Sistemas Operativos

Departamento de Computación – FCEyN – UBA
Primer cuatrimestre de 2018

Nombre y apellido: _____ _____
Nº orden: _____ L.U.: ____ Cant. hojas: 6

1	2	3	4	Nota
B-	B	B	B	A

Recuperatorio del primer parcial – 26/6 - primer cuatrimestre de 2018

ACLARACIONES: 1) **Numere** las hojas entregadas. Esta hoja se entrega y es la hoja cero. Complete en la primera hoja la cantidad total de hojas entregadas (sin contar el enunciado). 2) Realice cada ejercicio en **hojas separadas** y escriba **nombre, apellido y L.U. en cada una**. 3) Cada ejercicio se califica con **Bien**, **Regular** o **Mal**. La división de los ejercicios en incisos es meramente orientativa. Los ejercicios se califican globalmente. El parcial se aprueba con 2 ejercicios bien y a lo sumo 1 mal/incompleto. 4) El parcial **NO** es a libro abierto. 5) **Justifique adecuadamente cada una de sus respuestas.**

Ejercicio 1.

Se desea implementar la función `traductor()` que toma las primeras n líneas de texto de un archivo de entrada y las copia traducidas **en cualquier orden** a otro archivo. La función `traductor()` toma tres parámetros:

- `n`: Cantidad de líneas a traducir.
- `infile`: Nombre del archivo de entrada.
- `outfile`: Nombre del archivo de salida.

Cada línea será traducida por un proceso distinto. Para traducir las líneas se deberá contar con un proceso principal que envíe cada una de ellas a los otros `n` procesos. Cada uno de estos `n` procesos recibirá una línea, la traducirá mediante la llamada a la función `void traducir(char *linea, char *linea_traducida)` y la enviará traducida al proceso principal. Sólo el proceso principal será el encargado de leer el archivo de entrada `infile` y escribir las líneas traducidas en `outfile`. Es fundamental que los procesos puedan traducir líneas en simultáneo dado que la función `traducir` es muy lenta.

Importante: Toda la comunicación entre procesos debe hacerse mediante pipes.

Ejercicio 2.

Se cuenta con un sistema con 2 procesadores, y un scheduler con dos colas de prioridad. Ambas colas funcionan con Round Robin de quantum 5 y 10 respectivamente. Todos los procesos cuando llegan van a la cola de quantum 5 y una vez consumido su quantum o si se bloquean por E/S pasan a la cola de menor prioridad con quantum 10. Asuma que el cambio de contexto es de 1 ciclo, y que el costo de migración entre núcleos es de 3 ciclos.

a) Realizar el diagrama de Gantt para los siguiente procesos:

PID	Llegada	T. ejec	T. bloq (inc.)
1	0	9	3-3
2	0	5	
3	1	12	8-10
4	2	8	
5	8	10	

Donde el tiempo de bloqueo se contabiliza dentro del tiempo de ejecución y los tiempos de bloqueos están incluidos. Es decir, si una tarea tiene tiempo de ejecución de 5 segundos y bloquea en 2-3, ejecutará en $T = 0$, $T = 1$, bloqueará en $T = 2$, $T = 3$ y ejecutará en $T = 4$ (suponiendo que es la única tarea en el sistema). A su vez se contabilizará como que ejecuta 5s.

b) Calcular el **waiting time** y **turnaround** promedio. Justificar mediante las cuentas que avalan el resultado.

c) ¿Pueden los procesos sufrir de inanición con un scheduler como éste? Justifique.

Ejercicio 3.

a) Implementar un mutex **reentrante** con variables atómicas que implemente los siguientes tres métodos: `create()`, `lock()`, `unlock()`.

b) Explique el concepto de `local spinning`. ¿Cuáles son las diferencias entre un TASLock y un TTASLock?

Ejercicio 4.

Existen tres heladeros que quieren preparar helados. Los helados se preparan **uno por vez** y con tres elementos: un cucurucho, una bocha de helado y un baño de chocolate. Cada heladero tiene infinita cantidad de un sólo tipo de elementos. Es decir, tenemos un heladero con infinitos cucuruchos, otro con infinitas bochas de helado y otro con infinitos baños de chocolates.

Además existen tres proveedores. Cada proveedor trae dos elementos distintos para preparar helados. Esos dos elementos deberán ser usados por el heladero correspondiente para crear el helado.

Dado el siguiente código de proveedores:

```
llegaProveedor = Semaforo(1)
chocolate = Semaforo(0)
cucurucho = Semaforo(0)
bochaDeHelado = Semaforo(0)
```

ProveedorA

```
while (true) {
 llegaProveedor.wait()
 cucurucho.signal()
 bochaDeHelado.signal()
}
```

ProveedorB

```
while (true) {
 llegaProveedor.wait()
 bochaDeHelado.signal()
 chocolate.signal()
}
```

ProveedorC

```
while (true) {
 llegaProveedor.wait()
 chocolate.signal()
 cucurucho.signal()
}
```

Y el siguiente código de heladeros:

HeladeroConBochas

```
while (true) {
 cucurucho.wait()
 chocolate.wait()
 preparHelado()
 llegaProveedor.signal()
}
```

HeladeroConChocolate

```
while (true) {
 cucurucho.wait()
 bochaDeHelado.wait()
 preparHelado()
 llegaProveedor.signal()
}
```

HeladeroConCucuruchos

```
while (true) {
 chocolate.wait()
 bochaDeHelado.wait()
 preparHelado()
 llegaProveedor.signal()
}
```

Argumente si se cumplen o no las siguientes propiedades:

a) **EXCL**
b) **DEADLOCK-FREEDOM**

HOJA 1

```
1) void traductor (int n, char* infile, char* outfile) {
       char** lineas[n];
       for(int i=0; i<n; i++){
           lineas[i] = leer_linea(infile, i); // lee la linea
       }

       int* array_pipes[n], int* array2 = copiar
       for(int i=0; i<n; i++){
           int P[2]
           array_pipes[i] = asignar_pipe(P); // crea array de pipes
           pipe(array_pipes[i]);
       }
       int* array2 = array
int* array2 = copiar(array_pipes);
       for(int i=0; i<n; i++){
           dup2(STDOUT, array_pipes[i][1]);
           close(array_pipes[i][0])
           COUT << lineas[i];
       }
       int PID
int indice = -1
       for(int i=0; i<n; i++){
           PID = fork()
           if(PID==0) {
               indice = i
               break;
           }
       }
```

// asignar pipes e para asegurarse que queden en mem dinámica

no hace falta usar dup2 podes escribir directamente en el pipe

```
if (PID==0){
    dup2(STDIN, array_pipes[indice][0])
    close(array_pipes[indice][0])
    char* linea << CIN
    char* linea_traducida
    traducir(linea, linea traducida)
    dup2(STDOUT, array2[indice][1])
    close(array2[indice][0])
    COUT << linea traducida
}
else{
    for(int i=0; i<cn; i++){
        dup2(STDIN, array2[i][0])
        close(array2[i][1])
        char* a_imprimir <<CIN
        print(a_imprimir);
    }
}
}
```

a_imprimir <<CIN ← CIN → a_imprimir

↳ esta recibiendo el orden de salida

(podría recibir todo lo pronto al primer pipe y leer de uno solo)

Se debe escribir en el archivo de salida outfile

Leo las líneas del archivo; me crea 2 arreglos de pipes (la función copiar se encarga del segundo)

crea los procesos hijos; a c/u le paso por el pipe del 1° arreglo la línea; luego traducen y devuelven por pipe del segundo arreglo la traducción para que el padre imprima

HOJA 2

FECHA

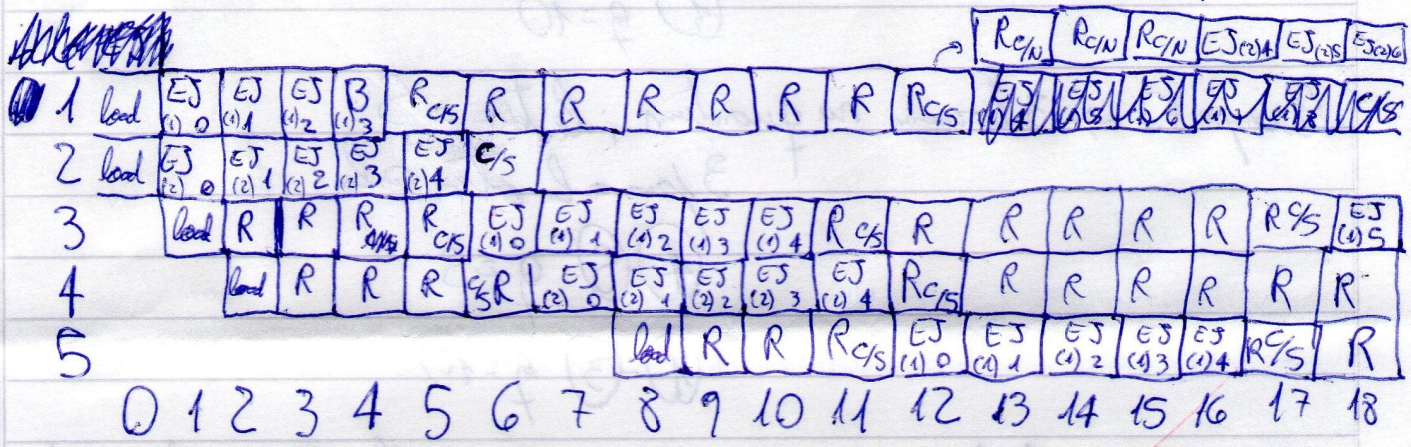
2) a) 2 procesadores

2 cola de prioridad → quantum = 5
" = 10

B

bloqueo E/S → pasan a cola de menor prioridad

CS → 1 ciclo migración de núcleo → 3 ciclos



(1) vuelve en otra vuelta

Tiempo 0 → llegan 1 y 2 ; los encola en la de quantum 5
(1)-(2)

Tiempo 1 → son puestos a ejecutar 1 y 2 c/u en un procesador
llega 3 que se encolada en la de quantum 5
(1)-(2)-(3)

Tiempo 2 → llega 4 , encolada en la de q 5
(1)-(2)-(3)-(4)

Tiempo 4 → 1 se bloquea por E/S se desalojado ; pasa a la otra cola
(2)-(3)-(4) q=5
(1) q=10

NOTA

Tiempo 6 → 3 comienza a ejecutarse
2 termina su ejecución ; 4 comienza a ejecutarse

③-④ q=5

① q=10

Tiempo 7 → 4 comienza a ejecutarse

③-④ q=5

① q=10

Tiempo 8 → llega 5 ③-④-⑤ q=5

① q=10

Tiempo 10 → 3 termina su quantum ; le toca a 5
3 pasa a la otra cola

④-⑤ q=5

①-③ q=10

Tiempo 12 → 4 termina su quantum ; pasa a la otra cola. le toca a 1
toma 3 ciclo de cambio de núcleo
+1 ciclo c/s

⑤ q=5

①-③-④ q=10

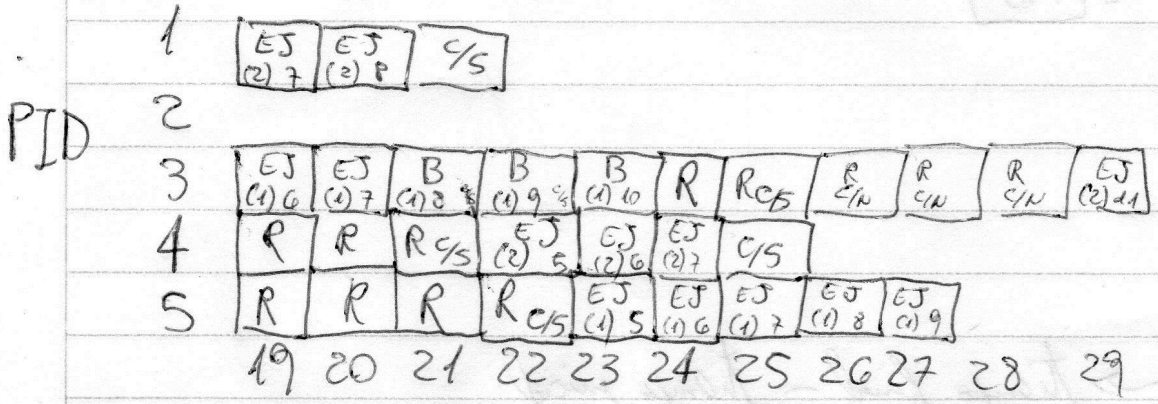
Tiempo 16 → 1 comienza a ejecutarse
5 termina su quantum ; le toca a 3 no hay cambio de núcleo (por suerte)

① ①-③-④-⑤ q=10

Tiempo 18 → 3 vuelve a ejecución

FECHA

~~8~~



Tiempo 20 termina 1

③—④—⑤ q = 10

Tiempo 21 se bloquea 3 ③—④—⑤

Tiempo 22 le toca a 4 que no cambia de nucleo ③—④—⑤

Tiempo 23 le toca a 5 que no cambia de nucleo

Tiempo 24 termina 4

Tiempo 26 llega 3 que cambia de nucleo

Tiempo 27 termina 5

Tiempo 29 termina 3

c) Se puede producir inanición si 1 proceso termina su quantum o se bloquea por E/S (para ala segunda cola) y llegan todo el tiempo procesos nuevos (que llenarán la cola de quantum 5)

b) waiting time: Numero de "ready" de 1 proceso y divido por

NOTA

5 para obtener el promedio

$$\frac{\overset{(1)}{11} + \overset{(2)}{0} + \overset{(3)}{16} + \overset{(4)}{14} + \overset{(5)}{9}}{5}$$

$$= \frac{50}{5} = \boxed{10}$$

$11+9=20$

$16+14=30$

Turnaround → tiempo final − tiempo inicial

$$\frac{\overset{(1)}{(20-0)} + \overset{(2)}{(5-0)} + \overset{(3)}{(29-1)} + \overset{(4)}{(24-2)} + \overset{(5)}{(27-8)}}{5}$$

$$= \frac{20+5+28+22+19}{5} = \frac{25+22+28+19}{5} = \frac{75+19}{5} = \boxed{\frac{94}{5}}$$

$$\begin{array}{r} {\scriptstyle 1\ \ \ \ } \\ 28 \\ +22 \\ \hline 50 \end{array} \qquad \begin{array}{r} {\scriptstyle 1\ \ \ \ } \\ 75 \\ +19 \\ \hline 94 \end{array}$$

FECHA

3) a) Declaro la variable

(5)

```
atomic<bool> b
atomic<int> contador
atomic<int> PID_actual

create(){
    b.set(false);
    contador.set(0);
    PID_actual.set(-1);
}
```

La función dame_PID devuelve el PID del proceso actual.

```
lock(){
    while(b.testAndSet()){
        if(PID_actual == dame_PID()){
            contador++;
            break;
        }
    }
    PID_actual = dame_PID();
}

unlock(){
    if(contador == 0){
        b.set(false);
    }
    else{
        contador--;
```

NOTA

b) Local spinning es una técnica para sincronizar procesos. Utiliza una variable booleana atómica, la cual es leída mediante la función get() en un while constantemente

b) Local spinning es una técnica para sincronizar procesos que se basa en estar leyendo constantemente una variable booleana atómica (utilizando la función "get()" dentro de un ciclo) esperando hasta que tome el valor deseado, momento en el que se debería realizar realiza un test and set para intentar alcanzar la sección crítica del código. Si logra (consiguiendo así) acceder, se volverá a leer la variable atómica.

```
while(true){                                  Un lock utilizando
    while(b.get() != valor_deseado){}         local spinning,
    if (b.testAndSet()){ break; }             llamado TTASLOCK
}
```

No se está llamando constantemente a la función testAndSet, a diferencia de los TASLOCKs donde sí se hace. Aquí, la función get mira la memoria caché en vez de la principal como lo hacen los TASLOCKs (ya que deben escribir en la variable constantemente) gastando por lo cual suelen ser más eficientes. Sin embargo, ambos producen busy waiting por lo que si la sección crítica demora mucho tiempo es preferible no usarlos ya que estarán consumiendo tiempo de la CPU.

HOJAS

4) a) NO HAY

4) b) Puede haber deadlock

4) b) Se cumple deadlock-freedom

Supongo una de la siguiente ejecución

⇒ Se ejecutan primero los 3 heladeros

quedan C/u esperando en su wait

Hlocha en cucurucho.wait()
Hchocolate en cucurucho.wait()
Hcucurucho en chocolate.wait()

} ya que los 3 semáforos comienzan en 0

Ahora supongo que se ejecuta proveedor C

Ejecuta Llega Proveedor.wait() → se queda esperando para entrar en 0 por lo que ningún otro proveedor podrá pasar del primer semáforo de la primer línea hasta que se ejecute un signal en dicho semáforo.

Luego de eso, continua ejecutando el proveedor C

chocolate.signal() → aquí se despierta a Hcucurucho que luego de hacer signal, queda esperando en el siguiente wait: lochaYHelado.wait()

Por último, termina de ejecutar el proveedor C

cucurucho.signal() → se despiertan Hlocha y Hchocolate

B

NOTA

Cada cual se despertado, volverán a quedar esperando en el siguiente wait

En el caso de H Lecha chocolate wait()
" " " " H chocolate lecha de helado wait()

Con lo cual se produce un deadlock ya que ninguno de los heladeros puede continuar su ejecución (como se demostró más arriba) ni ningún proveedor (ya que llega Proveedor queda en con valor 0 y no hubo ninguna señal) podrá pasar la primer línea. Ningún proceso entonces entrará en la sección crítica y todos quedarán eternamente esperando a ser habilitados para entrar

a) EXCL Nunca habrá más de 1 proceso en la sección crítica.

Supongamos que alguno de los heladeros está en la sección crítica, para esto antes necesariamente se tuvo que ejecutar un proveedor que haya liberado sus 2 recursos.

Un proveedor por sí solo no puede liberar a más de un heladero ya que el proveedor libera 2 recursos de los 3 disponibles y los recursos que necesitan los heladeros para entrar a la sección crítica no son iguales los mismos

H lecha necesita cucharacha y chocolate
H chocolate " " y lecha de helado
H cucharacha " chocolate y "

Luego no se libera nada

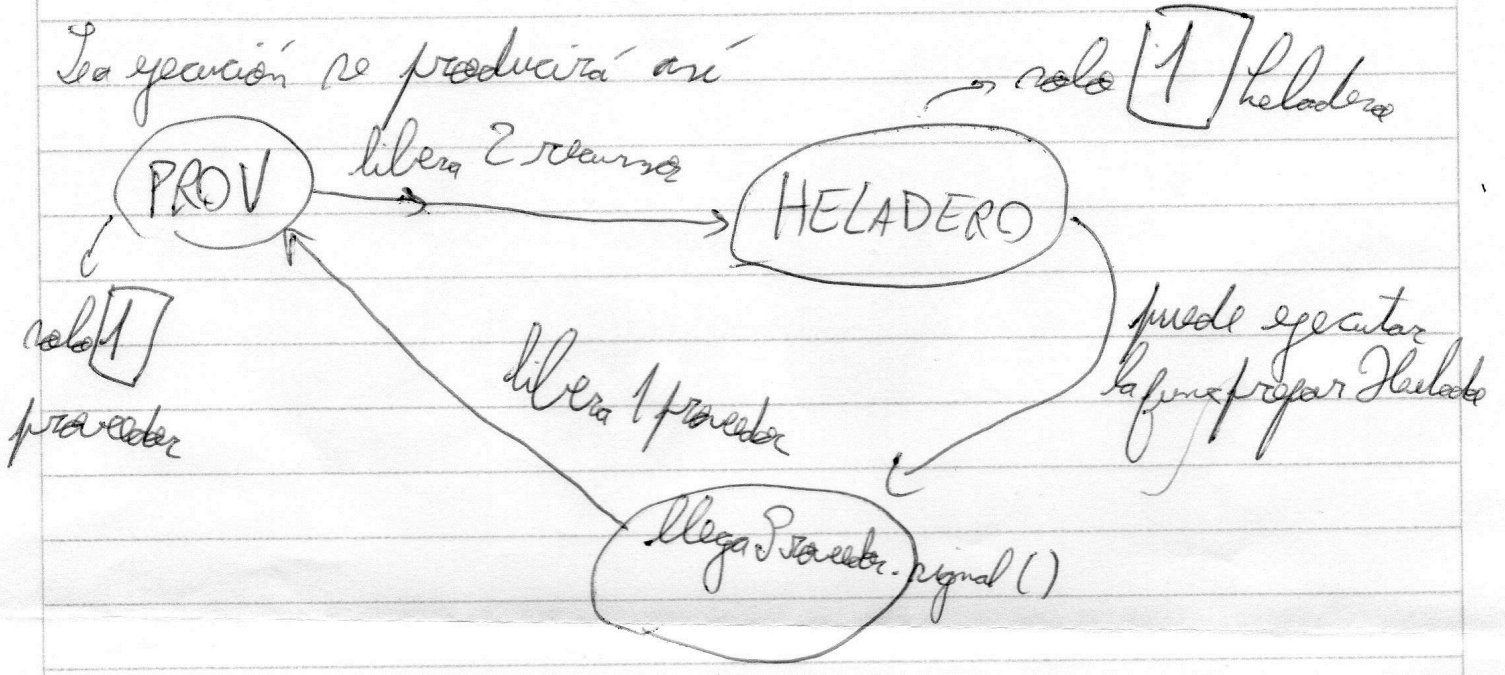
Como sólo se puede llamar a un proveedor después de que algún heladero ejecute la sección crítica (el signal está después

FECHA

de cada "preparar Helado()" y como al comenzar solo hay 1 proveedor y ningún recurso

(el semáforo de proveedor está en 1 y los de helado en 0 (chocolate, cucurucho, bocha de helado))

La ejecución se producirá así



Con lo cual si en algún punto de la ejecución llega a entrar a la sección crítica un heladero solo habrá podido hacerlo, ya que antes un proveedor liberó (ejecutó signal) un 2 recursos.

Si un proveedor libera los recursos de 2 heladeros distintos (ej: el proveedor C ~~gobernado~~ despierta al heladero con chocolate y al heladero con cucurucho y al hacer signal sobre cucurucho y chocolate respectivamente) se producirá un deadlock ya que no se podrá llamar a otro proveedor hasta que algún heladero pase la sección crítica (cosa que nunca sucederá al tener liberados solo 1 de los 2 recursos necesarios).