

Fundamentos de Software de Comunicaciones

Tema 2
Programación del
Sistema Operativo
(2^a parte)

Contenidos

- Señales
- Manejo del tiempo

Generación de avisos entre procesos

- Objetivo: enviar notificaciones (avisos o señales) entre procesos
- ¿Hemos enviado ya señales a procesos en la asignatura? ¿Os suena qué ocurre al pulsar las teclas Control + C en un terminal? ¿Qué ocurre?
- Mensajes que aparecen cuando se va a apagar el sistema
 - Es importante atender a este tipo de avisos para poder hacer un cierre ordenado del proceso
 - Cierre de ficheros o de canales de comunicaciones

Generación de avisos entre procesos

- Proceso: programa en ejecución
 - El ejecutable cargado en memoria
 - Datos
 - Recursos del sistema, como ficheros
 - Un contexto de seguridad (usuario asociado, permisos)
 - Una o más hebras de ejecución
 - Un ordenador virtual
- Gestión de procesos → sistema operativo
 - De forma jerárquica:
 - Todo proceso tiene un padre
 - Un proceso puede tener muchos hijos
 - Identificador de proceso: PID (Process IDentifier)
 - Número positivo mayor que cero
 - Init: PID = 1
 - Listado de procesos del sistema: el comando ps
 - ps aux
 - ps -ef
 - ps -ef | grep <nombre ejecutable>

Definición de señal

- En el mundo UNIX, una señal es similar a una interrupción hardware, que se envía a un proceso a través del S.O.
- Es un evento asíncrono que hace saltar el flujo de ejecución de un proceso
- Objetivo: informar a un proceso de situaciones que pueden ocurrir en tiempo de ejecución
 - Críticas para seguir o no con su funcionamiento normal
 - La mayoría notifican errores (sólo unas pocas son informativas)
- El sistema operativo determina un comportamiento por defecto cuando llega uno de estos avisos
 - En general, significa que el proceso termina → man 7 signal

Definición de señal

- man 7 signal
 - SIGINT: Interrumpir el proceso
 - SIGTERM: Señal de terminación cuando la máquina se apaga
 - SIGUSR1 y SIGUSR2: señales para que los usuarios asignen su significado
 - Determinar qué se hace en ese caso
- Esto nos permite programar el sistema de avisos
 - Enviando señales
 - Haciendo que nuestros programas puedan manejar esos avisos

Envío de señales

■ En el terminal

kill -s <señal> <PID>

■ En nuestro programa

- **pid_t**: tipo de datos para almacenar identificadores de proceso
- Llamada al sistema **kill()**
 - PID
 - Señal: definidas como constantes en **signal.h**
- **#define _POSIX_SOURCE**
 - Para evitar un warning

```
1 #define _POSIX_SOURCE
2 #include <stdio.h>
3 #include <signal.h>
4 #include <stdlib.h>
5 #include <sys/types.h>
6
7 int main(int argc, char * argv[]) {
8
9     if (argc < 2) {
10         printf("Uso: %s <PID>\n", argv[0]);
11         return 1;
12     }
13
14     pid_t pid = atoi(argv[1]);
15
16     int retorno = kill(pid, SIGTERM);
17     if (retorno < 0) {
18         perror("kill");
19         return 1;
20     }
21
22     return 0;
23 }
```

Manejo de señales

- El programador puede modificar este comportamiento por defecto e indicar al sistema operativo si desea IGNORAR la señal o ATENDERLA
 - Algunas NO se pueden ignorar o atender, como SIGKILL
 - La llamada al sistema **signal()** permite realizar estas modificaciones
- Una buena política de programación consiste en atender a las señales que afecten directamente al programa:
 - SIGTERM: el S.O. solicita que el proceso termine
 - SIGCHLD: un proceso hijo ha terminado su ejecución
 - SIGPIPE: el S.O. avisa de que se está escribiendo a un fichero cerrado

La función signal()

```
#include <signal.h>
typedef void (*sighandler_t) (int);
sighandler_t signal(
    int signo,
    sighandler_t handler);
```

```
7  void m(int signo) {
8      write(1,"Catched!\n",9);
9  }
10
11 int main() {
12     signal(SIGUSR1,m);
13 }
```

■ **Registra un manejador para la señal indicada**

- Cuando llega la señal al proceso, se suspende la ejecución del proceso
iiivaya por donde vaya!!!
- Entonces, se ejecuta la función manejadora (puntero a función)

■ Valor devuelto

- **SIG_ERR** (constante) si ha habido algún error
- El manejador de señal que atendía la señal previamente

■ Argumentos

- La señal que se quiere manejar: **SIGINT**, **SIGQUIT**, **SIGUSR1**, ...
- El manejador de señal: puntero a una función que tiene como argumento un entero y no devuelve nada

La función signal()

- Recordar: la función registra un manejador para la señal indicada, **en ningún caso dispara una señal**
- Manejadores especiales:
 - Ignorar: **SIG_IGN**
 - Restaurar el manejador por defecto: **SIG_DFL**
- Funcionamiento
 - Ignorar **SIGINT**
`signal(SIGINT, SIG_IGN) ;`
 - Restaurar el valor por defecto para **SIGINT**
`signal(SIGINT, SIG_DFL) ;`

La función signal()

- Recordar: la función registra un manejador para la señal indicada, **en ningún caso dispara una señal**
- La sección 2 del manual (en Español) está incompleta, hay que ir a la web
- Funcionamiento en Linux de la llamada **signal(s,m)** :
 1. En primer lugar, antes de la invocación del manejador, se restaura el manejador por defecto. Sería una llamada equivalente a

```
signal(s, SIG_DFL);
```
 2. Se invoca al manejador

```
(*m)(s);
```
 3. Se devuelve el control al programa principal

Señales en Unix System V vs BSD

- En el modelo **System V** de Unix, cuando el S.O. ejecuta el manejador asociado a una señal también restaura el manejador que tuviera por defecto
 - Por eso se vuelve a llamar a signal dentro del manejador si se quiere mantener éste cuando vuelva a aparecer la señal
 - Comprobable en Linux al compilar en C con la opción -ansi:
 - gcc -ansi -o exec programa.c
 - gcc -std=c99 -o exec programa.c
- En el modelo **BSD** las señales son más potentes
 - No hace falta realizar la restauración del manejador, ya que se mantiene para las siguientes veces en que aparezca la señal
 - Comportamiento por defecto en gcc (sin la opción -ansi ni std=c99) y en g++

Ejemplo de uso para signal()

```
void manejador(int signum) {
    write(1,"la tengo!!\n",11); /*escribe por pantalla*/
    signal(SIGINT, manejador); /*restaura el manejador*/
} /* manejador */
int main() {
    ...
    /* solicita al S.O. que ejecute la funcion manejador
cuando llegue la señal SIGINT */
    if (signal(SIGINT,manejador) == SIG_ERR) {
        perror("signal");
        exit(-1);
    }
    ...
} /* main */
```

Ejemplo de uso para signal() (II)

```
/*Se ignora el control de errores por comodidad */

void manejador(int signum) {
    write(1,"la tengo!!\n",11); /*escribe por pantalla*/
    signal(SIGPIPE, manejador); /*restaura el manejador*/
    signal(SIGTERM,SIG_DFL); /*vuelve a considerar TERM*/
} /* manejador */
int main() {
    ...
    /*ignora la señal SIGTERM*/
    signal(SIGTERM,SIG_IGN);
    /* solicita al S.O. que ejecute la función manejador
cuando llegue la señal SIGPIPE*/
    signal(SIGPIPE,manejador);
    ...
} /* main */
```

Diseño de manejadores de señal

■ Recordatorio

- La llegada de una señal interrumpe la ejecución del proceso en **cualquier lugar**
- Puede haber zonas críticas cuya interrupción lleven al proceso a un fallo irrecuperable

■ Reglas básicas para manejadores

- Reducir al máximo las operaciones que realizan
- Todo lo que se pueda hacer fuera del manejador, debe hacerse
- Si no hay más remedio, se deben usar funciones reentrantas
 - Por ejemplo: **write()** vs **printf()**

■ ¿Cómo hacer, por ejemplo, que nuestro programa no abra un fichero y lo muestre por pantalla hasta que no llega SIGUSR1?

Diseño de manejadores de señal

Opción 1

```
3 int ha_llegado_sigusr1 = 0;
4
5 void ejecutar_si_llega_sigusr1(int num_sig) {
6     ha_llegado_sigusr1 = 1;
7     signal(SIGURS1,ejecutar_si_llega_sigusr1);
8 }
9
10 int main() {
11     signal(SIGURS1,ejecutar_si_llega_sigusr1);
12
13     while(ha_llegado_sigusr1 == 0) {
14         pause();
15     }
16
17     //Ha llegado SIGUSR1
18     //Abro el fichero y lo muestro por pantalla
19     // int fd = open()
20     // ...
21
22     /* Resto de código de nuestro programa */
23     /* ...
24         ...
25         ... */
26
```

Opción 2

```
4 void ejecutar_si_llega_sigusr1(int num_sig) {
5     //Abro el fichero y lo muestro por pantalla
6     // int fd = open()
7     // ...
8     signal(SIGURS1,ejecutar_si_llega_sigusr1);
9 }
10
11 int main() {
12     signal(SIGURS1,ejecutar_si_llega_sigusr1);
13
14     /* Resto de código de nuestro programa */
15     /* ...
16         ...
17         ... */
18 }
```

Otras funciones útiles de <signal.h>

- **raise(signal_id)**
 - Envía una señal al proceso en ejecución
 - Ejemplo: te envías una señal TERM a ti mismo
`raise(SIGTERM);`
- **kill(pid_destino_signal, signal_id)**
 - Envía una señal a un proceso distinto al que ejecuta esta llamada
 - Ejemplo: envía TERM al proceso 6550:
`kill(6550, SIGTERM);`
- **pause()**
 - Hace que el proceso se duerma hasta que reciba una señal

Contenidos

- Señales
- **Manejo del tiempo**

Gestión del tiempo

- La gestión del tiempo es fundamental en el software de comunicaciones
- Necesitamos que el S.O. nos avise en determinados momentos, para poder realizar alguna tarea
 - Caso típico: retransmitir un paquete del que no tenemos confirmación
- El sistema operativo nos proporciona varios mecanismos. Veremos las llamadas al sistema:
 - `alarm()`
 - `setitimer()`

Gestión del tiempo

■ Formas de medir el tiempo

- Tiempo relativo: con respecto a un instante dado
 - 5 segundos desde ahora
 - Hace 10 minutos
- Tiempo absoluto: un instante en el tiempo
 - 10:00 del 16 de marzo de 2020
 - En Linux, se mide como el número de segundos desde las 00:00 del 01/01/1970 (epoch)

■ Medidas de tiempo

- Tiempo real (wall time)
- Tiempo de proceso: el tiempo que el proceso pasa ejecutando en el procesador

Gestión del tiempo: alarmas

- Se puede indicar al S.O. que genere una señal de alarma cuando transcurran X segundos, mediante:
unsigned int alarm(unsigned int seconds);
- Valor devuelto
 - El nº de segundos que quedaban para que una posible alarma previa se disparase
 - Cero, si no había alarma pendiente
- Cancelación de cualquier alarma: **alarm(0);**
- Antes de la llamada hay que indicar que se manejará la señal **SIGALRM**

Gestión del tiempo: alarmas

```
1 #include <signal.h>
2 #include <unistd.h>
3 #include <sys/time.h>
4
5 void manejador(int s){
6     write(1,"alarma!\n",8);
7     |    |    | alarm(3); /*para que lo haga periodicamente*/
8 }
9
10 int main(){
11
12     signal(SIGALRM,manejador);
13     alarm(3); /*alarma a los 3 segundos*/
14
15     while(1); /*bucle infinito de espera de señales*/
16     return 0;
17 }
```

Gestión de tiempo: temporizadores

- Un temporizador es similar a una alarma pero:
 - Tiene precisión de microsegundos
 - Se puede configurar para activarse periódicamente, evitando derivas de tiempo
- Llamada al sistema:
 - **int setitimer(int tipo, const struct itimerval * value, struct itimerval * oValue);**
 - Valor devuelto
 - 0: si todo ha ido bien
 - -1: si ha habido algún error al activar el temporizador
 - Argumentos
 - **Tipo:** tipo de temporizador: ITIMER_REAL, ITIMER_VIRTUAL, ITIMER_PROF
 - **Value:** estructura con la configuración temporal del temporizador

```
struct itimerval {  
    struct timeval it_interval; /* valor próximo */  
    struct timeval it_value;   /* valor actual */  
};  
struct timeval {  
    long tv_sec;           /* segundos */  
    long tv_usec;          /* microsegundos */  
};
```

- **Ovalue** (old Value): devuelve una estructura con el tiempo restante que quedase de un temporizador previo

Temporizadores a intervalos

■ TIPOS

- ITIMER_REAL:

- Mide tiempo real
 - Envía SIGALRM al proceso

- ITIMER_VIRTUAL:

- Mide sólo el tiempo cuando el proceso está ejecutando código de usuario en el espacio de usuario, pero no se decrementa cuando, por ejemplo, se hace una llamada al sistema
 - Envía SIGVTALRM al proceso

- ITIMER_PROF:

- Mide sólo tiempo cuando el proceso está ejecutando código de usuario o código del kernel en nombre el usuario, pero no cuando no se está ejecutando
 - Envía SIGPROF al proceso

■ Cancelación

- Un temporizador se cancela cuando el campo **it_value** del argumento **value** se fija a 0

Temporizadores a intervalos

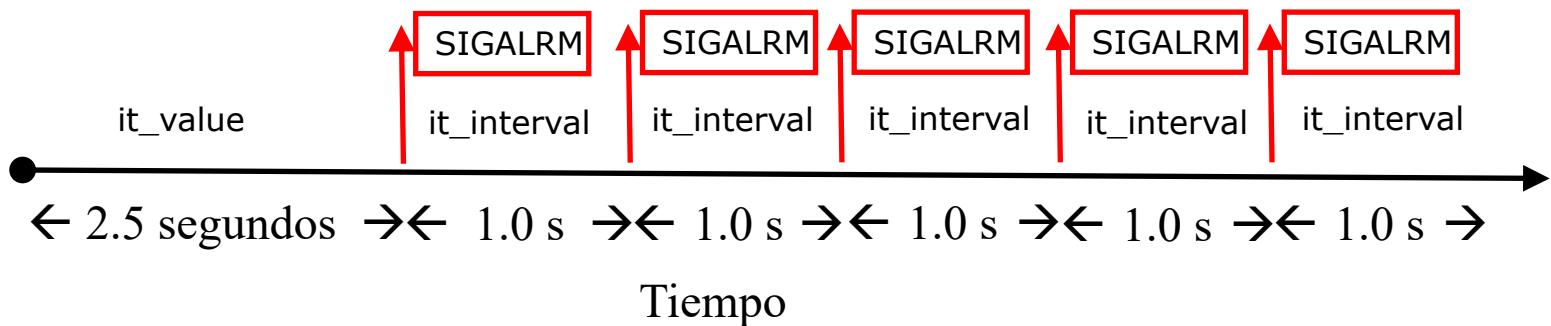
- Ejemplo sin reactivación: se activa una alarma a los 2.5 segundos

```
1 #include <sys/time.h>
2 ...
3 void manejador(int s){
4     /*hace lo que sea*/
5 }
6
7 int main(){
8     struct itimerval timer;
9     struct timeval valor;
10
11    valor.tv_sec = 2;          /*campo segundos*/
12    valor.tv_usec = 500000; /*campo microsegundos*/
13
14    timer.it_value = valor; /*campo valor del temporizador la primera vez*/
15    timer.it_interval = 0   /*campo con el valor del temporizador a partir de la
16                           primera vez y de forma periodica, si es cero, el
17                           temporizador no emitirá periódicamente más señales
18                           después de la primera*/
19    signal(SIGALRM,manejador);
20    setitimer(ITIMER_REAL, &timer, NULL);
21
22    while(1) ; /*bucle infinito de espera de señales*/
23    return 0;
24 }
```

Temporizadores a intervalos

- Ejemplo con reactivación: se activa una alarma a los 2.5 segundos inicialmente, y después cada 1.0 segundos

```
1 #include <sys/time.h>
2 ...
3 void manejador(int s){
4     /*hace lo que sea*/
5 }
6
7 int main(){
8     struct itimerval timer;
9     struct timeval inicial, intervalo;
10
11    inicial.tv_sec = 2;          /*campo segundos */
12    inicial.tv_usec = 500000;    /*campo microsegundos */
13    intervalo.tv_sec = 1;        /*campo segundos */
14    intervalo.tv_usec = 0;       /*campo microsegundos */
15
16    timer.it_value    = inicial; /* primer disparo */
17    timer.it_interval = intervalo; /* disparos sucesivos después del primero */
18
19    signal(SIGALRM,manejador);
20    setitimer(ITIMER_REAL, &timer, NULL);
21
22    while(1); /*bucle infinito de espera de señales*/
23    return 0;
24 }
```



Envíos y recepciones protegidos

- Sea la siguiente llamada:

```
r = read(fd, buf, len);
```

- Su resultado puede ser:

- Se lee todo lo esperado: **r = len**

- Resultado esperado.
 - La variable **buf** contiene los bytes leídos.

- Se lee algún byte, pero no todos: **0 < r < len**

- La variable **buf** contiene los bytes leídos
 - Escenarios:
 1. Se llega al final de fichero antes de leer **len** bytes
 2. Ha ocurrido un error en mitad de la lectura
 3. Ha llegado una señal que interrumpe la lectura
 4. Hay menos de **len** datos disponibles para lectura (pipe, socket, etc.)
 - Solución:
 - Volver a ejecutar el **read**, actualizando **buf** y **len**

- No se lee nada: **r = 0**

- Se ha llegado al final del fichero
 - La variable **buf** no se modifica

- **r = -1**, y **errno** vale **EINTR**

- Ha llegado una señal antes de poder leer nada (función reentrante)
 - La variable **buf** no se modifica
 - Solución:
 - Volver a ejecutar el **read**

Envíos y recepciones protegidos

- **readn()**
- Función propia de la asignatura
- Garantizar que se leen exactamente los datos que se solicitan
- Se recupera ante una interrupción

```
1 #include <unistd.h>
2 #include <errno.h>
3
4 ssize_t read_n(int fd, char * b, size_t n) {
5     ssize_t a_leer = n;
6     ssize_t total_leido = 0;
7     ssize_t leido;
8
9     do {
10         errno = 0;
11         leido = read(fd, b + total_leido, a_leer);
12         if (leido >= 0) {
13             total_leido += leido;
14             a_leer -= leido;
15         }
16     } while(
17         ((leido > 0) && (total_leido < n)) ||
18         (errno == EINTR));
19
20     if (total_leido > 0) {
21         return total_leido;
22     } else {
23         /* Para permitir que devuelva un posible
24          | error en la llamada a read() */
25         return leido;
26     }
27 }
```

Envíos y recepciones protegidos

- Precondición para usarla:
 - Hay que saber exactamente la cantidad de datos (bytes) a leer
- Funcionamiento
 - Línea 5: Se busca leer **n** bytes
 - Línea 11: La primera llamada a **read()** apunta al inicio de **b** (**total_leido = 0**)
 - Si no se lee todo debido a algún error en la lectura (línea 17) o por la llegada de una señal (línea 18), se vuelve a ejecutar **read()**, pero apuntando ahora a **b + total_leido**
 - Si se produce cualquier otro error, o se llega al final del fichero, devuelve lo leído
- Fuente de errores
 - Línea 10: falta inicializar **errno**
 - Líneas 17 y 18: desbalanceo de los paréntesis
- **written()** es idéntico, cambiando **read()** por **write()**
 - El copy/paste suele crear estragos

```
1 #include <unistd.h>
2 #include <errno.h>
3
4 ssize_t read_n(int fd, char * b, size_t n) {
5     ssize_t a_leer = n;
6     ssize_t total_leido = 0;
7     ssize_t leido;
8
9     do {
10         errno = 0;
11         leido = read(fd, b + total_leido, a_leer);
12         if (leido >= 0) {
13             total_leido += leido;
14             a_leer -= leido;
15         }
16     } while(
17         ((leido > 0) && (total_leido < n)) ||
18         (errno == EINTR));
19
20     if (total_leido > 0) {
21         return total_leido;
22     } else {
23         /* Para permitir que devuelva un posible
24          | error en la llamada a read() */
25         return leido;
26     }
27 }
```