de las llamadas "*read*" y "*write*" sobre tuberías. Indica en qué situaciones se bloquean, producen error….
Solución:
READ
Si el pipe no contiene datos
Si existe un descriptor de escritura ➔ la lectura se bloquea
Si no existe un descriptor de escritura ➔ la lectura devuelve 0 (EOF)
Si el pipe tiene datos lee la cantidad solicitada o la que haya si no hay suficiente
WRITE
Si no existe un descriptor de lectura ➔ devuelve error (-1) de tipo EPIPE

b) Proponer una implementación de las llamadas "lock" y "unlock" usando tuberías. ¿Cómo definirías en esa implementación el tipo mutex_t? ¿Cómo debería inicializarse?
Solución:
UNLOCK: write(fd[1], &c, 1); LOCK: read(fd[0], &c, 1);
typedef struct { int fd[2]; char c;} mutext_t;
mutex_init(mutext_t m) = {pipe(m.fd); write(m.fd[1], &c, 1); }

3. **[MEMORIA]**

```
char buf[4000] = {1,2,3,4…,4000};
int i;
char *m;

void main() {
  m = (char*) malloc(4000);
  for (i=0; i < N; i++) {
    pthread_create(….,opera,i);
  }
  // Espera a todos los hilos
  while ( pthread_join(..) … ) { };
}
```

```
void opera(int index) {
  int j,k;

  for (j=index*1000,k=0;k<1000;j++,k++)
      m[j] = buf[j]*2;
}
```

a) Dado el código anterior con N=1, indica qué regiones tiene el proceso antes de que termine la ejecución de *pthread_join.* Si el tamaño de página del sistema es de 1KB, ¿cuál es el tamaño de cada región en páginas? **Justifica tu respuesta.**
Solución:

Código, C: 1K (supuesto programa pequeño)
Datos inicializados, D: 4K (corresponde al array buf[]
Datos globales iniciados a 0, B: 1K (corresponde a las variables i y m
Pilas, P: 2K → 1K para el hilo main() y 1K para el hilo opera()
Heap, H: 4K (para la memoria dinámica malloc()

b) Si el proceso dispone de 6 marcos de página para su ejecución, ¿cuántos fallos de página se producen, como **mínimo**, durante la ejecución del proceso para N=4? ¿Bajo qué circunstancias se produciría un número superior de fallos?
Solución:

Ejecución de main() hasta pthread_join(): C+P0+H0+B
Ejecución del hilo 1 completo: C+P0+H0+B+P1+D0
Ejecución del hilo 2 completo: C+P0+B+H1+P2+D1
Ejecución del hilo 3 completo: C+P0+B+H2+P3+D2
Ejecución del hilo 4 completo: C+P0+B+H3+P4+D3

Se producirían más fallos si los hilos se alternasen en su ejecución

4. **[ENTRADA/SALIDA]**

a) Indica qué **información de control** que debería contener la cabecera de cada buffer (bloque) de la cache de bloques, suponiendo que se aplica una política LRU de reemplazo. **Justifica tu respuesta**
Solución:

Identificación del bloque de disco asociado: número de bloque y número de volumen
Puntero al bloque de datos asociado en memoria
Punteros para formar listas enlazadas: cola hash y cola "libre" LRU
Estado del bloque: válido o no válido; en uso o libre; modificado o limpio;
en operación de disco (lectura o escritura) o no;

b) Ordena las funciones encomendadas a un manejador de disco ("driver") que, a continuación, se presentan desordenadas:

1. Insertar la petición en la cola del dispositivo
2. Planificar la cola de peticiones con la estrategia pertinente: FCFS, SSTF, SCAN,…
3. Bloquear el manejador en espera de una interrupción de E/S procedente del disco
4. Procesar la petición de E/S de bloques
5. Indicar el estado de terminación de la petición al nivel superior del sistema de E/S
6. Traducir la petición a órdenes para el controlador
7. Enviar las órdenes al controlador, programando el DMA
8. Comprobar el estado de la operación cuando llega la interrupción
9. Gestionar los errores y resolverlos si es posible

Solución: 4 – 1 – 2 – 6 – 7 – 3 – 8 – 9 – 5

Recibe la petición de bloque, la redirige al dispositivo y la planifica: 4 + 1 + 2
La más prioritaria la envía al dispositivo: 6 + 7 + 3
Recibe respuesta por interrupción y la devuelve al nivel superior: 8 + 9 + 5

5. [**SISTEMA DE FICHEROS**]

a) Un dispositivo de memoria flash de 64 MB de capacidad y bloques de 1KB, contiene un sistema de ficheros FAT. ¿Cuántos bytes son necesarios para almacenar la tabla FAT? **Justifica tu respuesta**
   a. 64 KB
   b. 128 KB
   c. 1 MB
   d. 512 KB
   e. Ninguna de las respuestas anteriores es correcta

   Solución:
   Número de bloques de datos direccionables: 64 MB/1KB = 64K bloques
   La FAT debe poder contener 64K direcciones
   Cada dirección ocupa como mínimo: $\log_2$ 64K bits = 16 bits = 16/8 bytes = 2 bytes
   *Luego, la FAT ocupa: 64K direcciones * 2 bytes/dirección = 128KB (respuesta b)*

b) ¿Es posible realizar enlaces rígidos en un sistema de ficheros tipo FAT? ¿Por qué no se puede establecer enlaces rígidos a ficheros de volúmenes distintos en sistemas de ficheros tipo EXT2? **Justifica tu respuesta**

   Solución:
   NO – porque en FAT cada nombre de fichero dentro de un volumen tiene una entrada de atributos distinta y mantenerlas exactamente iguales representaría un problema de coherencia desorbitado
   PORQUE los enlaces rígidos están asociados a un inodo de un volumen específico, y los inodos no se pueden compartir entre volúmenes (cada volumen es un "espacio" de inodos independiente.

Ingeniería Informática. Curso 3º.
Sistemas Operativos
Examen Final. PROBLEMAS.
4 de Septiembre de 2009

1. Resolver el siguiente problema de concurrencia denominado "H_2O".
El programa principal genera constantemente hilos de dos clases: oxígeno e hidrógeno, con el objetivo de formar moléculas de agua.

A continuación se muestra un esquema del código de los hilos de tipo *oxígeno.* En primer lugar, un hilo oxígeno deberá esperar a que haya al menos dos hidrógenos en el sistema. Tras garantizar que hay dos hidrógenos y un oxígeno, esos tres hilos se esperarán mutuamente en una barrera para garantizar que ejecutarán la función "FormarMolecula()" concurrentemente.

```
main(){
 while(1) {
  n=random();
  if (n es par)
    pthread_create(…,oxigeno,…);
  else
    pthread_create(…,hidro,…);
 }
}
```

```
Oxigeno/Hidrogeno( ) {

     //EsperaRestantesComponentes

     Barrera(3);
     FormarMolecula();

}
```

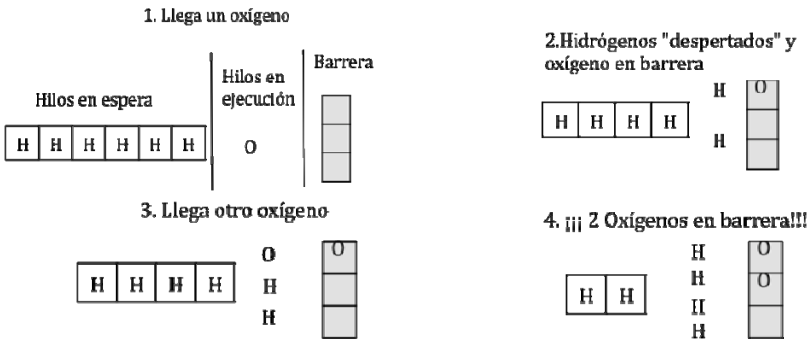
a) (2 puntos) Codificar, con ***mutexes y variables condicionales***, el código de los hilos de ***oxígeno*** e ***hidrógeno.*** En esta primera versión **NO es necesario garantizar** que haya 2 hidrógenos y un oxígeno en la barrera (y por tanto, ejecutando la función "FormarMolecula()" simultáneamente). Consulta el ejemplo posterior para ver una situación *conflictiva* que NO es necesario solucionar en este apartado.

b) (2 puntos) Codificar, mediante mutexes y variables condicionales, la función Barrera(N).

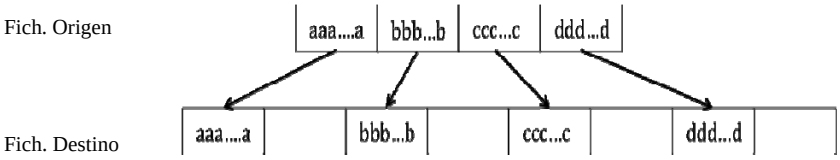
c) (1 punto) Partiendo de la implementación del apartado *a),* incluya la sincronización necesaria usando ***semáforos*** que garantice que la barrera sólo la superarán un oxígeno y dos hidrógenos, y sólo ellos ejecutarán la función "FormarMolecula()". Tras formar la molécula, se permitirá a otros hilos llegar a la barrera.

EJEMPLO (situación potencialmente *conflictiva).*

Considere que, en un instante dado, hay 6 átomos de hidrógeno en el sistema y ninguno de oxígeno. En el instante *t* llega un átomo de oxígeno, que deberá despertar dos hidrógenos para formar una molécula de agua. El oxígeno se dirigirá entonces a la barrera, donde esperará hasta que lleguen dos hilos más. Si antes de que alguno de esos dos hidrógenos llegue a la barrera, entra en el sistema un nuevo átomo de oxígeno, éste *podría* (todo depende de la planificación realizada por el SO) despertar a otros dos hidrógenos y dirigirse a la barrera, en la que tendríamos entonces dos oxígenos. Situaciones como ésta se pueden permitir en el apartado *a)* pero NO en el apartado *c).*



2. Un programador poco avezado pretende hacer una aplicación que realice *copias intercaladas* de ficheros en paralelo. El concepto de copia intercalada se ilustra en la figura: se copia el primer bloque íntegro, y se deja un bloque vacío. Se copia el segundo bloque del fichero origen al tercer bloque del destino, y así sucesivamente.



El código de su programa para la *copia intercalada* de un fichero de 4 bloques mediante 4 procesos, es el siguiente:

```
1: #define BLOCK 4096
2: char buf[BLOCK] = "xxxxxxx…xxxxx";
3: void main() {
4: pid_t pid;
5: int fdo,fdd;
6: fdo = open("Origen",O_RDONLY);
7: fdd = open("Destino",O_RDWR|O_CREAT|O_TRUNC,0666);
8: for (int i=0; i < 4; i++) {
9:    lseek(fdo,i*BLOCK, SEEK_SET);
10:   lseek(fdd,2*i*BLOCK,SEEK_SET);
11:   pid = fork();
12:   if (pid==0){
13:        copia_bloque(fdo,fdd);
14:        exit(0);
15:    }
16: }
17: while (wait(NULL)!=-1) { };
18: read(fdd,buf,BLOCK);
19: lseek(fdd,6*BLOCK,SEEK_SET);
20: read(fdd, buf,BLOCK);
21:}
```

```
22:
23:void copia_bloque(int fdo, int fdd)
24:{
25:    int size= read(fdo,buf,BLOCK);
26:    write(fdd,buf,size);
27:}
```

Responde a las siguientes preguntas, suponiendo que la **prioridad es para el proceso padre**.

a) (2 puntos) ¿Realiza el código la *copia intercalada* correctamente? Indica el contenido del array **`buf`** inmediatamente después de la ejecución de las líneas 18 y 20. Justifica tu respuesta.
b) (2 puntos) Sea sistema de ficheros de tipo Linux (nodos-i), con 4 punteros directos y un indirecto simple, tamaño de bloque de 4KB (4096 bytes) y 4bytes por puntero. Dibuja un posible estado final de toda la información del sistema de ficheros relativa al fichero "*Destino*":contenido relevante del nodo-i, bloques de datos ocupados…
c) (1 punto) Escribe una versión correcta de la *copia intercalada paralela.*

Solución Problema 1.

```
mutext_t mut;
cond_t cond_H, cond_O;
int num_H=0, num_O=0 ;
```

```
mutex_t mut_bar ;
cond_t c_bar ;
int num_bar=0 ;
```

```
Oxigeno() {
      lock(mut)
      num_O++;
      if (num_H < 2)
        cond_wait(cond_O, mut);
      else {
        num_H = num_H – 2;
        num_O = num_O – 1;
        cond_signal(cond_H);
        cond_signal(cond_H);
      }
      unlock(mut);
      Barrera(3);
      //FormarMolecula()
}
```

```
Hidrogeno() {
      lock(mut)
      num_H++;
      if (num_H < 2 || num_O < 1)
        cond_wait(cond_H, mut);
      else {
        num_H = num_H – 2;
        num_O = num_O – 1;
        cond_signal(cond_H);
        cond_signal(cond_O);
      }
      unlock(mut);
      Barrera(3);
      //FormarMolecula()
}
```

```
Barrera(int N) {
      lock(mut_bar;
      num_bar++;
      if (num_bar < N)
        cond_wait(c_bar, mut_bar);
      else {
        num_bar=0;
        cond_broadcast(c_bar);
      }
      unlock(mut_bar);
```

```
Oxigeno() {
      sem_wait(sem);
      lock(mut)
      num_O++;
      if (num_H < 2) {
        sem_post(sem);
        cond_wait(cond_O, mut);
      } else {
        num_H = num_H – 2;
        num_O = num_O – 1;
        cond_signal(cond_H);
        cond_signal(cond_H);
      }
      unlock(mut);
      Barrera(3);
      //FormarMolecula()
      sem_post(sem);
}
```

```
Hidrogeno() {
      sem_wait(sem);
      lock(mut)
      num_H++;
      if (num_H<2 || num_O<1) {
        sem_post(sem);
        cond_wait(cond_H, mut);
      } else {
        num_H = num_H – 2;
        num_O = num_O – 1;
        cond_signal(cond_H);
        cond_signal(cond_O);
      }
      unlock(mut);
      Barrera(3);
      //FormarMolecula()
}
```

Solución al problema 2

a) NO. La utilización compartida de los descriptores de fichero de "origen" y "destino" por parte de los 4 procesos hace que no se obtenga la necesaria independencia de operaciones de "copia" entre ellos.
18: read(fdd, buf, BLOCK); // posicionado a final del fichero
Puesto que lee de FIN-DE-FICHERO, buf queda inalterado: "xxx…x"
20: read(fdd, buf, BLOCK); // posicionado en el séptimo bloque
Lee en buf "ddd…d" que es la única escritura con éxito realizada por uno de los procesos

b) "Destino" → inodo X
nodo X → tipo: regular; permisos: rw-rw-rw-; tamaño: 7KB; uid; gid; device; link: 1
tiempos de A,C,M
punteros directos: 0, 0, 0, 0
puntero indirecto Y: 0, 0, Z
bloque de datos Z: "ddd…d"

c)

```
1: #define BLOCK 4096
2: char buf[BLOCK] = "xxxxxxx…xxxxx";
3: void main() {
4: pid_t pid;
5: int fdo,fdd;
6:
7: fdd = open("Destino",O_RDWR|O_CREAT|O_TRUNC,0666);
8: for (int i=0; i < 4; i++) {
9:
10:
11:   pid = fork();
12:   if (pid==0){
13:        copia_bloque(i);
14:        exit(0);
15:    }
16: }
17: while (wait(NULL)!=-1) { };
18: read(fdd,buf,BLOCK);
19: lseek(fdd,6*BLOCK,SEEK_SET);
20: read(fdd, buf,BLOCK);
21:}
```

```
22:
23:void copia_bloque(int i)
24:{
       int fdo, fdd;

       fdo = open("Origen",O_RDONLY);
       fdd = open("Destino",O_WRONLY);
       lseek(fdo,i*BLOCK, SEEK_SET);
       lseek(fdd,2*i*BLOCK,SEEK_SET);
25:    int size= read(fdo,buf,BLOCK);
26:    write(fdd,buf,size);
27:}
```