Modulo-II-Sesion-3.pdf

KIKONASO

Sistemas Operativos

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Módulo II. Uso de los Servicios del SO mediante la API

Sesión 3. Llamadas al sistema para el Control de Procesos

Ejercicio 1. Implementa un programa en C que tenga como argumento un número entero. Este programa debe crear un proceso hijo que se encargará de comprobar si dicho número es un número par o impar e informará al usuario con un mensaje que se enviará por la salida estándar. A su vez, el proceso padre comprobará si dicho número es divisible por 4, e informará si lo es o no usando igualmente la salida estándar.

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


void main(int argc, char * argv[]) {
    if (argc!= 2){
        printf("ERROR: Uso: %s <numero_entero>\n" , argv[0]);
        exit(-1);
    }

    int numero= atoi(argv[1]);

    pid_t pid_hijo=fork();

    if (pid_hijo < 0){
        perror("Error en el fork ");
        exit(-1);
    }

    else if(pid_hijo == 0){
        if (numero % 2 ==0 ){
            printf("El hijo dice que el número %d es par. \n", numero);
        }
        else{
            printf("El hijo dice que el número %d es impar. \n", numero);
        }
    }
    else /* Separa explícitamente lo que hace el padre y lo que hace el hijo.*/{
        wait(NULL); //Espera a que el hijo termine
        if (numero % 4 == 0 ){
            printf("El padre dice que el número %d es divisible por 4. \n", numero);
        }
        else{
            printf("El padre dice que el número %d no es divisble por 4. \n", numero);
        }
    }


}
```

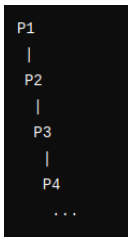
Ejercicio 2. ¿Qué hace el siguiente programa? Intenta entender lo que ocurre con las variables y sobre todo con los mensajes por pantalla cuando el núcleo tiene activado/desactivado el mecanismo de buffering.

```c
/*
tarea4.c
Trabajo con llamadas al sistema de Control de Procesos "POSIX 2.10 compliant"
Prueba el programa tal y como está. Después, elimina los comentarios (1) y
pruébalo de nuevo.
*/

#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<errno.h>

int global=6;
char buf[]="cualquier mensaje de salida\n";

int main(int argc, char *argv[])
{
int var;
pid_t pid;
var=88;
if(write(STDOUT_FILENO,buf,sizeof(buf)+1) != sizeof(buf)+1) {
    perror("\nError en write");
    exit(-1);
}

//(1)if(setvbuf(stdout,NULL,_IONBF,0)) {
//      perror("\nError en setvbuf");
//   }

printf("\nMensaje previo a la ejecución de fork");
if( (pid=fork())<0) {
    perror("\nError en el fork");
    exit(-1);
} else if(pid==0) {
    //proceso hijo ejecutando el programa
    global++;
    var++;
} else //proceso padre ejecutando el programa
    sleep(1);
    printf("\npid= %d, global= %d, var= %d\n", getpid(),global,var);
    exit(0);
}
```

En este programa lo que se hace es que se llama a un proceso hijo que modifique las variables globales, y luego el proceso padre muestra los valores de los mismos, como el proceso padre es diferente, sus variables no han sido modificadas, por lo que se muestran igual que estaban antes. Por otro lado, el setvbuf lo que hace es desactivar el buffer y directamente se imprime en la terminal, por lo que el hijo no hereda el buffer lleno y no vuelve a escribirlo seguidamente.

Ejercicio 3. Indica qué tipo de jerarquías de procesos se generan mediante la ejecución de cada uno de los siguientes fragmentos de código. Comprueba tu solución implementando un código para generar 20 procesos en cada caso, en donde cada proceso imprima su PID y el del padre, PPID.

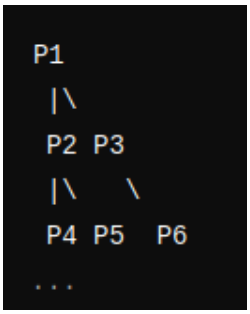
```
/*
Jerarquía de procesos tipo 1
*/

for (i=1; i < nprocs; i++) {
  if ((childpid= fork()) == -1) {
    fprintf(stderr, "Could not create child %d: %s\n",i,strerror(errno));
    exit(-1);
  }

  if (childpid)
    break;
}

/*
```



- Cada proceso que se crea (excepto el último) rompe el bucle inmediatamente después de crear su hijo, porque la condición **if (childpid)** se evalúa como verdadera en el padre (ya que **fork()** devuelve el PID del hijo en el proceso padre).

- Esto da lugar a una jerarquía lineal o en cadena, en la que cada proceso es padre del siguiente.

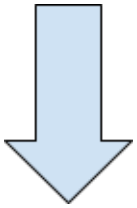
```
/*
Jerarquía de procesos tipo 2
*/

for (i=1; i < nprocs; i++) {
  if ((childpid= fork()) == -1) {
    fprintf(stderr, "Could not create child %d: %s\n",i,strerror(errno));
    exit(-1);
  }

  if (!childpid)
    break;
}
```



- En este caso, cada hijo recién creado continúa ejecutando el bucle porque **if (!childpid)** se evalúa como verdadera solo en el proceso hijo (donde **fork()** retorna 0).

- Esto da lugar a una jerarquía **en árbol**, donde cada proceso existente genera un nuevo proceso en cada iteración del bucle.

A continuación dejo el código que lo prueba:



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

void tipo1(int nprocs) {
    printf("Jerarquía de procesos tipo 1:\n");
    for (int i = 1; i < nprocs; i++) {
        pid_t childpid = fork();
        if (childpid == -1) {
            fprintf(stderr, "Could not create child %d: ", i);
            perror(""); // Imprime el mensaje de error asociado a errno
            exit(-1);
        }
        if (childpid) // Si es el padre, rompe el bucle
            break;
    }
    printf("Proceso: PID=%d, PPID=%d\n", getpid(), getppid());
    sleep(1); // Para permitir ver la salida correctamente
}

void tipo2(int nprocs) {
    printf("\nJerarquía de procesos tipo 2:\n");
    for (int i = 1; i < nprocs; i++) {
        pid_t childpid = fork();
        if (childpid == -1) {
            fprintf(stderr, "Could not create child %d: ", i);
            perror(""); // Imprime el mensaje de error asociado a errno
            exit(-1);
        }
        if (!childpid) // Si es el hijo, rompe el bucle
            break;
    }
    printf("Proceso: PID=%d, PPID=%d\n", getpid(), getppid());
    sleep(1); // Para permitir ver la salida correctamente
}

int main() {
    int nprocs = 20;

    tipo1(nprocs);
    sleep(2); // Separar claramente las salidas de las dos jerarquías
    tipo2(nprocs);

    return 0;
}
```

Consulta en el manual en línea las llamadas wait, waitpid y exit para ver sus posibilidades de sincronización entre el proceso padre y su(s) proceso(s) hijo(s):

Estas funciones permiten sincronizar la ejecución entre procesos padres e hijos en Linux:

1. **exit:**
    - Termina el proceso que la invoca y devuelve un código de salida al sistema.

- Un proceso que termina pero cuyo estado no ha sido consultado por su padre se convierte en un **zombie**. Esto ocupa recursos del sistema hasta que el proceso padre llama a alguna función de la familia wait.

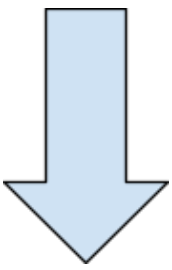
2. **wait**:
   - Permite al proceso padre bloquearse y esperar a que uno de sus hijos termine.
   - Devuelve el PID del hijo que cambió de estado. Si no hay hijos vivos, devuelve -1 y establece errno en ECHILD.
   - Si no se utiliza adecuadamente y el hijo ya ha terminado, puede quedar como un zombie.
3. **waitpid**:
   - Similar a wait, pero más flexible:
     - Permite especificar qué hijo esperar (usando su PID).
     - Ofrece opciones como WNOHANG, que no bloquea si el hijo no ha terminado, y WUNTRACED, para monitorizar hijos detenidos.
   - Devuelve el PID del hijo o 0 si se usa WNOHANG y no hay cambios de estado.

Posibilidades de sincronización:

- **Con wait o waitpid**, un proceso padre puede gestionar de manera ordenada la terminación de sus hijos:
  - Evaluar el estado de salida con macros como WIFEXITED (salida normal), WIFSIGNALED (terminado por señal) y WEXITSTATUS (código de salida).
  - Permite manejar hijos en señales con SIGCHLD, que se puede combinar con waitpid para evitar zombies automáticamente.
- **Con señales y opciones avanzadas**:
  - Configuraciones como ignorar SIGCHLD (SIG_IGN) o usar la bandera SA_NOCLDWAIT pueden prevenir zombies sin usar wait. Sin embargo, esto significa que se pierden los códigos de salida.

Ejercicio 4. Implementa un programa que lance cinco procesos hijo. Cada uno de ellos se identificará en la salida estándar, mostrando un mensaje del tipo Soy el hijo PID. El proceso padre simplemente tendrá que esperar la finalización de todos sus hijos y cada vez que detecte la finalización de uno de sus hijos escribirá en la salida estándar un mensaje del tipo:

Acaba de finalizar mi hijo con <PID>
Sólo me quedan <NUM_HIJOS> hijos vivos



1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en ing.es

Que te den **10 € para gastar** es una fantasía. ING lo hace realidad.

Abre la **Cuenta NoCuenta** con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

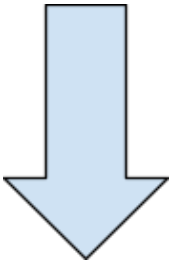
Quiero el cash

Consulta condiciones **aquí**

do your thing

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>

int main() {
    int n_hijos = 5;
    int hijos_restantes = n_hijos;
    pid_t pids[n_hijos];

    // Crear los hijos
    for (int i = 0; i < n_hijos; ++i) {
        pid_t proceso_hijo = fork();

        if (proceso_hijo == -1) {
            perror("Error al crear un hijo");
            exit(EXIT_FAILURE);
        }

        if (proceso_hijo == 0) {
            // Código del hijo
            printf("Soy el hijo %d, mi padre es %d\n \n", getpid(), getppid());
            exit(0); // El hijo termina aquí
        }

        // El padre guarda el PID del hijo
        pids[i] = proceso_hijo;
    }

    // El padre espera a sus hijos
    while (hijos_restantes > 0) {
        int status;
        pid_t hijo_terminado = wait(&status);

        if (hijo_terminado > 0) {
            printf("Acaba de finalizar mi hijo con PID %d\n \n", hijo_terminado);
            hijos_restantes--;
            printf("Sólo me quedan %d hijos vivos\n \n", hijos_restantes);
        } else if (hijo_terminado == -1) {
            if (errno == ECHILD) {
                // No hay más hijos que esperar
                break;
            } else {
                perror("Error al esperar por un hijo");
                exit(EXIT_FAILURE);
            }
        }
    }

    printf("Todos los hijos han terminado, el padre finaliza.\n \n");
    return 0;
}
```

Ejercicio 5. Implementa una modificación sobre el anterior programa en la que el proceso padre espera primero a los hijos creados en orden impar (1º,3º,5º) y después a los hijos pares (2º y 4º).



```c
int main() {
    int n_hijos = 5;
    int hijos_restantes = n_hijos;
    pid_t pids[n_hijos];

    // Crear los hijos
    for (int i = 0; i < n_hijos; ++i) {
        pid_t proceso_hijo = fork();

        if (proceso_hijo == -1) {
            perror("Error al crear un hijo");
            exit(EXIT_FAILURE);
        }

        if (proceso_hijo == 0) {
            // Código del hijo
            printf("Soy el hijo %d, mi padre es %d\n \n", getpid(), getppid());
            exit(0); // El hijo termina aquí
        }

        // El padre guarda el PID del hijo
        pids[i] = proceso_hijo;
    }

    //Esperamos a los hijos impares primero
    printf ("Esperando a los hijos impares... \n");
    for(int i = 0; i < n_hijos; ++i){
        if((i+1) % 2 == 1){
            int status;
            pid_t terminado = waitpid(pids[i], &status, 0);
            if (terminado > 0){
                printf("Acaba de finalizar mi hijo con PID %d . \n", terminado);
            }
        }
    }

    //Esperamos a los hijos pares
    printf ("Esperando a los hijos pares... \n");
    for(int i = 0; i < n_hijos; ++i){
        if((i+1) % 2 == 0){
            int status;
            pid_t terminado = waitpid(pids[i], &status, 0);
            if (terminado > 0){
                printf("Acaba de finalizar mi hijo con PID %d . \n", terminado);
            }
        }
    }



    printf("Todos los hijos han terminado, el padre finaliza.\n \n");
    return 0;
}
```

Ejercicio 6. ¿Qué hace el siguiente programa?

```
/*
tarea5.c
Trabajo con llamadas al sistema del Subsistema de Procesos conforme a POSIX 2.10
*/
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>
#include<errno.h>

int main(int argc, char *argv[]){

pid_t pid;
int estado;
if( (pid=fork())<0) {
    perror("\nError en el fork");
    exit(-1);
}
else if(pid==0) { //proceso hijo ejecutando el programa
    if( (execl("/usr/bin/ldd","ldd","./tarea5",NULL)<0)) {
        perror("\nError en el execl");
        exit(-1);
    }
}
wait(&estado);
/*
<estado> mantiene información codificada a nivel de bit sobre el motivo de
finalización del proceso hijo que puede ser el número de señal o 0 si alcanzó su
finalización normalmente.
Mediante la variable estado de wait(), el proceso padre recupera el valor
especificado por el proceso hijo como argumento de la llamada exit(), pero
desplazado 1 byte porque el sistema incluye en el byte menos significativo el
código de la señal que puede estar asociada a la terminación del hijo. Por eso se
utiliza estado>>8 de forma que obtenemos el valor del argumento del exit() del
hijo.
*/

printf("\nMi hijo %d ha finalizado con el estado %d\n",pid,estado>>8);
exit(0);

}
```

1. **Creación de un proceso hijo:**
    - El programa comienza llamando a fork(). Esto crea un nuevo proceso. Si fork() falla, se imprime un mensaje de error y se termina el programa.
    - Si fork() tiene éxito, el proceso se divide en dos: un proceso padre (que recibe el PID del hijo) y un proceso hijo (que tiene un PID de 0).
2. **En el proceso hijo:**
    - El hijo ejecuta la llamada execl() para reemplazar su imagen de proceso actual con el programa ldd, que se usa para mostrar las bibliotecas compartidas de un ejecutable.
    - El programa hijo intenta ejecutar el comando ldd ./tarea5, que verifica las dependencias de la biblioteca compartida del programa tarea5. Si execl() falla, se imprime un mensaje de error y el proceso hijo termina.
3. **En el proceso padre:**
    - El padre espera a que el proceso hijo termine utilizando la llamada wait(&estado). Esto permite que el padre recupere el estado de finalización del hijo.
    - El estado de finalización del hijo es codificado en estado, donde el código de salida del hijo se encuentra desplazado 8 bits (debido a la forma en que el sistema operativo

codifica la información de la señal que podría haber causado la finalización del hijo). El valor del código de salida del hijo se obtiene haciendo un desplazamiento a la derecha (estado >> 8).

- Finalmente, el padre imprime un mensaje indicando que el hijo ha finalizado, mostrando su PID y el estado de finalización.

Flujo del programa:

- El padre crea un hijo con fork().
- El hijo ejecuta execl() para ejecutar ldd ./tarea5.
- El padre espera a que el hijo termine con wait() y luego muestra el estado de finalización del hijo.

Ejercicio 7. Escribe un programa que acepte como argumentos el nombre de un programa, sus argumentos si los tiene, y opcionalmente la cadena "bg". Nuestro programa deberá ejecutar el programa pasado como primer argumento en foreground si no se especifica la cadena "bg" y en background en caso contrario. Si el programa tiene argumentos hay que ejecutarlo con ésto.

```c
int main(int argc, char *argv[]){

    if(argc < 2){
        perror("ERROR: Debes pasar al menos un programa como argumento. \n");
        exit(-1);
    }

    char *programa = argv[1];

    int bg=0;

    if (argc > 2 && strcmp(argv[argc - 1], "bg" ) == 0){
        bg = 1;
        argc--;
    }

    char **argumentos = argv + 1 ; // Los argumentos para el programa (excluyendo el nombre del programa)

    pid_t pid = fork();

    if (pid < 0 ){
        perror("ERROR: No se pudo crear un proceso hijo. \n");
        exit(-1);
    }

    if (pid == 0){  //Proceso hijo
        // Si es en background no hace falta que esperemos

        if(bg == 1){
            freopen("/dev/null", "a", stdout);  // Redirige stdout a /dev/null
            freopen("/dev/null", "a", stderr);  // Redirige stderr a /dev/null

            if(execv(programa, argumentos) < 0){
            perror("ERROR: No se pudo ejecutar el programa. \n");
            exit(-1);
            }
        } else{
            if(execv(programa, argumentos) < 0){
            perror("ERROR: No se pudo ejecutar el programa. \n");
            exit(-1);
            }
        }

    }

    else{

        if (bg == 1){
            printf("Ejecutando en background... \n");
        } else{
            //Si se hace en la terminal, esperamos a que el hijo termine
            wait(NULL);
            printf("\n El programa ha terminado. \n");
        }
    }

    return 0;
}
```

LA LLAMADA A CLONE NO ES IMPORTANTE