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

IIC2333 — Sistemas Operativos y Redes — 1/2018

Interrogación 1

Viernes 6-Abril-2018

Duración: 2 horas
SIN CALCULADORA

1. **[15p]** Responda brevemente las siguientes preguntas:

   1.1) **[3p]** Los sistemas operativos modernos de propósito general incluyen mecanismos de protección entre el *hardware* y el usuario. Describa cómo se implementa esta protección, y por qué es necesaria la participación del *hardware*.

   R. 1p. Se implementa mediante un *mode bit*. También pueden decir *kernel*, *privileged*, *system bit*, etc., o mencionar que hay múltiples modos.

   1p. Mencionar que (1) se implementa en *hardware* y (2) mediante interrupciones. Adicional: Si un proceso ejecuta una instrucción privilegiada, estando en modo no privilegiado, el *hardware* levanta una interrupción y llama al sistema operativo (que usualmente lo mata o lo ignora, pero esa es decisión del sistema).

   1p. Mencionar que el *hardware* es necesario porque, si no participa, el sistema operativo tendría que verificar cada instrucción a través de una *syscall* y sería muy lento. También se puede argumentar que si no hay soporte de *hardware*, el proceso de usuario puede hacer lo mismo que el sistema operativo y no habría protección.

   1.2) **[3p]** ¿Qué diferencia hay entre un sistema de tipo *batch* y un sistema *interactivo*, y cómo eso influye en el tipo de *scheduler* que se utiliza para cada uno de ellos? ¿Qué tipo de sistema constituyen los sistemas operativos más comunes? (Linux, Windows, macOS, Android, iOS)

   R. 1p. El sistema *batch* está pensado para procesos intensivos en CPU que trabajan sin interactuar con el usuario, o bien, privilegian el tiempo de permanencia en el sistema (*turnaround time*), mientras que los sistemas interactivos privilegian tiempos de respuesta cortos.

   1p. Para *batch* se usan *schedulers* que privilegian *turnaround time*, como SJF, o bien FCFS con prioridades. Para *interactivos* se prefieren *schedulers* que privilegian a procesos intensivos en I/O como Round-Robin o MLFQ.

   1p. Los sistemas operativos mencionados son interactivos. En ningún caso son *batch*. A lo más se puede mencionar que soportan algunas tareas *real-time*.

   1.3) **[3p]** Los sistemas computacionales poseen un *timer* que, al cumplirse, levanta una interrupción. La operación que permite establecer el valor de ese *timer* es, normalmente, una instrucción privilegiada. ¿Por qué los fabricantes proveen esta operación como una instrucción privilegiada?

   R. La instrucción que permite modificar el *timer* sirve para que el *scheduler* (que corre en *kernel mode*) pueda interrumpir a un proceso en ejecución. 3p. Si no fuera privilegiada, el usuario podría modificar este *timer* y evitar que ocurra la interrupción que llamaría al *scheduler*.

   1.4) **[3p]** Un proceso ejecutando en modo *kernel* puede acceder a todo el *hardware* del computador. ¿Cómo es posible aprovechar esto para atacar al sistema? Ej: ejecutar un miner, ejecutar un virus, enviar información de comportamiento de usuario por la red, keylogging, etc. ¿Influye la arquitectura del sistema (monolítico o *microkernel*)?

   R. 2p. Construyendo un programa que ejecute acciones sobre el *hardware* y conseguir que se instale como módulo del *kernel*.

   1p. Es más fácil en *kernel* monolítico porque el módulo se ejecuta en *kernel space*. En *microkernel* las extensiones deben comunicarse con el *kernel* mediante paso de mensajes (IPC), y el *kernel* puede protegerse

ante la ejecución de un componente (*server*) malicioso. Habría que aprovechar algún error (*bug*) del *kernel* para ejecutar el proceso malicioso.

1.5) **[3p]** Considere un sistema operativo interactivo diseñado para ejecutar de manera óptima procesos intensos en operaciones matemáticas. Las operaciones matemáticas se encuentran implementadas de manera óptima en *kernel space*, y se ha añadido la interfaz apropiada para que el programa de usuario las pueda invocar. ¿Qué ventajas provee este diseño respecto a sistemas que no poseen estas operaciones matemáticas optimizadas? Justifique su respuesta. Si no hay ventajas, indique por qué.

R. 1p. No provee ventajas. Al contrario, es desventajoso.

2p. Introduce dentro del *kernel*, código que no tiene ninguna razón de estar ahí. Hay que ejecutar las llamada via *syscall* y eso es más costoso. También se puede agregar que es mejor una biblioteca optimizada a nivel de usuario.

2. **[11p]** Escriba un código que reciba un comando del usuario, un entero N y un entero T. El código debe ejecutar el comando N veces **de manera concurrente** durante un máximo de T segundos. Si ese tiempo se cumple, los procesos que no hayan terminado deben recibir una señal `SIGTERM`.

```
// no es necesario indicar los headers
char *command
char *arguments;
int N, T;

// puede agregar mas variables aqui
read_command(command, arguments, &N, &T); // lee correctamente el comando y sus argumentos
  /** completar codigo aqui **/
```

R. Una posible solución es la siguiente:

```
// no es necesario indicar los headers
char *command
char *arguments;
int N, T;
pid_t child = (pid_t*)malloc(sizeof(pid_t)*T); // también puede ser 'new pid_t[T];'
\
// puede agregar mas variables aqui
read_command(command, arguments, &N, &T); // lee correctamente el comando y sus argumentos
for(int i=0; i<N; i++) {
   child[i] = fork();
   if(child[i] == 0) {
     exec(command, arguments);
   }
}
sleep(T);
for(int i=0; i<N; i++) {
  kill(child[i], SIGTERM);
}
free(child);
```

Esta solución genera todos los hijos, y recuerda sus *id*s. Una vez que los ha creado todos, el padre duerme por `T` segundos. Cada hijo creado ejecuta el comando entregado. Una vez que el padre termina de dormir, envía una señal `SIGTERM` a cada hijo. Es posible que pasen más de `T` segundos entre la creación de los hijos y el término del *sleep*, pero al menos padre habrá dejado pasar de `T` segundos antes de enviar la seál a los hijos. También es posible hacer un *thread* en que el padre se duerma por `T` segundos y luego haga el `kill`.

5p. La solución permite que los procesos hijo se encuentren al mismo tiempo en estado *ready*. Una solución en que el padre a cada hijo individualmente no tiene puntos en esta parte. Una solución en que cada nuevo procesos crea un nuevo hijo, y lo espera, tiene solo puntaje parcial.

2p. Se dejan pasar al menos `T` segundos antes de enviar la señal `SIGTERM`. Esto puede hacerlo directamente el procesos *padre*, o bien un *thread*.

4p. El padre almacena el `PID` de cada hijo y lo usa para enviar la señal a cada uno.

[4p] ¿Se pueden generar procesos *zombie* o huérfanos con su solución? Si es así, indique bajo qué condiciones. Si no es así, indique por qué.

R. Esta solución permite generar procesos *zombie* en la medida que el tiempo `T` sea tan grande que los procesos hijo alcanzan a terminar antes que el padre. No se producen huérfanos porque el padre solo termina después de haber enviado el `SIGTERM` a todos los hijos, suponiendo que ellos terminan a causa del `SIGTERM`.

2p. Correcta respuesta de acuerdo a la solución.

2p. Correcta justificación de la respuesta.

3. **[15p]** Responda las siguientes preguntas respecto a algoritmos de *scheduling*.

   3.1) **[5p]** El *scheduling* MLFQ fue diseñado pensando en una mezcla de procesos intensos en I/O, y procesos intensos en CPU. Una decisión de diseño es que si un proceso ejecuta una operación de I/O en tiempo t, antes de consumir su quantum q ($t < q$), entonces en su próximo turno tendrá solamente $q - t$ para ejecutar antes de ser transferido a la siguiente cola. ¿Por qué se elige este diseño en lugar de uno más simple en que el proceso siempre tiene q para ejecutar en la misma cola, independientemente de cuánto gastó en el turno anterior?

   R. Supongamos un proceso intenso en I/O que se ejecuta una operación de I/O cada $q - e$, donde e es un valor muy pequeño. Este proceso podría ejecutar muchas veces sin bajar de cola. Los procesos que se encuentren en las colas inferiores tendrán menos posibilidades de ejecutar porque el proceso que se encuentra en la cola superior tendría cada vez un q completo para ejecutar independiente de la suma del tiempo efectivamente ocupado en todos sus turnos. Esta medida termina siendo injusta para los procesos de las colas inferiores.

   2p. Mencionar que el proceso en la primera cola nunca bajaría de cola.

   1p. Mencionar que el proceso en la primera cola podría ejecutar más tiempo en la CPU. También se puede decir que este proceso estaría "engañando" a la CPU porque sería un proceso con mucho tiempo de ejecución pero que no baja de cola.

   2p. Mencionar perjuicio para procesos en colas inferiores. El proceso en la cola superior.

   3.2) **[5p]** ¿Por qué se incorpora el mecanismo de *aging* (A) en el *scheduling* MLFQ? ¿Qué ocurre si A es muy bajo, o si A es muy alto respecto al quántum de la primera cola?

   R. El mecanismo de *aging* permite que los procesos con mayor tiempo de ejecución, y que, por lo tanto, han llegado a la cola inferior, tengan posibilidades de ejecutar. Si no hubiese *aging*, los procesos de la cola con menor prioridad tendrían pocas posibilidades de ejecutar si permanentemente llegan procesos a la primera cola.

   Si A es muy alto, los procesos que quedan en la cola inferior tiene menos probabilidad de ejecutar y tendrían algo de inanición.. Si A es muy bajo, esto se comporta como un *round-robin* porque todos los procesos que gastan su quántum, y llegan a la cola inferior, volverían rápidamente a la primera cola.

   3p. Explicar razón de que haya *aging*, por ejemplo, mencionando que pasaría si no existiese el mecanismo.

   1p. Explicar qué pasa si A es muy alto.

   1p. Explicar qué pasa si A es muy bajo.

   3.3) **[5p]** Considere dos procesos de tiempo real, con $\{p_A = 50, t_A = 25\}$, y $\{p_B = 75, t_B = 30\}$. Suponiendo que el período es igual al *deadline*, determine la secuencia de ejecución usando *Rate Monotonic* (RM) y usando *Earliest Deadline First* (EDF), hasta el tiempo 150. De acuerdo a ello, ¿es posible ejecutarlos en

cada caso (RM y EDF) cumpliendo sus restricciones? Por los casos en que la respuesta es positiva, ¿es posible ejecutar un tercer proceso?, y de ser así, ¿cuál debería ser la utilización máxima de ese tercer proceso? La utilización de un proceso es $\frac{t_i}{p_i}$.

R. 1p. No se puede con RM. Basta que ilustren hasta el momento en que se pierde un *deadline*.

1p. Se puede con EDF. Debe estar hasta el tiempo 150.

1p. No se puede un tercer proceso con RM. Si dijo que no se podía planificar con RM y no dice nada en esta parte, también se considera OK. Si dijo que se podía con RM y hace un cálculo en base a la utilización máxima de RM, está OK, siempre que el análisis esté correcto para el tercer proceso.

1p. Se puede un tercer proceso con EDF.

1p. Utilización máxima para ese tercer proceso: 10 %.

4. **[15p]** Responda brevemente las siguientes preguntas respecto al uso de *threads*

   4.1) **[2p]** Describa dos ventajas de usar *threads* (sean *kernel* o *user*) en un programa de usuario.

   R. Se puede mencionar 1p cada una, máximo 2p:

   - Facilidad de diseño. Tareas separadas, desde el punto de vista de la ingeniería de *software*.
   - Posibilidad de aprovechar ejecución concurrente.
   - Espacios de memoria compartidos (data, *heap*), y privados *stack*, sin necesitar paso de mensajes.
   - Más livianos de crear que un proceso.

   NO es una ventaja que el programa sea más rápido. Eso depende de las condiciones en que se ejecuta, en particular de la cantidad de unidades de ejecución físicas (*cores*) disponibles.

   4.2) **[4p]** Considere el diseño de un editor de texto con múltiples *tabs* (pestañas) para cada archivo abierto. ¿Qué utilizaría usted para implementar cada *tab*: *threads* o procesos? ¿Por qué?

   R. Ambas respuesta son posible, siempre que estén bien justificadas.

   Si indica *threads*: destacar que se pueden compartir bibliotecas o plugins fácilmente, que la creación es barata.

   Si indica procesos: destacar que hay independencia de ejecución en caso que alguno se bloquee, ya que no bloquea a la aplicación completa.

   4.3) **[9p]** Considere un sistema con 4 *cores* físicos. Un usuario ejecuta un proceso P con 20 *threads*, y el sistema utiliza una implementación de *threads* híbrida. No hay más procesos ejecutando que puedan afectar el rendimiento de P. Describa el comportamiento del sistema en cuanto a: (1) utilización de CPU por parte de los *threads*, y (2) uso de cores por parte de los *kernel threads*, para cada uno de los siguientes escenarios:

   a) **[2p]** El sistema asigna menos de 4 *kernel threads* a P.
   R. 2p. Todos los *threads* ejecutan el $100\,\%$ de su tiempo, y hay *cores* no utilizados.

   b) **[2p]** El sistema asigna exactamente 4 *kernel threads* a P.
   R. 2p. Todos los *threads* ejecutan el $100\,\%$ de su tiempo, y todos los *cores* están utilizados con un *kernel thread*.

   c) **[2p]** El sistema asigna entre 4 y 20 *kernel threads* a P.
   R. 2p. Los *threads* ejecutan menos del $100\,\%$ de su tiempo, y todos los *cores* están utilizados con un *kernel thread*.

   d) **[2p]** El sistema asigna más de 20 *kernel threads* a P.
   R. 2p. Los *threads* ejecutan menos del $100\,\%$ de su tiempo, todos los *cores* están utilizados con un *kernel thread*, pero hay *kernel threads* no utilizados.

   [1p] ¿Cuál escenario permite un **MENOR**[1] tiempo de ejecución?

   Estrictamente es el escenario *b*. Puntaje parcial si ponen al mismo nivel los escenarios *c*, *d*, *e*, sin dejar claro que el *b* es mejor, porque aquellos pueden incurrir en demoras por *scheduling* de *threads*. El *a* por ningún motivo.

[1] En el enunciado decía "mayor"

API de procesos y *threads*

- `pid_t fork()` retorna 0, en el contexto del hijo; retorna *pid* del hijo, en el contexto del padre.
- `int exec(char *command, char *argumentos)` recibe como parámetro un *string* con la ruta del archivo a ejecutar y sus argumentos. Si hay error retorna -1. De lo contario, no retorna.
- `pid_t wait(pid_t p, int *exitStatus)` espera por el proceso `p`, y guarda el estado de salida de `p` en `exitStatus`. Si `p` es -1, espera por cualquiera. Retorna el `pid` del proceso que hizo `exit`.
- `exit(exitCode)` termina al proceso entregando un `exitCode` para su padre.
- `sleep(int secs)` duerme al proceso que lo invoca, durante `secs` segundos
- `kill(pid_t pid, int sig)` envía la señal `sig` al proceso `pid`. Si `pid` no exista, retorna -1.
- `tid_t thread_create(f, args)` crea un nuevo (kernel) *thread* que empieza ejecutando la función `f(args)`. Retorna un *thread Id*.
- `thread_join(tid_t tid)` espera que el *thread* `tid` termine
- `thread_exit()` termina al *thread actual*
- `thread_yield()` entrega la CPU y vuelve a la cola *ready*.