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

IIC2333 — Sistemas Operativos y Redes — 2/2017

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*)?

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

   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 se *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.

      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.

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?

      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?

   2.2) **[8p]** Considere una arquitecura *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) $T_K < C$

      b) $C < T_K < T_U$

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.

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

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

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

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

a) **[3p]** ¿Qué rango de valores podría tomar `s` después que ambos *threads* han terminado?

b) **[3p]** Modifique `functT1()` 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.

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

```
void threadFunc(int threadId) {
  L(threadId)
  barrier(N)
  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.

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.

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.

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.

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.