

PRÁCTICA 2: Problema de los Productores/Consumidores (conurrencia de procesos)

INTRODUCCIÓN: En esta práctica se tratará de resolver el problema de los productores/consumidores utilizando semáforos como medio de sincronización y utilizando memoria compartida en los elementos comunes a los distintos procesos.

Cada productor escribirá por pantalla el instante en el que ha introducido un ítem en el búffer, el ítem introducido, y el contenido de dicho búffer. Asimismo, cada consumidor indicará el instante en el que obtiene un ítem del búffer compartido, el ítem obtenido, y el estado del búffer.

Para la creación del búffer compartido, se utilizarán las funciones IPC para el manejo de memoria compartida (shmget, shmat, shmdt, shmctl,...). Para el manejo de semáforos, se deben utilizar, entre otras, las funciones: semget, semop y semctl.

DESCRIPCIÓN GENERAL: La práctica consistirá en la creación de 4 programas:

Inicializa.c: Crea los semáforos necesarios para la sincronización de productores y consumidores. Además **crea el búffer** en el que éstos añadirán o extraerán ítems. El tamaño del búffer vendrá indicado por el parámetro `-items`. Por defecto se asumirá que el búffer tiene capacidad para 15 ítems (cada ítem es un carácter). En ningún caso, el tamaño del búffer podrá exceder el valor indicado por una constante `MAX_ITEMS`.

Sintaxis: `$inicializa -items 15`

Productor.c: Crea p procesos productores (parámetro `-p`). Cada proceso creado simula el funcionamiento de un productor, produciendo ítems (1 de cada vez) y añadiéndolos al búffer compartido. Un parámetro “times” indica el número de elementos que cada productor añadirá al búffer.

Sintaxis: `$productor -p 3 -times 30`

Consumidor.c: Crea c procesos consumidores (parámetro `-c`). Cada proceso extrae ítems del búffer (1 de cada vez) y muestra su contenido. Un parámetro “times” indica el número de elementos que cada consumidor quitará del búffer.

Sintaxis: `$consumidor -c 7 -times 30`

Finaliza.c: Una vez que todos los productores y consumidores hayan finalizado su trabajo, se encarga de liberar todos los recursos utilizados (creados por “inicializa.c”). Esto es, los semáforos creados, y el búffer compartido.

Sintaxis: `$finaliza`

PRODUCTORES/CONSUMIDORES: Los programas `productor.c` y `consumidor.c` se encargarán respectivamente de crear p procesos productores y c procesos consumidores. Para la creación de un número concreto de productores o consumidores, puede usarse la función `fork()`, como se muestra a continuación:

```
main() {
    pid_t pid;
    int i;
    int np=10; //número de procesos a crear.
    for (i=0;i<np;i++) {
        pid=fork();
        if (pid == 0) break; //un hijo
    }
    printf("\n el padre del proceso %d es %d",getpid(),getppid());
}
```

Nota: El proceso “padre” no se convertirá en un productor/consumidor, sino que esperará a que cada “hijo” termine y mostrará su status. Véase **man 2 wait** para más información.

Cada proceso productor o consumidor, en lugar de un bucle infinito, se ejecuta N veces (en función del parámetro *-times*) ejecutando el código asociado a un productor o consumidor respectivamente.

Cada vez que el productor “quiera producir” un ítem, obtendrá un carácter del siguiente modo: el primer productor siempre dará lugar a una ‘A’, el segundo a una ‘B’, el tercero a una ‘C’, y así sucesivamente (se puede asumir que no habrá más de 25 productores distintos). Esto es, si hay 3 productores, el 1º escribirá siempre ‘A’s, el 2º escribirá ‘B’s y el 3º ‘C’s.

Una vez que cada proceso accede a la sección crítica, las funciones *anadirItem()* y *obtenerItem()* dejan al productor/consumidor en espera un tiempo aleatorio entre 1 y 5 segundos (*sleep*). Además, se imprime la hora a la que empieza y termina de producir o consumir el correspondiente proceso. También se muestra el estado del búffer (número de ítems que contiene actualmente y los distintos ítems).

Todas las salidas de información a pantalla se realizarán mediante la llamada al sistema *write*.

```
char produceItem(int id_proc) {
    ...
    ...
}

void anadirItem(int id_proc, char item) {
    char mensaje [MAXCADENA];
    time_t t;
    pid_t p;
    p=getpid();
    t=time(NULL);
    sprintf (mensaje,"productor %d (%u): Comienzo anadir item %c
                a las %s",id_proc,p,item,ctime(&t));
    write (STDOUT_FILENO,mensaje,strlen(mensaje));
    /*esperar tiempo aleatorio entre 1 y 5 segundos*/

    anhadCola(item); //añade el ítem producido al búffer compartido
    mostrarBuffer();

    sprintf (mensaje,"productor %d (%u): termino de ahadir %c a las %s"
                ,id_proc,p,item,ctime(&t));
    write (STDOUT_FILENO,mensaje,strlen(mensaje));
}
```

BÚFFER COMPARTIDO: COLA CIRCULAR. El búffer que productores y consumidores comparten y donde se almacenan los diferentes ítems, se implementará como una cola circular.

```
Struct tColaChar {
    char items[MAX_ITEMS]; //vector de longitud indicada en "inicializa.c"
    int maxNum, num, ppio, final;
}
typedef struct tColaChar tBuffer
```

El número máximo de ítems que caben en el búffer (*maxNum*) se le indica al programa “inicializa”; nótese que ha de cumplirse que $maxNum \leq MAX_ITEMS$. El valor *num*, indica el número de elementos que actualmente están almacenados en la cola. Los valores *ppio* y *final* indican el principio y final de la cola; al ser una cola circular, al incrementar *ppio* y *final* de debe hacer posteriormente una operación de módulo con “*maxNum*” ($ppio = (ppio + 1) \% maxNum$).

Puesto que queremos que el búffer (la cola de ítems) sea compartido por todos los procesos consumidores y productores, no es posible declararlo simplemente como:

tBuffer buffer;

ya que esto daría lugar a que cada proceso tuviese su propio búffer (una variable de tipo *tBuffer* llamada *buffer*), pero no sería compartida entre todos ellos. Esto haría, p ej, que los procesos consumidores no podrían ver los ítems generados por los productores.

Para poder compartir el búffer, usaremos las funciones de la librería IPC para manejo de memoria compartida *shmget*, *shmat*, *shmdt*, *shmctl* del siguiente modo:

1. El proceso padre crea un área de memoria compartida con tamaño suficiente para almacenar el búffer (`sizeof(tBuffer)`), usando la llamada *shmget*(`key`). *Shmget* devuelve un número (`shmid`) que será usado como identificador de la zona de memoria compartida que se ha creado. A la función *shmget*(`key`) debe pasársele una clave (`key`), que puede obtenerse mediante la función *ftok*(`key`).

```
key_t key = ftok("/tmp", 'B');
shmid = shmget(key, .....);
```

2. Usando el identificador que devuelve *shmget*, cada proceso podrá obtener un puntero a la zona de memoria compartida (el búffer), y a través de dicho puntero acceder a dicho búffer para añadir o quitar items.

```
tBuffer *pbuffer;
pbuffer=(tBuffer *) shmat(shmid,0,0);
```

3. A partir de ahí, cada proceso podrá usar `pbuffer` como si fuese un puntero a `tBuffer` "normal", pero que se comporta ahora como una estructura compartida.
4. Al finalizar cada proceso productor o consumidor, se "desasigna" el puntero `pbuffer` usando *shmdt*.
5. Por último, el programa "finaliza.c" destruye el área de memoria compartida usando *shmctl*.

UTILIZACIÓN DE SEMÁFOROS. La implementación propuesta requiere el uso de 3 semáforos: *vacío*, *lleno* y *búffer*.

La creación de semáforos se hará mediante la llamada al sistema *semget*(`key`). Se pueden implementar las operaciones básicas para la manipulación de cada semáforo haciendo uso de la llamadas *semop*(`semid`) y *semctl*(`semid`). Finalmente la llamada *semctl*(`semid`) permitirá liberar dicho recurso. Más concretamente:

- *semget*: Permite obtener un grupo de `k` semáforos que será identificado por un identificador `semid`.
- *semop*: Opera sobre uno de los semáforos (`i`) del grupo obtenido con *semget* (grupo identificado por `semid`), y permite entre otras, hacer las operaciones P y V sobre alguno de los semáforos del grupo.
- *semctl*: se encarga de liberar un grupo de semáforos identificado por `semid` cuando ejecuta el comando `IPC_RMID`. También permite establecer un valor a un semáforo determinado, cuando a través de *semctl* se ejecuta el comando `SETVAL`.

NOTA:

Al igual que *shmget*(`key`), *semget*(`key`) recibe como parámetro un valor `key_t key`, que puede obtenerse previamente con la función *ftok*(`key`). Debemos asegurarnos de que la clave que devuelva *ftok* sea distinta que la obtenida para crear el búffer_compartido, y la misma para todos los procesos que han de "compartir" el mismo semáforo.

Básicamente *semget*(`key`) permite obtener un grupo de `k` semáforos. En nuestro caso será necesario simplemente disponer de un grupo de `k=3` semáforos (*vacío*, *lleno* y *búffer*). Como cada uno de estos semáforos ha de ser accedido utilizando un número `{0,1,2}`, se aconseja definir constantes para identificar cada semáforo de modo que:

- el semáforo "*vacío*" se identificará con un 0.
- el semáforo "*lleno*" se identificará con un 1.
- el semáforo "*mutex*" se identificará con un 2.

MÁS INFORMACIÓN Y COMANDOS DE INTERÉS.

Véase manual en línea para las funciones comentadas.

Ipcs e **iperm** para ver y eliminar recursos compartidos (semáforos o segmentos de memoria compartida) desde la línea de comandos.

FECHA LÍMITE DE ENTREGA.

La fecha límite de entrega será el: **Miércoles 22 de diciembre de 2010** (23:59h hora española)

FORMA DE ENTREGA.

Las prácticas se entregarán a antonio.fari.so@gmail.com antes de proceder a su defensa. Se deberá enviar un fichero comprimido `p2.tar.gz` que contendrá: a) código fuente e b) fichero Makefile (el profesor compilará los 4 programas tecleando `$make`). El asunto del mensaje deberá ser el siguiente:

`[SO1P2]::[EI]::[Nombre 1er Integrante Grupo][Nombre 2º Integrante Grupo] // de ser sólo un alumno → 1 sólo nombre.`

El nombre del fichero comprimido debe tener los códigos de usuario de la red de docencia concatenados: `nombre1.apellido.apellido-nombre2-apellido.apellido.tar.gz`

La práctica será defendida ante el profesor, en el aula de prácticas. Todos los miembros de cada grupo deberán estar presentes para la defensa, de forma que el profesor pueda revisar su correcto funcionamiento, y realizar comentarios/preguntas a los integrantes del grupo, así como la realización de cambios en el código que se consideren pertinentes.

Las prácticas entregadas y que no sean defendidas con solvencia “no será bien vistas”, y en función de las circunstancias, podrán implicar un “no apto” para **todos** los miembros del grupo.

Los programas deberán:

- Compilar correctamente: se usará `gcc` con la opción `-Wall`, y no deben contener errores.
- Ejecutarse correctamente:
 - Seguirá las especificaciones marcadas.
 - Funcionar correctamente. El alumno debe tener en cuenta los valores que devuelvan todas las funciones que utilice (OJO a los valores devueltos por las funciones usadas!!).
 - Evitar la existencia de *memory-leaks* (el profesor usará **valgrind** para revisar este aspecto).

Se valorará favorablemente el que se evite la repetición innecesaria de código entre los 4 programas a implementar.

- P.ej: Se puede implementar un módulo “`semáforos.h/semáforos.c`” que se encargue de implementar las funciones de manejo de semáforos. Entre ellas:

```
void P(int semid, int sem_i);
void V(int semid, int sem_i);
void setValue (int semid, int sem_i, int value);
int getValue (int semid, int sem_i); //sólo para debugging.
int loadSemaphores(uint numberOfsemaphs, int mode); //crear/acceder
```
- Igualmente es aconsejable utilizar un módulo “`buffer.h/buffer.c`” que contendrá funciones de creación/manejo del buffer compartido.

DETALLES A RECORDAR.

- El valor máximo de las prácticas es de 1.5 puntos. Todas las prácticas deben haber recibido la calificación de “aptas”.

NOTAS:: Problema de productores/consumidores (búffer limitado).

Var

Semáforo mutex = 1 // exclusión mutua en acceso a búffer
 Semáforo vacio = n; // número de celdas vacías en el búffer
 Semáforo lleno = 0; //número de celtas llenas en el búffer.

Productor	Consumidor
Var it: t_item Begin Repeat Producir(it); P(vacio); P(mutex); Añadir it al búffer; V(mutex); V(lleno); Until false End;	Var it: t_item Begin Repeat P(lleno); P(mutex); It ← elemento del búffer; V(mutex); V(vacio); Consumir (it); Until false; End