Sistemas Operativos
Práctica 2

Escuela Politécnica Superior

Pareja 9
Lucía Fuentes
Mihai Blidaru

Contenidos

Ejercicio 2

Cada uno de los hijos creados solo imprime el mensaje “Soy el proceso hijo <pid>”:

```
root@soper-5929753:/home/ubuntu/workspace/practica2# ./ejercicio2
Soy el proceso hijo 3774
Soy el proceso hijo 3775
Soy el proceso hijo 3776
Soy el proceso hijo 3777
root@soper-5929753:/home/ubuntu/workspace/practica2#
```

Esto ocurre porque la acción por defecto que se toma cuando se recibe la señal SIGTERM es de terminar el proceso y por tanto nunca llega a imprimir el segundo mensaje.

Ejercicio 3

a. ¿La llamada a signal supone que se ejecute la función captura?

La llamada a *signal* no supone la ejecución de captura, solamente cambia el manejador de la señal, de forma que, si un proceso recibe la señal SIGINT, ejecuta la acción que se le indica. En este caso, cuando el proceso recibe SIGINT es cuando se ejecuta la función captura.

b. ¿Cuándo aparece el printf en pantalla?

Cuando el proceso recibe un SIGINT, una señal que se puede lanzar por teclado (C^), en vez de interrumpir la ejecución del programa se ejecuta la función de captura y aparece el *printf*.

c. ¿Qué ocurre por defecto cuando un programa recibe una señal y no la tiene capturada?

Cuando un programa recibe una señal que no tiene capturada se ejecuta la acción por defecto que tiene asignada esa señal. En la mayoría de los casos esa acción consiste en terminar el proceso, aunque hay señales que por defecto se ignoran (SIGCLD o SIGPWR) o que terminan con un core dump.

El código de un programa captura la señal SIGKILL y la función manejadora escribe “He conseguido capturar SIGKILL”. ¿Por qué nunca sale por pantalla “He conseguido capturar SIGKILL”?

SIGKILL es la señal que se envía a un proceso para que termine inmediatamente. SIGKILL junto a SIGSTOP son dos señales que no se pueden capturar, bloquear o ignorar. Por eso el programa nunca llega a imprimir por pantalla.

Ejercicio 4

Este ejercicio es un ejemplo del intercambio de señales entre un proceso padre y sus procesos hijos. Cada hijo después de imprimir un numero predeterminado de mensajes, manda una señal al padre para pedir un relevo. El padre (bloqueado esperando señales) al recibir la señal de desbloquea, y crea otro hijo. Ese nuevo hijo, manda una señal al hijo anterior para que termine. Cuando el padre ha acabado de crear todos los hijos y recibe una señal de relevo del ultimo, responde con una señal, indicando que termine. Finalmente recogemos el estado de cada hijo con wait(NULL), para que no queden procesos zombies activos.

En el siguiente ejemplo de ejecución vemos como ocurre este proceso (limitamos el número de segundos a 3 para que la salida del programa no sea excesivamente larga):

```
root@soper-5929753:/home/ubuntu/workspace/practica2# ./ejercicio4
[H]Soy 17641 y estoy trabajando
[H]Soy 17641 y estoy trabajando
[H]Soy 17641 y estoy trabajando
[H]Enviar señal de relevo
[H]Soy 17641 y estoy trabajando
[P]Señal de relevo recibida
[H]Hijo 17642 creado, terminar proceso 17641
[H]Soy 17642 y estoy trabajando
[H]Soy 17642 y estoy trabajando
[H]Soy 17642 y estoy trabajando
[H]Enviar señal de relevo
[H]Soy 17642 y estoy trabajando
[P]Señal de relevo recibida
[H]Hijo 17643 creado, terminar proceso 17642
[H]Soy 17643 y estoy trabajando
[H]Soy 17643 y estoy trabajando
[H]Soy 17643 y estoy trabajando
[H]Enviar señal de relevo
[H]Soy 17643 y estoy trabajando
[P]Señal de relevo recibida
[H]Hijo 17644 creado, terminar proceso 17643
[H]Soy 17644 y estoy trabajando
[H]Soy 17644 y estoy trabajando
[H]Soy 17644 y estoy trabajando
[H]Enviar señal de relevo
[H]Soy 17644 y estoy trabajando
[P]Señal de relevo recibida
root@soper-5929753:/home/ubuntu/workspace/practica2#
```

Podemos observar también que los procesos hijos no terminan instantáneamente, sino que todavía les queda tiempo para imprimir el mensaje “[H]Soy <pid> y estoy trabajando” una vez, dado que el tiempo que se tarda en manejar la señal, crear un nuevo hijo y que ese hijo mande la señal de terminar al hijo antiguo es mayor que el tiempo que tarda ese proceso en llamar a *printf*.

Ejercicio 6a

¿Qué sucede cuando el hijo recibe la señal de alarma?

Solo se va a ejecutar la función de alarma (imprimir “Recibo la señal de alarma”) cuando coincida que la señal SIGALRM esté desbloqueada. Supongamos que el hijo programa una alarma en el segundo 0 para el segundo 40. La señal se bloquea en el segundo 0 hasta el segundo 5, y luego se desbloquea del segundo 5 al 8, si hacemos cuentas, en el segundo 40 (cuando se recibe la señal de alarma), no podemos definir la situación del proceso: la señal de alarma llega al proceso después de 40 segundos reales, y coincide con el momento donde la señal se desbloquea por lo que no queda claro cual es el orden de los eventos. Si el proceso no tiene la señal bloqueada, el mensaje se imprimirá en el instante que reciba la señal. En caso contrario, la señal se procesará cuando se desbloque.

Este programa es bastante complejo como para observar claramente este comportamiento. Por eso hemos creado otro programa de prueba:

```
void print_time(char* msg){
    time_t      now;
    struct tm *ts;
    char        buf[80];
    now = time(0);
    ts = localtime(&now);
    strftime(buf, sizeof(buf), "%H:%M:%S", ts);
    printf("[%s]%s\n", buf, msg);}

void handler(){ print_time("ALAAAAARMAAAA");}
int main(){
    int i;
    sigset_t lock;
    sigset_t unlock;

    sigemptyset(&lock);
    sigemptyset(&unlock);

    sigaddset(&lock, SIGUSR1);
    sigaddset(&lock, SIGUSR2);
    sigaddset(&lock, SIGALRM);

    sigaddset(&unlock, SIGALRM);
    sigaddset(&unlock, SIGUSR1);

    print_time("Alarma para dentro de 20 segundos");
    signal(SIGALRM, handler);
    alarm(20);
    print_time("Bloqueando señales");
    sigprocmask(SIG_BLOCK, &lock, NULL);
    sleep(30);
    print_time("Desbloqueando señales");
    sigprocmask(SIG_UNBLOCK, &unlock, NULL);
    sleep(5);
    print_time("Saliendo");
}
```

Que produce la siguiente salida:

```
root@soper-5929753:/home/ubuntu/workspace/practica3# ./a.out
[22:33:49]Alarma para dentro de 20 segundos
[22:33:49]Bloqueando señales
[22:34:19]Desbloqueando señales
[22:34:19]ALAAAAARMAAAA
[22:34:24]Saliendo
root@soper-5929753:/home/ubuntu/workspace/practica3#
```

Como podemos ver, aunque la alarma estaba programada dentro de 20 segundos, el manejador no se ejecuta hasta que la señal no se desbloquea.

Además, si miramos la máscara de señales dentro de "`/proc/<PID>/status` ", podemos ver que la señal SIGALRM se marca como pendiente después de 20 segundos hasta que la señal se desbloquea.

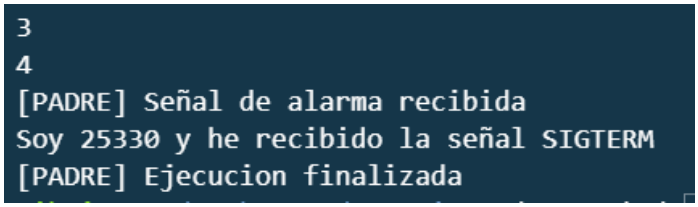
Salida del ejercicio 6a:

```
root@soper-5929753:/home/ubuntu/workspace/practica2# ./ejercicio6a
Lock
1
2
.
.
Varios segundos más tarde...
.
.
2
3
4
5
Unblock
[21328]Recibo la señal de alarma
Espera +3
Lock
```

En este caso, no podemos decir con seguridad si la señal ha llegado mientras estaba bloqueada o no. En nuestro caso, la espera de 3 segundos no se ve interrumpida por la llegada de la señal por lo que o ha llegado mientras estaba bloqueada o ha llegado antes de llegar a la instrucción sleep. En el manual de sleep se indica que, si llega una señal, la espera se interrumpe, pero en este caso no ocurre.

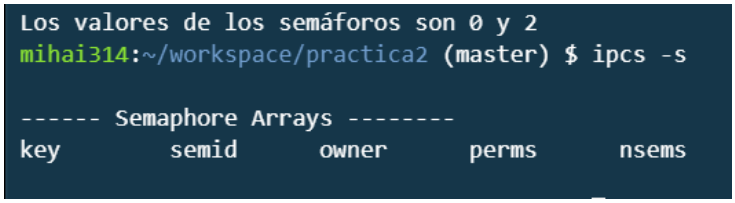
Ejercicio 6b

Es un ejercicio muy parecido al anterior en el que el padre se pone una alarma mientras el hijo trabaja. Pasados los 40 segundos de alarma el padre envía un SIGTERM al hijo. Cuando el hijo termina, el padre recoge su estado y termina también. En este caso podríamos no haber cambiado el manejador de SIGTERM en el hijo, ya que por defecto es una señal que termina un proceso, pero no podría imprimir antes el mensaje “He recibido la señal SIGTERM”.



Ejercicio 7

Comprobamos efectivamente que tras la ejecución del programa no hay ningún semáforo, todos han sido liberados.



Ejercicio 9

Durante el desarrollo de este programa hemos encontrado varios problemas:

Primero, al usar srand(time(NULL)) para definir una semilla para los números aleatorios, todos los hijos realizaban las mismas esperas, porque time(NULL) devuelve un *timestamp* en segundos. Si todos los procesos se crean en el mismo segundo, generaran la misma secuencia de números aleatorios. Para resolver este problema usamos otra fuente para la semilla de los números aleatorios: el valor de los nanosegundos devuelto por la función **clock_gettime**.

Otro de los problemas encontrados fue encolar varias señales del mismo tipo: si un proceso recibe una señal SIGUSR1 mientras se encuentra bloqueado, la señal de marca como pendiente. Al recibir una segunda señal del mismo tipo, esta segunda, es ignorada y solo se procesa la primera. Esto genera un problema, ya que con este diseño podría darse el caso donde el dinero de una caja nunca se recoge.

Para resolverlo, nuestra solución fue usar señales REAL TIME, que permiten formar una cola y además se garantiza que se van a procesar en el orden en el que se han mandado. De esta forma, un hijo, cuando supera la cantidad de 1000, manda una señal al padre, que va a ser añadida a la cola de señales. Mientras el padre no recoja el dinero de una caja, esta volverá a mandar la señal otra vez. Si el padre recibe otra señal del mismo tipo del mismo proceso, pero el dinero de la cuenta es inferior a 1000, el padre cierra ese fichero y pasa a la siguiente señal (este caso es bastante poco probable ya que la espera mínima que hace el hijo es de 1 segundo y el padre tiene suficiente tiempo para procesar las señales).

Cuando un proceso hijo termina de procesar todos los clientes, manda una señal distinta al padre para indicarlo. El padre, al manejar esta señal distinta, decrementa el numero de hijos activos, y cuando ya no queda ninguno, imprime el total recibido y el total real, libera los recursos usados y sale.