Sistemas Operativos
Tercer curso GITT
Departamento de Ingeniería Telemática
Escuela Superior de Ingenieros
Universidad de Sevilla

PRÁCTICA 1. PROCESOS

1. OBJETIVO

El objetivo es entender las llamadas al sistema operativo para la identificación y ejecución de procesos. Para ello se van a estudiar algunas de las llamadas de la familia **exec**, así como la llamada a **fork** y otras llamadas al sistema relacionadas con los procesos.

2. DESCRIPCIÓN

Identificación de procesos:

Cada proceso se identifica mediante un número entero único denominado *identificador de proceso* (de tipo pid_t). Los servicios relativos a la identificación de procesos son:

pid_t **getpid**(void);

Devuelve el identificador del proceso que realiza la llamada.

pid_t **getppid**(void);

Devuelve el identificador del proceso padre.

⇒Ejemplo:

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

int main(void)
{
   pid_t id_proceso;
   pid_t id_padre;

   id_proceso = getpid();
   id_padre = getppid();

   printf("Identificador de proceso: %d\n", id_proceso);
   printf("Identificador del proceso padre: %d\n", id_padre);

   return 0;
}
```

p1_prueba1.c

Además de la identificación del proceso, un proceso tiene asociado un *identificador de usuario real* (identificador de usuario del propietario), un *identificador de usuario efectivo* (que determina los privilegios que un proceso tiene cuando se ejecuta), un *identificador de grupo* (al que pertenece el usuario) y un *identificador de grupo efectivo*.

Los servicios relacionados para obtener estos identificadores son:

uid_t **getuid**(void);

Devuelve el identificador de usuario real del proceso que realiza la llamada.

uid_t **geteuid**(void);

Devuelve el identificador de usuario efectivo del proceso que realiza la llamada.

gid_t **getgid**(void);

Devuelve el identificador del grupo del proceso que realiza la llamada.

gid_t **getegid**(void);

Devuelve el identificador del grupo efectivo del proceso que realiza la llamada.

⇒ Ejemplo:

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
   printf("Identificador de usuario: %d\n", getuid());
   printf("Identificador de usuario efectivo: %d\n",  geteuid());
   printf("Identificador de grupo: %d\n", getgid());
   printf("Identificador de grupo efectivo: %d\n", getegid());

   return 0;
}
```

p1_prueba2.c

Creación de procesos:

Para la creación de procesos se utiliza el siguiente servicio:

pid_t **fork**(void);

Devuelve la identificación del proceso creado en el padre y 0 en el proceso hijo.

La llamada a fork lo que hace es crear una copia del proceso que ha realizado la llamada. Se puede decir que se realiza una clonación del proceso. El proceso que hace la llamada a fork se convierte en el proceso padre del proceso creado. Una vez realizada la copia, tanto padre e hijo

continúan de forma independiente la ejecución en el mismo punto del programa, es decir, en la siguiente instrucción al fork. Es un error pensar que el hijo comienza la ejecución por el principio del programa. Esto es así porque el proceso hijo hereda del padre los datos y la pila que tuviera en el momento de la ejecución del fork, así como el valor de los registros.

⇒Ejemplo:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void)
{
   int estado;

   pid_t pid;
   pid = fork();

   switch(pid)
   {
      case -1:    /* error del fork() */
         perror("fork");
         break;
      case 0:    /* proceso hijo */
         printf("Proceso %d; padre = %d \n", getpid(), getppid());
         break;
      default:   /* padre */
         printf("Proceso %d; padre = %d \n", getpid(), getppid());
         wait(&estado);

   }

   return 0;
}
```

p1_prueba3.c

Se ha utilizado la función perror para imprimir un mensaje de error por la salida estándar de errores.

Ejecución de un programa:

El servicio exec tiene por objetivo cambiar el programa que está ejecutando un proceso. No es como el servicio fork ya que exec no crea un nuevo proceso, sino que permite que un proceso pase a ejecutar un programa diferente. Recuerde que fork crea un nuevo proceso que ejecuta el mismo programa que el proceso padre.

Algunas de las funciones exec son las siguientes:

int **execlp**(char *file, const char *arg, …);
int **execvp**(char *file, char **argv);

Estas funciones reemplazan la imagen del proceso actual por una nueva imagen. Esta imagen se construye a partir de un archivo ejecutable (file). Si la llamada se ejecuta con éxito, no devolverá ningún valor puesto que la imagen del proceso habrá sido reemplazada, es decir, de una llamada con éxito no hay retorno, en caso contrario devuelve –1.

El argumento arg es la dirección del primer elemento de una cadena de caracteres que contiene el primer argumento que se le pasa al programa. Los puntos suspensivos indican que hay que poner también los demás argumentos de la misma forma.
El argumento argv es la dirección de comienzo de un vector de cadenas de caracteres que contiene los argumentos pasados al programa y debería acabar con un NULL.
La diferencia entre execlp y execvp es únicamente la forma de dar los argumentos del programa.

La función main del nuevo programa llamado tendrá la forma:

int main(int argc, char **argv)

donde argc representa el número de argumentos que se pasan al programa, incluido el propio nombre del programa, y argv es un vector de cadenas de caracteres, conteniendo cada elemento de este vector un argumento pasado al programa. El primer componente de este vector (argv[0]) representa el nombre del programa.

⇒ Ejemplo de execlp:

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

int main(void)
{
   pid_t pid;

   pid = fork();
   switch(pid)
   {
      case -1:      /* error del fork() */
         perror("fork");
         break;

      case 0:     /* proceso hijo */
         execlp("ls","ls","-l",NULL);
         perror("exec");
         break;
      default:    /* padre */
         printf("Proceso padre\n");
    }

    // getchar();

    return 0;
}
```

p1_prueba4.c

⇒ Ejemplo de execvp:

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

int main(int argc)
{
   pid_t pid;
   char *argumentos[3];
```

```
   argumentos[0] = "ls";
   argumentos[1] = "-l";
   argumentos[2] = NULL;

   pid = fork();
   switch(pid)
   {
      case -1:    /* error del fork() */
         perror("fork");
         break;
      case 0:    /* proceso hijo */
         execvp(argumentos[0], argumentos);
         perror("exec");
         break;
      default:   /* padre */
         printf("Proceso padre\n");
   }

   return 0;
}
```

p1_prueba5.c

⇒ Otro ejemplo de execvp:

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
   pid_t pid;

   pid = fork();

   switch(pid)
   {
      case -1:   /* error del fork() */
         perror("fork");
         break;
      case 0:    /* proceso hijo */
         if (execvp(argv[1], &argv[1])< 0)
            perror("exec");
         break;
      default:   /* padre */
         printf("Proceso padre\n");
   }

   return 0;
}
```

p1_prueba6.c

Espera por la finalización de un proceso:

El servicio wait permite a un proceso padre esperar hasta que termine la ejecución de un proceso hijo. El proceso padre se queda bloqueado hasta que termina un proceso hijo. La forma de wait es la siguiente:

Pid_t **wait**(int *status);

Esta llamada espera la finalización de un proceso hijo y permite obtener información sobre el estado de terminación del mismo. Devuelve el identificador del proceso hijo cuya ejecución ha finalizado. Si status es distindo de NULL, entonces se almacena en esta variable información relativa al proceso que ha terminado. Se puede usar esta variable para ver como ha terminado el proceso utilizando macros. Algunas de estas macros son las siguientes:

- WIFEXITED(status): devuelve un valor verdadero (distinto de cero) si el hijo terminó normalmente.
- WEXITSTATUS(status): permite obtener el valor devuelto por el proceso hijo. Sólo puede ser utilizada cuando WIFEXITED devuelve un valor verdadero.
- WIFSIGNALED(status): devuelve un valor verdadero si el proceso hijo finalizó su ejecución como consecuencia de la recepción de una señal para la cual no se había programado manejador.
- WTERMSIG(status): devuelve el número de la señal que provocó la finalización del proceso hijo. Sólo puede ser utilizada si WIFSIGNALED devuelve un valor verdadero.

⇒ Ejemplo:

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

int main(int argc, char **argv)
{
   pid_t pid;
   int valor;

   pid = fork();

   switch(pid)
   {
      case -1:    /* error del fork() */
         perror("fork");
         break;
      case 0:    /* proceso hijo */
         if(argc>1)
            if (execvp(argv[1], &argv[1]) < 0)
               perror("exec");
         break;
      default:   /* padre */

         while (wait(&valor) != pid);
         if (valor == 0)
            printf("El mandato se ejecuto de forma normal\n");
         else
         {
```

```
            if (WIFEXITED(valor))
               printf("El hijo termino normalmente
                       y su valor devuelto fue %d\n", WEXITSTATUS(valor));
            if (WIFSIGNALED(valor))
               printf("El hijo termino al recibir
                       la señal %d\n", WTERMSIG(valor));
         }
   }
   return 0;
}
```

p1_prueba7.c

TAREAS:

1. Descarga el fichero procesos.zip correspondiente a la primera práctica.
2. Compila el programa `p1_prueba1.c` mediante la siguiente instrucción:

```
gcc -Wall -W  p1_prueba1.c  -o p1
```

3. Ejecuta varias veces el programa generado.
4. ¿Qué proceso es el padre del proceso p1? Se puede ver con la orden `ps`
5. Compila y ejecuta p1_prueba2.c. Indica el identificador de usuario real y efectivo del proceso y también el identificador de grupo real y efectivo.
6. Compila y ejecuta p1_prueba3.c. Indica de las líneas de salida cuál corresponde al padre y cuál corresponde al hijo.
7. Observa que se ha puesto wait en el padre, esto es para que el proceso padre espere a que el hijo termine antes de terminar. Indica el resultado de la ejecución si comentas el wait.
8. En p1_prueba3.c crea una variable local a main llamada a, iníciala a 0. Haz varias pruebas imprimiendo el valor de la variable a antes del fork, en el hijo, en el padre y después del switch. Prueba también a modificar su valor en el hijo y en el padre.
9. Recuerda como se utilizan los argumentos en la línea de órdenes en C compilando y ejecutando el programa argumentos.c proporcionado.
10. Compila y ejecuta p1_prueba4.c. Indica cuál es el resultado de la ejecución.
11. Quita el comentario en p1_prueba4.c de la instrucción getchar para que pida un carácter antes de terminar. Ejecuta ps –a para ver todos los procesos.
12. Cambia en p1_prueba4.c la instrucción execlp por esta otra y observa el resultado:
    ```
    execlp("ls","ls","-j",NULL);
    ```
    Comenta el resultado.
13. Cambia en p1_prueba4.c la instrucción execlp por esta otra y observa el resultado:
    ```
    execlp("ls","gg","-j",NULL);
    ```
    Comenta el resultado.
14. Compila y ejecuta p1_prueba5.c. Indica cuál es el resultado.
15. Compila p1_prueba6.c y ejecútalo para que el resultado sea similar al resultado de p1_prueba5.c.
16. Compila p1_prueba7.c y ejecútalo con los siguientes argumentos:
    p1_prueba1
    sin argumentos
    kk
    ls –j
    Para probar la terminación con una señal : ls –lR /
    y a continuación: Crt-z / ps / kill -9 idProcesoHijo / jobs / fg idTrabajoPadre