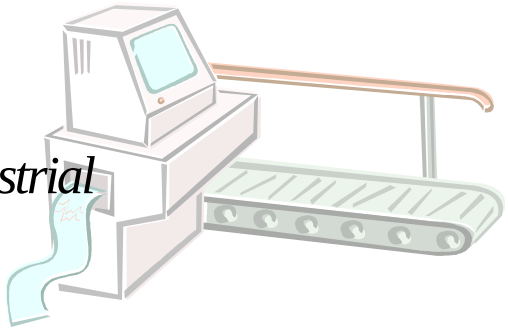
Página Web: https://webct08.uah.es/
e-mail: eliseo.garcia@uah.es
Despacho: N-245
Departamento de Automática

Práctica 7

Programación concurrente

Hilos

Laboratorio de Informática Industrial



Concurrencia entre aplicaciones

Introducción

Una de las funciones más características de los SS.OO. modernos es la posibilidad que nos brindan de poder ejecutar varias aplicaciones independientes compartiendo un mismo hardware. Para ello el S.O. ha de actuar de árbitro para asignar y retirar recursos. A menudo existe la necesidad de que las aplicaciones compartan información entre sí, por lo que los SS.OO. proporcionan todo un repertorio de posibilidades de comunicación entre procesos. En esta práctica veremos la comunicación mediante señales.

En ocasiones es necesaria la existencia de diferentes tareas dentro de cada aplicación para poder atender simultáneamente a las entradas/salidas con las que cuente. Se dice entonces que se están utilizando varios hilos por cada proceso, cada uno de ellos atendiendo posiblemente a un dispositivo de respuesta lenta. Si la comunicación entre distintos procesos es necesaria, la comunicación entre los distintos hilos que se integran en un proceso es fundamental ya que tienen mayor cohesión entre sí. Además los hilos han de disponer de mecanismos que les permitan sincronizarse para repartirse adecuadamente los recursos del proceso sin la intervención directa del S.O. Para este fin, se utilizan profusamente los mutex, variables condicionales y semáforos. En esta práctica veremos el uso básico de los objetos de exclusión mutua que proporciona la biblioteca Pthreads.

Señales

Las señales son un mecanismo básico de comunicación entre procesos en UNIX. Su objetivo es comunicar la existencia de un evento, por lo que no transmiten datos; son tan solo un aviso de que cierto suceso ha ocurrido. Podemos hacernos una idea conceptual de lo que son las señales asimilándolas a las interrupciones que recibe la CPU. En efecto el S.O. utiliza señales para comunicar a los procesos que han ocurrido situaciones excepcionales, como división por cero, fallo en un cálculo de coma flotante, o acceso no permitido a una región de memoria restringida.

Uno de los usos más habituales de las señales es el de comunicar a un proceso que ha de terminar. Para ello otro proceso le envía la señal SIGTERM o bien SIGKILL. La señal SIGTERM se interpreta como una petición de terminación que pude utilizar el proceso que la recibe para cerrar los documentos que tuviera abiertos. SIGKILL es una orden irrevocable que hace que el S.O. termine el proceso forzosamente.

Podemos obtener un listado de las señales que se pueden usar en un sistema concreto introduciendo el siguiente comando:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      32) SIGRTMIN    33) SIGRTMIN+1
34) SIGRTMIN+2  35) SIGRTMIN+3  36) SIGRTMIN+4  37) SIGRTMIN+5
38) SIGRTMIN+6  39) SIGRTMIN+7  40) SIGRTMIN+8  41) SIGRTMIN+9
42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14
50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10
54) SIGRTMAX-9  55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6
58) SIGRTMAX-5  59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2
62) SIGRTMAX-1  63) SIGRTMAX
```

También podemos consultar el archivo `/usr/include/bits/signum.h` donde encontramos una pequeña descripción del propósito de cada una de ellas:

```
/* Signals.  */
#define     SIGHUP          1       /* Hangup (POSIX).  */
#define     SIGINT          2       /* Interrupt (ANSI).  */
#define     SIGQUIT         3       /* Quit (POSIX).  */
#define     SIGILL          4       /* Illegal instruction (ANSI).  */
#define     SIGTRAP         5       /* Trace trap (POSIX).  */
#define     SIGABRT         6       /* Abort (ANSI).  */
#define     SIGIOT          6       /* IOT trap (4.2 BSD).  */
#define     SIGBUS          7       /* BUS error (4.2 BSD).  */
#define     SIGFPE          8       /* Floating-point exception (ANSI).  */
#define     SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define     SIGUSR1         10      /* User-defined signal 1 (POSIX).  */
#define     SIGSEGV         11      /* Segmentation violation (ANSI).  */
#define     SIGUSR2         12      /* User-defined signal 2 (POSIX).  */
#define     SIGPIPE         13      /* Broken pipe (POSIX).  */
#define     SIGALRM         14      /* Alarm clock (POSIX).  */
#define     SIGTERM         15      /* Termination (ANSI).  */
#define     SIGSTKFLT       16      /* Stack fault.  */
#define     SIGCLD          SIGCHLD /* Same as SIGCHLD (System V).  */
#define     SIGCHLD         17      /* Child status has changed (POSIX).  */
#define     SIGCONT         18      /* Continue (POSIX).  */
#define     SIGSTOP         19      /* Stop, unblockable (POSIX).  */
#define     SIGTSTP         20      /* Keyboard stop (POSIX).  */
#define     SIGTTIN         21      /* Background read from tty (POSIX).  */
#define     SIGTTOU         22      /* Background write to tty (POSIX).  */
#define     SIGURG          23      /* Urgent condition on socket (4.2 BSD).  */
#define     SIGXCPU         24      /* CPU limit exceeded (4.2 BSD).  */
#define     SIGXFSZ         25      /* File size limit exceeded (4.2 BSD).  */
#define     SIGVTALRM       26      /* Virtual alarm clock (4.2 BSD).  */
#define     SIGPROF         27      /* Profiling alarm clock (4.2 BSD).  */
#define     SIGWINCH        28      /* Window size change (4.3 BSD, Sun).  */
#define     SIGPOLL         SIGIO   /* Pollable event occurred (System V).  */
#define     SIGIO           29      /* I/O now possible (4.2 BSD).  */
#define     SIGPWR          30      /* Power failure restart (System V).  */
#define     SIGSYS          31      /* Bad system call.  */
#define     SIGUNUSED       31
```

Como vemos a cada señal le corresponde un número. En principio sólo disponemos de hasta 64 señales distintas.

La propia orden `kill` sirve para enviar señales a procesos. La siguiente secuencia de órdenes crea un proceso de procesamiento de fondo que lo único que hace es dormir durante 1000 segundos, luego obtiene el identificador de proceso (PID) que le ha asignado el sistema, y por último le envía la señal de terminación:

```
$ sleep 1000 &
[1] 10147
$ ps
  PID TTY          TIME CMD
  694 pts/0    00:00:00 bash
10147 pts/0    00:00:00 sleep
10148 pts/0    00:00:00 ps
$ kill -SIGTERM 10147
 [1]+  Terminated              sleep 1000
```

Se podría haber enviado la misma señal especificando su número en lugar de su nombre:

```
$ kill -15 10147
```

O bien suprimiendo las letras SIG:

```
$ kill -TERM 10147
```

En el caso de que el proceso no respondiera a la petición de terminación, sería el momento enviarle la señal SIGKILL.

Las señales son procesadas desde una rutina dedicada llamada manejador de señal. Estas comportan de forma similar a las rutinas de atención a interrupción, y como tales deben respetar algunas restricciones:

- Han de devolver el control lo más rápidamente posible a la aplicación.
- No deberían llamar a funciones de biblioteca peligrosas, sobre todo a las de respuesta lenta o aquellas que pudieran bloquear la llamada.

En general los manejadores de señales tan solo deberían ser usados para transmitir el evento al programa principal que estaría encargado de realizar el procesamiento oportuno, comprobando tal vez el estado de una variable compartida. Dada la naturaleza asíncrona de las interrupciones sería peligroso que se accediera a datos de forma no atómica desde hilos del proceso y desde los manejadores de señales. Si esto fuera necesario GNU/Linux proporciona el tipo de dato `sig_atomic_t` con el que se garantiza que cualquier operación de incremento o decremento se realizarán de manera atómica. En el caso de las versiones x86 de **gcc**, este tipo es un sinónimo de un entero volátil sobre el que dichas operaciones se implementan con las instrucciones máquina `inc` y `dec` respectivamente.

Para que un proceso pueda atender a una señal ha de disponer de un manejador para esa señal. A continuación se muestra el programa `mi_sleep.c` que instala un mismo manejador para las señales SIGTERM y SIGUSR1:

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

sig_atomic_t sigusr1_count = 0, num_seg = 0;

/* Manejador de las señales SIGTERM y SIGUSR1 */
void handler (int signal_number)
{
  if(signal_number == SIGTERM)
    printf ("Se ha recibido la señal SIGTERM.\n");
  else if(signal_number == SIGUSR1)
  {
    ++sigusr1_count;
    printf ("Se ha recibido la señal SIGUSR1.\n");
  }

  printf("Quedan %d segundos hasta finalizar.\n", num_seg);
}
```

```
int main (int argc, char** argv)
{
  struct sigaction sa;
  memset (&sa, 0, sizeof (sa));   // Inicia a 0 la estructura

  if (argc == 1)
      printf("Sintaxis:\n\t%s NUM_SEG\n\n\tNUM_SEG - "
             "Indica el número de segundos a dormir.\n", argv[0]);
  else if (argc == 2)
      num_seg = atoi(argv[1]);

  sa.sa_handler = &handler;       // Establece el manejador de la interrupción
  sigaction (SIGTERM, &sa, NULL); // Configura la interrupción

  sa.sa_handler = &handler;       // Establece el manejador de la interrupción
  sigaction (SIGUSR1, &sa, NULL); // Configura la interrupción

  /* Aquí vendría el resto del programa.  */
  while (--num_seg) // Dormir num_seg segundos
    sleep (1); // Las señales interrumpen la siesta
  /* ...  */

  printf ("Se ha recibido la señal SIGUSR1 %d veces\n", sig_usr1count);

  return 0;
}
```

El mecanismo de señales existe desde las primeras versiones de UNIX. Fue implementado sin tener en cuenta la posibilidad de que existiera más de un hilo por cada proceso, como consecuencia generalmente son incompatibles: supóngase por ejemplo qué ocurriría si se intentara adquirir un mutex que ya estuviera bloqueado desde un manejador de señales. El problema es aun más acusado en Linux con kernels inferiores al 2.6, ya que implementan hilos de núcleo mediante procesos diferentes que comparten el mismo área de memoria. El estándar POSIX asume que todos los hilos deben compartir un mismo proceso por lo que no tendría que existir manejadores de señales privados a cada hilo.

La biblioteca **Pthreads** propone un método para gestionar señales que consiste en llamar a una función que bloquee al hilo hasta que ocurre la señal. Esto supone que debería existir al menos un hilo dedicado al tratamiento de cada una de las señales que se quisieran admitir.

Mutex

Uno de los mecanismos más básicos de sincronización son los *mutex* o semáforos binarios. El mutex es útil para proteger las estructuras de datos compartidas de modificaciones concurrentes, implementar secciones críticas (no interrumpibles), y crear monitores o áreas de código en las que sólo puede estarse ejecutando un único hilo.

Los mutex sólo tienen dos estados posibles: desbloqueado (no poseído por hilo alguno), y bloqueado (poseído por un hilo). Cualquier intento de bloquear a un mutex que ya esté de hecho bloqueado podría causar que el hilo sea suspendido hasta que el mutex sea liberado por cualquier otro hilo.

La especificación Pthreads implementa algunas funciones básicas para operar con mutex:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *pmutex); /* Bloquea al mutex */
int pthread_mutex_trylock(pthread_mutex_t *pmutex); /* Intenta bloquear al mutex */
int pthread_mutex_unlock(pthread_mutex_t *pmutex); /* Desbloquea al mutex */
```

La manera más sencilla de crear un nuevo mutex es asignando la constante PTHREAD_MUTEX_INITIALIZER a una variable del tipo `pthread_mutex_t`:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

La llamada a `pthread_mutex_trylock()` intenta bloquear al mutex. Si ya estuviera bloqueado por algún hilo simplemente devolverá EBUSY pero no suspenderá al hilo que la llame.

Por razones de diseño, y teniendo en cuenta la implementación de **Pthreads** ofrecida en Linux mediante la biblioteca **LinuxThreads**, es recomendable configurar todas las señales que vaya a necesitar el proceso al principio del hilo principal. Por otra parte, ya que en Linux 2.4- los hilos son internamente procesos, el envío de una señal ha de realizarse sobre el hilo que tenga el manejador correspondiente.

El siguiente programa gestiona señales mediante hilos. Apréciese que no hay restricciones en el tipo de funciones que pueden ser llamadas desde manejadores basados en hilos, no obstante deben continuar siendo lo más rápidos posible:

```
#include <pthread.h>
#include <signal.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

pthread_t hilo_sennal;
pthread_mutex_t mut_sennal = PTHREAD_MUTEX_INITIALIZER;

void* atiende_SIGUSR1(void* arg);
void muestra_estadistica(void);

int main (int argc, char** argv)
{
    sigset_t sennales_a_bloquear;
    char sz_entrada[128];

    /* Establecer la máscara de interrupciones del hilo principal
       para que se ignoren SIGUSR1 y SIGTERM */
    sigemptyset(&sennales_a_bloquear);
    sigaddset(&sennales_a_bloquear, SIGUSR1);
    sigaddset(&sennales_a_bloquear, SIGTERM);
    pthread_sigmask(SIG_BLOCK, &sennales_a_bloquear, NULL);

    pthread_create(&hilo_sennal, NULL, atiende_SIGUSR1, NULL);

    printf("Para salir del programa introduzca \"quit\".\n"
           "Para imprimir un mensaje introduzca \"mensaje\".\n");

    for(;;)
    {
        // Esta función puede bloquear al hilo que la llama si no hay entrada
        gets(sz_entrada);

        if (strcmp(sz_entrada, "quit") == 0)
            break;
        if (strcmp(sz_entrada, "mensaje") == 0)
        {
            pthread_mutex_lock(&mut_sennal);
            muestra_estadistica();
            pthread_mutex_unlock(&mut_sennal);
        }
    }

    return 0;
}

void* atiende_SIGUSR1(void* arg)
{
    sigset_t sennales_a_atender;

    printf("Hilo preparado para recibir señales.\n");

    pthread_detach (pthread_self()); // No es unible (joinable)

    /* Establecer la máscara de interrupciones del hilo principal
       para que se atienda a SIGUSR1 y SIGTERM*/
    sigemptyset(&sennales_a_atender);
    sigaddset(&sennales_a_atender, SIGUSR1);
```

```
    sigaddset(&sennales_a_atender, SIGTERM);

    for(;;)
    {
        int num_sennal;
        sigwait(&sennales_a_atender, &num_sennal);

       printf("Recibida señal.\n");

        /* Acaba de recibirse una señal y podemos usar libremente todas las funciones Pthreads */
        switch (num_sennal)
        {
            case SIGUSR1:
                printf("Se ha recibido la señal SIGUSER1.\n");

                pthread_mutex_lock(&mut_sennal);
                muestra_estadistica();
                pthread_mutex_unlock(&mut_sennal);
                break;
           case SIGTERM:
                printf("Se ha recibido la señal SIGTERM.\n");
                /* Desde aquí se le podría comunicar a otro hilo que
                pregunte al usuario si desea guardar los documentos abiertos */
                break;
            default:
                ;
        }
    }

    return NULL;
}

/* Esta función podría acceder a recursos compartidos, por lo que debe ser protegida con
 * un objeto de exclusión mutua.
 */
void muestra_estadistica(void)
{
    printf("Esta función puede ser llamada desde el hilo principal y "
                  "desde un hilo de atención a señales.\n");
}
```

Para compilar el programa necesitamos enlazar con la biblioteca `libpthread.so` como aparece en la siguiente orden:

```
$ gcc -Wall -lpthread -othread_signal threaded_signal.c
```

Podemos ejecutar el programa y desde otra consola enviar mensajes al hilo.

```
$ ps -am  # Muestra todos los hilos de los procesos
  PID TTY          TIME CMD
 4452 pts/1    00:00:03 vim
 4519 pts/0    00:00:00 thread_signal
 4520 pts/0    00:00:00 thread_signal
 4521 pts/0    00:00:00 thread_signal
 4538 pts/2    00:00:00 ps
$ ps -a  # Muestra sólo los hilos principales de cada proceso
  PID TTY          TIME CMD
 4452 pts/1    00:00:03 vim
 4519 pts/0    00:00:00 thread_signal
 4539 pts/2    00:00:00 ps
$ kill -TERM 4521
$ kill -USR1 4521
$ kill -KILL 4521  # Esta llamada hace que termine todo el proceso
```

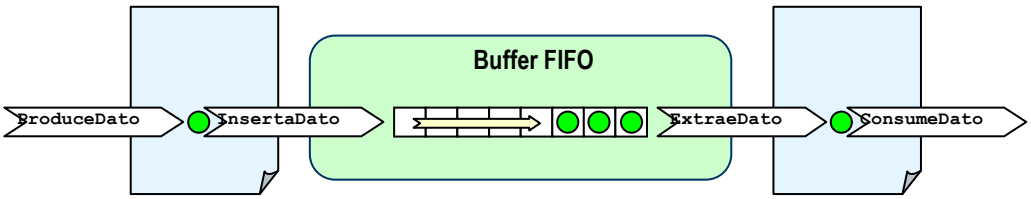
Ejercicio

El problema del Productor/Consumidor Dormilón

Este problema es muy habitual en sistemas informáticos que consiste en atender la entrada de datos que proporciona un productor para posteriormente proporcionárselos a un receptor que los consume a un ritmo distinto. Para minimizar la pérdida de datos, productor y consumidor han de ser comunicados a través de un buffer FIFO que pueda absorber los datos generados aunque el consumidor no pueda procesarlos al mismo ritmo.

Este se trata de un contenedor que permite introducir datos, para recuperarlos posteriormente respetando el orden de entrada, es decir el primero en entrar es el primero en salir (*First In First Out* - FIFO). Este tipo de *buffer* o memoria intermedia se emplea para adaptar la velocidad de un productor/generador de datos con la de un consumidor/receptor.

En la siguiente figura se muestra un sistema de producción de datos ajeno al sistema informático.



Esquema del problema del productor-consumidor.

Para ilustrar este caso se ha hecho un programa que contiene dos funciones que serán usadas por los supuestos productores (Productor()) y consumidores (Consumidor()). Para cada uno de los dos, es necesario mantener un índice (IndiceProductor e IndiceConsumidor) que apunte al siguiente elemento a ser escrito, y al siguiente elemento a ser leído respectivamente. Además se incorpora un contador que indica la cantidad de elementos libres disponibles (libres).

Ya que tanto las funciones de producción y consumo de datos pueden emplear un tiempo indeterminado, y funcionan sobre dispositivos distintos en instantes no necesariamente sincronizados, es recomendable que cada una de ellas sea llamada desde hilos independientes. Se debe tener en cuenta que cuando no quede información por consumir el hilo consumidor debería permanecer suspendido para no utilizar CPU innecesariamente.

PRACTICA A REALIZAR

Realizar un programa que realice las acciones especificadas anteriormente. En él, además se deberá preparar un manejador para procesar la señal SIGTERM. Cuando sea recibida, el hilo productor dejara de Producir datos e insertarlos en el buffer, y el hilo consumidor extraerá los elementos que restan en el buffer y los procesara. Finalmente, el hilo que ejecute en el main() escribirá por pantalla "Programa terminado" y finalizara la ejecución.

Las funciones a llamar son las mostradas a continuación. Se deben modificar todas ellas salvo la función ProduceDato y ConsumeDato.

```
void * Productor(void *);
void * Consumidor(void *);
void ProduceDato(int *dato);
void InsertaDato(int dato);
```

```
void ConsumeDato(int dato);
void ExtraeDato(int *dato);


int fdProd,fdCons;
int indiceProductor,  indiceConsumidor,libres;

//_____________________________________________________________

void main()
{
     fdProd=open("Productor.txt", O_RDWR | O_CREAT);
     fdCons=open("Consumidor.txt", O_RDWR | O_CREAT);



    printf("Programa Terminado\n");
}

//_____________________________________________________________

void * Productor(void *)
{
    int dato;
// Codigo a ser ejecutado por el hilo productor

    while(1)
    {


    ProduceDato(&dato);


    InsertaDato(dato);



    }
}
//_____________________________________________________________


void * Consumidor(void *)
{
 int dato;
// Codigo a ser ejecutado por el hilo consumidor
    while(1)
    {
       ExtraeDato(&dato);


       ConsumeDato(dato);

    }
}
//_____________________________________________________________


void ProduceDato(int *dato)
{  //Para la realización de la practica no se puede modificar esta funcion
    dato=rand();
    write(fdProd,dato,1);
    usleep ((unsigned long) rand () % 5000000UL);

}

//_____________________________________________________________

void InsertaDato(int dato)
{
// Codigo que inserta un dato en el buffer
}


//_____________________________________________________________
```

```
void ConsumeDato(int dato)
{//Para la realización de la practica no se puede modificar esta funcion
    write(fdCons,dato,1);
    usleep ((unsigned long) rand () % 5000000UL);
}

//_____________________________________________________________


void ExtraeDato(int *dato)
{
// Codigo que extrae un dato del buffer

}
```