# Programación Concurrente

Resúmenes

Jesús Sanz Marcos

e-mail: jesus.sanz@upcnet.es

Barcelona, Spain. 01/01/2001

## Procesos en UNIX

### Entrada salida

```
if open(nombre,{0='r';1='w';2='rw'})=-1
then error.

if create(nombre,{0='r';1='w';2='rw'})=-1
then error.

actual = read(fd,dirección,tamaño);
actual = write(fd,dirección,tamaño);
(actual>0) bytes transferidos.
(actual=0) EOF.
(actual<0) algún error.

posicion = lseek(df,desplazamiento,
  {from 0=BOF,1=posición actual;2=EOF});
 (posicion>=0) posición actual.
 (posicion<0)  algún error.

if close(fd)<>0 then error.
```

### Ejecución de programas

```
#include <stdio.h>
```

Estos programas sustituyen el programa actual por el nuevo, lo ejecutan y finalizan en el segundo sin retornar al primero.

```
execl("bin/date/","date",NULL);
 conocemos el número de parámetros

execv(char *path, char *argv[]);
 no conocemos el númeoro de parámetros.

execv(char *path, char *arg1, char *arg2, …);
 busca el fichero ejecutable en el PATH.
```

### Creación y control de procesos

```
proc_id = fork();
 divide el proceso en dos copias que continuan
ejecutándose y que son idénticas.

 En el proceso hijo: proc_id = 0.
 En el procedo padre:
   proc_id = identificador del proceso hijo.

int status;
wait(&status);
 espera a que alguno de sus hijos finalice,
devolviendo el identificador del último proceso
hijo que ha acabado.

exit(0=todo bien,<>0 algún error);
 Finaliza el proceso actual.
```

### Gestión de hilos básica

```
#include <pthread.h>

void * funcion (void *arg) { … }

pthread_t funcion_id;
char * arg;
if pthread_create(&funcion_id, NULL,
  funcion, (void*) arg)<>0 then error.
 Crea un proceso nuevo (funcion) y lo ejecuta
concurrentemente con el proceso llamante.

pthread_exit(&result);
 Finaliza el proceso actual y pasa el resultado al
proceso padre.

pthread_join(funcion_id, void**thread_return);
 Suspende la ejecución del proceso llamante hasta
que el proceso funcion finaliza o es cancelado.

pthread_detach(funcion_id);
 Pone al proceso en el modo DETACHED. Es decir,
garantiza que todos los recursos de memoria se
liberarán después de la ejecución del proceso pero
no será posible la sincronización con pthread_join.
```

### Mútex

```
#include <pthread.h>
```

Un Mútex es un dispositivo de Exclusión Mutua y es útil para proteger estructuras de datos compartidas de modificaciones concurrentes y para implementar zonas críticas y monitores.

Un mútex tiene dos posibles estados: bloqueado (propiedad de un hilo) y desbloqueado –locked/unlocked-. Un mutex jamás puede ser propietario de dos hilos diferentes simultáneamente. Cuando un hilo intenta bloquear a un mutex que ya está bloqueado por otro hilo, es suspendido hasta que el hilo-propietario desbloquea el mutex.

```
pthread_mutex_t fastmutex =
PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t recmutex  =
PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;

pthread_mutex_t errchkmutex =
PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;


int pthread_mutex_init(pthread_mutex_t   *mutex,
        const pthread_mutexattr_t *mutexattr);

PTHREAD_MUTEX_INITIALIZER
 mutex tipo 'fast',

PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP
 mutex tipo 'recursive'

PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP
 Mutex tipo 'error-checking'


int pthread_mutex_lock(pthread_mutex_t *mutex);
 Bloquea el mutex. Si el mutex estaba desbloqueado,
pasa a ser propiedad del hilo que lo ha llamado y
la función finaliza inmediatamente. Si el mutex
estaba bloqueado por otro hilo, entonces se
suspende el hilo que lo ha llamado hasta que el
mutex esté desbloqueado.

 Si el mutex estaba ya bloqueado por el hilo
llamante, el comportamiento depende del tipo de
mutex.
'fast': se produce deadlock.
'error checking': la función finaliza
inmediatamente devolviendo el código EDEADLK.
'recursive': la función finaliza inmediatamente y
devuelve el número de veces que se ha llamado a la
función. Es decir, el número de veces que se ha
desbloquear para que el mutex esté desbloqueado
desde el punto de vista de los demás threads.
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```
Actúa igual que pthread_mutex_lock(), pero si el mutex se encuentra bloqueado, finaliza inmediatamente y devuelve el valor EBUSY.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```
Desbloquea el mutex.
'**fast**': desbloquea siempre el mutex.
'**recursive**': decrementa el contador del mutex (número de veces que el mútex propietario ha llamado a pthread_mutex_lock) y sólo cuando alance cero se desbloqueará el mútex.
'**error checking**': comprueba en tiempo real que el mutex haya sido bloqueado por el mismo thread que ahora lo llama.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```
Destruye el objeto mutex, liberando sus recursos de memoria. El mútex tiene que haber sido desbloqueado antes.

## Semáforos

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```
(pshared = 0) solamente lo pueden utilizar los hilos del proceso.

```
int sem_destroy(sem_t *sem);
```
Destruye el semáforo, excepto si hay todavía algún hilo esperando ya que entonces finaliza y devuelve EBUSY.

```
int sem_post(sem_t *sem);
```
Automáticamente incrementa el contador del semáforo. Esta función nunca se bloquea. Devuelve cero si OK y –1 si el contador del semáforo fuese a exceder de SEM_VALUE_MAX.

```
int sem_wait(sem_t *sem);
```
Suspende el hilo llamante hasta que el contador del semáforo tenga un valor superior a cero. Entonces *decrementa* atómicamente el contador del semáforo.

```
int sem_trywait(sem_t *sem);
```
variante *no-blocking* de sem_wait. Si el semáforo tiene un valor mayor que cero entonces *decrementa* atómicamente este valor y finaliza. Si el contador del semáforo es cero, finaliza y devuelve –1.

```
int sem_getvalue(sem_t *sem, int *sval);
```
Almacena en sval el valor actual del semáforo.

**(Ejemplo)**

```
sem_t candado;
sem_init(&candado, 0, 1);
/*…*/

sem_wait(&candando);
/* Zona crítica */
sem_post(&candado);

/*…*/
sem_destroy(&candado);
```

## Condiciones

```
#include <pthread.h>
```

Una variable de condición es un dispositivo de sincronización que permite a los hilos suspender su ejecición hasta que un predicado en los datos compartidos se satisfaga. El procedimiento báscio es el siguiente: señalizar la condición (cuando el predicado sea cierto) y esperar hasta que otro hilo señalice la condición.

```
int pthread_cond_init(pthread_cont_t *cond,
                      pthread_condattr_t *cont_attr);
```
Inicializa la condición. Devuelve cero. El segundo parámetro debe ser NULL. Las condiciones e pueden inicializar también mediante la constante PTHREAD_COND_INITIALIZER.

```
int pthread_cond_signal(pthread_cond_t *cond);
```
Activa uno de los hilos que están esperando a la condición. Si no hay hilos esperando la condición, no pasa nada. Si más de un thread está esperando, exactamente uno se activa, pero no se especifica cual. Devuelve siempre 0.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```
Activa todos los hilos que están esperando la condición. No ocurrre na da si no hay ningún hilo. Devuelve siempre 0.

```
int pthread_cond_wait(pthread_cond_t *cond,
pthread_mutex *mutex);
```
Desbloquea el mutex y espera a que la condición se señalice. Se suspende la ejecución del thread. El mutex debe ser bloqueado por el hilo llamante a la entrada de esta función. Antes de finalizar, se bloquea de nuevo el mutex. Devuelve siempre 0.

```
int pthread_cond_timedwait(pthread_cond_t *cond,
pthread_mutex *mutex, const struct timespec
*abstime);
```
Igual que pthread_cond_wait pero limita el tiempo de espera, devolviendo el código ETIMEDOUT si se supera este espacio de tiempo en espera.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```
Destruye la condición si no hay hilos esperándola. En este caso, devuelve EBUSY.

```
int pthread_cond_empty(pthread_cond_t *cond);
```
Devuelve TRUE si no hay ningún hilo esperando la condición.

**(ejemplo)**

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER:
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

/* … */

pthread_mutex_lock(&mut);
while (x<=y)
{
   pthread_cond_wait(&cond, &mut);
}
/* operar x,y */
pthread_mutex_unlock(&mut);

proceso:
        pthread_mutex_lock(&mut);
        /* modificar x,y */
        if (x>y) pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&mut);
```

## Paso de mensajes

```
#include <sys/socket.h>
#include <sys/poll.h>

int socket(int domain, int type, int protocol);
```
Crea un socket (enchufe) en un dominio de comunicación y devuelve un descriptor de archivo que puede ser utilizado para posteriores operaciones con sockets.

**domain**: espeficia la familia de direcciones utilizada en el dominio de comunicaciones. Depende de la aplicación y del sistema.

**type:**
*SOCK_STREAM*
Proporciona una flujo (stream) secuencial, fiable, bidireccional y orientado a bytes.
*SOCK_DGRAM*
Proporciona datagramas, que son mensajes no fiables y no orientados a conexión de un tamaño máximo.
*SOCK_SEQPACKET*
Proporciona una camino secuencial, fiable, bidireccional y orientado a conexión para la

transmisión de registros. Cada operación sólo puede enviar un registro.

```
int socket(int domain, int type, int protocol,
           int sockets[2]);
```
Crea un par de sockets conectados. Los dos sockets son idénticos.

**(ejemplo)**

```
#include" <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#define DATA1 "In Xanadu, did Kubla Khan . . ."
#define DATA2 "A stately pleasure dome decree"

main()
{
  int sockets[ 2] , child;
  charbuf [1024] ;

if(socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) <0)
   Error();

if ((child = fork( ) ) = = -1) Error();

if (child) { /* padre */

close (sockets[ 0] );
if (read(sockets[1] , buf, 1024) < 0) Error();
if (write(sockets [1] , DATA2, sizeof(DATA2)) < 0)
   Error();
close(sockets [1] ); }

if (!child) { /* hijo */

close(sockets[ 1 ]);
if (write(sockets[ 0] , DATA1, sizeof(DATA1)) < 0)
  Error();
if (read(sockets [0] , buf, 1024) < 0) Error();
close(sockets [0 ]);  }
}
```

```
int poll (struct pollfd fds[], nfsdt_t nfds,
          int timeout);
```
Proporciona a las aplicaciones un mecanismo para multiplexar entradas y salidas sobre una serie de descriptores de archivo. Para cada miembro del array apuntado por fds, poll() examina el descriptor de archivo para los acontecimientos especificados por nfds. La función poll() identifica estos descriptores de archivos en los que la aplicación puede leer o escribir datos.

POLLIN
Todos los datos excepto los de alta prioridad se podrán leer sin bloqueo.
POLLRDNORM
Los datos normales (prioriddad=0) se porán leer sin bloqueo.
POLLRDBAND
Los datos de prioridad>0 se podrán leer sin bloqueo.
POLLPRI
Los datos de alta prioridad se podrán recibir sin bloqueo.
POLLOUT
Los datos normales (prioridad=0) se podrán escribir sin bloqueo.

Si ninguno de los acontecimientos se producen en el tiempo en milisegundos espeficiado por timeout. Si timeout=0 poll() finaliza inmediatamente, mientras que si timeout=-1 se bloquea hasta que algún acontecimiento se produce o la llamada es interrumpida.

Si alguno de los acontecimientos se ha producido, poll() devuelve un entero no negativo. Un valor positivo indica el numero total de descriptores que se han seleccionado. Un valor de cero indica que hay time_out. Si hay fallo devuelve –1.

```
#include <unistd.h>

int pipe(int filedes[2]);
```
Crea un par de descriptores tales que filedes[0] es para leer y filedes[1] es para escribir.

**(ejemplo)**

```
int fd[2];
int lee;
int salida;
struct pollfd poll_task;

salida = socketpair(PF_LOCAL, SOCK_STREAM, 0, fd);

poll_task.fd = fd[1];
poll_task.events = POLLIN;

/*...*/
write(fd[1],cadena,len(cadena));
/*...*/

while (...)
{
lee = poll(&poll_task, 1, 100);
if (!lee) read(fd[1],ch,1);
}
```

# El Shell del UNIX

## Ejecución en segundo plano

```
gcc pgm.c &
```

## Redirección entrada/salida

```
ls –l >file
/* si file existe se borra y si no existe se crea. La salida se escribe en file. */
ls –l >>file
/* si file existe se borra y si no se mantiene. La salida se añade al final de file. */
gcc <file
/*  la entrada estándar de un programa se redirecciona al contenido del archivo file. */
```

## Pipes y filtros

```
ls-l | gcc
/* equivale a ls-l>file; gcc<file. A excepción que no se crea ningún archivo. Los dos programas se ejecutan a la vez y escriben y leen en paralelo. */

ls | grep old | gcc -l
/* grep escoge aquellas cadenas de la salida ls que contengan old y gcc compila las salidas de old. */
```