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

IIC2333 — Sistemas Operativos y Redes — 2/2017

Soluciones Interrogación 2

Jueves 28-Septiembre-2017

Duración: 2 horas
SIN CALCULADORA

1. **[15p]** *Scheduling*

   1.1) **[3p]** En un sistema operativo de propósito general, bajo qué condiciones un *scheduling* de tipo *Round-Robin* se puede comportar peor que un *scheduling* FCFS (*First-Come First-Serve*)?

   R.

   Si los procesos utilizan el mismo tiempo T, y el *quantum* $q >= T$, entonces RR se comporta **igual** que FCFS. Para que el tiempo que permanecen en el sistema (*turnaround time*) aumente, entonces q debe ser mucho más pequeño que T. Con esas condiciones aumenta el *overhead* por cambios de contexto.

   También es correcto mencionar que q debe ser menor al tiempo del proceso más corto.

   1.2) **[4p]** Describa dos procesos del tipo *CPU-bound* y dos procesos del tipo *IO-bound*.

   R.

   1p por cada uno. Procesos *CPU-bound*, deben tener poca o ninguna interacción con el usuario. Ej: compresión de audio/imágenes, compilación, cómputos científicos, simulaciones (incluye juegos que requieren simular eventos), *rendering*, pruebas de fuerza bruta, ...

   Procesos *IO-bound* deben ser principalmente interactivos. Ej: procesadores de texto, navegadores web, procesos de transmisión de datos (*streaming*), procesos con alta lectura/escritura de archivos, ...

   1.3) **[8p]** Considere un procesador que utiliza un *scheduler* Round-Robin con prioridades. Cada proceso nuevo recibe un *quantum* inicial q. Cuando un proceso consume completamente su *quantum* sin bloquearse, su próximo *quantum* se establece al doble del actual. Si un proceso se bloquea antes que su *quantum* expire, su nuevo *quantum* se restablece a q. Para efectos de esta pregunta, suponga que cada proceso utiliza una cantidad finita de tiempo de CPU.

   a) **[4p]** Suponga que el *scheduler* da mayor prioridad a los procesos con mayor *quantum*. ¿Es posible que se produzca **inanición** en este sistema? Justifique.

   R.

   NO. Los procesos que tienen mayor prioridad son los procesos *CPU-bound*, ya que ven aumentado su *quantum*. Sin embargo, estos procesos eventualmente terminan. Los procesos *IO-bound* permanecen siempre con la menor prioridad, pero los nuevos procesos *CPU-bound* también entran a la cola con la menor prioridad donde compiten en modalidad Round Robin con los *IO-bound*. El peor caso es que lleguen procesos *CPU-bound* de manera contínua, pero éstos no pueden dejar postergados a los *IO-bound* ya que entran a la misma cola. Los procesos *IO-bound* podrían verse postergados pero no de manera indefinida.

   b) **[4p]** Suponga que el *scheduler* da mayor prioridad a los procesos que tienen menor *quantum*. ¿Es posible que se produzca **inanición** en este sistema? Justifique.

   R.

   SÍ. En este caso lo peor que puede pasar es que lleguen contínuamente procesos *IO-bound*. Estos procesos se mantendrán en la cola con menor *quantum* (y mayor prioridad). Los procesos *CPU-bound* verán su *quantum* incrementado y pasarán a colas com menor prioridad. Si llegan contínuamente procesos *IO-bound*, los procesos con menor prioridad quedarán postergados sin opción de ejecutar.

2. **[15p]** *Threads*

2.1) **[7p]** Considere la implementación de *threads* a nivel de *kernel*.

a) **[4p]** ¿Qué información necesita ser almacenada y restaurada durante un cambio de contexto (*context switch*) entre dos *threads* del mismo proceso?

R.

Para un *thread* se debe incluir al menos, registros (*Program Counter*, *Stack Pointer*, ...), estado, identificador de *thread*. Todo esto debe ser almacenado en el TCB (*Thread Control Block*). No es correcto mencionar contenido del *stack*, pues no se copia. Se puede contar información de contabilidad (tiempo de ejecución, tiempo en *waiting*).

Se debe mencionar al menos 3 estos ítemes. Se puede considerar PC ó SP fuera del conjunto de registros.

b) **[3p]** Respecto a la pregunta anterior, ¿qué ocurre si el cambio de contexto se produce entre *threads* de procesos distintos? ¿hay alguna diferencia, o es lo mismo?

R.

Si el cambio de contexto se produce entre *threads* de procesos distintos no es necesario almacenar nada adicional. Información propia del proceso como espacios de memoria (registro de ubicación de tabla de páginas), archivos abiertos, señales, etc, ya se encuentran almacenadas en el PCB. La información de espacios de memoria no puede ser modificada por cada *thread* (la asigna el sistema operativo), por lo que no cambia en cada turno de ejecución.

Sí es necesario restaurar información de espacios de memoria del proceso nuevo (registro de ubicación de tabla de páginas), ya que estos registros se deben consultar en *hardware*.

2.2) **[8p]** Considere una arquitectura *multicore* y un programa *multithreaded* ejecutando sobre un sistema operativo que utiliza el modelo de *threads* híbrido. Sea C la cantidad de cores, T_K la cantidad de *kernel threads* disponibles en el sistema, y T_U la cantidad de *user threads* generados por el programa, con $1 < C < T_U$. Para cada escenario a continuación, describa como se ve afectado el rendimiento del sistema (por ejemplo: ¿algo se satura?, ¿hay thrashing?, ¿no pasa nada?, ¿funciona más rápido/lento?)

a) $T_K < C$

R.

En este caso hay algunos *cores* ($C - T_k$) que no tendrán *threads* en ejecución. El sistema queda subutilizado, ya que el sistema operativo solamente planifica *kernel threads*. Aunque haya más *user threads*, éstos quedarán no asignados y tendrán que esperar hasta que un *kernel thread* quede liberado, aún habiendo *cores* libres.

b) $C < T_K < T_U$

R.

En este caso se puede argumentar que el *hardware* estará completamente utilizado; que todos los *kernel threads* estarán ocupados y asignados a *user threads* (salvo que los *user threads* ejecuten muchas operaciones de entrada/salida). Sin embargo se debe considerar que pasado cierto límite la cantidad de *kernel threads* puede saturar el sistema operativo y provocar *thrashing*. Por esta razón usualmente los sistemas operativos limitan la cantidad de *kernel threads* que pueden ser creados.

3. **[17p]** Sincronización

3.1) **[3p]** Considere un sistema operativo *multithreaded* que provee *locks*. `Lock.acquire()` bloquea un proceso hasta que `Lock` está disponible, y luego lo toma. `Lock.release()` libera `Lock`. Además se define una primitiva `Lock.isFree()`, que **no** bloquea, sino que retorna `True` si `Lock` está disponible, y `False` si no lo está. ¿Hay alguna ventaja para el programador en usar `Lock.acquire()` para tomar `Lock` con seguridad, basado en el resultado de un llamado a `Lock.isFree()`? Si la hay, describa la ventaja. Si no la hay, indique por qué.

R.

No hay ninguna ventaja. En el intertanto en que se ha obtenido la respuesta de `Lock.isFree()` y se ejecute `Lock.acquire()`, otro *thread* podría haber tomado el `Lock`, y la ejecución de `Lock.acquire()` se bloquearía de todas maneras. Por esta razón los sistemas no suelen implementar un `Lock.isFree()`.

3.2) **[6p]** Considere dos *threads* concurrentes `T1` y `T2` pertenecientes al mismo proceso, donde cada uno ejecuta una de las siguientes funciones:

```
1  int s = 0;    // variable compartida
```

```
1  void funcT1() {
2    int i;
3    for(i=1;i<5;i++)
4      s += 1
5  }
```

```
1  void funcT2() {
2    int i;
3    for(i=1;i<5;i++)
4      s -= 1
5  }
```

a) **[3p]** ¿Qué rango de valores podría tomar `s` después que ambos *threads* han terminado?
R.
Cualquier valor entero en el intervalo $[-5, 5]$. Como vimos en los ejemplos de clase, cada operación de la línea 4 puede ser interrumpida entre el momento que lee el valor actual de `s` en un registro, y que aplica la operación. El valor que queda es el valor del último *thread* que ejecuta la asignación de un nuevo valor a `s`, y eso permite que algunas operaciones sean sobre escritas.

b) **[3p]** Modifique `funcT1()` ó `funcT2()`, ó ambas para que `T2` se ejecute **siempre** antes que `T1`. Puede agregar más variables globales, pero no puede modificar el orden de creación de los *threads*, ni utilizar `wait()` o `join()` entre ellos. *Hint:* use primitivas de sincronización.
R.

```
1  int s = 0;    // variable compartida
2  Semaphore sem(0); // semáforo compartido, inicializado en 0
```

```
1  void funcT1() {
2    sem.P();  // se bloquea aqui
3    int i;
4    for(i=1;i<5;i++)
5      s += 1
6  }
```

```
1  void funcT2() {
2    int i;
3    for(i=1;i<5;i++)
4      s -= 1
5  }
6  sem.V(); // permite que T1 ejecute
```

3.3) **[8p]** Considere un conjunto de N `threads`, donde cada *thread* ejecuta el código:

```
1  void threadFunc(int threadId) {
2    L(threadId)
3    barrier(N)
4    M(threadId)
```

donde `barrier()` es una primitiva de sincronización que detiene la ejecución de todos los *threads* hasta que N la han invocado, y luego permite que todos los *threads* continúen. De esta manera ningún *thread* puede ejecutar `M` hasta que los `N` *threads* hayan ejecutado `L`.

Implemente un código (C-like) para `void barrier(int N)` utilizando *locks*, semáforos o variables de condición.

R. Varias alternativas de implementación. Una es:

```
1  int arrived = 0;          // variable compartida
2  Lock lock = new Lock();   // lock de acceso, para las variables de condicion
```

```
3
4   void barrier(int N) {
5     lock.acquire();
6     arrived += 1;
7     if(arrived < N)
8       cond.wait(lock);          // si no es el ultimo, espera
9     else {
10      for(int i=0; i<N-1; i++)  // libera N-1 threads
11        cond.signal(lock);
12      arrived = 0;
13    }
14    lock.release();
15  }
```

Otras alternativas válidas incluyen utilizar un arreglo de N semáforos o *locks*. La implementación presentada no soporta bien la reentrada a la barrera, esto es, si un *thread* vuelve a entrar a la barrera antes que todos hayan tenido la oportunidad de salir de ella. Para soportarlo, se debe utilizar dos *locks*, de manera de asegurarse que un *thread* que salió de una barrera no se vuelve a bloquear en la misma barrera. Sin embargo, la solución presentada cumpe con lo solicitado, ya que `threadFunc` solo ejecuta un llamado a la barrera.

4. **[13p]** *Deadlocks*

4.1) **[4p]** "Una dependencia circular siempre produce *deadlock*". Indique bajo qué condiciones esta aseveración es verdadera o falsa, o si siempre lo es.

R.

La afirmación es verdadera si y solamente si existe una instancia de cada recurso a compartir.

Cuando existe más de una instancia de un recurso, la existencia de un ciclo no asegura *deadlock*, pues un recurso asignado a un proceso que no es parte del ciclo, podría ser liberado, y entonces el ciclo se rompe. Lo que sí es siempre cierto es que "si hay *deadlock* entonces debe haber un ciclo en alguna parte"

4.2) **[9p]** Considere un sistema con 4 procesos $\{P_1, P_2, P_3, P_4\}$, y 3 tipos de recursos $\{R_1, R_2, R_3\}$, donde la cantidad de recursos **totales** de cada tipo es $existent = \{12, 9, 12\}$. El sistema se encuentra en esta situación:

Allocated	R_1	R_2	R_3
P_1	2	1	3
P_2	1	2	3
P_3	5	4	3
P_4	2	1	2

Max	R_1	R_2	R_3
P_1	4	9	4
P_2	5	3	3
P_3	6	4	3
P_4	4	8	2

donde *Allocated* indica cuántos recursos de cada tipo se encuentra asignados a cada procesos, y *Max* indica la cantidad máxima de recursos de cada tipo que un proceso podría solicitar.

a) **[3p]** ¿Se encuentra el sistema en un estado seguro? Justifique su respuesta.

R.

El sistema se encuentra en un estado seguro. Es más fácil de ver si se considera la matriz *Need* con la cantidad de recursos que queda por solicitar.

Need	R_1	R_2	R_3
P_1	2	8	1
P_2	4	1	0
P_3	1	0	0
P_4	2	7	0

Y considerando la cantidad de recursos disponibles de cada tipo: Available $= \{2, 1, 1\}$, los procesos pueden terminar en la siguiente secuencia:

Proceso	Available ($\{2,1,1\}$)
P_3 termina	$\{7,5,4\}$
P_2 termina	$\{8,7,7\}$
P_4 termina	$\{10,8,9\}$
P_1 termina	$\{12,9,12\}$

b) **[3p]** A partir de la situación inicial, suponga que P_1 solicita 2 instancias de R_1. De acuerdo al algoritmo del banquero, ¿debe el sistema satisfacer la solicitud, o dejar a P_1 en espera? Justifique su respuesta.

R.

Si se asigna la solicitud $Req_1 = \{2,0,0\}$, entonces quedaría Available $= \{0,1,1\}$, y

Need	R_1	R_2	R_3
P_1	0	8	1
P_2	4	1	0
P_3	1	0	0
P_4	2	7	0

Con estas condiciones, ningún procesos que solicite todos los recursos que le faltan podría terminar, por lo tanto el sistema llegaría a un estado **inseguro**. El sistema debe dejar la solicitud de P_1 en espera.

c) **[3p]** A partir de la situación inicial, ¿cuál es la cantidad máxima de recursos de cada tipo que P_1 podría solicitar simultáneamente de manera que el sistema la satisfaga inmediatamente sin arriesgar un *deadlock*? Justifique su respuesta.

R.

A partir de Available $= \{2,1,1\}$. Sabemos que si P_1 pide 2 instancias de R_1, queda en un estado inseguro. Si P_1 solicita 1 instancia de R_1, entonces Available $= \{1,1,1\}$ y

Need	R_1	R_2	R_3
P_1	1	8	1
P_2	4	1	0
P_3	1	0	0
P_4	2	7	0

Con lo cual pueden terminar:

Proceso	Available ($\{1,1,1\}$)
P_3 termina	$\{6,5,4\}$
P_2 termina	$\{7,7,7\}$
P_4 termina	$\{9,8,9\}$
P_1 termina	$\{12,9,12\}$

Si P_1 solicita 1 instancia de R_2, entonces Available $= \{1,0,1\}$, y:

Proceso	Available ($\{1,0,1\}$)
P_3 termina	$\{6,4,4\}$
P_2 termina	$\{7,6,7\}$

con lo cual ni P_1 ni P_4 pueden terminar, por lo tanto P_1 no puede pedir instancia de R_2.

Finalmente, si P_1 solicita 1 instancia de R_3, entonces Available $= \{1,1,0\}$, y:

Proceso	Available ($\{1,1,0\}$)
P_3 termina	$\{6,5,3\}$
P_2 termina	$\{7,7,6\}$
P_4 termina	$\{9,8,8\}$
P_1 termina	$\{12,9,12\}$

Por lo tanto, se obtiene que la máxima cantidad de cada recurso que puede solicitar P_1 sin pasar a un estado inseguro es: MaxReq$_{P_1} = \{1,0,1\}$