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

IIC 2333 — Sistemas Operativos y Redes — 1/2016
Interrogación 1

Miércoles 06-Abril-2016

Duración: 2 horas

1. **[12p]** Considere un supermercado con múltiples cajas abiertas y una única cola de atención a clientes. Suponga que cada cliente es un proceso y que cada caja es una CPU.

   1.1) ¿Cómo se reflejan en esta situación los conceptos de *multiprogramación* y *multitasking*? ¿Ocurren?

   1.2) Suponga ahora que hay solamente una caja abierta. ¿En qué cambia la respuesta a la pregunta anterior?

   1.3) Suponga que nuevamente hay múltiples cajas abiertas. ¿Cómo se pueden representar en este escenario los conceptos de *affinity* y *load balancing*?. Haga los supuestos que considere razonables.

2. **[12p]** Respecto a los conceptos de sistemas operativos y procesos:

   2.1) El sistema operativo permite que un usuario utilice el computador. Los primeros sistemas de cómputo no poseían sistema operativo, sin embargo eran utilizados. ¿Cómo se explica esto?

   2.2) En sus inicios muchos sistemas operativos permitían a los programas de usuario acceder directamente al *hardware* (video, disco, red). Actualmente esas instrucciones solo se pueden hacer en *kernel mode*, lo que implica ejecutar *syscalls*. Explique dos razones por las que se tomó esta decisión de diseño.

   2.3) Algunas *syscall*s de la API *POSIX* (como `mount` o `kill`) no tienen un equivalente en (*WinAPI*), y viceversa. ¿Qué consecuencias tiene esto para el programador?

3. **[18p]** Respecto de *threads*, *scheduling* y *sincronización*:

   3.1) **[3p]** Una condición de competencia (*race condition*) ocurre cuando dos procesos o *threads* intentan acceder a un recurso compartido. Sin embargo, en sistemas con una CPU (1 *core*), solamente un proceso puede estar en ejecución en cualquier instante de tiempo. ¿por qué, aún en estos sistemas se necesitan instrucciones de sincronización para entrar a una sección crítica?

   3.2) **[3p]** En un algoritmo de *scheduling* con *round-robin*, *quantum* Q, tiempo S de *context-switch*, llegan procesos con ráfagas de CPU (*CPU-burst*) de promedio T. Describa brevemente el comportamiento del sistema para las siguientes situaciones: (a) $Q >> T$, (b) $S < Q < T$, (c) $Q = S << T$

   3.3) **[6p]** Considere la siguiente extensión de la solución de Petersen, para el problema de exclusión mutua, con N procesos. ¿Cumple con las propiedades de exclusión mutua? Indique por qué sí o por qué no.

```
1   int turn = 0; /* arbitrario */
2   int flag[3] = {false, false, false};
3   int me = ? // me = {0, 1, 2}
4   do {
5     flag[me] = true;
6     turn = (me+1)%3;
7     while( (flag[(me+1)%3] || flag[(me-1)%3]) && (turn != me) ); {}
8     /* Seccion critica */
9     flag[me] = false;
10    /* ... ... ... */
11  } while ('3');
```

3.4) **[6p]** ¿Qué algoritmo usaría para un sistema donde la mayoría de los procesos son *CPU-bound*? ¿Qué algoritmo usaría para un sistema donde la mayoría de los procesos son *I/O bound*? Explique por qué.

4. **[18pt]** 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.
   - `pid_t waitpid(pid_t p, int *exitStatus, int options)` espera por el proceso `p`, y guarda el estado de salida de `p` en `exitStatus`. `options=0` es para opciones adicionales.
   - `create_thread( void(*f)())` recibe como parámetro una función a ejecutar por un nuevo *thread*, y lo deja en estado *ready*.

4.1) **[6p]** Explique las diferencias en la salida de los siguientes códigos.

```c
int i=0;

void f() {
  for( ; i<3; i++) {
    fork();
    printf("%d: Wow!\n", i);
  }
}

int main(int argc, char *argv[]) {
  f();
  scanf("%c");
  printf("Such PintOS!\n");
  return 0;
}
```

```c
int i=0;

void f() {
  for( ; i<3; i++) {
    create_thread(f);
    printf("%d: Wow!\n", i);
  }
}

int main(int argc, char *argv[]) {
  f();
  scanf("%c");
  printf("Such PintOS!\n");
  return 0;
}
```

4.2) **[6p]**Para este código visto en clases, explique por qué se generan procesos *zombie*, y cuantos se generan:

```c
int main(void) {
        pid_t pids[10];
        int i;
        for (i = 9; i >= 0; --i) {
           pids[i] = fork();
           if (pids[i] == 0) {
              sleep(i+1);
              exit(0);
           }
        }
        for (i = 9; i >= 0; --i)
           waitpid(pids[i], NULL, 0);
        return 0;
}
```

4.3) **[6p]** Un desglose del tiempo de ejecución del código de la pregunta anterior (una ejecución real) es:

```
cruz@cruzpro:examples$ time ./waitZombies
real    0m10.007s
user    0m0.002s
sys     0m0.006s
```

Expliqué por qué los valores de `user` y `sys` no suman el tiempo real (o *wall-clock time*), y qué pasa con esa diferencia.