Grupo : 2202
Asignatura : Sistemas operativos
Alumnos : Lucía Asencio y Rodrigo De Pool

**Introducción:**

En el siguiente documento haremos una breve explicación de cada uno de los ejercicios propuestos en la práctica 3.

**Nota:**

La documentación del código se encuentra en el directorio 'Documentación' en formato tanto pdf como html (abriendo el index.html en el directorio correspondiente).

El directorio txt contiene ficheros de texto con información que será explicada en el apartado 6.

Para crear los ejecutables basta con entrar en el directorio 'codigo' y ejecutar el comando 'make'. La explicación del funcionamiento de cada ejecutable se explicará a continuación en los apartados.

**Ejercicio 2:**

Para este ejercicio realizamos las siguiente suposiciones: el valor de la variable 'id' es inicializada a 0, el tiempo aleatorio que esperan los hijos es un valor entre 0 y 1 segundos ( 0 , 0.1, 0.2 …) y la impresión no se realiza desde el manejador de la señal para evitar el uso de variables globales, en vez de ello realizamos un 'pause()' seguido de la impresión.

El principal fallo en el planteamiento de este ejercicio es no hacer referencia a la sincronización de los procesos, tanto entre los hijos como entre los hijos y el padre. Si no se tiene en cuenta la concurrencia de procesos pueden ocurrir comportamientos inesperados, entre ellos: Al solicitar información por línea de comando la impresión de la solicitud y la lectura se ven totalmente descoordinados, el acceso a la zona de memoria compartida puede ser corrompida si varios hijos intentan escribir al mismo tiempo sobre la variable o si intentan escribir mientras el padre lee, además puede ocurrir que la información de un hijo no se imprima si el padre esta imprimiendo mientras el hijo termina su ejecución, de este modo perdiendo información.

Podemos concluir que la sincronización entre procesos es fundamental cuando se trata con el acceso a memoria compartida.

**Libreria de semáforos:**

La implementación está comentada en su totalidad en el fichero semaforos.c (ó en el correspondiente doxygen).

El test de esta librería se genera como el ejecutable 'main_test' tras utilizar el comando 'make'. Este main realiza llamadas a funciones de la librería de test de semaforos, la especificación del funcionamiento y test realizados esta comentada en el fichero 'semaforostest.c'.

**Ejercicio 6:**

IMPORTANTE: Esta funcion puede recibir 0, 2 o 3 argumentos de entrada. Leer documentacion del main.

Para empezar, intentamos explicar la implementacion del ejercicio, para mostrar luego una serie de pruebas que ejecutamos sobre el mismo.

Grupo : 2202
Asignatura : Sistemas operativos
Alumnos : Lucía Asencio y Rodrigo De Pool

En primer lugar, nuestro proceso es un padre, que crea la memoria compartida y los 3 semáforos que heredan los dos hijos que genera después.

El padre hace la labor de poductor, el primer hijo que genera es el consumidor, y el tercero es un "temporizador" que permite que la interaccion produccion-consumicion se ejecute hasta que pase un determinado tiempo, que decide el usuario mediante el tercer argumento de entrada al main (este tercer hijo ejecuta un usleep(<3º argumento>) antes de frenar a los otros dos).

Los tres semaforos no tienen gran explicacion (el problema es un problema tipo, y la solucion escogida es la que aparece en todos los libros: un seaforo controla que no se consuma si no hay productos en el almacen, otro que no se produzca si el almacen esta lleno y un mutex que protege la memoria compartida).

Nuestra memoria compartida, el almacen, tiene una estructura de pila (es decir, los elementos se añaden y se consumen del top del almacen). Tenemos un array de caracteres con todo el abecedario, y una variable 'end' que apunta al top de la pila en ese momento (el productor incrementa el end, el consmidor lo decrementa). Adicionalmente, usamos una variable temp, que el hijo "temporizador", el único en escribir en esta variable una vez comienza la ejecución concurrente, emplea como flag para indicar a productores y consumidores que ha cesado su tiempo de trabajo. Por tanto, productor y consumidor trabajan en un while(temp == XX). Cuando se les acaba el tiempo, los 3 procesos hacen un detach de la shmem y los dos hijos vuelven al padre, que elimina tanto la memoria como los semaforos antes de salir.

Hay dos casos en los que el uso de este temporizador puede originar problemas: uno es cuando el almacen esta vacío (si damos suficiente tiempo, este no debería ocurrir: aun así, está tratado) y cuando el almacen esta lleno.

Explicamos el del almacen lleno, y el otro es equivalente.

Cabe la siguiente posibilidad: el consumidor consume la Z, y hace Up a sus dos semáforos. Antes de volver a ejecutar su while (temp == XX), el productor se ejecuta, produce de nuevo la Z, que es la última letra, y vuelve a entrar en su bucle. Como el almacén está lleno, se bloquea al hacer "down(vacio)". En ese momento, pasa a ejecutarse el hijo temporizador, que cambia la bandera. El procesador vuelve al consumidor, que ejecuta su while(temp == XX) y, como temp ha cambiado, sale del bucle y termina. El productor ha quedado bloqueado en la cola de vacio.

Para evitar esto hacemos que, una vez cambiada la variable temp, el hijo temporizador haga un Up(vacio) (par el caso almacen lleno) y un Up(lleno) (para el caso de almacen vacio) y que, en la linea siguiente a sus respectivos downs (donde el proceso ha quedado bloqueado y despierta), se ejecute un if(temp == XX), para distinguir el caso en que el proceso ha sido despertado por el temporizador, y debe hacer el detach y el return, o porque debe puede continuar su trabajo.

No es una solución muy bonita pero, hasta lo que hemos podido pensar (y probar) funciona. La opcion de hacer que el padre fuera temporizador e hiciera volver a sus dos hijos por medio del envio de señales nos pareció peor, ya que los hijos no podrían entonces (sin el uso de variables globales) hacer el correcto detach de la memoria compartida

Para apreciar el funcionamiento del programa, hemos añadido al final del bucle de produccion/consumicion unos usleeps que el usuario modifica mediante el primer y segundo argumento

Grupo : 2202
Asignatura : Sistemas operativos
Alumnos : Lucía Asencio y Rodrigo De Pool

de entrada (el primer argumento es el usleep del productor, el segundo argumento es el usleep del consumidor).

SI TE HAS ABURRIDO AQUI, DA IGUAL QUE SIGAS LEYENDO O NO. LO SIGUIENTE SOLO SON PRUEBAS CON LAS DISTINTAS ENTRADAS AL MAIN.

A continuacion, explicamos lo observado “jugando” con estos argumentos. Hemos impreso una parte de las salidas,que pueden comprobarse en el directorio txt, que contiene archivos de la forms ‘n1-n2.txt’, donde n1 es el usleep del productor y n2 es el usleep del consumidor.

Como no podemos hacer suposiciones de tiempo, sabemos que los resultados que damos aquí no son correctos, simplemente expresan lo que la ejecución devuelve “la mayoría de las veces”, aunque en algún momento el planificador podría querer llevarnos la contraria y hacerlo al revés.

**n1 >= n2:**
0-0, 1000-1000, 2000-1000, 10000-1000
El productor tiene más retardo que el consumidor, por lo que siempre que el productor genera una letra el consumidor se la lleva antes de dar tiempo a que el productor produzca la siguiente. Cuando ambos retardos son iguales, el resultado comprobado ha sido siempre equivalente.
Las salidas son del tipo: Productor produce A- Consumidor consume A- Productor produce A- Consumidor consume A…. y así eternamente, sin producir las demás letras del alfabeto.

**n1 < n2:**
El consumidor tiene un retardo mayor que el productor, por lo que thay tiempo para producir más antes de que el consumidor llegue a llevárselo. En todas estas ejecuciones, si dejamos suficiente tiempo antes de cortar el programa, hay un punto en el que el producor ha producido todas las letras, y el tira y afloja de los dos queda en: productor produce Z- Consumidor consume Z-Productor produce Z… y así. La diferencia es cuánto se tarda en llegar a ese punto.
1000-10000, 1000-5000:
En el primer caso, el productor produce todo el alfabeto en apenas 3 tandas (por ejemplo, produce de la A a la I seguido, llega el consumidor y consume la I, el productor vuelve y produce de la I a la Q, consumidor consume Q, y productor produce hasta el final). En el segundo, tarda 6/7 tandas del mismo tamaño en producirl el alfabeto. Si el planificador hace lo que intuitivamente parece, esto tiene sentido: si el consumidor duerme diez veces más que el productor, éste tarda la mitad en producir las letras que si el consumidor duerme cinco veces más que el productor.
1000-2000, 1000-1500:
Tenemos la misma situación, pero el productor tarda más “tandas” en producir todo el abecedario. En el 1000-2000, las salidas son del tipo Produce ABC- Consume C-Produce CDE-Consume E-Produce EFG… hasta el tira y afloja del Z; y en el 1000-1500, la “carrera” está más reñida: Produce A-Consume A-Produce A -Consume A-Produce AB-Consume B-Produce B-Consume B… al productor le cuesta más producir antes de que el consumidor vuelva a recoger letras.

Grupo : 2202
Asignatura : Sistemas operativos
Alumnos : Lucía Asencio y Rodrigo De Pool

Por último, un par de imágenes donde hemos omitido las frases de productor y consumidor, donde se muestra liberación de semáforos y shmem (se ven los semaforos y la memoria que utilizamos, que no se encuentran en la salida de ipcs)

```
lucia@lucia-SATELLITE-P50-B-11L:~/pr/soper/p3/codigo$ ./ejercicio6 1000 10000 80000
Semaforos: 1540104 1605642 1572873
Shmem: 3997718
El productor sale tras ser despertado por el proceso temporizador
Consumidor sale
lucia@lucia-SATELLITE-P50-B-11L:~/pr/soper/p3/codigo$ ipcs -ms

---- Segmentos memoria compartida ----
key        shmid      propietario perms      bytes      nattch     estado
0x0052e2c1 0          postgres   600        56         5
0x00000000 557057     lucia      600        524288     2          dest
0x00000000 917506     lucia      600        524288     2          dest
0x00000000 1015811    lucia      600        524288     2          dest
0x00000000 786436     lucia      600        524288     2          dest
0x00000000 819205     lucia      600        524288     2          dest
0x00000000 2949126    lucia      600        393216     2          dest
0x00000000 1146887    lucia      600        524288     2          dest
0x00000000 1736712    lucia      600        524288     2          dest
0x00000000 1277961    lucia      600        16777216   2
0x00000000 1507338    lucia      600        524288     2          dest
0x00000000 1605643    lucia      600        524288     2          dest
0x00000000 1638412    lucia      600        67108864   2          dest
0x00000000 1835021    lucia      600        524288     2          dest
0x00000000 1998862    lucia      600        524288     2          dest
0x00000000 2031631    lucia      600        4194304    2          dest
0x00000000 2555920    lucia      600        524288     2          dest
0x00000000 2588689    lucia      600        4194304    2          dest
0x00000000 2621458    lucia      600        33554432   2          dest
0x00000000 2654227    lucia      600        67108864   2          dest
0x00000000 3047444    lucia      600        524288     2          dest
0x00000000 3244053    lucia      600        524288     2          dest

------ Matrices semáforo -------
key        semid      propietario perms      nsems
0x0052e2c1 0          postgres   600        17
0x0052e2c2 32769      postgres   600        17
0x0052e2c3 65538      postgres   600        17
0x0052e2c4 98307      postgres   600        17
0x0052e2c5 131076     postgres   600        17
0x0052e2c6 163845     postgres   600        17
0x0052e2c7 196614     postgres   600        17
0x0052e2c8 229383     postgres   600        17
```