

Sistemas Operativos I

1er Control - 21 de marzo de 2002

Modelo 111

TEORÍA (5 puntos)

Se proponen 10 preguntas tipo test. Cada pregunta tiene cuatro opciones, de las cuales sólo una es válida. Cada pregunta acertada valdrá 0.5 puntos. Cada fallo restará un tercio del valor asignado a la pregunta. Una pregunta no contestada valdrá 0 puntos.

Nota aclaratoria: La llamada *fflush*(stdout) fuerza el volcado inmediato del buffer de pantalla.

1. Sean dos programas A y B, escritos en lenguaje C. Los dos procesos comparten las siguientes declaraciones variables:

```
#define FALSE 0
#define TRUE 1
int interesado[2] = {0,0}
int x = 0;
```

/* Código */

<pre>main() { int i=0; for (i=0; i<5; ++i) { interesado[0] = TRUE; while(interessado[1] == TRUE); ++x; if(i==4) printf("%d\n",x), fflush(stdout); interesado[0] = FALSE; } }</pre>	<pre>main() { int j=0; for (j=0; j<5; ++j) { interesado[1] = TRUE; while(interessado[0] == TRUE); ++x; if(j==4) printf("%d\n",x), fflush(stdout); interesado[1] = FALSE; } }</pre>
PROGRAMA A	PROGRAMA B

La instrucción de incremento en C se descompone en tres instrucciones máquina ininterrumpibles: 1) carga el valor de la variable *x* en el acumulador, 2) incrementa el acumulador y 3) almacena el valor del acumulador en la posición de memoria correspondiente a la variable *x*.

Suponiendo que los dos programas se ejecutan de manera concurrente, dí cual de las siguientes frases es verdadera:

- a) La salida por pantalla será siempre:
9
10
- b) La salida por pantalla será siempre dos números cualesquiera entre 2 y 10, imprimiéndose en la primera línea el número más bajo y en la segunda el número más alto.
- c) La salida será siempre:
5
10
- d) Es posible que ninguno de los procesos imprima un valor de salida

2. En un sistema operativo con dos estados de suspensión, tal como el correspondiente al diagrama de estados más completo explicado en clase, di cuál de las siguientes transiciones entre estados **NO** es posible:

- | | | |
|---------------------------|---|-------------------------------|
| a) NUEVO | → | LISTO Y SUSPENDIDO |
| b) <u>LISTO</u> | → | <u>BLOQUEADO Y SUSPENDIDO</u> |
| c) BLOQUEADO Y SUSPENDIDO | → | BLOQUEADO |
| d) EN EJECUCIÓN | → | BLOQUEADO |

3. Di cuál de las siguientes afirmaciones es **cierta**:

- a) Todo sistema operativo Multiprogramado es siempre un sistema operativo de Tiempo Compartido
- b) Un sistema operativo de Tiempo Compartido es Multiprogramado
- c) El objetivo prioritario de los sistemas operativos de Tiempo Compartido es maximizar la utilización del procesador
- d) Los sistemas operativos Multiprogramados no requieren protección de memoria alguna por software o por hardware.

4. Sea el siguiente programa:

```
main() {  
    int *var, *temp, id, i;  
    var = (int *)malloc (5 * sizeof(int));  
    temp = var;  
    for(i=0; i<5; ++ i) *var++ = i;  
    id = fork();  
    switch(id) {  
        case -1:  
            exit(-1);  
        case 0:  
            for(i=0; i<5; ++ i) *--var = i;  
            break;  
        default:  
            break;  
    }  
    printf("%d\n",*temp), fflush(stdout);  
}
```

Sin considerar el orden en que los procesos imprimen el resultado, se obtendrá por pantalla lo siguiente:

- a) Un cero y un cuatro
- b) Un cinco
- c) Dos cuatros
- d) Dos ceros

5. Di cuál de las siguientes afirmaciones es verdadera.

En los procesos UNIX:

- a) La memoria reservada mediante un malloc() para una variable global de tipo puntero a entero se reserva siempre en la Pila del proceso
- b) Las direcciones de retorno de las llamadas a los procedimientos se almacenan en el segmento de Datos del programa
- c) Dos procesos ejecutando el mismo código pueden compartir el segmento de texto mediante las Estructuras de Texto
- d) El tamaño de la Pila del proceso es fijo, y se establece en el momento de la carga del proceso

6. Teniendo en cuenta que TS es la instrucción Test and Set (Comparar y Fijar), indicar cuál es el valor que obtiene x tras ejecutar 10 veces el proceso de forma simultánea:

```
/*Variables compartidas */
```

```
int cerrojo=1;
```

```
int x=0;
```

```
/* Código */
```

```
proceso()
```

```
{
```

```
    register int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
    {
```

```
        while (!TS((cerrojo))
```

```
        {
```

```
            cerrojo = 0;
```

```
        }
```

```
    exit (0);
```

```
}
```

- a) El resultado es no determinista pudiendo dar cualquier valor entre 10 y 100
- b) Puede haber un interbloqueo.
- c) Es siempre 100.
- d) La sección crítica no está protegida y el resultado es, por tanto, impredecible.

7.- ¿Cuál es el resultado que se obtiene en el siguiente código?

```
Proceso ()
{
    int x=0;
    int status;

    printf ("%d ", x), fflush(stdout);
    if (x==10) exit(0)
    if (fork()) wait (&status), exit (0);
    else ++x, Proceso(), exit(0);
}
```

a) La serie ordenada 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

b) Se conseguirán infinitos ceros.

c) Se conseguirá un cero e infinitos unos.

d) Cualquier combinación aleatoria de los números 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 y 10.

8.- ¿Cuál es el resultado del siguiente código?

```
main()
{
    printf ("%d ",proceso()), fflush();
    exit(0);
}

int proceso ()
{
    if (!fork()) return 1;
    else return 0;
}
```

a) 0

b) 1

c) 0 1

d) No da un resultado determinista.

9.Cuál de las siguientes frases **no es cierta**, relativa a los Sistemas Operativos por capas:

a) Favorecen la modularidad, y por lo tanto la depuración del sistema operativo

b) Pierden eficiencia (con respecto a los sistemas monolíticos), debido a la necesidad de transferir solicitudes entre capas del sistema

c) Provocan problemas de diseño a la hora de distribuir los servicios del Sistema Operativo en capas

d) Favorece la distribución de tareas entre distintos ordenadores, y por lo tanto es la metodología habitual en sistemas operativos distribuidos

10.Cuál de las siguientes frases **es cierta**, relativa al algoritmo de la panadería (Bakery)

a) No garantiza la exclusión mutua de múltiples procesos concurrentes

b) Garantiza la exclusión mutua mediante el uso de recursos hardware

c) Previene la inanición asignando a cada proceso un identificador único que se utiliza como criterio de selección en el caso de que varios procesos tengan el mismo número de acceso a la sección crítica

d) Garantiza la exclusión mutua cuando existen sólo dos procesos concurrentes.

Sistemas Operativos I

1er Control - 21 de marzo de 2002

1

Nombre:

Apellidos:

Problema (5 ptos)

Escribe una función **crea_hijos**(int n) en C. Esta función va a crear procesos. El proceso llamante deberá comprobar su identificador de proceso. Si su identificador de proceso es un número par, el proceso creará **n** hijos (n es el único parámetro de la función), que constituirán la segunda generación de procesos. Si el identificador es un número impar, el proceso termina (más adelante se describe lo que los procesos tienen que hacer al terminar). Cada proceso hijo creado volverá a hacer lo mismo, es decir, comprobar su identificador de proceso, y si es un número par, crear otros **n** nuevos procesos, que constituirán la tercera generación. Esto se repite hasta crear la **décima (10)** generación de procesos. Los procesos de décima generación terminan, independientemente de si su identificador es par o impar.

Cada proceso que ha creado procesos hijo debe ponerse en espera hasta que todos sus hijos terminen.

Antes de terminar, la función deberá devolver el número total de procesos creados durante la ejecución de la misma. Para eso, cada proceso antes de salir debe devolver a su progenitor el número (acumulado) de procesos de generaciones inferiores que provienen de él. Esto lo harán a través del parámetro de retorno del **exit()**.

No se deben utilizar llamadas recursivas para resolver el problema.

```

crea_hijos(int n) {

register int i;
int fid; /* valor de retorno del fork() */
int hijos_creados=0; /* lleva cuenta del número de hijos creados */
int generacion=0; /* contador de generaciones */
int hijos_retorno=0; /* número de hijos que se deben esperar */
int status; /* valor de retorno utilizado en el wait() */
int error=0; /* condición de error en la terminación de los procesos de menor */
/* generación */

while(++generacion < 10) {
    hijos_creados=0;
    if((getpid()%2)==0) { /* El identificador de proceso es un número par */
        for(i=0;i<n;++i) {
            if((fid = fork()) == -1) printf("Error al crear proceso\n");
            else if(fid ==0) break;
            else ++hijos_creados;
        }
        if(i==n) break;
    } else break;

} /* fin del while que controla el número de generaciones de procesos */

if(generacion==10) exit(0); /* Los procesos de generación 10 salen */

hijos_retorno=hijos_creados;
while(hijos_creados) { /* bucle de espera a la terminación de procesos hijo */
    wait(&status);
    if(WIFEXITED(status)) {
        /* Si hay error en la salida activo un código de error que se transmitirá al padre */
        if(WEXITSTATUS(status)== -1) error=1;
        /* Sumo a hijos_retorno el número de descendientes del proceso que termina */
        else hijos_retorno += WEXITSTATUS(status);

    } else error=1;
    --hijos_creados;
}
if(error) hijos_retorno = -1;
if(generacion!=1) exit(hijos_retorno);
else return(hijos_retorno);
}

```