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*.

   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)

   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?

   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*)?

   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.

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 **/
```

   [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.

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?

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?

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 $\frac{t_i}{p_i}$.

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.

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?

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.

b) **[2p]** El sistema asigna exactamente 4 *kernel threads* a P.

c) **[2p]** El sistema asigna entre 4 y 20 *kernel threads* a P.

d) **[2p]** El sistema asigna más de 20 *kernel threads* a P.

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

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*.

[1] En el enunciado decía "mayor"