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

IIC 2333 — Sistemas Operativos y Redes — 2/2015

Interrogación 1

Jueves 27-Agosto-2015

Duración: 2 horas

1. **[14p]** 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) Un sistema operativo puede proveer *multitasking* sin proveer necesariamente *multiprogramación*

   1.2) Un sistema operativo monolítico es riesgoso porque no posee bit de protección (*dual mode*, ó *privileged mode*)

   1.3) Un sistema operativo basado en microkernel que provee la mismas funcionalidades que otro sistema basado en un kernel con módulos, llegará a ser mucho más rápido ya que es más pequeño.

   1.4) El vector de interrupciones del sistema operativo mantiene, entre otras cosas, los Program Counter (PC) de cada proceso que ha sido interrumpido.

   1.5) Al ejecutar la instrucción `kill PID` (enviar la señal `SIGTERM`) al proceso `PID`, el PCB del proceso se sacado de la tabla de procesos.

   1.6) Una de las maneras de deshacerse de un proceso *zombie* es que su padre haga `wait` por él.

   1.7) Un *scheduler* que asigna poco tiempo a cada proceso mejora el *multitasking*, pero puede provocar contención en la cola.

   1.8) Un proceso no puede acceder a memoria de otro proceso, a menos que éste haya creado una región de memoria compartida y le haya dado acceso.

   1.9) Un proceso perteneciente al usuario `root` o `Administrator` puede acceder en todo momento a memoria de cualquier proceso.

   1.10) Un proceso que está en estado *waiting*, al recibir la señal que estaba esperando, pasará al estado *running*.

   1.11) Si un *thread* intenta modificar el contenido del *stack* de otro *thread* dentro del mismo proceso, el sistema operativo genera una excepción y lo termina.

   1.12) Una desventaja de la asignación de *threads* en el modelo *many-to-one* es que, si un *user thread* ejecuta una llamada bloqueante al sistema, esto hace que todos los otros *user threads* asignados al mismo *kernel thread* también queden bloqueados.

   1.13) Una llamada a `fork()` a partir de un proceso que utiliza una alta cantidad de memoria es siempre costosa porque se debe copiar toda la memoria desde el proceso padre al proceso hijo.

   1.14) El valor de retorno de una llamada a `exec()` nunca es leído por ningún proceso.

2. **[24p]** Para las siguientes preguntas considere las descripciones de `fork()`

   - `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()`

   2.1) **[6p]** Construya un código único en C que permita crear un árbol ternario de procesos con 3 niveles, usando `fork()`, `exit()`, `exec()`, `wait()`. En el primer nivel, el proceso padre P_0 crea tres hijos, P_1, P_2 y P_3. En el segundo nivel, los hijos P_1, P_2 y P_3 crean, cada uno, tres hijos (procesos hoja). En el tercer nivel los procesos hoja imprimen su PID, su PPID, y un mensaje indicando que terminarán. Cuando **todos** los descendientes de P_0 han terminado, éste deberá ejecutar el comando `date` y terminar.

   2.2) **[6p]** Dibuje el árbol de procesos que resulta de ejecutar el siguiente código, e indique cuantas veces se imprime el texto de la línea 4, y cuántas el texto de la línea 6. Puede considerar que el proceso principal es P_0 y numerar secuencialmente los siguientes.

```
1 int main(int argc, char *argv[]) {
2   for(int i=0;i<3;i++) {
3       fork();
4       printf("Wow!\n")
5   }
6   printf("Such Banner!\n");
7   return 0;
8 }
```

   2.3) **[6p]** Dibuje el árbol de procesos que resulta de ejecutar el siguiente código, e indique cuántas veces se imprime el texto de la línea 3. Considerar que el proceso principal es P_0 y numere secuencialmente los siguientes.

```
1 int main(int argc, char *argv[]) {
2   fork() && (fork() || fork());
3   print("Forked!\n");
4   return 0;
5 }
```

   2.4) **[6p]** Para el siguiente código indique cuál de las siguientes salidas son posibles:

   - (1) Se imprime `A(0)` ... `A(99)`, y luego 100 veces `B(0)` ... `B(99)`
   - (2) Se imprime `B(0)` ... `B(99)`, y luego 100 veces `A(0)` ... `A(99)`
   - (3) Se imprime `A(0)` ... `A(99)`, y nada más porque el padre termina.
   - (4) Se imprime `A(0)B(0)` ... `A(99)B(99)`.
   - (5) Se imprimen los 100 `A(i)` y los 100 `B(i)` en orden creciente de i
   - (6) Se imprimen los 100 `A(i)` y los 100 `B(i)` en cualquier orden (Ej: `A(4)A(2)B(1)B(99)...B(10)A(3)`)

```
1 int main(int argc, char *argv[]) {
2   if(pid=fork())
3     for(int i=0;i<100;i++) printf("A(%d)",i);
4   else
5     for(int i=0;i<100;i++) printf("B(%d), i");
6   if(pid) wait(NULL); else exit(0);
7   return 0;
8 }
```

3. **[22p]** Responda brevemente las siguientes preguntas:

   3.1) **[4p]** Para las siguientes operaciones, indique si ella deben ser ejecutadas solamente en modo privilegiado (*kernel*), o si pueden ser ejecutadas en modo usuario (no es necesario justificar).

   - (1) Deshabilitar las interrupciones durante un tiempo limitado
   - (2) Aplicar una operación sobre un rango de posiciones de memoria del proceso
   - (3) Detener el sistema operativo (instrucción *halt*)
   - (4) Modificar el *bit mode*

   3.2) **[4p]** Los sistemas monolíticos se caracterizan por ser más eficientes que los sistemas basados en módulos o los sistemas basados en microkernel. Sin embargo, la mayoría de los sistemas operativos actuales no han sido diseñados siguiendo puramente el modelo monolítico. Mencione y justifique dos razones (distintas) por las cuales el modelo monolítico es menos popular hoy.

   3.3) **[4p]** Suponga que desa implementar un sistema de IPC llamado *memoria compartida por mensajes*. Este sistema permite que un proceso A defina una región de memoria compartida en un proceso y le envíe el puntero de esa región a otro proceso B (sin necesidad de hacer un *syscall* para crear la región) usando `send()`. Cada vez que el proceso B desea utilizar una porción de la memoria compartida de A, le envía una solicitud a A usando `send()`, e indicando la dirección a la que desea acceder.

   Mencione y justifique una ventaja y una desventaja de este esquema respecto a los modelos individuales de memoria compartida y de paso de mensajes.

   3.4) **[6p]** Considere un proceso intensivo en CPU que toma tiempo T para ejecutar.

   a) ¿Cuál es la aceleración máxima que se puede obtener si solamente el $25\,\%$ de su código puede ser paralelizado?

   b) Suponiendo que el $75\,\%$ del código es paralelizable. ¿Cuántos *threads* se necesitan para que la nueva versión sea 3 veces más rápida que la secuencial?

   c) Suponiendo que el $75\,\%$ del código es paralelizable. ¿Cuántos *threads* se necesitan para que la nueva versión sea 5 veces más rápida que la secuencial?

   3.5) **[4p]** ¿Por qué los sistemas operativos limitan la cantidad máxima de *kernel threads* que pueden ser creados?