PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC 2333 — Sistemas Operativos y Redes

Interrogación 1 – Soluciones

Miércoles 07-Abril-2015

Duración: 2 horas

Esta pauta debe ser tomada como una guía de respuestas y no como una planilla absoluta de soluciones. Las respuestas en esta pauta pueden ser intencionalmente más largas de lo solicitado a los estudiantes, para efectos de dar explicaciones adicionales y respuestas alternativas.

1. **[10p]** Indique si las siguientes afirmaciones son verdaderas o falsas. En caso que sean falsas, debe explicar por qué lo son, de lo contrario no tendrán puntaje. (1 cada una)

   1.1) Una situación de *deadlock* puede ser resuelta utilizando un algoritmo de *scheduling* expropiativo.
   R. FALSO. Una manera de resolver un *deadlock* es expropiar recursos a un proceso, pero el *scheduling* expropiativo no está relacionado con la expropiación de recursos, sino con la expropiación de la CPU a un proceso. En cualquier caso, ante una situación de *deadlock*, expropiar la CPU no soluciona el problema.

   1.2) Un sistema operativo que intenta otorgar una buena respuesta a la interacción del usuario con el escritorio, debe ser capaz de privilegiar los procesos intensivos en E/S.
   R. VERDADERO. Los procesos de interacción con el usuario son inherentemente procesos intensivos en E/S, por lo tanto si éstos tienen prioridad para ejecutar, el usuario percibirá que el sistema responde más rápidamente.

   1.3) Un sistema operativo puede funcionar sin interrupciones por *timer*
   R. VERDADERO. El sistema operativo puede funcionar, sin embargo será difícil tener un *multitasking* efectivo. El sistema operativo puede en cualquier momento consultar el *timer*, pero deberá hacerlo regularmente dentro de su código.

   1.4) El *stack pointer* (SP) de un proceso sólo puede ser modificado en modo monitor.
   R. FALSO. Cada proceso modifica su SP durante la ejecución, y eso ocurre en modo usuario.

   1.5) Es posible tener un sistema con multiprogramación, pero sin *multi-tasking*.
   R. VERDADERO. Puede haber múltiples procesos cargado en memoria pero atendido en orden FCFS. En esto caso no *multitasking*.

   1.6) Sistemas operativos como Windows o Linux evitan *deadlock*s utilizando el algoritmo del banquero.
   R. FALSO. El algoritmo de banquero suele provocar subutilización de recursos ya que asigna los recursos de manera bastante conservadora para transitar solamente en estado seguros. Además su ejecución es más lenta a medida que hay más procesos en ejecución.

   1.7) En un *scheduler Round-Robin*, mientras más pequeño sea el *quantum* mayor será la frecuencia a la cual los procesos son asignados a la CPU.
   R. VERDADERO. Mientras más pequeño es el *quantum*, menos tiempo pasa entre el paso de procesos consecutivos a la CPU. Aún cuando el *quantum* sea menor que el tiempo necesario para efectuar un cambio de contexto, la frecuencia de acceso a la CPU seguirá aumentando (lo que, en cualquier caso, no implica que los procesos terminen más rápido).

   1.8) Un programa, originalmente secuencial, ha sido modificado de manera que el $95\,\%$ de sus instrucciones pueden ser ejecutadas en paralelo. Al ejecutarlo en un cluster con miles de *cores*, la ejecución toma un 1/20 del tiempo original.
   R. VERDADERO. De acuerdo a la ley de Amdahl:

$$\mathit{Speedup} = \frac{1}{S + \frac{1-S}{N}} = \frac{1}{0{,}05 + \frac{0{,}95}{N}} \xrightarrow[N\to\infty]{} \frac{1}{0{,}05} = 20$$

1.9) Cada vez que un proceso hace una llamada al sistema, se ejecuta un cambio de contexto (i.e., se llama al *scheduler*)

R. FALSO. No todas las llamadas al sistema implican un cambio de contexto entre procesos.

1.10) Un sistema que se encuentra en un estado *inseguro*, en el futuro se encontrará en *deadlock*

R. FALSO. Un estado *inseguro* solo implica que no hay ningún proceso que pueda ser atendido si éste hace una solicitud por todos los recursos que le restan por pedir, pero el sistema aún no se encuentra en *deadlock*. Es posible que el sistema salga del estado inseguro sin llegar a estar en *deadlock* si algún proceso libera algunos de sus recursos voluntariamente antes de solicitar el máximo declarado.

2. **[10p]** Responda de manera breve y precisa las siguientes preguntas:

2.1) **[4p]** Suponga que tiene una línea de comandos con 3 operaciones: `head -n X file`, que retorna las primeras `X` líneas del archivo `file`; `tail -n X file`, que retorna las últimas `X` líneas de `file`; y `grep patron file` que retorna solamente las líneas de `file` que contienen el *substring* `patron`. En todos los casos, si se omite `file`, el comando lee desde la entrada estándar (`stdin`).

Describa el resultado de las operaciones[1]:

a) `head -n 100 thread.c | tail -n 40 | grep ``struct thread''`
R. Este comando imprime en pantalla todas las líneas de `thread.c` que contienen el *substring* `struct thread`, pero solo considerando las líneas 61 a la 100 de `thread.c`. Si el archivo tiene menos de 100 líneas, se consideran solamente las últimas 40. Si el archivo tiene menos de 40 líneas, se consideran todas.

b) `grep ``ticks'' thread.c | head -n 20 | tail -n 1`
R. Este comando imprime en pantalla la última línea de `thread.c`, de entre las 20 primeras que contienen el *substring* `ticks`.

2.2) **[3p]** Un *lock* implementado usando espera ocupada se conoce como *spinlock*. Se argumenta que los *spinlock*s pueden ser convenientes para ser utilizados en multiprocesadores. Justifique si esto es una buena idea o no.

R. Un *spinlock* es una mala idea en un monoprocesador pues consume ciclos innecesariamente, los que podrían ser ocupados por otro proceso. Sin embargo cuando un proceso se bloquea por poco tiempo (menos tiempo que lo demora hacer un cambio de contexto), por ejemplo en el caso que hay otro proceso ejecutando en otro procesador, que podría liberar al que está en el *spinlock*, entonces un *spinlock* es una solución más eficiente que bloquear al proceso con un *lock* tradicional.

2.3) **[3p]** Para el siguiente código, considera los siguientes prototipos:

- `pid_t fork()` retorna 0 en el caso del hijo y el *pid* del hijo, en el caso del padre.
- `int exec(command)` recibe como parámetro la ruta del archivo con el código a ejecutar
- `pid_t wait(*exitStatus)` recibe como parámetro un puntero donde se guarda el estado de salida del proceso que ha terminado (puede usar `NULL` si no le interesa ese valor). Retorna el *pid* del proceso que terminó, ó -1 si el proceso no ha hecho `fork()`

Para el siguiente código escriba su salida en pantalla. Puede haber muchas salidas válidas. Escriba cualquiera de ellas, pero sólo una.

```
1 int main(int argc, char *argv[]) {
2   pid_t pid;
3   int a = 42;
4   printf("Wow!\n");
5   pid = fork();
6   if(pid == 0) {
7     pid = fork();
```

[1]Ejemplo: `head -n 100 thread.c` muestra las primeras 100 lineas de `thread.c`

```
8       a++;
9       if(pid == 0) {
10        printf("3(%d)\n",a);
11      }
12      else {
13        pid = wait(NULL);
14        if(pid > 0) {
15          printf("4(%d)\n",a);
16        }
17      }
18    }
19    else {
20      pid = fork();
21      if(pid == 0) {
22        a++;
23        printf("1(%d)\n",a);
24        execl("/bin/date", "date", NULL);
25        printf("5\n");
26      }
27      printf("2(%d)\n",a);
28    }
29    printf("0\n");
30    return 0;
31
32  }
```

R. Una salida posible es la siguiente. Se ha agregado la identificación P0: proceso padre; P1: primer hijo de P0 (línea 5); P2: segundo hijo de P0 (línea 20); P3: hijo de P1 (línea 7).

```
P0: Wow!
P0: 2(42)
P0: 0
P1: 4(43)
P1: 0
P2: 1(43)
P2: Wed Apr  8 20:47:48 CLST 2015
P3: 3(43)
P3: 0
```

Es importante notar que P2 no escribe `5` ni `0`, pues una vez que ejecuta `/bin/date` usando `exec`, su código ha sido reemplazado. Los valores no cambian, pero la intercalación entre líneas de distintos procesos puede variar. Asimismo, el orden entre líneas de un mismo proceso debe ser respetado.

3. **[14p]** Considere un tramo de la Ruta 5 Norte con dos pistas, una en cada sentido. Un tramo de ella, por razones climáticas, se ha dañado de manera que en ese tramo solo es posible circular en un sentido a la vez. La longitud del tramo dañado permite que haya un máximo de N vehículos circulando en él.

   3.1) **[7p]** Escriba un código para `vehiculoNorte` y uno para `vehiculoSur` de manera que en todo momento solo haya un tipo de vehiculo circulando por ese tramo de la carretera. Los vehículos que no están circulando deben esperar a que pasen todos los del otro sentido antes de intentar entrar. Especifique las variables compartidas y su inicialización.

   R. Estado compartido:

```
1     const int N;              // numero maximo de autos en la pista
2     int nAutos = 0;           // numero de autos en la pista
```

```
3    int arrivedNorte = 0;      // numero de autos esperando en direccion NORTE
4    int arrivedSur = 0;        // numero de autos esperando en direccion SUR
5    int turno = {NORTE, SUR};
6    semaphore mutex = 1;
7    condition norte, sur;
```

Función para cada dirección:

```
1  vehiculoNorte() {                          vehiculoSur() {
2    mutex.P()                                  mutex.P()
3    arrivedNorte++                             arrivedSur++
4    while(turno == SUR || nAutos > N) {        while(turno == NORTE || nAutos > N) {
5      mutex.V()                                  mutex.V()
6      norte.wait()                               sur.wait()
7      mutex.P()                                  mutex.P()
8    }                                          }
9    nAutos++                                   nAutos++
10   arrivedNorte--                             arrivedSur--
11   mutex.V()                                  mutex.V()
12   ...                                        ...
13   // conducir                                // conducir
14   ...                                        ...
15   mutex.P()                                  mutex.V()
16   nAutos--                                   nAutos--
17   if(nAutos == 0) {                          if(nAutos == 0) {
18     turno = SUR                                turno = NORTE
19     for(i=0;i<arrivedSur;i++)                  for(i=0;i<arrivedNorte;i++)
20       sur.signal()                               norte.signal()
21   }                                          }
22   mutex.V()                                  mutex.V()
23 }                                          }
```

3.2) **[7p]** Modifique su solución para que, después que un máximo de M vehículos $(M < N)$ vehiculos pasen en un sentido, se le de la oportunidad a los del sentido opuesto. Esta modificación le agrega justicia (espera acotada) al problema.

R. Esta solución agrega un límite a la cantidad de autos en cada sentido con la variable `passed`. Además se agregan variable `passingNorte` y `passingSur` para contar la cantidad de vehículos que se encuentran dentro de la ruta en cada dirección.

```
1    const int N;               // numero maximo de autos en la pista
2    int nAutos = 0;            // numero de autos en la pista
3    int arrivedNorte = 0;      // numero de autos esperando en direccion NORTE
4    int arrivedSur = 0;        // numero de autos esperando en direccion SUR
5    int passed = 0;            // numero de autos que han atravesado la carretera
6    int passingNorte = 0;      // numero de autos circulando en direccion NORTE
7    int passingSur =0;         // numero de autos circulando en direccion SUR
8    int turno = {NORTE, SUR};
9    semaphore mutex = 1;
10   condition norte, sur;
```

La variable `passed` permite contar cuantos autos han pasado en cada sentido. Al momento de superar la cantidad límite, se modifica `turno`. Sin embargo, mientras no termine el último vehículo que estaba pasando, los de la dirección opuesta no deben poder entrar.

```
vehiculoNorte() {
  mutex.P()
  arrivedNorte++
  while(turno == SUR || nAutos > N ||
        passingSur > 0) {
    mutex.V()
    norte.wait()
    mutex.P()
  }
  nAutos++
  arrivedNorte--
  passingNorte++
  mutex.V()
  ...
  // conducir
  ...
  mutex.P()
  nAutos--
  if(passed < M)
    passed++
  else
    turno = SUR
  if(nAutos == 0) {
    turno = SUR
    for(i=0;i<arrivedSur;i++)
      sur.signal()
      passed = 0
  }
  passingNorte--
  mutex.V()
}
```

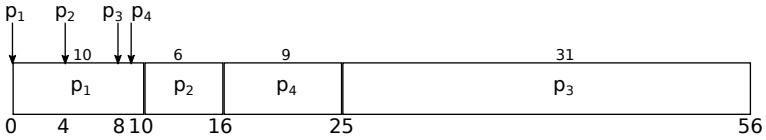
```
vehiculoSur() {
  mutex.P()
  arrivedSur++
  while(turno == NORTE || nAutos > N ||
        passingNorte > 0) {
    mutex.V()
    sur.wait()
    mutex.P()
  }
  nAutos++
  arrivedSur--
  passingSur++
  mutex.V()
  ...
  // conducir
  ...
  mutex.V()
  nAutos--
  if(passed < M)
    passed++
  else
    turno = NORTE
  if(nAutos == 0) {
    turno = NORTE
    for(i=0;i<arrivedNorte;i++)
      norte.signal()
      passed = 0
  }
  passingSur--
  mutex.V()
}
```

La solución puede utilizar las primitivas de software vistas en clase: *locks*, semáforos, monitores/variables de condición, pero no *busy-waiting*. También puede implementar un solo tipo de proceso `vehiculo` que sea capaz de comportarse correctamente en cada sentido.

4. **[16p]** Considere un conjunto de procesos $P = \{p_1, p_2, p_3, p_4\}$ con ráfagas (*burst-time*) $T = \{10, 6, 31, 9\}$, tiempos de llegada $A = \{0, 4, 8, 9\}$. Determine el tiempo de espera (*waiting time*) promedio, y la cantidad de cambios de contexto generados por los siguientes algoritmos de *scheduling* (los tiempos están en *ms*):

   4.1) **[4p]** SJF sin expropiación (*non-preemptive*)

   R. Se genera la siguiente ejecución:



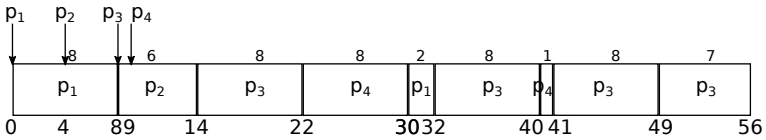
   Los tiempos de espera para proceso son:

   $w(p_1) = 0$
   $w(p_2) = 6$
   $w(p_3) = 17$
   $w(p_4) = 7$

   Tiempo promedio: $30/4 = 7{,}5$. Cambios de contexto: 4

   4.2) **[4p]** Round Robin, con $q = 8$

R. Se genera la siguiente ejecución:



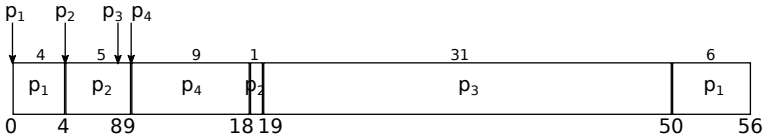
Los tiempos de espera para proceso son:

$w(p_1) = 22$
$w(p_2) = 4$
$w(p_3) = 6 + 10 + 1 = 17$
$w(p_4) = 13 + 10 = 23$

Tiempo promedio: $66/4 = 16{,}5$. Cambios de contexto: 9

4.3) **[4p]** Prioritario con expropiación, con prioridades $Pr = \{5, 40, 30, 50\}$

R. Se genera la siguiente ejecución:



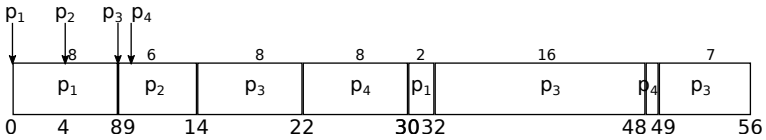
Los tiempos de espera para proceso son:

$w(p_1) = 46$
$w(p_2) = 9$
$w(p_3) = 11$
$w(p_4) = 0$

Tiempo promedio: $66/4 = 16{,}5$. Cambios de contexto: 6

4.4) **[4p]** Round Robin en que $q = 8$, y cada vez que un proceso utiliza todo su *quantum*, para el próximo turno su *quantum* se duplica.

R. Se genera la siguiente ejecución:



Los tiempos de espera para proceso son:

$w(p_1) = 22$
$w(p_2) = 4$
$w(p_3) = 6 + 10 + 1 = 17$
$w(p_4) = 13 + 18 = 31$

Tiempo promedio: $74/4 = 18{,}5$. Cambios de contexto: 8

Cuente los cambios de contexto aunque el proceso elegido sea el mismo que estaba ejecutando. Incluya el cambio de contexto en el instante 0.

5. **[10p]** Considere un pub que cuenta con 7 mesas, 14 sillas, y 14 vasos para sus clientes. Tres grupos de amigos (G_1, G_2, G_3) se encuentran en el pub. Al llegar, cada grupo declaró cuantos recursos ocuparían. G_1 declaró que ocuparía 2 mesas, 6 sillas y 7 vasos; G_2 declaró que ocuparía 5 mesas, 8 sillas y 7 vasos; y G_3 declaró que ocuparía 3 mesas, 7 sillas y 10 vasos. Sin embargo, como no todos los miembros del grupo han llegado, el pub les ha dado solamente algunos de los recursos que necesitan y les asignará el resto a medida que lleguen los miembros faltantes.

   Hasta el momento, G_1 está ocupando 1 mesa, 4 sillas y 7 vasos; G_2 está ocupando 2 mesas, 2 sillas, y 5 vasos; y G_3 está ocupando 2 mesas, 6 sillas, y 1 vaso. Ningún grupo se retirará mientras no haya utilizado todos los recursos que declaró, pero es posible dejarlos en espera (el mesero puede decidir ignorarlos temporalmente). En cualquier caso, se requiere que todos los grupos puedan ser atendidos completamente en algún momento y que ninguno se quede esperando hasta el cierre del pub.

R. Caso de asignación de recursos. Se puede resolver utilizando el algoritmo del banquero. En este caso hay 3 tipos de recursos R_1: mesas, R_2: sillas, R_3: vasos. Los vectores iniciales:

Total: $(7, 14, 14)$
Available: $(2, 2, 1)$

Y las matrices de asignación:

Max	R_1	R_2	R_3
G_1	2	6	7
G_2	5	8	7
G_3	3	7	10

,

Allocated	R_1	R_2	R_3
G_1	1	4	7
G_2	2	2	5
G_3	2	6	1

,

Need	R_1	R_2	R_3
G_1	1	2	0
G_2	3	6	2
G_3	1	1	9

5.1) **[4p]** Dada esta situación, ¿es posible satisfacer las demandas restantes de todos los grupos? Justifique por qué si o por qué no.

R. La pregunta es si el pub se encuentra en un estado seguro en cuanto a capacidad de asignación de recursos. La respuesta es sí, si hay una secuencia en la cual todos los grupos pueden ser atendidos:

G_1: puede ser atendido con su vector Need$(G_1) = (1, 2, 0)$, Available $= (3, 6, 8)$
G_2: puede ser atendido con su vector Need$(G_2) = (3, 6, 2)$, Available $= (5, 8, 13)$
G_3: puede ser atendido con su vector Need$(G_3) = (1, 1, 9)$, Available $= (7, 14, 14)$

Por lo tanto el sistema sí se encuentra en un estado seguro.

5.2) **[3p]** A partir de la situación inicial, si G_2 solicita dos mesas más, ¿debe el mesero atender su solicitud?

R. G_2 hace la solicitud $r(G_2) = (2, 0, 0)$. Es posible atenderla ya que hay recursos suficientes. El vector Available queda como Available $= (0, 2, 1)$, y los vectores Need$(G_2) = (1, 6, 2)$, Allocated$(G_2) = (4, 2, 5)$.

Con este vector Available no es posible atender el vector Need de ningún grupo, por lo tanto el sistema quedaría en un estado inseguro, y el mesero no debe atender (aún) la solicitud del grupo.

5.3) **[3p]** A partir de la situación inicial, si G_3 solicita 1 mesa y 1 un vaso más, ¿debe el mesero atender su solicitud?

R. G_3 hace la solicitud $r(G_3) = (1, 0, 1)$. Es posible atenderla ya que hay recursos suficientes. El vector Available queda como Available $= (1, 2, 0)$, y el vector Need$(G_3) = (0, 1, 8)$, y Allocated$(G_3) = (3, 6, 2)$.

Con este vector Available es posible atender:

G_1: puede ser atendido con su vector Need$(G_1) = (1, 2, 0)$, Available $= (2, 6, 7)$
G_2: no puede ser atendido con su vector Need$(G_2) = (3, 6, 2)$
G_3: no puede ser atendido con su vector Need$(G_3) = (0, 1, 8)$

Aún cuando se podría atender a un grupo, el sistema quedaría en un estado inseguro, y propenso a *deadlock* (no está en *deadlock* aún). El mesero debe postergar la solicitud del grupo.

6. **[5p] Esta pregunta es OPCIONAL**. Describa de la manera más breve y precisa posible, el mecanismo que utiliza Pintos para efectuar los cambios de contexto (lo que ocurre desde que se llama a `schedule`).

No responda esta pregunta si no está seguro o no recuerda nada del código.