2001 | ELIRAR 3 y 4. | SEPTIEMBRE SEPT 2001

AMPLIACION DE SISTEMAS OPERATIVOS 11/9/2001

Alumno/a D./Dña.

Ejercicio 1.- (2 puntos, 1 por apartado)
a) Algoritmo de anillo de fichas para resolver un problema de exclusión mutua en Sistemas Distribuidos. Ventajas e inconvenientes del mismo.
b) Implemente en seudocódigo el algoritmo anterior
Solución
Este algoritmo surge como un método de resolución del problema de Sección Crítica Distribuida (SCD), a través de la exclusión mutua en un sistema distribuido. La idea del algoritmo se basa en la construcción de un anillo lógico y asignarle a cada proceso una posición en el anillo como se muestra en la Fig 1. El orden asignado a los procesos no es relevante puesto que en el algoritmo tan solo es necesario que cada proceso sepa quien es el siguiente, a él, en el anillo.

Al comenzar el algoritmo, se le da a un determinado proceso, proceso 0, una ficha, la cual circulara a través del anillo, desde el proceso i al proceso i+1 (modulo, el número de procesos del anillo) a través de mensajes entre procesos.

Cuando un proceso recibe la ficha de su vecino verifica si desea entrar en la SC. Si es así, el proceso entra en ella, ejecuta el código de la SC realizando las operaciones pertinentes, sale de la SC y pasa la ficha a su correspondiente vecino. No se permite entrar a una segunda sección crítica con la misma ficha.

Si un proceso, al recibir la ficha no desea entrar en su SC, la vuelve a pasar, por lo que si ningún proceso desea entrar en su SC, la ficha circulara a lo largo del anillo hasta encontrar un proceso que desee, en un instante, entrar en la SC.

El algoritmo resuelve el problema de la Sección Crítica Condicional puesto que:
- Presenta exclusión mutua (para que un proceso entre en la SC debe de tener la ficha que es única).
- Presenta espera limitada (ya que como máximo un proceso de anillo de tamaño n deberá esperar a que los otros n-1 procesos entren y salgan de la SC), por lo que no se produce inanición.
- Presenta progresión (a un proceso que no desee entrar en la SC no se le obliga a entrar).

El algoritmo presenta algunos inconvenientes:
- En el caso de que no aparezca la ficha, puede ser que esta se haya perdido, por lo que hay que determinar cuando se ha perdido; o puede ser que este siendo retenida por un proceso.
- Si alguno de los procesos produce un error, entonces el algoritmo también falla; aunque en este caso es suficiente con no considerar ese proceso.
- Es posible que alguno de los procesos muera, en ese caso al mandarle la ficha tendríamos un error, pero esto se subsana ya que si al cabo de un cierto tiempo de mandarle la ficha a un proceso no recibimos respuesta del mismo, consideraremos que esta muerto, y obraremos en consecuencia, y le pasaremos la ficha al siguiente proceso. Para poder hacer esto cada proceso del anillo deberá de tener información de la situación de todos los procesos en el anillo.

El algoritmo esta formado por:
- Un programa principal, que es el encargado de crear los procesos que forman el anillo, así como de mandar a cada uno de los procesos el identificador del proceso (TID) que se encuentra a su izquierda.
- Un programa denominado proceso que será el encargado de simular las actividades realizadas por cada uno de los procesos del anillo.
- Un programa llamado SC que se encargará de simular las operaciones de los procesos una vez que están en la Sección Crítica.

2001 – SEPTIEMBRE

Existe una posibilidad de recuperación de errores, de forma tal que, en el caso de que ningún proceso entre en la SC durante un tiempo predeterminado se proceda a la eliminación de todos los procesos y se vuelva a reiniciar el programa principal.

Ejercicio 2.- (2 puntos, 1 por apartado)
Analice el siguiente código respondiendo a las siguientes preguntas:

a) ¿Resuelve algún problema de sincronización entre procesos?. ¿Las tareas que aparecen en el código tienen una ejecución secuencial o paralela?. Justifique las respuestas
b) Reescriba el procedimiento Compra para que contenga tres tareas locales para mostrar una solución simétrica en la compra.

```
procedure Compra is
        task Obtener_Ensalada;

        task body Obtener _Ensalada is
        begin
                Comprar_Ensalada;
        end Obtener_Ensalada;

        task Obtener_Pescado;

        task body Obtener _Pescado is
        begin
                Comprar_Pescado;
        end Obtener_Pescado;

        begin
        Comprar_Carne;
        end Compra
```

Solución
En este código se presenta una ejecución paralela donde existe un procesador principal que llama directamente a Comprar_Carne desde el procedimiento Compra. Los otros procesadores se consideran Ccomo subsirvientes y realizan las tareas declaradas localmente como Obtener_Ensalada, y Obtener_Pescado las cuales llaman a los procedimientos respectivos de Comprar_Ensalada y Comprar_Pescado.

El ejemplo presenta la declaración, activación y terminación de tareas. Una tarea es una componente del programa como un paquete y se declara de una forma similar dentro de un subprograma, bloque, paquete o dentro del cuerpo de otra tarea. La activación de una tarea es automática . En nuestro ejemplo las tareas locales se activan cuando la unidad padre ejecuta el mandato begin siguiente a la declaración de tarea.

Debemos de considerar que el subprograma principal es considerado en si mismo como llamado por una hipotética tarea principal

Ejercicio 3.- (2 puntos, 1 por apartado)
Analice el siguiente código

```
try     {
    ServerSocket  httpd = new ServerSocket (5776, 100);
}
catch (IOException e)   {
    System.err.println(e);
}
```

y responda a las siguientes preguntas:

2001 – SEPTIEMBRE

a) ¿Cuál es su misión? Justifique la respuesta
b) Sockets en Java, diferencias entre socket cliente y socket servidor, ponga ejemplos

Ejercicio 4.- (4 puntos, 2 por apartado)
a) Analice el siguiente código indicando cuál es su cometido y si resuelve algún tipo de problema
b) Indique que misión tienen las funciones de pvm que aparecen en el mismo. Justifique las respuestas.
Solución
El ejemplo fallo muestra como matar tareas, y como notificar la salida o fallo de estas tareas. En el ejemplo se expanden varias tareas, y se muestra como una de estas tareas es eliminada por el padre.

El ejemplo de notificación de fallo
muestra como notificar cuando una tarea sale
*/

```
/* Definiciones y Prototipos para la librería PVM*/
#include <pvm3.h>
#define MAXNCHILD 20        /* Máximo número de hijos */
                            /* que generará el programa */
#define TASKDIED      11    /* Tag a usar por la tarea que manda el mensaje */

int
main(int argc, char* argv[])
{

int ntask = 3;              /* Número de tareas a generar, poner 3 por defecto */
int info;                   /* Devuelve código desde las llamadas pvm */
int mytid;                  /* Identificador de la tarea */
int myparent;               /* Identificador de la tarea padre */
int child[MAXNCHILD]        /* Array identificador de las tareas hijos */
int i, deadtid;
int tid;
char *argv[5];

mytid = pvm_mytid( )        /* Enterarse del número identificador de tarea */

/* Chequeo de error */
if (mytid < 0) {
pvm_perror(argv[0]);        /* Impresión del error */
return -1;                  /* Salida del programa */
}

myparent = pvm_parent( )    /* Encontrar el número identificador de tarea del padre */

/* Salir si existe algún error aparte de PvmNoParent */
if ((myparent < 0) && (myparent != PvmNoParent)) {
        pvm_perror(argv[0]);
        pvm_exit( );
        return -1;
        }
/* Si yo no tengo un padre entonces yo soy el padre */
if (myparent == PvmNoParent) {
        if (argc == 2) ntask = atoi(argv[1]);   /* Si en la línea de comandos se indica el */
                                                /* número de tareas a generar, obtener este número */

        /* Seguridad de que ntaks es correcto */
        if ((ntask < 1) || (ntask > MAXNCHILD)) { pvm_exit( ); return 0; }
```