Comunicación Padres-Hijos con Señales

C Prácticas

Os comento un programa para comunicar padres he hijos mediante señales, utilizando señales de tiempo real y normales. Los hijos envían al padre y el padre envía ACK a los hijos. Veremos cómo instanciar el manejador para ambos, como enviar y recibir las señales.

Hay que comentar que la solución tiene un problema de sincronización, que comentaré al final y que, aunque se ha resuelto cambiando la velocidad relativa entre padres e hijos veremos (en otro artículo) que esto se puede solucionar simplemente estableciendo una barrera (con semáforos) u otra forma de sincronización.

El enunciado de este problema de ejemplo sería el siguiente: *(ATENCION – esto es un ejemplo para aprender, distinto de la práctica propuesta, del cual podéis sacar mucho código, pero no es exactamente la solución a la práctica. Analizarlo y determinar qué y cómo podéis usar lo aquí expuesto.*)

Tenemos un proceso padre que tiene que esperar a que un conjunto de procesos hijos, por ejemplo 5, le envíen una señal, una vez que el proceso padre recibe todas las señales de los hijos, entonces comienza a enviar ACKs a todos los hijos.

De entrada tenemos que controlar varias cosas en relación a las señales y sus manejadoras:

Por un lado tenemos que plantearnos cuantas señales necesitaremos. En principio puede parece que necesitaremos una señal para enviarla al padre y otra señal para enviarla al hijo.

Por otro lado, hay que instanciar, como decimos, la o las funciones manejadoras para las señales. Tenemos que considerar varias cosas, si definimos una única función manejadora o varias (una para cada señal).

También hay que plantearse dónde se deben instanciar las funciones manejadoras, antes o después del fork. Si lo hacemos antes, la o las funciones manejadoras son heredadas por los hijos. Si lo hacemos después podemos definir funciones manejadoras desconocidas para el otro (padre o hijo).

Todo esto no está exento de problemas en función de cómo lo hagamos. Supongamos que decidimos instanciarlas en el código propio del padre o del hijo, es decir, tras el fork en la rama propia de hijo o padre. En este caso tenemos que tener cuidado de que el código del padre no envíe la señal antes de que el hijo haya isntanciado su manejador.

En general hay que tener presente esta posibilidad. En este problema en concreto parece que no va a pasar puesto que si el padre espera a las señales para enviar el ACK y el hijo lo primero que hace antes de enviar la señal al padre es instanciar el manejador podemos estar tranquilos que cuando llegue el padre al punto de su código donde envía la señal, la manejadora ya estará instalada.

Para asegurarnos de que el padre o el hijo, al recibir las señales del otro tengan las manejadoras instanciadas, lo que haremos será instanciarlas antes del fork. Así nos garantizamos que están instanciadas antes de que comience la comunicación.

En cuanto a si instanciamos una o varias manejadoras; si tenemos en cuenta que hemos decidido instanciarlas antes del fork, y por tanto se van a heredar por los hijos, parece razonable una única manejadora que determine que tiene que hacer en función de la señal que llega. Esta es la técnica habitual, una manejadora única que discrimina en función de la señal a manejar, y no una manejadora para cada señal.

También hay que tener en cuenta aspectos relacionados con la evolución del padre y los hijos:

Podemos crear un bucle que use fork para crear a los hijos. Como el padre no tiene nada que hacer hasta que los hijos le envíen la señal, podemos colocar todo el código del padre después del bucle y dentro de este un if determina si es código de hijo, que hará lo que proceda y terminará con exit() sin iterar.

El enunciado comenta que el padre debe esperar a que todos los hijos envíen la señal para comenzar a enviar los ACKs. Por lo que tendremos que colgar al padre en un bucle a la espera de la llegada de todas las señales.

El hijo debe esperar al ACK del padre antes de terminar su código (o continuar si tuviera que hacer algo más). Es decir, envía la señal al padre, espera el ACK, y continúa.

Cuando el padre comienza a enviar los ACKs a los hijos, estos podrán ir terminando (pues llegan a su exit) antes que el padre termine de enviar los ACKs. Esto no ocasiona problema si pudiéramos garantizar que todos los hijos terminan antes que el padre. Pero puesto que eso no es así, tras enviar los ACKs, el padre tendrá que comenzar un bucle de espera con wait por la terminación de todos sus hijos, con el fin de no dejarlos huérfanos.

Otra cosa que hay que tener en cuenta es cómo sabemos el pid de los hijos a la hora de enviarles el ACK:

Cuando el padre tiene que enviar el ACK tiene que hacerlo con kill o sigqueue (en función del tipo de señal), pero en ambas llamadas, tiene que conocer el pid el hijo al que enviarlo. Por tanto, tenemos que almacenar de alguna forma los pids de los hijos que se van creando.

Una forma de hacerlo es, dentro del bucle donde se usa fork, determinar cuándo estamos en código padre con el if, y en ese momento almacenar en un array los pid's de los hijos que se van creando.

Otra forma sería realizar este almacenamiento del pid en el manejador de la señal que se recibe del hijo, puesto que podemos saber en ese momento quién envía la señal, con lo que se almacenan los pid's en el array conforme van enviando la señal.

En la solución que os pongo a continuación voy a usar esta segunda opción puesto que la primera es más fácil.

Veamos una solución y la comentamos un poco más poco a poco a continuación. (*RECORDAR que no se debe usar printf() en la función manejadora de la señal, aquí lo estoy haciendo por sencillez y para mostrar texto de debug, es preferible usar write para ello… pero por no liar más …*)

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>

#define NUMHIJOS 5

//Definimos una unión usada en sigqueue
union sigval value;

//fijamos el número de hijos
const int numHijos = NUMHIJOS;
int pids[NUMHIJOS]; //guarda los pid's de los hijos de los que se recibe señal

//variable contador de señales recibidas
int senyalesRecibidas;

//*****************************************************************
//FUNCIÓN que envía la señal al padre
//*****************************************************************
void EnviarSenyal(){
 value.sival_int = getpid(); //Asignamos el valor al dato
 //Enviamos una señal al padre con el pid del hijo para que me envíe el ACK
 sigqueue(getppid(), SIGRTMIN, value);  //Usamos sigqueue para enviar señales en tiempo real
}

//*****************************************************************
//FUNCIÓN que envía el ACK al hijo
//*****************************************************************
void EnviarAck(int pid){
 //Enviamos el ACK al hijo.
 kill(pid,SIGUSR1); //usamos kill para enviar señales normales
 printf("Enviado ACK a hijo %d\n",pid);
}

//*************************************************
//MANEJADORA que gestiona las señales recibidas en padre e  hijos
//*************************************************
void Handler(int signo, siginfo_t *info, void *context){
 if (signo==SIGUSR1){ //Señal recibida por el hijo
 printf ("Hijo %d recibe ACK del padre usando la señal %d.\n",getpid(),signo);
 }
 else if (signo==SIGRTMIN){ //Señal recibida por el padre
 pids[senyalesRecibidas]=info->si_pid;
 senyalesRecibidas++; // incrementamos el contador de señales recibidas
 printf ("Recibida señal %d (#%d) del hijo %d.\n",signo,senyalesRecibidas,info->si_pid);
```

```
 }
}

//FUNCION que muestra el array de pid's recibidos (simplemente con ánimo de debug)
void MostrarPids(){
 int i;
 for(i=0; i<numHijos-1; i++)
 printf("%d:",pids[i]);
 printf("%d\n",pids[numHijos-1]);
}


int main( int args, char *argv[] )
{
 int i;
 int pid;
 senyalesRecibidas=0;
 struct sigaction action;
 int status;
 char buffer[100];

 //Inicializamos el array de pids que envían
 for(i=0; i<numHijos; i++) pids[i]=0;

 //Tenemos un manejador (Handler) que atenderá a las distintas señales, el manejador se instala antes del fork

 //Instanciamos en el manejador la señal que usará el padre en la estrucutra Action y registramos el manejador de esa señal
 action.sa_sigaction = Handler;
 action.sa_flags = SA_SIGINFO;
 sigaction(SIGRTMIN, &action, NULL); //Usaremos una señal de tiempo real

 //Instanciamos en el mismo manejador la señal que usará el hijo en la estrucutra Action y registramos el manejador de esa señal
 action.sa_sigaction = Handler;
 action.sa_flags = SA_SIGINFO;
 sigaction(SIGUSR1, &action, NULL); //Usaremos una señal normal

 //---***Bucle para crear a los hijos***---
 //En el bucle los hijos envían su señal, reciben su ACK y terminan.
 for(i=0; i < numHijos; i++) { //El padre itera creando hijos
 pid=fork();
 if(pid == -1){
    sprintf(buffer,"ERROR. Fork ha fallado al crear al hijo %d\n",i);
    perror(buffer);
    exit(-1);
 }
 if(pid == 0){
 //Código hijo
    printf("Hijo %d envía señal.\n", getpid());
    EnviarSenyal();
```

```
    printf("Hijo %d esperando ACK\n", getpid());
    pause();
    printf("El hijo %d termina.\n",getpid());
    exit(0); //El hijo termina, no itera.
  }
  }
     //Aquí sólo llega el padre
  //Bucle para esperar las señales de los hijos
  while(senyalesRecibidas < numHijos){
  pause(); //Bloquea al padre hasta que llegue una señal que debe ser tratada en el manejador, o morirá.
      MostrarPids();
  }

  printf("El padre envía ACKs\n");
  sleep(1);
  //Enviamos los ACKS
  for (i=0; i<numHijos; i++)
  EnviarAck(pids[i]);

  printf("El padre entra a esperar la terminación de los hijos\n");
  //Bucle para esperar la terminación de los hijos y notificarlo
  for(i=0; i < numHijos; i++){
  pid=waitpid(-1,&status,0); //Esperamos la terminación de cualquier hijo recogiendo el pid del mismo.
  printf("El hijo con pid %d ha terminado\n",pid);
  }

  return 0;
}
```

Revisemos el código poco a poco....

Tras los includes defino el número de hijos en NUMHIJOS, por ejemplo, 5

Declaramos la unión que se usará en sigqueue para enviar la señal de tiempo real.

Declaramos una constante numero de hijos y un array para los pids de los hijos que envían su señal.

Declaramos la variable contador de señales recibidas que se usará para que el padre sepa si ya han terminado todos los hijos de enviar su señal.

A continuación definimos una serie de funciones, entre ellas la función manejadora:

EnviarSenyal() : la utilizarán los hijos para enviar la señal al padre. Como vemos usamos una señal de tiempo real que son encoladas y no se pierden aunque el padre esté dentro del manejador en el momento de la llegada de la señal. Esto es así porque hay mucha probabilidad de que todos los hijos envíen casi simultáneamente sus señales al padre y lo pueden pillar dentro del manejador. Cargamos en value.sival_int el pid del hijo en cuestión, ya que este valor le llegará al padre como parámetro de la función manejadora. El envío se realiza con sigqueue (para señales de tiempo real) cuyo primer parámetro es el pid del padre, el segundo la señal en concreto y el último la unión value que lleva datos adicionales, en este caso el pid del hijo.

EnviarAck() : La utilizará el padre para enviar el ACK a cada uno de los hijos, recibe el pid del hijo al que enviarlo. Como usamos SIGUSR1, que es una señal normal, podemos utilizar kill() para el envío de la señal.

Handler() : Es la función manejadora de ambas señales. Como vemos, con el primer parámetro determinamos de qué señal se trata y con un if hacemos una cosa u otra. Recordar que esta manejadora la vamos a instanciar en el main antes del fork, por lo que los hijos la heredarán.

Si se trata de la señal de ACK, no tenemos que hacer nada, simplemente el manejador la captura para no morir cuando llegue. Mostramos información de debug.

Si se trata de la señal que envían los hijos, almacenamos en el array de pids el pid del hijo que envía. El índice del array lo determina la variable senyalesRecibidas que cuenta cuantas vamos recibiendo. El valor del pid del hijo, que pusimos en la unión value, viene en la estructura siginfo_t, parámetro info, en el campo si_pid. Incrementamos el contador de señales recibidas y mostramos información de debug.

MostrarPids() : Una función meramente para información de debug, que muestra el array de pids tal como lo tenemos en el momento de llamarla.

Por fin llegamos a main.

Inicializamos las variables y el array de pids lo ponemos a ceros.

Instanciamos el mismo manejador para las dos señales, recordar que con sigaction decimos cuál es la función manejadora para qué señal. Ponemos la misma manejadora para ambas señales, puesto que lo hacemos antes del fork.

En el bucle para crear a los hijos con fork determinamos si ha fallado o es código del hijo. En este caso enviamos la señal al padre usando la función EnviarSenyal, esperamos con un pause() a que llegue el ACK (señal enviada por el padre) y finalmente podemos terminar.

Tras el bucle tenemos código del padre únicamente, pues el hijo ha hecho exit() en su iteración. Aquí vemos como el padre entra en bucle mientras que el contador de señales recibidas sea menor que el número de hijos. Dentro del bucle llamamos a pause(), cuando llegue una señal del hijo saldrá del pause() e imprimirá el array para que lo veamos.

Cuando todos los hijos han enviado su señal, el padre sale del bucle y entra en otro bucle para esperar la terminación de todos sus hijos. Como hemos comentado esto es necesario puesto que si el padre terminara antes sin hacerlo, los hijos quedarían huerfanos. Así, solo estarán zombies mientras el padre vaya capturando su estado de terminación con wait (revisar el post sobre zombies y huerfanos).

En la discusión he obviado deliberadamente el sleep que hay antes del bucle de envío de los ACKs puesto que es la solución a un problema difícil de ver y que comentaremos a continuación. Sin el sleep, en ocasiones el código se cuelga y el último hijo, a pesar de haber mostrado su info de debug diciendo que ha recibido su ACK, se queda colgado esperando .... y por tanto el padre también se cuelga en el bucle de terminación puesto que el hijo no termina.

Es decir, el sleep esta puesto para solucionar la siguiente cuestión. ¿Qué pasa si cuando el último hijo envía su señal, se planifica al padre, que por tanto puede salir del bucle de espera de señales, y se pone a enviar todos los ACKs, incluido el que le corresponde a este hijo, y mientras tanto el hijo se ha quedado en el punto del código antes de hacer pause()?

Pues en ese caso, el hijo recibe el ACK, y cuando le toque CPU saltará primero a su manejador, mostrará su mesaje de que ha llegado el ACK y luego continuará, entonces hace su pause(), pero el padre ya envió la señal (por eso entró el hijo en el manejador) y por tanto no vuelve a enviar la señal que necesitaría el hijo para salir del pause().

Es por ello que una posible solución es forzar al padre a esperar un tiempo para que todos los hijos lleguen a colgarse en su pause() antes de que él envíe los ACKs. Esto lo conseguimos con el que ponemos antes de este bucle.

Esto es uno de los efectos colaterales y peligrosos que tiene usar pause() para sincronizar procesos esperando la llegada de señales para hacer algo, aquí se explica este hecho y propone como solución el uso de sleep.

Tener en cuenta que en la práctica, en su primera versión, tendremos este tipo de comunicación / sincronización entre padres e hijos, es decir, el hijo envía una señal para indicar que ha apostado y el padre le envía una señal cuando ha generado la combinación ganadora. En esta primera versión hay que tener en cuenta si ponemos o no lo que explico aquí sobre el sleep que ponemos para solucionar el problema.

En la segunda versión, sin embargo, los hijos se tiran directamente a leer del pipe, y como hemos visto en el post sobre pipes, cuando se lee de un pipe que no tiene datos el proceso se bloquea. Por tanto, en esta segunda versión, el padre no necesita indicarle al hijo que la combinación ganadora está lista, puesto que estos estarán esperando pacientemente en el pipe a la llegada de la combinación. Esto hace irrelevante el uso de la señal que el padre envía a los hijos, por lo que el problema de sincronización desaparece.

← Entrada anterior

Entrada siguiente →