Sistemas informáticos en tiempo real

Programación

1º Procesos concurrentes

- ***Creación de un proceso hijo a partir de un proceso padre***

  Lo primero que tenemos que hacer es añadir dos librerías

  #include<sys/types.h> → Incluye el tipo pid_t
  #include<unistd.h> → Incluye la función fork y la función execl

  Después, para crear el proceso hijo tenemos que seguir dos pasos:

  Primer paso: llamada a fork

  1. Creamos un identificador de nuestro proceso hijo:

     pid_t hijo;

     pid_t es un tipo de variable que identifica al proceso con un número

  2. Llamamos a fork:

     hijo = fork (void);

     La función fork tiene la siguiente estructura:

     pid_t fork (void)

     Crea el proceso hijo a partir del proceso padre

Segundo paso: activación del proceso deseado

3. Creamos la división padre hijo:

```
if(!hijo)
        {
                Código del hijo
        }
else
        {
                Código del padre
        }
```

4. Para que el hijo sea el que corra el proceso deseado, se usa:

   execl("./nombredelejecutable","argumento1","argumento2", …, NULL);

   Esto se incluye en el código del hijo, claro está.

   La función execl tiene la siguiente estructura:

   *pid_t execl (const char*ejecutable, const char*arg0, … , NULL)*

   Siendo el primer valor que se le pasa un puntero al ejecutable que se va a ejecutar, en general mediante el método "./nombredelejecutable"

   El resto de parámetros son los argumentos que se le pasan a dicho ejecutable. El último argumento que se le pasa a execl SIEMPRE será NULL.

   Explicación del proceso completo: en el primer paso, la función fork crea una copia del proceso padre. Si esta función falla, devuelve el valor -1. Si no, devuelve dos valores: en el proceso padre devuelve el valor del PID (identificador) de proceso hijo; en el proceso hijo, devuelve el valor 0. Estos valores los asignamos al identificador que hemos creado previamente.

   Esto explica cómo se estructura el punto 3, inicio del paso dos: el if depende del valor del identificador. Si estamos en el proceso hijo, su valor será cero, por ello la condición para entrar en el código del proceso hijo es el negado del identificador, es decir, 1, que en C significa verdadero. Cualquier otro valor nos llevaría al código del proceso padre.

- ***Terminación y espera de procesos***

  Lo primero que tenemos que hacer es añadir dos librerías más:

  #include<stdlib.h> → Incluye la función exit.
  #include<sys/wait.h> → Incluye las funciones wait y waitpid.

  La forma normal de finalizar un proceso es utilizar la función exit:

  exit(1);

  Al final del código del hijo siempre se incluye exit.

  La función exit es de la forma:

  void exit (int status)

  int status es un valor al que tienen acceso diversos comandos

  Dentro del proceso padre pueden incluirse dos funciones para saber si el proceso hijo ha terminado o no, que son las funciones "wait" y "waitpid". Estas son muy útiles porque aparte liberan completamente los recursos del ordenador que el proceso hijo estaba usando. La diferencia entre uno y otro es que waitpid es más flexible, pues permite añadirle más opciones.

  En cualquier caso, siempre que usemos wait/waitpid, tenemos que inicializar una variable int donde se guardará el valor que devuelve exit.

```
<cabeceras>
main()
{
        pid_t   pid,pidfin;
        int     estado;

        pid=fork();

        <resto del código>

        pidfin=wait(&estado);

        Alternativa: pidfin=waitpid(pid,&estado,0);
}
```

La función wait es de la forma:

*pid_t wait (int*status)*

Espera la terminación de cualquier hijo. El puntero suele introducirse como &variable.

La función waitpid es de la forma:

*pid_t waitpid (pid_t, int*status, int options)*

En el primer argumento se introduce el número que identifica al proceso hijo al que se espera. Si se introduce un -1, se espera a que acabe cualquier proceso hijo. El segundo argumento funciona igual que en wait. En ambas funciones, el valor de retorno es el del proceso que acaba.

- ***Identificación de procesos***

Hay dos funciones para identificar procesos:

1. ***La función getpid:*** esta función devuelve el identificador del proceso que se está ejecutando. Su forma es la siguiente:

   pid_ t getpid()

2. ***La función getppid:*** esta función devuelve el identificador del proceso padre del proceso que se está ejecutando. Su forma es la siguiente:

   pid_t getppid()

Una forma de utilizarlos puede ser la siguiente:

```
…
pid_t  proceso;
proceso=getpid();
…
```

En cuyo caso la variable proceso identifica al proceso actual. Otra forma muy común de usarlos es con los moldes de C, para pasarlos por pantalla. Siguiendo el ejemplo anterior:

printf("El proceso actual es el número %d",(int)getpid());

Lo que se ha puesto antes de getpid es el molde.

- ***<u>Usos del valor de la variable estado de exit y opciones de waitpid</u>***

  Nosotros usaremos tres macros con la variable “estado” que almacena el valor que devuelve exit. Estas son:

  WIFEXITED(estado): este comando da un número, que será distinto de 0 siempre que se saliera del proceso a través de exit de forma normal.

  WIFSIGNALED(estado): si su valor es 1, indica que se salió por culpa de una señal, aunque otros números indican más posibilidades.

  WEXITSTATUS(estado): recupera los 8 bits menos significativos del valor que se pasó a exit, es decir, si escribimos exit(2), nos daría 2.

  Las opciones de waitpid son el último argumento de la función, que normalmente escribiremos como 0, aunque podríamos poner:

  WNOHANG: evita que el padre espere a ningún hijo. En este caso, se devuelve 0 si existen hijos activos, y si no devolverá -1.

  WUNTRACED: el padre recibe información adicional del hijo si se recibe alguna de las siguientes señales: SIGTTIN, SIGTTOU, SIGSSTP o SIGTSTOP.

  Las señales se tratan a partir de la página 11.

2º Hilos

- ***Creación de un proceso hijo a partir de un proceso padre***

  Lo primero que tenemos que hacer es añadir una librería:

  #include<pthread.h> → Incluye las funciones pthread_create, pthread_join, y los tipos pthread_t y pthread_attr_t.

  Después, para crear el hilo hay que seguir una serie de pasos:

  Primer paso: crear el puntero a la rutina de arranque

  Por cada hilo que vayamos a crear, tenemos que crear un puntero antes de definir el hilo principal (main). Para ello, escribimos:

  ```
  void *hilo(void *arg);
  ```

  Evidentemente, en vez de "hilo" podemos escribir lo que queramos. Después ya se crea el hilo principal, esto es:

  ```
  void main (void)
  {
          …
  }
  ```

  Segundo paso: crear el identificador del hilo, los atributos y el argumento de rutina

  Son imprescindibles el identificador del hilo y el argumento para la rutina, los atributos sin embargo no, porque siempre podemos dejar los que vienen por defecto. Es simplemente añadir tres líneas (ya dentro de la rutina main)

  ```
  pthread_t thr1;
  pthread_attr_t attr1;
  int i;
  ```

  Los atributos se modifican con dos funciones que nosotros no vamos a ver.

Tercer paso: crear el hilo en sí

Para crear el hilo tenemos que usar la función pthread_create:

pthread_create(&thr1,NULL,hilo,(void *)&i);

La función pthread_create es de la forma:

*int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*rut_com)(void*), void*arg)*

El entero que devuelve es distinto de cero si ha habido algún problema. De esta manera, podemos detectar errores en la creación de hilos.

El primer argumento es el identificador del hilo. El segundo es la variable que contiene los argumentos del hilo, si se escribe NULL se ponen los que vienen por defecto. El tercer argumento es el que indica qué hilo va a crearse. El cuarto argumento es un puntero que recibe el hilo nuevo.

Cuarto paso: definir el hilo

Un hilo se define de forma parecida a las funciones, después de cerrar el hilo principal main. Es decir, agrupando todo el código:

```
#include<pthread.h>

void *hilo(void *);

void main(void)
{
        pthread_t thr1;
        pthread_attr_t attr1;
        int i;

        pthread_create(&thr1,NULL,hilo,(void *)&i);

}
```

void *hilo(void *pi) → Definición en sí del hilo que se crea

{

<código del hilo>

}

- ***Identificación de hilos***

  La función que se utiliza para identificar hilos tiene la siguiente forma:

  pthread_t pthread_self (void)

  Devuelve el número del hilo que se está ejecutando en este momento. Por ejemplo:

  ```
  pthread_t thr;
  thr=pthread_self();
  ```

  Hay otra función que compara un identificador con otro y que dice si son iguales o no:

  int pthread_equal (pthread_t t1, pthread_t t2)

  Si son iguales devuelve un valor distinto de 0.

- ***Terminación y espera de hilos***

  Hay varias formas de terminar con un hilo:

  Terminación con retorno

  Puede hacerse de dos formas:

  1º) Usando la función return en cualquier momento del código:

  return (void *)p;

  2º) Usando la función pthread_exit:

  void pthread_exit (void *valor)

  Ej: pthread_exit((void *)p);

  Ambos usan el molde (void *) porque recordemos que los hilos devuelven ese tipo de puntero. Una vez acabado el hilo de esta manera, puede recogerse dicho valor usando la función:

  Int pthread_join (pthread_t *thread, void **valor)

El primer argumento de entrada es el identificador del hilo del que se recoge la información, y el segundo es la variable donde se almacenará dicho valor (si no se desea almacenar, se escribe NULL)

```
int *p;
pthread_t thr1;

…
pthread_join(thr1,(void **)&p);
```

Esta función, además, tiene el mismo uso que "wait" en procesos, ya que espera a que el hilo acabe para seguir adelante.

Terminación sin retorno

Una vez se llega al final de la rutina del hilo, este termina. Para evitar que queden "restos" de los hilos en el sistema, es conveniente invocar a:

```
int     pthread_detach      (pthread_t thread)
```

Que lo único que hace es que al acabar el hilo, todos los recursos que utilizaba queden libres. El argumento que recibe es el identificador del hilo del que se desean liberar los recursos al acabar.

Terminación desde otro hilo

Hay que utilizar la función:

```
int     pthread_cancel      (pthread_t thread)
```

Cuyo único argumento de entrada es el identificador del hilo que se desea terminar. No siempre es posible cancelar un hilo de esta forma, pues está ligado a dos conceptos relativos a la cancelabilidad del hilo:

-Estado: puede tomar dos valores, "disabled" y "enabled". En el primero no se puede cancelar, en el segundo se produce en función del tipo de cancelabilidad.

-Tipo de cancelabilidad: define si la cancelación se produce de forma "síncrona" o de forma "asíncrona". En el primero solo se puede cancelar en determinados puntos del código llamados "puntos de cancelación", en el segundo caso se puede cancelar en cualquier instante.

Por defecto todos los hilos comienzan con estado “enabled” y tipo “síncrono”. Para modificar ambos conceptos, se usan dos funciones:

-Para cambiar el estado:

int pthread_setcancelstate (int state, int *oldstate)

El primer argumento puede ser “PTHREAD_CANCEL_ENABLE” (estado enabled) o “PTHREAD_CANCEL_DISABLE” (estado disabled). El entero que devuelve la función es el estado anterior, a menos que se use NULL en el segundo argumento de entrada.

-Para cambiar el tipo:

int pthread_setcanceltype (int type, int *oldtype)

El primer argumento puede ser “PTHREAD_CANCEL_DEFERRED” (tipo síncrono) o “PTHREAD_CANCEL_ASYNCRONOUS” (tipo asíncrono). El entero que devuelve la función es el estado anterior, a menos que se use NULL en el segundo argumento de entrada.

Para crear un punto de cancelación donde las peticiones de cancelación de hilos síncronos se cumplan, se usa la función:

void pthread_testcancel (void)

Simplemente invocándola se crea dicho punto, y en ese momento se comprueba si hay alguna petición de cancelación pendiente, y si es así, se resuelve.

3º Señales

- ***Concepto de señal***

  Las señales son un medio de comunicación entre procesos e hilos. Básicamente se utilizan para notificar eventos excepcionales, es decir, asíncronos, pues no sabemos cuándo pueden ocurrir. Todo su código está incluido en la librería <signal.h>, es decir, que a la hora de usar señales siempre empezaremos con:

  #include<signal.h>

  Cuando se genera una señal, se encuentra pendiente hasta que le "llega" a un proceso, que actuará en consecuencia. Sin embargo, la llegada de señales depende de la máscara del proceso. La máscara del proceso puede entenderse como un número formado por bits, uno para cada señal, y en función de si el bit está a 0 ó a 1, la señal estará o no bloqueada.

  Cuando una señal está bloqueada, esa señal no llega al proceso. Si por el contrario la señal no está bloqueada, una vez que se genera, pueden ocurrir varias cosas:

  -El proceso es **terminado**.
  -La señal es **ignorada**.
  -El proceso se **detiene**.
  -El proceso **continúa** si había sido detenido.
  -Una vez que llega la señal, la recibe un **manejador** ("handler"), una especie de función.

  Hay dos tipos de señales: las señales definidas en la norma 1003.1a de POSIX y las definidas en la norma 1003.1b, que están orientadas a tiempo real. Las diferencias entre uno y otro son: las señales tienen prioridad (a menor número de señal, más prioritarias) desde SIGRTMIN a SIGRTMAX; las señales se encolan, hasta un máximo de SIGQUEUE_MAX; pueden llevar datos adicionales; y se puede esperar una señal en vez de interrumpir una función cuando llega una señal (se entenderá más adelante)

  En cualquier caso, siempre que vayamos a utilizar las señales, hay que hacer dos cosas:

  1º) Programar la máscara del proceso. Paso indistinto del tipo de señal.

  2º) Especificar qué se hace cuando "llega" la señal. Diferente en 1003.1a y en 1003.1b.

- ***Programación de la máscara del proceso***

  Para programar la máscara del proceso haremos uso de "conjuntos de señales" (también son máscaras) que están definidos por el tipo de variable sigset_t. En estos conjuntos añadiremos o eliminaremos señales, que nos servirán para definir la máscara del proceso:

  -Para añadir todas las señales del conjunto:

  int sigfillset (sigset_t *pset)

  El argumento de entrada es un puntero al conjunto al que queremos añadir todas las señales. Por ejemplo:

  sigset_t conjunto;

  sigfillset(&conjunto);

  -Para eliminar todas las señales del conjunto:

  int sigemptyset (sigset_t *pset)

  El funcionamiento es idéntico al de la función anterior.

  -Para añadir una señal concreta:

  int sigaddset (sigset_t *pset, int sig)

  El primer argumento es de nuevo un puntero al conjunto al que vamos a añadirle la señal, mientras que el segundo es el nombre de la señal que queremos añadir (es una macro). Por ejemplo, para bloquear SIGALRM:

  sigset_t conjunto;

  sigaddset(&conjunto,SIGALRM);

  -Para eliminar una señal concreta:

  int sigdelset (sigset_t *pset, int sig)

  El funcionamiento es idéntico al de la función anterior.

  Todas estas funciones devuelven 0 cuando todo va bien, y un -1 si hay algún error.

Una vez definido los conjuntos de señales con las señales que queremos modifica en nuestra máscara del proceso, usaremos la siguiente función para modificarla:

int sigprocmask (int how,const sigset_t *set, sigset_t *oset)

-Primer argumento: es el que modifica en sí la máscara. Podemos escribir tres órdenes:

1. SIG_BLOCK: las señales bloqueadas en la máscara de señales son las que ya tiene bloqueadas más las que están incluidas en el conjunto que se coloca en el segundo argumento de sigprocmask.

2. SIG_UNBLOCK: se desbloquean las señales que están incluidas en el conjunto que se coloca en el segundo argumento de sigprocmask.

3. SIG_SETMASK: se borra la máscara actual y la nueva máscara sólo tiene bloqueadas las señales incluidas en el conjunto que se coloca en el segundo argumento de sigprocmask.

-Segundo argumento: es el conjunto cuyas señales se toman como referencia para las órdenes del primer argumento. Si se pone NULL, no se modifica la máscara.

-Tercer argumento: en este conjunto se guarda la máscara del proceso actual antes de la modificación. Si se escribe NULL, no se guarda.

Esta función también devuelve 0 si todo va bien, y -1 para algún error.
Ejemplo: desbloqueo de la señal SIGALRM.

```
sigset_t  conjunto;
sigemptyset(&conjunto);
sigaddset(&conjunto,SIGALRM);

sigprocmask(SIG_UNBLOCK,&conjunto,NULL);
```

Las señales SIGKILL y SIGSTOP son las únicas que NO pueden ser bloqueadas.

Como hemos dicho, la definición de la máscara es común a las señales de 1003.1a y 1003.1b, pero la programación de lo que se hace una vez que llegan las señales no. Por eso, ahora vamos a explicarlo de forma separada.

- ***Programación para señales POSIX 1003.1a***

Lo primero que hay que hacer es definir una variable estructura “sigaction”, que será utilizada en una función también llamada “sigaction”. Esta estructura está definida de la siguiente forma:

```
struct   sigaction  {
                        void( *sa_handler)      ();
                        sigset_t        sa_mask;
                        int             sa_flags;
                    };
```

-Primer elemento: hay tres posibles opciones:

1. SIG_DFL: se ejecuta la acción por defecto para la señal asociada (las señales definidas tienen una acción asociada)

2. SIG_IGN: se ignora que la señal ha llegado.

3. El nombre de una función manejadora previamente definida.

-Segundo elemento: es un conjunto que se modifica como tal, y que en caso de usar un manejador se utiliza como máscara del proceso.

-Tercer elemento: en 1003.1a, simplemente lo ponemos a 0.

Una vez hemos definido nuestra estructura sigaction, tenemos que utilizar la función sigaction para asociarle una señal. Esta función es de la forma:

```
int     sigaction       (int sig,struct sigaction *act,struct sigaction *oact)
```

El primer argumento es tan solo el nombre de la señal que va a disparar la acción definida en la estructura sigaction que se coloca en el segundo argumento. El último argumento es simplemente la acción que estaba previamente programada, almacenada también en una estructura sigaction. Puede ponerse NULL.

Para entender bien todo esto, lo mejor es usar un ejemplo, en el que supondremos que hay un manejador previamente definido llamado “manejador1”. Ejemplo, pues:

```
struct sigaction        ejemplo;
ejemplo.sa_handler=manejador1;
ejemplo.sa_flags=0;
sigemptyset(&ejemplo.sa_mask);
sigaction(SIGALRM,&ejemplo,NULL);
```

¿Qué es lo que ocurrirá? Cuando la señal SIGALRM se active, al estar asociada al manejador “manejador1”, este se disparará y todo lo que esté definido en él se ejecutará.

- ***Programación para señales POSIX 1003.1b***

La estructura sigaction se define ahora del siguiente modo:

```
struct   sigaction  {
                        void( *sa_handler)      ();
                        sigset_t        sa_mask;
                        int             sa_flags;
                        void(*sa_sigaction)     (int signo, siginfo_t *datos,void *extra);
                    };
```

-Primer elemento: NO se usará.

-Segundo elemento: se usa igual.

-Tercer elemento: hay que ponerla siempre a “SA_SIGINFO”.

-Cuarto elemento: se utiliza exactamente igual que sa_handler en 1003.1a, la diferencia es que una vez se asigna un manejador automáticamente almacena el número de la señal en “signo”, un puntero a una estructura tipo siginfo_t (que ahora definiremos) llamado “datos” en el que se almacenan datos adicionales, y por último un puntero a vacío llamado “extra” cuyo uso no está definido.

La estructura siginfo_t está definida de la forma:

```
typedef struct   {
                        int       si_signo;
                        int       si_code;
                        union sigval      si_value;
                } siginfo_t;
```

El tipo “union sigval” se define así:

```
union sigval {
                int       sival_int;
                void *    sival_ptr;
            };
```

¿Qué es lo que ocurre?

1. El número de la señal que recibe el manejador ahora se encuentra en el entero "signo" y en el valor apuntado por "datos->si_signo".
2. El puntero de la variable "si_value" no suele usarse. Ambos valores se le pasan al manejador a través de la función "sigqueue".
3. El código de "datos->si_code" indica el motivo de la activación del manejador, que puede ser distinto de la activación de "kill":

   SI_QUEUE: señal que genera la función sigqueue.
   SI_TIMER: señal que genera timer.
   SI_ASYNCIO: señal que genera una entrada/salida por pantalla asíncrona.
   SI_MESGQ: señal que genera la llegada de un mensaje a una cola vacía.
   SI_USER: señal que genera un kill u otras llamadas similares.

- ***Generación de señales***

Vamos a explicar primero la función que vale para 1003.1a y para 1003.1b, y después la que solo vale para 1003.1b.

Función kill

Es de la forma:

```
int     kill     (pid_t pid,int sig)
```

El primer argumento es el identificador del proceso en el que vamos a mandar la señal, aunque si escribimos 0 se envía a todos los procesos a la vez. El segundo parámetro es la señal que se va a enviar. Como siempre, devuelve 0 si va bien y -1 si hay algún error.

Función sigqueue

Permite no solo mandar una señal a un proceso sino también mandar datos adicionales. Además, las señales se encolan. Solo se puede usar en 1003.1b. Su forma es:

```
int     sigqueue     (pid_t pid,int sig,const union sigval val)
```

Las dos primeras entradas funcionan igual que en kill. La última es una variable tipo "union sigval" que habremos definido previamente y que se le pasará al manejador, por lo tanto será accesible su variable union sigval. De nuevo devuelve 0 cuando funciona y -1 si no.

Ejemplos de ambas funciones:

```
pid_t    pid;
union sigval valor;
pid=getpid();
valor.sival_int=2;

kill(pid,SIGALRM);
sigqueue(pid,SIGALRM,valor);
```

En ambos casos mandamos la señal SIGALRM al proceso pid, pero en el Segundo además mandamos el dato adicional de "valor".

- ***Espera de señales***

Usamos diferentes funciones para 1003.1a y 1003.1b.

Espera en 1003.1a

Se usa la función sleep, que espera el tiempo que se le pasa a menos que llegue una señal. Es de la forma:

```
unsigned int     sleep    (unsigned int seg)
```

Si acaba el tiempo de espera devuelve 0, y si llega una señal devuelve el tiempo que le quedaba por esperar todavía.

También puede utilizarse la función sigsuspend:

```
int        sigsuspend        (const sigset_t *nueva_mascara)
```

Recibe un conjunto de señales, y lo que hace es bloquear el programa hasta que llega alguna señal que no esté incluida en ese conjunto de señales, en cuyo caso retorna el valor -1, se ejecuta el manejador de esa señal si lo hay, y entonces el programa sigue.

Ejemplo:

```
sigset_t          conjunto;
sigfillset(&conjunto);
sigdelset(&conjunto,SIGALRM);
sleep(30);
sigsuspend(&conjunto);
```

Primero esperaría 30 segundos por sleep, a menos que llegara una señal, y después esperaría hasta que se enviara la señal SIGALRM.

Espera en 1003.1b

Se usa sobre todo la función sigwaitinfo, de la forma:

```
int     sigwaitinfo     (const sigset_t *estas_sg, siginfo_t *info)
```

En este caso, el conjunto de señales del primer argumento debe estar bloqueado previamente con sigprocmask. Al recibirlas, sigwaitinfo las desbloquea y bloquea el resto, continuando el proceso sólo cuando llega una de esas señales que estaban bloqueadas. Cuando la señal llega, sigwaitinfo devuelve el número de esa señal, y además guarda en el puntero siginfo_t la información adicional de esa señal que se hubiera mandado con sigqueue.

Ejemplo: imaginemos que la señal es la que se mandaba en el ejemplo de sigqueue:

```
siginfo_t       info;
sigset_t        conjunto;
sigemptyset(&conjunto);
sigaddset(&conjunto,SIGALRM);
sigprocmask(SIG_BLOCK,&conjunto,NULL);

sigwaitinfo(&conjunto,&info);
```

Cuando se mande la señal SIGALRM, entonces dejará de esperarse, y en la variable "info" se guardará el valor 2 que se mandó, concretamente en info.si_value.sival_int.

También se puede usar la función sigtimedwait:

```
int     sigtimedwait    (const sigset_t *estas_sg, siginfo_t *infop, const struct
                         timespec *timeout)
```

Funciona igual que sigwaitinfo (incluído el valor devuelto), solo que puede añadirse un tiempo límite de espera con una estructura tipo timespec, que es de la forma:

```
struct timespec{
                time_t  tv_sec;
                long    tv_nsec;
                };
```

En el primer elemento se guardan los segundos y en el segundo los nanosegundos. El máximo de ambos es 10^9. Si ambos son cero, lo único que hace la función es comprobar si hay señales pendientes. Del ejemplo anterior:

```
struct timespec t; t.tv_sec=40;
sigtimedwait(&conjunto,&info,&t);
```

- ***Manejadores***

  Los manejadores son funciones que se activan cuando la señal asociada a ellos con la función sigaction se activa. La forma de definirlos es muy sencilla, ya que simplemente se escribe su código entero antes del hilo main. Solo hay que distinguir entre los manejadores para señales definidas en 1003.1a y los que son para señales de 1003.1b.

  Manejadores 1003.1a

```
void    nombremanejador      (int signo)

        {
        <código del manejador>
        }
```

  Manejadores para 1003.1b

```
void    nombremanejador      (int signo,siginfo_t *datos,void *extra)
        {
        <código del manejador>
        }
```

- ***Temporizador basado en señales***

  Se utiliza la llamada:

```
int     alarm   (unsigned int segs)
```

  Cuenta los segundos que recibe, y cuando acaba active la señal SIGALRM. Devuelve 0 si va bien y -1 si hay error, y además los segundos que le quedaban por contar. Para desactivar la señal SIGALRM, o detener el contador, lo único que hay que hacer es introducir como tiempo de espera 0.

  Ejemplo:

```
alarm(100);
sigwaitinfo(&conjunto,&info);
alarm(0);
```

  Activa el temporizador para 100 segundos, pero si la señal que espera sigwaitinfo llega antes, pasa a la siguiente línea de código que desactiva el temporizador, luego no se activaría SIGALRM. En caso de que tardara más de 100 segundos en terminar de esperar sigwaitinfo, SIGALRM se activaría.

4º Temporizadores

- ***Concepto de temporizador***

  Los temporizadores sirven para generar señales tras un cierto tiempo establecido, ya sea una única vez (disparo único) o repetidas veces. Igual que ocurría con las señales, hay una versión básica definida en 1003.1a y otra para tiempo real definida en 1003.1b, bastante más avanzada. En ambos casos, la librería que hay que incluir es la siguiente:

  #include<time.h>

  Los temporizadores no se acumulan, es decir, una vez que disparan su señal, si esta no es atendida y se dispara otro temporizador la señal anterior se pierde. Entre las ventajas de los temporizadores en 1003.1b se encuentra el hecho de que aunque no se traten estas señales, sí se contabiliza el número de ellas que se pierde ("overrun"). Además, aumenta el rango de precisión de tiempo de 1 segundo a 1 nanosegundo, y por último permiten disparar cualquier señal, no sólo SIGALRM.

- ***Temporización en 1003.1a***

  La función time es de la siguiente forma:

  time_t time   (time_t *tiempo)

  La función devuelve un el número de segundos transcurridos desde una referencia fija, que es el 1 de enero de 1970 a las 0:00 horas. En el contenido apuntado por el puntero también se almacena, pero puede ponerse NULL.

  > *El tipo time_t es un entero long, definido en la librería time.h. Puede usarse en cualquier caso un entero long definido como tal, esto es, (long int). Para referenciar el dato por pantalla o almacenarlo, se usa "%ld". El molde de long int es (long).*

  El resto de llamadas de temporización de 1003.1a son sleep y alarm que ya han sido explicadas. No es aconsejable usarlas porque su rango de precisión es 1 segundo. Además, ninguna de las dos puede actuar de forma repetitiva, son de disparo único, y ambas usan la señal SIGALRM luego no pueden utilizarse a la vez sin riesgo de fallos.

- ***Temporización en 1003.1b***

  Para definir los tiempos utilizaremos siempre la estructura "timespec", que ya hemos definido pero la repetimos por comodidad:

```
struct timespec{
        time_t  tv_sec;
        long    tv_nsec;
        };
```

  Recordamos que tanto en el campo de los segundos (tv_sec) como el de los nanosegundos (tv_nsec) el máximo permitido es 10e9.

  Los pasos para programar un temporizador son los siguientes:

  Primer paso: definición de las variables a utilizar

  Tendremos que crear una variable estructura "timespec" para definir el tiempo del temporizador, otra variable estructura "itimerspec" donde se indicarán los tiempos de espera, una última variable estructura "sigevent" donde especificaremos la señal que envía el temporizador y los datos adicionales de ésta, y otra variable "timer_t" que será el identificador de nuestro temporizador. Vamos a explicarlas una a una, salvo timespec que ya lo hemos hecho.

  - ***itimerspec:*** es de la siguiente forma:

```
struc itimerspec        {
                struct  timespec        it_value;
                struct  timespect       it_interval;
                };
```

    En "it_value" ponemos el tiempo que se espera para disparar la señal desde que iniciamos el temporizador; en caso de que valga 0, el temporizador se desactiva. En "it_interval" ponemos el tiempo que espera entre una repetición y otra; en caso de que valga 0, el temporizador será de un solo disparo.

    Por ejemplo:

```
struct  timespec        val={10,0};    /* Para contar 10 segundos
struct  timespec        inter={5,50000000000L};   /* Para contar 5'5 segundos
struct  itimerspec      ejemplo;

ejemplo.it_value = val;   /* La primera vez, esperaré 10 segundos para disparar
ejemplo.it_interval = inter;   /* Después, cada 5'5 segundos dispararé
```

- ***Sigevent:*** es de la siguiente forma:

```
struct   sigevent        {
                         …
                         int       sigev_notify;
                         …
                         int       sigev_signo;
                         …
                         union sigval      sigev_value;
                         };
```

Esta estructura tiene muchos más campos (de ahí los "…") pero a nosotros sólo nos interesan esos. En "sigev_notify" definimos cómo vamos a notificar el evento, en nuestro caso escribimos "SIGEV_SIGNAL", que es lo que se escribe cuando se notifica mediante una señal, o si no queremos que se notifique de ninguna manera, escribimos "SIGEV_NONE". En el campo "sigev_signo" metemos la señal que queremos que dispare el temporizador posteriormente. Por último, en "sigev_value" metemos la información adicional que mandamos con la señal.

Por ejemplo:

```
struct   sigevent        exp;

exp.sigev_notify = SIGEV_SIGNAL;  /* Uso como notificación una señal
exp.sigev_signo = SIGRTMIN;  /* La señal SIGRTMIN, en este caso
exp.sigev_value.sival_int = 8;  /* Mandaré el número 8 con la señal
```

Segundo paso: creo el temporizador

Se usa una función que crea el temporizador pero no lo inicializa:

```
int       timer_create     (clockid_t reloj, struct sigevent *aviso, timer_t *tempo)
```

El primer argumento se introduce el reloj que se va a utilizar. Si bien se pueden definir relojes, nosotros SIEMPRE utilizaremos para cualquier función que reciba un "clockid_t reloj" la macro "CLOCK_REALTIME". El segundo argumento será un puntero a la estructura "sigevent" que hayamos definido, con la señal que queremos que active el temporizador y los valores adicionales que mande. El último argumento es un puntero al identificador de nuestro temporizador, la variable "timer_t" que creamos antes, que será lo que nos permitirá referenciar el temporizador a otras funciones. El valor de retorno, como siempre, será 0 si todo va bien, y -1 si hay algún error.

Por ejemplo, siguiendo con los descritos en el primer paso:

```
timer_t          tempor;  /*  Identificador de nuestro temporizador
timer_create(CLOCK_REALTIME,&exp,&tempor);   /* Creamos el temporizador
```

Tercer paso: arrancar el temporizador

Se usa la función:

```
int      timer_settime   (timer_t tempo, int flags, const struct itimerspec *spec, struct
                          itimerspec *spec_ant)
```

El primer argumento de esta función es el identificador de nuestro temporizador. El segundo argumento indica cómo se interpreta el valor del tercer argumento: nosotros siempre escribiremos "0", que significa que espera lo que indica el "it_value" del tercer argumento para disparar por primera vez; si por el contrario escribimos "TIMER_ABSTIME", lo que hace es que coge el valor de "it_value", que son segundos, y le suma esos segundos a la fecha de referencia (las 0:00 horas del 1 de enero de 1970) y compara el resultado con el reloj de tiempo real, disparando por primera vez si la fecha ha sido superada, y esperando en caso de que no. En cualquiera de los dos casos, en el tercer argumento se coloca el puntero al "imiterspec" que creamos antes, que indica tanto cuando se dispara por primera vez como el intervalo entre los disparos, si los hay. El último argumento es un puntero a otro "itimerspec" para guardar el valor antiguo, puede dejarse a NULL si no se quiere guardar. Como siempre, la función retorna 0 si todo ha ido bien y -1 si ha habido algún error.

Para no tener que calcular los segundos desde una determinada fecha, usamos la estructura "tm" para definir nuestra fecha y la función "mktime" que la convierte en segundos, ambas definidas en la librería time.h:

```
struct   tm      {
                 int      tm_hour;  /* Horas de 0 a 23
                 int      tm_min;  /* Minutos de 0 a 59
                 int      tm_sec;  /* Segundos de 0 a 59
                 int      tm_mon;  /* Mes de 0 a 11
                 int      tm_mday;  /*Día del mes de 1 a 31
                 int      tm_year; /* El año, siendo 0 el año 1900
                 };

time_t   mktime(struct tm *fecha)
```

La función recibe el puntero de la estructura "Tm" y devuelve la fecha convertida en segundos desde la referencia (0:00 horas del 1 de enero de 1970).

Vamos a continuar primero el ejemplo anterior para inicializar el temporizador, y después escribiremos un ejemplo con struct tm y mktime. La continuación sería:

```
timer_settime(&tempor,0,&ejemplo,NULL);  /* Inicializamos "tempor" con lo indicado
                                            en la estrucutura "ejemplo"
```

Para que disparara por primera vez el 03-07-13 a las 20:30 horas, por ejemplo, tendríamos que modificar "ejemplo" de la siguiente forma:

```
time_t          tiempo;
struct  tm      fecha = {20,30,0,6,3,1013};
tiempo=mktime(&fecha);
ejemplo.it_value = tiempo;

timer_settime(&tempor,"TIMER_ABSTIME",&ejemplo,NULL);
```

Cuarto paso: destruir el temporizador

Es tan sólo usar la función:

```
int     timer_delete    (timer_t tempo)
```

El argumento que recibe es el identificador del temporizador que queremos eliminar, y el que devuelve es 0 si todo va bien, y -1 si hay algún error.

Continuando el ejemplo:

```
timer_delete(tempor);
```

Además, hay dos funciones que hay que conocer porque nos dan información sobre un determinado temporizador en funcionamiento. Estas son:

- ***timer_gettime:*** nos permite saber cuánto falta para el próximo disparo. Es de la forma:

  ```
  int     timer_gettime  (timer_t tempo, struct itimerspec *queda)
  ```

  El primer argumento es el identificador del temporizador del que queremos saber cuánto tiempo falta para que dispare. El segundo argumento es un puntero a una estructura "itimerspec", donde en el valor "it_value" guarda el tiempo que resta para disparar, y en el valor "it_interval" guarda el intervalo de disparo, que puede ser diferente del que se programó porque el ordenador lo redondea a un número entero de periodos de su reloj. Como siempre, devuelve 0 si va bien y -1 si no.

- ***timer_getoverrun:*** permite saber cuántas veces se ha disparado el temporizador sin que su señal se trate. Es de la forma:

  int timer_getoverrun(timer_t tempo)

  Recibe el identificador del temporizador que vamos a comprobar, y devuelve el número de veces que se ha disparado. Si se supera el límite, lo que devuelve es "DELAYTIMER_MAX".

- ***Funciones que llaman al reloj de referencia en 1003.1b***

  Las siguientes funciones nos permiten trabajar con el reloj de tiempo real:

  - *Conocer el tiempo actual:* para ello se usa la función

    int clock_gettime (clockid_t reloj, struct timespec *tiempo)

    El primer argumento como hemos dicho será "CLOCK_REALTIME". El segundo argumento es un puntero a la estructura "timespec" donde se guardará el tiempo actual. De nuevo, devuelve 0 si va bien y -1 si no.

  - *Conocer la resolución/precisión del reloj:* para ello se usa la función

    int clock_getres (clockid_t reloj, struct timespec *resol)

    El primer argumento como hemos dicho será "CLOCK_REALTIME". El segundo argumento es un puntero a la estructura "timerspec" donde se guardará la precisión del reloj. De nuevo, devuelve 0 si va bien y -1 si no.

  - La función "clock_settime" permite modificar el reloj, pero no la usaremos.

5º Mensajes

- ***Concepto de mensajes***

  Los mensajes son un tipo de comunicación entre diferentes procesos. Lo mejor son las colas de mensajes de 1003.1b, frente a PIPEs y FIFOs.

  Una cola de mensajes es como un espacio en memoria que crea un proceso pero no dentro de él, esto es, la cola de mensajes existe independientemente del proceso. Una vez creada la cola, cada proceso puede "abrir" dicha cola con unos permisos (ya sea de lectura, escritura o ejecución) y a partir de ahí intercambiar mensajes con dicha cola. Por ejemplo, si un proceso abre una cola con un permiso de escritura, a partir de ahora puede mandar mensajes a la cola, pero no recibirlos, pues necesitaría un permiso de lectura. Posteriormente, un proceso puede "cerrar" la cola, en cuyo caso **no destruye** la cola, simplemente interrumpe su comunicación con ella. No obstante, hay una función determinada que sí destruye colas desde los procesos que la invocan.

- ***Creación y apertura de una cola de mensajes***

  Lo primero que tenemos que hacer es añadir la librería correspondiente:

  #include<mqueue.h>

  En esta librería se incluyen dos elementos importantes:

```
mqd_t
struct   mq_attr        {
                        long     mq_maxmsg;
                        long     mq_msgsize:
                        long     mq_flags;
                        long     mq_curmsgs;
                        …
                        };
```

  El primer elemento es un identificador de una cola de mensajes. El segundo es una estructura donde especificaremos las características que queremos para nuestra cola de mensajes, siendo respectivamente:

  1. El máximo número de mensajes que puede almacenar la cola.
  2. El máximo tamaño de un mensaje que almacena la cola.
  3. El único flag disponible es O_NONBLOCK, pero nosotros siempre pondremos 0.
  4. Indica cuántos mensajes quedan por recibir en la cola (se actualiza solo)

La función que se utiliza para crear y abrir una cola es la misma, mq_open, de la forma:

```
mqd_t  mq_open      (const char *mq_name, int oflag, mode_t modo, struct
                    mq_attr *atributos);
```

Vamos a explicar uno a uno los argumentos que recibe:

- El primero es el nombre que le vamos a poner a la cola. Si queremos ponerle de nombre “Paco”, tendremos que escribir “/Paco”. Es recomendable que no contenga más que al principio el carácter ‘/’.

- El segundo es donde se especifica para qué se abre la cola, que pueden ser las siguientes:

  - O_RDONLY: abre la cola solo para lectura.
  - O_WRONLY: abre la cola solo para escritura.
  - O_RDWR: abre la cola para escritura y lectura.
  - O_CREAT: si no existe la cola, se crea la cola.
  - O_EXCL: si se pone con O_CREAT, la función devolverá un fallo llamado EEXIST a través de errno si ya existía la cola.
  - O_NONBLOCK: pase lo que pase, el programa no se bloqueará cuando se llame a esta función.

- El tercero indica los permisos que tendrá el proceso que abre la cola sobre la misma, siendo estos definidos de la siguiente forma:

  S_I + R,W,X + USR/GRP

  El primer sumando siempre es igual. El segundo es una de esas letras, que dan permiso de lectura (R), escritura (W) o ejecutar (X). El último sumando es quién tiene el permiso, el usuario (USR) o el grupo de usuarios (GRP). Por ejemplo, S_IRUSR permite leer al usuario. Si queremos leer y escribir, ponemos S_IRUSR | S_IWUSR.

  En caso de que queramos habilitar los tres permisos, existen constantes especiales: S_IRWXU (usuario) y S_IRWXG (grupo).

  Para poder hacer todo esto, deben estar incluidas las librerías sys/stat.h y sys/types.h

- El cuarto argumento es un puntero a los atributos definidos en una estructura del tipo mq_attr. Si ya está creada la cola, pondremos NULL.

El argumento que devuelve es el identificador de la cola creada/abierta, y será -1 siempre que haya habido algún error.

Ejemplo:

```
mqd_t                   cola;
struct  mq_attr         atributos;
atributos.mq_maxmsg = 100;
atributos.mq_msgsize = 128;
atributos.mq_flags = 0;

cola = mq_open("/Paco", O_CREAT | O_RDWR, S_IRWXU, &atributos);
```

Creamos una estructura atributos, donde definimos el número de mensajes máximo que puede almacenar la cola (100), el tamaño máximo de cada mensaje (128) y ponemos las flags a 0. Después, creamos al cola "Paco", cuyo identificador es la variable "cola", y la hemos creado para que se pueda leer y escribir en ella, además de abrirla a nuestro proceso para que el usuario pueda leer, escribir y ejecutar.

- ***Cerrar y eliminar colas de mensajes***

  Las dos funciones que vamos a utilizar son, respectivamente, mq_close y mq_unlink:

  - ***mq_close:*** esta función es de la forma

    ```
    int     mq_close        (mqd_t cola)
    ```

    Recibe el identificador de la cola a la que se va a dejar de acceder, es decir, con la que se interrumpe la comunicación, pero NO la destruye. Devuelve un -1 si hay algún error, 0 si va bien.

  - ***mq_unlink:*** esta función es de la forma

    ```
    int     mq_unlink       (const char *nombre)
    ```

    Recibe el nombre de la cola que se quiere eliminar, es decir, destruir permanentemente, liberando los recursos asociados a ella. Si algún proceso estaba usándola, se destruye cuando el último proceso que la esté usando use "mq_close", pero una vez invocada "mq_unlink" ningún otro proceso puede abrirla. Devuelve un 0 si todo va bien y -1 si hay error.

    NOTA: en general, antes de crear una cola, por mera seguridad se escribe una línea de comando con mq_unlink destruyendo esa cola, por si ya existía.

  Vamos a ver un ejemplo de las dos funciones, continuando con el anterior:

  mq_close(cola); mq_unlink("/Paco"); →Cierro primero y después destruyo.

- ***Envío y recepción de mensajes***

  Las dos funciones que se utilizan son muy similares, por eso las explicaremos juntas:

  ```
  int      mq_send      (mqd_t cola, const char *datos, size_t longitud,
                         unsigned int prioridad);

  size_t   mq_receive   (mqd_t cola, const char *datos, size_t longitud,
                         unsigned int *prioridad);
  ```

  - El primer argumento es el identificador de la cola a la que va a mandarse o de la que se va a recibir un mensaje.

  - El segundo argumento es un puntero a una cadena de caracteres, que en el caso de "mq_send" es el mensaje que se manda a la cola, y en el caso de "mq_receive" es el mensaje que se recibe de la cola.

  - El tercer argumento es el tamaño máximo del mensaje enviado. Para quitarse de problemas, lo lógico es poner siempre el tamaño máximo definido en los atributos de la cola a la que se mandan o de la que se reciben mensajes.

  - El cuarto argumento es la prioridad, y está definida entre 0 y un número máximo definido como MQ_PIO_MAX (al menos 32). Cuanto mayor sea el número más prioritario es el mensaje, y se trata antes. En "mq_send" ponemos directamente la prioridad que queremos. En el caso de "mq_receive" se pasa el puntero a una variable unsigned int, y en él se guarda la prioridad del mensaje recibido.

  En ambos casos, si algo va mal se devuelve un -1, pero en el caso de "mq_receive" si todo va bien se almacena la longitud del mensaje recibido. Es importante destacar que las llamadas bloquean el proceso si no pueden bien mandar el mensaje (no hay espacio en la cola) bien recibir un mensaje (no hay mensajes en la cola), pero no bloquearan el proceso si se activó el flag O_NONBLOCK, solo devolverán un error EAGAIN en errno.

  Ejemplo con la cola "Paco" suponiendo que no la hemos cerrado ni destruido:

  ```
  char     mensaje[128],recibido[128];
  unsigned int     prio;
  mensaje="Esto es un ejemplo";
  mq_send(cola,mensaje,128,0);
  mq_receive(cola,recibido,128,&prio);
  ```

  Ahora "recibido" tiene también el mensaje "Esto es un ejemplo".

- ***Activación de señales por mensajes***

  Lo que se consigue es que cuando llegue un mensaje a una cola, automáticamente se active una señal, que puede ser tratada por un manejador. Para ello, hay que hacer lo siguiente:

  1. ***Utilizar una estructura sigevent:*** recordamos su forma

     ```
     struct   sigevent        {
                              …
                              int       sigev_notify;
                              …
                              int       sigev_signo;
                              …
                              union sigval      sigev_value;
                                       };
     ```

     En ella tenemos que hacer:

     - *Primer argumento:* ponerlo a "SIGEV_SIGNAL"
     - *Segundo argumento:* poner el nombre de la señal que queremos que se active cuando llegue un mensaje a la cola.
     - *Tercer argumento:* tenemos que pasar como dato adicional no un entero, sino un puntero a vacío de la cola con la que trabajamos.

  2. ***Utilizar mq_notify:*** esta función es de la siguiente forma

     ```
     int       mq_notify       (mqd_t cola, const struct sigevent *espec)
     ```

     Recibe como primer argumento el identificador de la cola en la que la recepción de un mensaje activa la señal. El segundo argumento es un puntero a la estructura sigevent que hemos creado previamente. Devuelve como siempre un -1 si hay algún error.

NOTA IMPORTANTE: una vez que llega por primera vez un mensaje y se activa la señal, el efecto de mq_notify desaparece, es decir, que cuando llegue otro mensaje la señal no se activará. Para que eso ocurra habría que volver a invocar a mq_notify, y suele hacerse desde el manejador que trata la señal que se activa al llegar el mensaje.

Ejemplo utilizando la cola "Paco" de los ejemplos anteriores:

```
struct sigevent  senal;
senal.sigev_notify = SIGEV_SIGNAL;    senal.sigev_signo = SIGRTMIN;
senal.sigev_value.sival_ptr = (void *)&cola;
mq_notify(cola,&senal);
```

Lo que pasará cuando llegue un mensaje a la cola “Paco” es que se activará la señal SIGRTMIN. Si quisiéramos que siempre que llegaran mensajes se activara SIGRTMIN, tendríamos que hacer algo así en el manejador asociado a SIGRTMIN:

```
void    man     (int sig, siginfo_t *info, void *nada)
{
        mqd_t  *cola;
        struct sigevent  espec;
        cola = (mqd_t *)info->sival_ptr;

        espec.sigev_notify = SIGEV_SIGNAL;
        espec.sigev_signo = SIGRTMIN;
        espec.sigev_value.sival_ptr = (void *)cola;

        mq_notify(*cola,&espec);
}
```

- ***Conocer los atributos de una cola***

Para ello se utiliza la siguiente función:

```
int     mq_getattr      (mqd_t cola,struct mq_attr *atributos)
```

Recibe el identificador de la cola cuyos atributos se quieren conocer, y un puntero a una estructura mq_attr donde se almacenan los atributos de la cola consultada. Si todo va bien, devuelve un 0, y si no un -1.

Esta función es sobre todo interesante para conocer el valor de “mq_curmsgs”, que recordemos son los mensajes pendientes de llegar a la cola.

Ejemplo de uso, de nuevo usando la cola “Paco”:

```
struct mq_attr  receptor;

mq_getattr(cola,&receptor);
```

Ahora, receptor.mq_maxmsg vale 100, y receptor.mq_msgsize vale 128, como los atributos que definimos para la cola “Paco”.

7º Semáforos

- ***Semáforos: qué son y cómo funcionan***

  Un semáforo es un elemento que da acceso a un recurso de la memoria a un proceso y se lo niega al resto hasta que el anterior termine de utilizarlo. Es pues una forma de comunicación y sincronización entre procesos.

  Su funcionamiento es el siguiente: un semáforo se inicializa con un valor entero no negativo, y siempre que dicho valor sea no negativo, los procesos pueden acceder al recurso que controla el semáforo. Siempre que un proceso vaya a acceder a un recurso, tiene que restarle una unidad al semáforo, y cuando acabe de usarlo, sumarle una unidad. Veámoslo con un ejemplo:

  Vamos a suponer que inicializamos un semáforo "SEM" con valor inicial "1" que controla el acceso al recurso "A". El proceso "PACO" quiere acceder al recurso "A", así que lo primero que hace es restarle una unidad al semáforo "SEM", con lo cual el valor del entero del semáforo es "0". Al ser un valor no negativo, "PACO" puede utilizar el recurso "A". Acto seguido el proceso "PEPE" quiere acceder al recurso "A", y para ello resta una unidad al semáforo "SEM", y el valor del entero sería ahora "-1", y como es un valor negativo, el proceso "PEPE" no accede al recurso "A" y se queda bloqueado en una cola de espera. Cuando el proceso "PACO" acabe de usar el recurso "A", le sumará "1" al semáforo "SEM" y entonces el valor del entero será "0" y al ser no negativo, el proceso "PEPE" podrá acceder al recurso "A", y al acabar, sumará también "1" al semáforo "SEM" dejando todo como estaba al principio.

  Hay que aclarar que si el entero del semáforo llega un valor "-2", por ejemplo, en cuanto otro proceso sume "1" al mismo entrará alguno de los procesos que estaban esperando, aunque el valor del entero sea negativo.

  Evidentemente, somos nosotros los que en el código indicamos que cada proceso primero reste una unidad al semáforo, haga lo que tenga que hacer con el recurso, y después sume una unidad.

  Por último, cabe mencionar que existe un tipo de semáforos llamados "binarios", que sólo pueden tomar los valores 0 y 1, pero no los vamos a usar.

- ***Creación, acceso y destrucción de semáforos***

  Sólo están definidos en 1003.1b. Hay dos tipos de semáforos, con nombre y sin nombre, y para cada uno se hace una cosa diferente, pero ambos usan la librería

  #include<semaphore.h>

  - ***Semáforos con nombre:*** funciona de forma similar a las colas de mensajes, es decir, una vez creado, cualquier proceso puede obtener permiso para utilizarlo. Para ello se usan las siguientes funciones:

    ```
    sem_t *     sem_open    (const char *nombre,int oflags,
                             mode_t modo,  Unsigned int inicial)
    ```

    En el primer parámetro de esta función, hay que escribir, como en las colas, el nombre del semáforo, por ejemplo "SEM", y para ello de nuevo lo haremos escribiendo "/SEM". El segundo y tercer parámetros se usan exactamente igual que en las colas, salvo que sólo pueden usarse los flags O_CREAT y O_EXCL (el resto no tiene sentido). Por último, el cuarto parámetro es el valor inicial que va a tener nuestro semáforo. El puntero tipo sem_t que devuelve apunta al semáforo creado.

    ```
    int     sem_close     (sem_t *semaforo)

    int     sem_unlink    (const char *nombre)
    ```

    Estas dos funciones funcionan exactamente igual que en las colas: "sem_close" cierra el semáforo para el proceso que lo invoca, y "sem_unlink" lo elimina del sistema, una vez que todos los procesos que lo usaban han invocado a "sem_close".

    El tipo de variable "sem_t" es un identificador del semáforo.

  - ***Semáforos sin nombre:*** se crean en una zona de memoria que si es compartida por varios procesos pueden utilizarlo para sincronizarse, sino sólo tiene sentido para procesos que tengan muchos hilos de ejecución (threads). Si va a usarse memoria compartida, hay que inicializar esta, que ya veremos más adelante cómo se hace. Las funciones en este caso son:

    ```
    int     sem_init      (sem_t *semaforo,int compartido,
                           Unsigned int inicial)
    ```

    Esta función recibe el puntero de tipo sem_t que previamente tenemos que haber puesto apuntando a la memoria compartida creada para almacenar el semáforo. Esto normalmente se hace así:

```
sem_t  *psem;
Memoria = <código para crear la memoria compartida>;  /* Memoria es un
                                                          Puntero */
psem = (sem_t *)memoria;
```

El segundo argumento que recibe vale “1” si va a usarse el semáforo en memoria compartida, y “0” si va a ser utilizado por diferentes hilos del mismo proceso. El último argumento es de nuevo el valor inicial del semáforo.

Para destruir el semáforo se usa la función:

```
int       sem_destroy    (sem_t *semáforo)
```

Que sólo recibe el puntero que apunta a la memoria compartida. Es importante notar que si se elimina dicha memoria compartida, también desaparece el semáforo.

- ***<u>Sincronización con semáforos: aumento y decremento de su valor</u>***

Las siguientes funciones son comunes para ambos tipos de semáforos:

```
int       sem_wait       (sem_t *psem)
int       sem_trywait    (sem_t *psem)
int       sem_post       (semt_t *psem)
int       sem_getvalue   (sem_t *psem,int *value)
```

La primera función es la que decrementa en uno el valor del entero del semáforo, y si dicho valor queda como negativo, pues bloquea al proceso. La segunda función hace lo mismo salvo que no bloquea al proceso, se limita a devolver -1 con el error EAGAIN. La tercera función es la que suma “1” al valor del entero, y la última función almacena en el contenido apuntado por el puntero que recibe como segundo argumento el valor del entero del semáforo que recibe en el primero. Es obvio que todas estas funciones reciben como puntero sem_t el del semáforo que se usa.

8º Memoria compartida

- ***Pasos para crear una memoria compartida***

  Las operaciones que vamos a realizar son las siguientes:

  1. Abrir un objeto compartido de memoria, que se manejará como un pseudoarchivo.
  2. Fijar la longitud de dicho objeto.
  3. "Mapear" el objeto sobre el espacio de direcciones del proceso.
  4. Cuando deje de usarse, cerrar el pseudoarchivo.

  Las librerías que tenemos que incluir en cualquier caso son las siguientes:

  #include<sys/mman.h>
  #include<sys/stat.h>
  #include<fcntl.h>
  #inlcude<limits.h>

- ***Creación del objeto de memoria compartido***

  Para ello se utiliza la función shm_open:

  int shm_open (const char *nombre,int oflags,mode_t modo)

  De nuevo el primer argumento es un nombre, que al igual que en colas de mensajes y semáforos, se escribe entrecomillado y con el carácter "/" antes del nombre, por ejemplo, para llamarlo "MEMORIA" pondríamos "/MEMORIA". Los flags y el modo se usan exactamente igual, aunque no puede usarse O_WRONLY. El entero que devuelve es un descriptor de archivo que utilizaremos hasta el paso de "mapeo".

- ***Fijar la longitud del objeto de memoria***

  Inicialmente el objeto de memoria tiene una longitud cero. Aunque se puede poner cualquier tamaño, nosotros siempre pondremos un número entero de páginas de la memoria virtual. El tamaño de estas páginas puede obtenerse con:

  long intsysconf (int name)

El argumento que le pasaremos será "_SC_PAGESIZE" y la dimensión de la página de la memoria virtual se almacenará en el entero long, que para nosotros será una variable tipo "off_t".

Una vez obtenida la dimensión de la página de la memoria virtual, para fijar la longitud del objeto de memoria usaremos:

int ftruncate (int fd,off_t tamaño)

Recibe como primer argumento el descriptor de archivo del objeto compartido del cual se va a fijar la longitud, y como segundo argumento el tamaño que queremos para el mismo (un número entero de páginas)

- ***"Mapeo" del objeto en el espacio de direcciones***

Vamos a usar la función mmap, que recibe muchos argumentos pero casi siempre usaremos los mismos:

void * mmap (void *donde,size_t long,int protec,int mapflags,int fd,off_t offset)

El primer argumento es un puntero que indica al sistema dónde ubicar el objeto de memoria, pero no implica que lo vaya a colocar ahí porque puede que no pueda. Lo normal es que este argumento valga 0, que significa que el sistema colocará el objeto donde quiera.

El segundo argumento indica el tamaño de la zona mapeada, nosotros siempre colocaremos el mismo tamaño que le pasamos como segundo argumento a ftruncate, porque otro valor se usaría simplemente si quisiéramos acceder a un solo fragmento de la memoria compartido.

El tercer argumento son las protecciones de memoria que queremos establecer. Pueden ser: PROT_READ (habilita lectura), PROT_WRITE (habilita escritura), PROT_EXEC (habilita ejecución) y PROT_NONE (no habilita ningún modo de acceso). Pueden usarse varias, pero deben concordar con los permisos dados en shm_open.

El cuarto argumento puede tomar tres valores: MAP_SHARED (permite compartir el mapeo con otros procesos, nosotros SIEMPRE usaremos este); MAP_PRIVATE (se crea una copia privada del objeto mapeado, así que los cambios que haga el proceso que ejecuta mmap no los verán el resto de procesos); MAP_FIXED (obliga al sistema a aceptar el valor del primer argumento como dirección para almacenar el objeto de memoria).

El quinto argumento es el descriptor de archivo que nos devolvió shm_open. Es la última vez que lo usamos antes de destruirlo. El último argumento siempre será 0.

El valor de retorno será lo que utilicemos a partir de ahora para identificar al objeto de memoria compartida, por ejemplo para usarlo en los semáforos sin nombre, y será un puntero a entero (para que sea así utilizaremos un molde).

Una vez hayamos usado mmap, ejecutaremos la siguiente función:

```
int     close   (int fd)
```

Que elimina el descriptor de archivo, puesto que ya no es necesario.

- ***Cierre del pseudoarchivo***

Primero tenemos que liberar la memoria que no vayamos a usar, de forma similar a lo que hacíamos con "mq_close", con la función:

```
int     munmap      (void *comienzo,size_t long)
```

Los dos argumentos trabajan conjuntamente: el primero es un puntero que apunta al principio de la memoria desde donde se empezará a borrar, y se borrará desde donde apunta dicho puntero hasta esa posición más la cantidad fijada en el segundo. Para liberar toda la memoria, el primer argumento será el identificador del objeto de memoria que nos devuelve "mmap" y el segundo el tamaño que le pusimos.

Para destruir completamente el objeto de memoria se utilizará la función:

```
int     shm_unlink      (const char *nombre)
```

Que sólo recibe como argumento el nombre del objeto de memoria que le pusimos en shm_open, escrito también como "/nombre". Como siempre, hasta que todos los que abrieron no liberen toda la memoria con "munmap", no se eliminará por completo el objeto, sólo se impedirá el acceso.

Todos los procesos que vayan a usar el objeto de memoria compartida tienen que llevar a cabo estos pasos.

Vamos a ver un ejemplo en el que se utilizan todas estas funciones explicadas:

```
<se introducen las cabeceras>

int     descriptor;
int     *objeto;
off_t   tamañoobjeto;

Tamaño = sysconf(_SC_PAGESIZE);
```

```
descriptor=shm_open("/Ejemplo",O_CREAT | O_RDWR, S_IRWXU);
ftruncate(descriptor,tamaño);
objeto=(int *)mmap(0,tamaño,PROT_READ | PROT_WRITE,
                   MAP_SHARED,descriptor,0);
close(descriptor);

<lo que sea que se haga con el objeto de memoria>

munmap(objeto,tamaño);
shm_unlink("/Ejemplo");
```

9º Mutex y variables de condición

- ***Mutex: qué son y para qué se usan***

  Un mutex es parecido a un semáforo binario, pues sólo toma dos valores, "cerrado" (lock) o "abierto" (unlock) pero se usa exclusivamente para impedir o permitir el acceso a parte de su código a los hilos de un mismo proceso. Una característica muy importante de los mutex es que si hay varios hilos bloqueados, al activarse el mutex que los bloquea solo pasará uno de ellos, y dependerá de la prioridad de cada uno y del algoritmo de planificación elegido. Estos parámetros se fijan en los atributos del mutex. La librería necesaria es pthread.h (ya utilizada en hilos)

- ***Inicialización de un mutex***

  Hay dos formas de inicializar un mutex:

  - ***Inicialización estática:*** es el método más sencillo pero no permite modificar los atributos del mutex. Simplemente tenemos que escribir:

    ```
    pthread_mutex_t        mut = PTHREAD_MUTEX_INITIALIZER;
    ```

    Donde pthread_mutex_t es una variable identificador de nuestro mutex, "mut" es el nombre que le hemos puesto a dicha variable y PTHREAD_MUTEX_INITIALIZER es la constante con la que inicializamos estáticamente nuestro mutex.

  - ***Inicialización dinámica:*** tenemos que escribir un poco más de código pero podemos modificar los atributos del mutex, si bien nosotros no vamos a hacerlo. Para ello usamos la función

    ```
    int      pthread_mutex_init     (pthread_mutex_t *mutex,const
                                     pthread_mutexattr_t *attr)
    ```

    Recibe como primer argumento el puntero a mutex con el que identificaremos a nuestro mutex, y que tenemos que inicializar antes (ahora lo vemos con el ejemplo). El segundo argumento son los atributos que queremos otorgarle a nuestro mutex, o NULL. Vemos un ejemplo de inicialización dinámica:

    ```
    pthread_mutex_t        *mut;
    mut = (pthread_mutex_t *)malloc(sizeof(pthread_mutex_t));  /* Para que
                        nuestro puntero apunte a suficiente espacio en la memoria */
    pthread_mutex_init(mut,NULL);
    ```

- ***Abrir y cerrar un mutex***

  Usaremos dos funciones muy sencillas. Para cerrar un mutex usaremos la función:

  int pthread_mutex_lock (pthread_mutex_t *mutex)

  Recibe un puntero al mutex que vamos a cerrar. Si el mutex está ya cerrado, la función bloquea al hilo hasta que dicho mutex se abra, en cuyo caso el hilo que ha invocado pthread_mutex_lock es el que cierra el mutex. Recordemos que cerraremos un mutex cuando queramos usar variables que queramos vetar al resto de hilos.

  Para abrir de nuevo el mutex y permitir que otro hilo acceda a dichas variables, usaremos la función:

  int pthread_mutex_unlock (pthread_mutex_t *mutex)

  Que funciona de idéntica forma. Un ejemplo de ello es, siguiendo con la inicialización estática antes realizada:

  pthread_mutex_lock(&mut);

  <lo que sea que hagamos>

  pthread_mutex_unlock(&mut);

  Existe una función similar a sem_trywait en semáforos, y es:

  int pthread_mutex_trylock(pthread_mutex_t *mutex)

  En este caso si el mutex está cerrado ya, en vez de bloquear al hilo simplemente devuelve el error EBUSY.

- ***Destruir un mutex***

  Para liberar los recursos asociados a un mutex, se utiliza la función:

  int pthread_mutex_destroy (pthread_mutex_t *mutex)

  Que de nuevo recibe el puntero que identifica al mutex.

- ***Variables de condición: qué son y para qué se usan***

Las variables de condición sirven para sincronizar hilos eficazmente. Cada variable de condición está ligada a un mutex y a una condición binaria (sí o no), cuyo estado depende de unos datos que protegerá el mutex, para que no puedan activarlo muchos hilos a la vez, sólo uno, el que permite el mutex.

De esta forma, imaginemos que tenemos dos hilos, "PACO" y "PEPE", que usan una variable global "A", cuyo valor incrementa "PACO", y cuyo acceso controla el mutex "MUT" y que existe una variable de condición "COND", que se activa cuando "A" vale 5 y permite que "PEPE" ponga "A" al valor 0. En ese caso, el pseudocódigo que gobierna a cada hilo sería:

```
"PACO"
Bucle infinito (while(1), por ejemplo)
        Bloqueo MUT
        Incremento el valor de A
        Compruebo la condición: si A = 5 -> activo la variable de condición
                                si no: no hago nada
        Libero MUT y acabo
Fin bucle

"PEPE"
Bucle infinito
        Bloqueo MUT
        Bucle: mientras A sea menos de 5, hago
                Espero la variable de condición (esta espera libera el mutex asociado
                                                 Mientras espera, y lo bloquea cuando
                                                 Se termina de esperar)
        Fin bucle
        Pongo A a cero
        Libero MUT
Fin bucle
```

Cuando se espera a que se active una variable de condición, se hace dentro de un bucle que comprueba dicha condición porque una vez se activa la variable todos los hilos que están esperando compiten para cerrar el mutex asociado, y el primero que lo consiga cerrará el mutex y modificará variables y por tanto, cuando acabe de hacer lo que haga, la condición que gobierna la variable de condición puede que no se cumpla, pero al liberar el mutex otro hilo de los que competían antes continuará sus acciones, a pesar de que la condición puede no cumplirse. Por eso se hace el bucle: porque al continuar, comprueba que se cumpla, y si es así, continúa, si no, vuelve a esperar.

El código de las variables de condición es muy parecido al de los mutex, como vamos a ver a continuación.

- ***Inicialización de una variable de condición***

  Igual que con los mutex, hay dos formas de hacerlo:

  - ***Inicialización estática:*** es el método más sencillo pero no permite modificar los atributos de la variable de condición. Simplemente tenemos que escribir:

    ```
    pthread_cond_t          cond  = PTHREAD_COND_INITIALIZER;
    ```

    Donde pthread_cond_t es una variable identificador de nuestra condición, "cond" es el nombre que le hemos puesto a dicha variable y PTHREAD_COND_INITIALIZER es la constante con la que inicializamos estáticamente nuestra variable.

  - ***Inicialización dinámica:*** tenemos que escribir un poco más de código pero podemos modificar los atributos de la variable, si bien nosotros no vamos a hacerlo. Para ello usamos la función

    ```
    int     pthread_cond_init     (pthread_cond_t *cvar,const
                                   pthread_condattr_t *attr)
    ```

    Recibe como primer argumento el puntero a variable de condición con el que identificaremos a nuestra variable, y que tenemos que inicializar antes (ahora lo vemos con el ejemplo). El segundo argumento son los atributos que queremos otorgarle a dicha variable, o NULL si queremos los que vienen por defecto (es como la inicialización estática). Vemos un ejemplo de inicialización dinámica:

    ```
    pthread_cond_t          *cond;
    cond = (pthread_cond_t *)malloc(sizeof(pthread_cond_t));  /* Para que
                        nuestro puntero apunte a suficiente espacio en la memoria */
    pthread_cond_init(cond,NULL);
    ```

- ***Espera de una condición***

  Para ello, recordemos que siempre que esperemos una condición lo haremos dentro de un bucle que comprueba que esa condición se cumple, ya que una vez que se termina la espera de la activación de la variable de condición, es posible que se nos adelante otro hilo y ya no se cumpla la condición.

  La función que se utiliza para esperar que se active la variable de condición es:

  ```
  int     pthread_cond_wait     (pthread_cond_t *cond,pthread_mutex_t *mutex)
  ```

Esta función recibe como primer parámetro el puntero a la variable de condición inicializada, y como segundo el mutex asociado a ella. De esta forma, cuando se invoca esta función, lo primero que hace ella es liberar el mutex, y después se pone a esperar. Cuando se active la variable de condición, intenta cerrar el mutex. Si alguien lo ha cerrado ya, sigue esperando hasta que vuelva a liberarse el mutex, momento en el que cual intenta cerrarlo de nuevo. Una vez lo consiga, acaba.

Existe otra función que permite poner una limitación de tiempo a la espera de la activación de la variable de condición, y es:

```
int     pthread_cond_timedwait     (pthread_cond_t *cond,pthread_mutex_t
                                    *mutex,const struct timespec *abstime)
```

Los dos primeros parámetros que recibe son idénticos a los de la función anterior. El tercero es el tiempo ***absoluto*** en el que cual se acaba la espera, no un intervalo de tiempo. Es decir, tenemos que obtener el tiempo del reloj con "clock_gettime" (página 25) y después sumarle el tiempo que queramos esperar. El valor de retorno es cero si la causa por la que termina la espera es que se active la variable de condición, y el código de error ETIMEDOUT si es que se ha alcanzado el tiempo marcado.

- ***Activación de una variable de condición***

Cuando un hilo se bloquea esperando con "pthread_cond_wait", solo puede ser desbloqueado si otro hilo activa la variable de condición (y además el mutex está abierto claro) ya sea porque dicho hilo es el que hace que se cumpla la condición, o porque detecta que dicha condición ha sido cumplida por un tercero. En cualquier caso, la función que se utiliza para activar la variable de condición es:

```
int     pthread_cond_signal    (pthread_cond_t *cond)
```

Recibe únicamente el puntero de la variable de condición que activa. Esta función hace que al menos uno (y en general sólo uno, aunque pueden ser más) de los hilos bloqueados se desbloquee y continúe, cerrando el mutex, etc

La otra función que se puede utilizar para activar la variable de condición es:

```
int     pthread_cond_broadcast     (pthread_cond_t *cond)
```

Funciona igual que la anterior, pero en este caso TODOS los hilos sean desbloqueados, y que compitan todos por intentar cerrar el mutex, y después de que el que gane lo libere, sigan compitiendo.

Lo normal es que el hilo que activa la variable de condición haya cerrado previamente el mutex, así que después de usar una de estas dos funciones, debe liberarlo.

Teoría

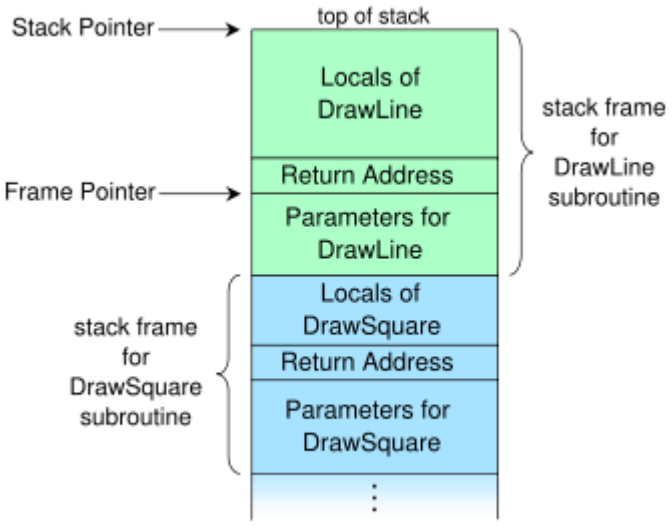
1º La pila del sistema

Una pila de llamadas es una estructura dinámica de datos LIFO (“Last In First Out”) que almacena información sobre las subrutinas activas de un programa de ordenador. La principal razón por la que se usa es para seguir el curso del punto al cual cada subrutina activa debe retornar el control cuando termine de ejecutarse, es decir, su principal función es almacenar la dirección de retorno. Aparte, también puede usarse para almacenar datos locales, pasar parámetros, etc. En general, hay una pila asociada a cada tarea o hilo de un proceso.

La pila del sistema se divide en registros de activación (“stack frames”) donde cada uno corresponde a una llamada a una subrutina activa, es decir, una subrutina que no ha terminado con un retorno a quién la llamó. El otro elemento fundamental es el puntero de pila (“stack pointer”) que apunta a la posición del último elemento que entró en la pila, es decir, el que debe ejecutarse ahora. En otras palabras, el valor del puntero de pila es el límite actual de la pila.

Cuando un programa llama a una subrutina, el programa que llama empuja (“push”) la dirección de retorno, es decir: si antes el puntero de pila apuntaba a la posición 0, ahora apunta a la posición 1 que es donde está la subrutina llamada. Cuando la subrutina finaliza, retira (“pop”) la dirección de retorno de la pila y transfiere el control a esa dirección, volviendo hacia atrás, por lo que el puntero de pila apunta de nuevo a la posición 0.

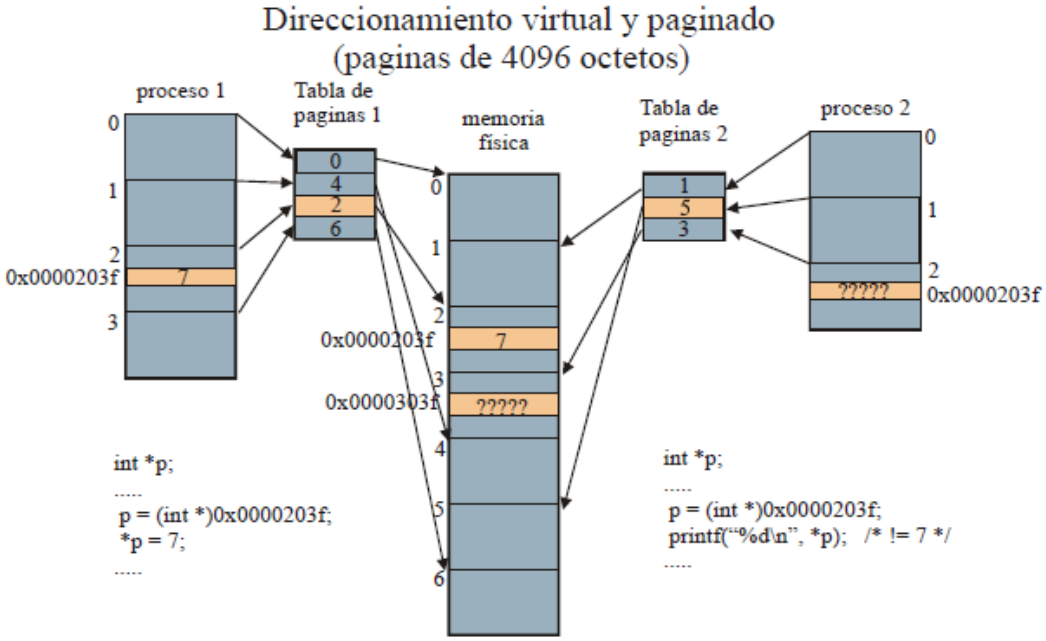
En este ejemplo, la subrutina “dibuja cuadrado” llama a la subrutina “dibuja línea” que es la subrutina activa. Cuando se la llama, en la pila se guardan los parámetros de dicha subrutina y la dirección de retorno de la subrutina que la invocó, es decir, a donde debe apuntar el puntero de pila una vez que se retire “dibuja línea”. De este modo, al retirar “Locals of draw line” se retorna a “locals of Draw Square”.



2º Memoria virtual y direccionamiento paginado

El direccionamiento virtual consiste en asignarle a cada proceso un espacio de direccionamiento propio, siendo necesario que se traduzcan después las instrucciones en dicho espacio en direcciones reales. Cuando hacemos un direccionamiento paginado, lo que hacemos es dividir la memoria disponible en páginas de una longitud igual a una cantidad que debe ser potencia de 2.

Para traducir en el direccionamiento virtual y paginado, se utilizan las tablas de página. Cada tabla de página tiene una entrada para cada página virtual, y almacena los números de páginas reales. Cuando se invoca una instrucción en una memoria virtual, esta se traduce en la tabla de páginas para hacérsela llega a la memoria real:



Los problemas que soluciona la memoria virtual y el direccionamiento paginado son:

1. La gestión de memoria, es decir, buscar sitio para un proceso.
2. La protección de datos, es decir, crear "barreras" entre procesos.
3. La reubicación, es decir, evitar que un proceso cambie de dirección.

3º Propiedades

- ***Thread-safe:*** en un proceso con varios hilos, es posible que varios de ellos intenten acceder a la vez a la misma función. Esta propiedad exige que las funciones no vean alterado su funcionamiento por este hecho. Por ejemplo, cuando dentro de una función se accede a datos globales, esta propiedad impide que varias invocaciones accedan a los mismos.

- ***Async-safe:*** es una propiedad que define a la rutina como segura, es decir, que no van a detenerse por la llegada de señales. Esto es muy importante en los manejadores, que no deben invocar a funciones "inseguras" (como printf). La manera más fácil de hacer que una rutina sea "async-safe" es bloquear las señales antes de llamarla y desbloquearlas después.

4º Pipes y FIFOs

Son colas circulares de datos que permiten conectar procesos. Los pipes son más antiguos quelas FIFOs (First In First Out), y son como una "tubería" que conecta un proceso con otro. El problema de las pipes es que son unidireccionales, luego para una comunicación bidireccional (por ejemplo, que ambos procesos se envíen datos) necesitas dos, y además, los procesos conectados tienen que tener por fuerza la relación "padre-hijo".

Las FIFOs funcionan de forma similar pero son más flexibles. Los principales inconvenientes de las FIFOs son:

- No existe el concepto de prioridad (lógico: first in first out)
- No se puede controlar o conocer de cuánto espacio se dispone para almacenar mensajes en la cola que estén pendientes de recibir.
- No se puede saber cuántos mensajes hay encolados ni otros estados.
- No se puede definir una estructura de los mensajes que llegan (como su tamaño)

5º Monitores

Los monitores agrupan los datos compartidos y las rutinas (funciones) que permiten realizar las operaciones con los datos compartidos. De esta forma, varios procesos o hilos pueden acceder a unas mismas variables, pero sólo uno de ellos estará a la vez dentro de él.

6º Modelos de concurrencia

Para conseguir que la concurrencia en un programa, pueden hacer estas tres cosas:

- ***Utilizar un solo proceso:*** el tiempo de que se dispone se divide entre todas las tareas cuidadosamente. Es viable, pero es muy complejo de diseñar y lo normal es que de fallos. Es lo que se hace cuando se diseña un ejecutivo cíclico.

- ***Utilizar un sistema "monotarea" con dos actividades concurrentes:*** una de las actividades concurrentes realiza las tareas normales, mientras que la otra sólo funciona cuando hay alguna interrupción. Viable para sistemas sencillos, pero no para los complejos.

- ***Una tarea para cada actividad o grupo de actividades relacionadas:*** habrá un sistema operativo que distribuye los recursos con algoritmos genéricos (no para una solución concreta) Además, las tareas concurrentes las pueden llevar a cabo "hilos". La principal pega es que el modelo impuesto por el lenguaje puede ser difícil de implementar en un determinado sistema operativo y el resultado puede ser ineficiente.

7º Procesos e hilos

- ***Proceso:*** una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados

- ***Hilos:*** son parte de un proceso, comparten el espacio de direcciones, pero se crea una pila nueva y sólo los datos globales son accesibles desde todos, porque el resto de datos son locales. Es menos costoso crear hilos que procesos nuevos, pero no existe protección entre hilos.

8º Restricciones temporales

- ***Coste (C):*** tiempo de ejecución máximo sin interferencias de otras actividades. Es difícil de medir.

- ***Tiempo de respuesta(R):*** tiempo de ejecución en condiciones reales, obviamente nunca es inferior al coste.

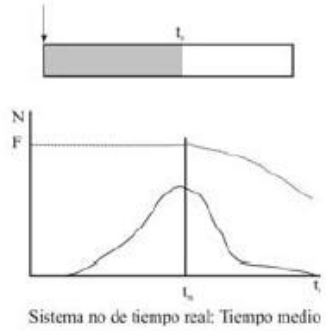
- ***Tiempo límite ("deadline")(D):*** máximo tiempo de respuesta admisible, ídem.

9º Tipos de sistemas en tiempo real

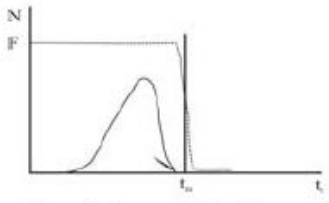
- ***Sistemas críticos ("hard real time"):*** el tiempo de respuesta debe ser siempre menor que el coste tiempo máximo.

- ***Sistemas no críticos ("soft real time"):*** se pueden tolerar retrasos ocasionales, pero con algunas restricciones:

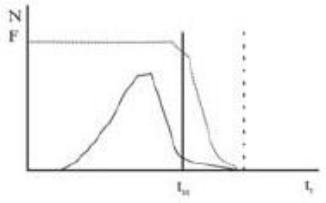
    - *Restricciones firmes ("firm"):* si se incumplen, el resultado carece de valor.
    - *Restricciones no firmes:* si se incumplen, el resultado es útil pero no óptimo.



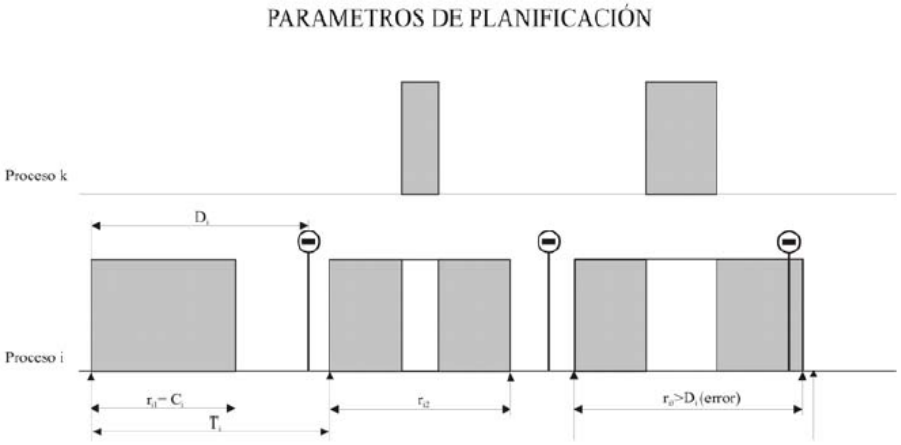
Sistema no de tiempo real: Tiempo medio



Sistema de tiempo real crítico: Tiempo máximo



Sistema de tiempo real no crítico



10º Modelos de sincronización por paso de mensajes

- ***Asíncrono:*** el emisor no espera a que los datos sean recibidos y continúa con sus operaciones. Requiere el uso de un elemento denominado “cola de espera” donde se almacenan los mensajes no recibidos. Es el que usa POSIX.

- ***Síncrono:*** el emisor sí espera a que se reciban los datos para continuar con sus operaciones.

- ***Invocación remota:*** usa un protocolo de llamada-respuesta, el emisor no sólo se limita a esperar la recepción del mensaje, también espera a que el destinatario lo procese y le responda después de realizar sus operaciones. También se conoce como “cita”. Está presente en ADA.

Notas generales sobre programación
Main(int argc,char **argv)

El entero argc devuelve el número de argumentos que se le han pasado a la función main. La cadena de caracteres argv es un vector char y cada elemento corresponde a uno de esos argumentos pasados a main.

Apuntar cómo se pasan los arguementos (Ver las prácticas)

Mirar también nanosleep

Ver continue()