EXPLICACION PRACTICA 2 SO2 2010/2011
20031707-H JOSE MANUEL MORATO ESCANDELL

1. ENUNCIADO DE LA PRÁCTICA

Escribir un programa en C de nombre practica2.c y cuyo ejecutable se llame creasin que reciba a través de la línea de ordenes de la consola de un sistema Linux el número n de procesos hijo que debe crear (1 <= n <= 15). La forma de llamar al programa debe ser:

$ creasin n

El programa debe realizar las siguientes acciones:

A) Creación de procesos. El primer proceso o proceso patriarca (i=0) debe encargarse de crear un proceso hijo (i=1). A su vez el proceso (i=1) debe crear un proceso hijo (i=2), que será el nieto del proceso (i=0). El proceso (i=2) debe crear un proceso hijo (i=3), que será nieto del proceso (i=1) y biznieto del proceso (i=0). La creación de procesos se debe detener cuando se hayan creado un total de n+1 procesos (el proceso patriarca i=0 más sus n descendientes).

B) Creación de un primer punto de sincronización. Una vez que todos los n+1 procesos han sido creados empezarán a comunicarse entre si. Para ello es necesario introducir un primer punto de sincronización mediante el cual los procesos que se van a comunicar indican que están disponibles para iniciar la comunicación. Para esta primera sincronización cada proceso debe realizar las siguientes operaciones:

1) Esperar a que su proceso hijo le comunique que ha sido creado.
2) Transcurrido un segundo (llamada al sistema `sleep(1);`) comunicarle a su proceso padre que él ya ha sido creado.

Para realizar esta comunicación, cada proceso va a esperar la recepción de la señal SIGUSR1 y una vez recibida y se la va a enviar a su proceso padre. La señal SIGUSR1 actúa a modo de testigo único que se intercambian los distintos procesos.

C) *Comunicación y sincronismo entre procesos*. Una vez realizado el apartado b) todos los procesos están creados y listos para iniciar la comunicación, la cual va a ser iniciada por el proceso patriarca (i=0). La secuencia de sincronismo que debe realizar cada proceso es:

1) Esperar a que su proceso padre le envíe la señal SIGUSR1.
2) Una vez recibida la señal SIGUSR1 y transcurrido un segundo enviársela a su proceso hijo para pasarle el turno.

En esta secuencia hay dos excepciones:

1) El proceso patriarca (i=0) debe recibir la señal desde el proceso hijo final (i=n).
2) El último proceso hijo (i=n) no tiene descendencia por lo tanto le pasará la señal al proceso patriarca (i=0).

Se establece así una trayectoria circular en el paso de la señal que actúa de testigo, donde el último proceso creado se la pasa al primero. Este circuito se debe repetir un total de *tres veces* al final de las cuales se iniciará una secuencia de terminación.

D) *Secuencia de terminación*. La terminación se debe realizar en orden inverso al de creación. Así, el último proceso creado (i=n) debe ser el primero en terminar. La secuencia de sincronismo que se debe ejecutar es:

1) Cuando el ciclo de sincronismo del apartado c) se haya realizado tres veces, el

proceso patriarca (i=0) le enviará la señal SIGTERM a su proceso hijo (i=1).
2) Cada proceso hijo debe esperar la recepción de la señal SIGTERM procedente de su proceso padre.
3) Una vez que reciba esta señal y transcurrido un segundo, se la enviará a su proceso hijo.

Con la secuencia anterior se consigue que la orden de terminar se transmita desde el proceso patriarca (i=0) hasta el hijo final (i=n), atravesando toda la estructura de procesos creada.
Cuando el último proceso hijo recibe la señal SIGTERM, se encarga de devolvérsela a su proceso padre, iniciándose así un retorno de la señal en sentido contrario. La secuencia de sincronismo que debe ejecutar cada proceso es:

1) Esperar a recibir la señal SIGTERM.
2) Una vez recibida la señal SIGTERM y transcurrido un segundo, enviársela a su proceso padre.
3) Terminar su ejecución con una llamada a `exit`.

Hay que tener cuidado de que el proceso patriarca (i=0) no le envíe la señal SIGTERM a su proceso padre, ya que al ser éste el interprete de órdenes, el resultado que se obtienen es la terminación de la sesión de trabajo.
Una vez se haya terminado de ejecutar `creasin` el usuario debe comprobar que no existe ningún proceso de los creados vivos, es decir, se realizado la terminación de los procesos correctamente. Para ello basta teclear desde la línea de ordenes el comando `ps`.

2. EXPLICACIÓN

Las señales son interrupciones software que pueden ser enviadas a un proceso para informarle de algún evento asíncrono o situación especial.

Cada señal tiene asociado un número entero positivo y, cuando un proceso le envía una señal a otro, realmente le está enviando ese número. Afortunadamente para los programadores, cada señal tiene asignada una constante simbólica común en todas las distribuciones.

Los procesos pueden enviarse señales unos a otros y es bastante frecuente que, durante su ejecución, un proceso reciba señales procedentes del núcleo.

Para enviar una señal desde un proceso a otro o a un grupo de procesos, emplearemos la llamada a kill:

```
#include <signal.h>
int kill (pid_t pid, int sig);
```

donde pid identifica el proceso o grupo de procesos al que queremos enviarle la señal. Este parámetro es un número entero y los distintos valores que puede tomar son:

- pid > 0, es el PID del proceso al que le enviamos la señal.
- pid = 0, la señal es enviada a todos los procesos que pertenecen al mismo grupo que el proceso que la envía
- pid = -1, la señal es enviada a todos aquellos procesos cuyo identificador real es igual al identificador efectivo del proceso que la envía.
- pid < -1, la señal es enviada a todos los procesos cuyo identificador de grupo coincide con el valor absoluto de pid.

El parámetro sig es el número de la señal que queremos enviar. Si sig vale 0, se efectúa una comprobación de errores pero no se envía ninguna señal.

En todos los casos, si el identificador efectivo del proceso no es el del superusuario o si el proceso que envía la señal no tiene privilegios sobre el proceso que la va a recibir, la llamada a kill falla, devuelve -1 y en errno estará el código de error producido; en el caso de que el envío se realice

satisfactoriamente la llamada devolverá el valor 0.

Si queremos que un proceso se envíe señales a sí mismo, podemos usar la llamada a raise:

```
#include <signal.h>
void raise (int sig);
```

Cuando un proceso recibe una señal puede reaccionar de tres formas diferentes:

1. Ignorar la señal.
2. Invocar una rutina de tratamiento por defecto.
3. Invocar a una rutina propia que se encarga de tratar la señal.

Para especificar qué tratamiento debe realizar un proceso al recibir una señal, se emplea la llamada signal:

```
#include <signal.h>
void (*signal (int sig, void (*action)())) ();
```

Como vemos, signal es del tipo función que devuelve un puntero a una función void y recibe dos parámetros; el parámetro sig es el número de la señal cuya forma de tratamiento queremos especificar y el parámetro action es la acción que se tomará al recibir la señal y puede tomar tres clases de valores:

- SIG_DFL: Se ejecutará la acción por defecto asociada a la señal
- SIG_IGN: indica que la señal se debe ignorar
- dirección, es la dirección de la rutina de tratamiento de la señal suministrada por el usuario y se debe ajustar al siguiente modelo:

```
#include <signal.h>
void handler (int sig |, int code, struct sigcontext *scp|);
```

La llamada a signal devuelve el valor que tenía action, que puede servirnos para restaurarlo en cualquier instante posterior. En caso de error, signal devuelve SIG_ERR y en errno estará el código del error producido.

Con toda la teoría en mente podemos empezar a crear el código que cumpla con los requisitos especificados en el enunciado de la práctica.

Si examinamos el archivo practica2.c, en primer lugar, encontramos la inclusión de las librerías que utilizamos para el programa:

- stdio.h: donde encontramos las llamadas principales relacionadas con la entrada/salida
- stdlib.h: donde se encuentran algunas funciones comunes propuestas por el estandard ANSI; entre ellas la función exit()
- signal.h: donde se encuentran las definiciones de las constantes y rutinas relacionadas con el tratamiento de señales

A continuación, vemos la declaración de las variables globales del programa:

- int n: se utilizará para guardar el numero de procesos hijos que deben crearse y que será recibido desde en el segundo parámetro de la llamada al programa.
- int pid: se utilizará para guardar el pid del proceso. Cada proceso hijo debe guardar su pid en esta variable realizando una llamada a getpid().
- int pid_padre: se utilizará para guardar el pid del proceso padre. Cada proceso hijo debe guardar el pid de su proceso padre en esta variable realizando una llamada a getppid().
- int pid_patriarca: el proceso principal guardara su pid en esta variable que sera compartida por todos los proceso hijos creados.
- int id: se utiliza para guardar un identificador del proceso. Se inicializará a 0, por lo tanto para el proceso patriarca este valor sera su id. Cada vez que se crea un proceso hijo, se incrementará en una unidad este valor, así pues, el primer proceso hijo hereda el

valor 0 desde el proceso patriarca al crear su contexto y inmediatamente después lo incrementara en una unidad. Cuando el proceso 1 cree su hijo, el contexto del hijo se creara con el valor de id = 1 y sera aumentado en una unidad, y así sucesivamente hasta crear el número de procesos indicados por n.

- int contador: es el contador del número de veces que se a repetido la secuencia descrita en el apartado C) del enunciado de la práctica.

Después de las variables globales se puede ver el prototipo de las funciones que se utilizarán en el programa. Se han utilizado dos manejadores diferentes para las señales SIGUSR1 y SIGTERM para poder diferenciar el flujo ascendente o descendente del las señales, tal y como se sugiere en la ayuda del enunciado de la práctica.

En la próxima línea de código encontramos la función main, que sirve como punto de entrada de un programa en c y toma como argumentos:

- argc: Número de parámetros con el que se ha llamado al programa
- argv: Array de cadenas que contiene los parámetros de entrada al programa

La primera tarea al entrar en la función main es comprobar el número de argumentos con el que se ha llamado el programa. Si este numero es diferente de 2 (argc != 2) imprimimos en pantalla un mensaje de error, informamos al usuario del modo correcto de llamar al programa y salimos del programa con el código de salida -1.

En el caso de que argc sea igual a dos, argv[0] contendrá el nombre del programa y argv[1] debe contener la representación de un numero entero comprendido entre 1 y 15. Para realizar esta comprobación, transformamos y guardamos en la variable n el valor de argv[1] utilizando la función atoi. En el caso de que argv[1] no contenga una representación valida de un número entero la función devolverá 0, en otro caso devolverá el valor entero representado en argv[1]. Si el valor de n, después de la transformación, es menor que 1 o mayor que 15, se imprimirá un mensaje de error, se informará al usuario del modo correcto de utilización del programa y se saldrá del programa con el código -1.

Una vez realizada dichas comprobaciones, el programa guardara en la variable pid_patriarca su pid mediante la llamada a getpid y asignará los manejadores para las señales SIGUSR1 y SIGTERM. Si se produce un error durante la llamada a signal en cualquiera de las dos asignaciones, se imprimirá una descripción del error en pantalla y se saldrá del programa con el código -1. En el caso de que todo funcione correctamente, todos los procesos hijos, que se creen a partir de ese momento, heredarán los valores para estos manejadores y el valor de la variable pid_patriarca al crearse la copia del contexto.

Para la creación de los procesos hijos he utilizado un bucle con dos condiciones. La primera condición nos asegura que el bucle se ejecutará como máximo n veces; la segunda nos asegura que cada proceso solo ejecutará una vez el interior del bucle ya que la variable id solo se incrementará en el contexto del hijo.

Para explicar mejor el funcionamiento del bucle, vamos a suponer que se llama al programa como sigue:

```
$creasin 2
```

Al llegar al bucle, el proceso patriarca contendrá tanto en id como en la variable i el valor 0 y la variable n contendrá el valor 2. Como i < n (0 < 2) y i == id (0 == 0), entonces el proceso entrará en el interior del bucle, creando un proceso hijo y aumentando el valor de la variable i (i++). En la siguiente comprobación del bucle, en el contexto del proceso patriarca la variable i será menor que n (1 < 2) pero la variable i será mayor que id (1 > 0) ya que la variable id no ha sido nunca modificada para el proceso patriarca; por lo tanto el proceso patriarca no ejecutará de nuevo el interior del bucle y continuará con la ejecución del código (en este caso, un bucle infinito que mantiene vivos a los procesos a la espera de señales ya que la condición del if siguiente al bucle de creación no se cumplirá). Por su parte, el proceso hijo hereda los valores i = 0 y id = 0, pero la primera acción que hace al ejecutar el código del proceso hijo es aumentar en una unidad la variable id; más tarde, al igual que el proceso patriarca, aumentará el valor de i antes de comprobar de nuevo las condiciones del bucle; el contexto del hijo 1 en el momento de las comprobaciones del bucle contendrá i = 1, id = 1 y n = 2, por lo tanto, las condiciones del bucle devolverán verdadero y el hijo 1 volverá a ejecutar el cuerpo del bucle, creando

un nuevo proceso hijo. Al igual que paso con el proceso patriarca, cuando el proceso hijo 1 compruebe de nuevo las condiciones del bucle devolverán falso y por lo tanto el proceso hijo 1 entrará en el bucle infinito en espera de señales ya que tampoco cumple con la condición del siguiente bloque if (id == n). El proceso hijo 2 también aumenta el valor de la variable id y el de la variable i antes de volver a comprobar la condiciones del bucle; por lo tanto el contexto del hijo 2 en el momento de realizar las comprobaciones contendrá i == 2, id == 2 y n == 2. En este caso la segunda condición se cumple ya que id == i, pero no así la primera ya que 2 no es menor que 2 (i < n), así pues el hijo 2 sale del bucle de creación sin crear un nuevo hijo.

Si examinamos detenidamente el interior del bucle, vemos que la primera acción es crear un proceso mediante la llamada al sistema fork. Esta llamada al sistema devuelve -1 en caso de error y se imprimirá en pantalla la descripción del error producido antes de salir del proceso. En caso de que la llamada a fork se ejecute correctamente, la variable pid_hijo valdrá 0 en el contexto del proceso del hijo y el valor del pid del proceso hijo en el caso del contexto del proceso que realiza la llamada a fork.

Las acciones que se ejecutarán dentro del contexto del hijo (else if (pid_hijo == 0)) son principalmente, aumentar el valor de la variable id y guardar los valores de la variable pid y pid_padre con las llamadas al sistema getpid() y getppid() correspondientemente.

En el caso de que id sea igual a n (es el ultimo proceso hijo que se debe crear), antes de entrar en el bucle infinito, empezaremos la secuencia de sincronismo descrita en el apartado B. Para ello, lo único que tendrá que hacer es enviarse una señal SIGUSR a si mismo por medio de la llamada raise. Al recibir esta señal se ejecutará el código de la función Manejador_SIGUSR1_1 que es el que ha sido configurado por el proceso patriarca al principio del programa.

La función Manejador_SIGUSR1_1 es la que se encargará de informar a su proceso padre de que esta preparado para empezar (requisito B) y esta señal se irá propagando de hijo a padre hasta que sea recibida por el proceso patriarca. Al entrar en la función de tratamiento de señales, es buena práctica desactivar la señal que se esta tratando para que se ignoren las señales de ese tipo mientras esta dentro de la rutina de tratamiento. A continuación, esperamos un segundo y si el id es diferente de 0 (no es el proceso patriarca) enviamos la señal SIGUSR1 a su proceso padre y re-configuramos el manejador de la señal SIGUSR1 para que se ejecute la rutina Manejador_SIGUSR1_2 la próxima vez que se reciba esta señal. Si es el proceso patriarca quien recibe la señal por primera vez, entonces significa que todos los procesos hijos están preparados y el proceso patriarca se encargará de empezar la fase de sincronismo y comunicación (requisito C) al re-configurar su manejador de la señal SIGUSR1 a Manejador_SIGUSR1_2 y enviarse a si mismo la señal SIGUSR1 mediante la llamada a raise.

El manejador Manejador_SIGUSR1_2 será el encargado de cumplir los requisitos del apartado C) e iniciar la secuencia de terminación cuando se hayan cumplido por completo. Al igual que en el manejador anterior, deshabilitamos la recepción de señales SIGUSR1 mientras se esta ejecutando el manejador. Acto seguido comprobamos el estado de la variable global contador (recordamos que esta variable fue inicializada a 3 para simular tres pasos de testigo por el anillo); en el caso de que el contador sea mayor que cero, escribimos el mensaje correspondiente en pantalla dependiendo de si es el proceso patriarca (id == 0) o un hijo y pasamos la señal al siguiente proceso (el flujo normal es de padre a hijo excepto si es el último hijo que se la devolverá al patriarca). Si por el contrario el contador es igual a cero, significa que el anillo se ha completado 3 veces y por lo tanto, imprimimos los mensajes correspondientes en pantalla y iniciamos la secuencia de terminación al enviarse la señal SIGTERM mediante la llamada a raise. El manejador termina decrementando la variable contador en una unidad y reasignando de nuevo el manejador de la señal SIGUSR1 a Manejador_SIGUSR1_2.

El manejador que fue configurado por el proceso patriarca para la señal SIGTERM, y que fue heredado en el contexto de todos los hijos, fue la función Manejador_SIGTERM_1. La tarea principal de este manejador sera reenviar la señal SIGTERM a su proceso hijo y reconfigurar la señal para que la próxima que se reciba se ejecute la función Manejador_SIGTERM_2. Si el proceso es el último hijo creado (id == n), después de reconfigurar la señal SIGTERM, se volverá a enviar una señal SIGTERM mediante la llamada a raise que, cuando sea recibida, será tratada mediante la función Manejador_SIGTERM_2.

Finalmente, el manejador Manejador_SIGTERM_2 se encargará de informar por pantalla de la terminación del proceso y terminarlo literalmente mediante la llamada al sistema exit.