s

Instituto Politécnico Nacional
Escuela Superior de Computo

Sistemas operativos (2CM9)

Maestra Ana Belem Juárez Méndez

Practica 2: Procesos

Alumno:

García González Aarón Antonio

Septiembre 05, 2019

Índice

Objetivo

Aplicar los conocimientos sobre la programación de procesos en situaciones que se pueden presentar en el Sistema Operativo UNIX- LINUX.

Introducción

Función perror

La función perror transforma el número de error en la expresión entera de errno a un mensaje de error. Escribe una secuencia de caracteres al stream estándar de errores, esto es: primero (si cadena no es un puntero nulo y el carácter apuntado por cadena no es el carácter nulo), la cadena apuntada por cadena seguido de dos puntos (:) y un espacio; entonces un mensaje de errores apropiado seguido por un carácter de línea nueva. El contenido de los mensajes de errores es el mismo que aquello retornado por la función strerror con el argumento errno, que están definidos según la implementación.

Funciones exec

La familia de funciones exec reemplaza el proceso actual en ejecución con un nuevo proceso. Se puede usar para ejecutar un programa en C usando otro programa en C. Viene bajo el archivo de encabezado unistd.h. Hay muchos miembros en la familia exec que se muestran a continuación.

Execvp: Con este comando, el proceso hijo creado no tiene que ejecutar el mismo programa que el proceso padre. Las llamadas al sistema de tipo exec permiten que un proceso ejecute cualquier archivo de programa, que incluye un ejecutable binario o un script de shell. Sintaxis:

```
int execvp (const char *file, char *const argv[]);
```

Execv: Esto es muy similar a la función execvp () en términos de sintaxis también. La sintaxis de execv () es la siguiente: Sintaxis:

```
int execv(const char *path, char *const argv[]);
```

execlp and execl: Estos dos también tienen el mismo propósito, pero la sintaxis de ellos es un poco diferente, como se muestra a continuación: Sintaxis:

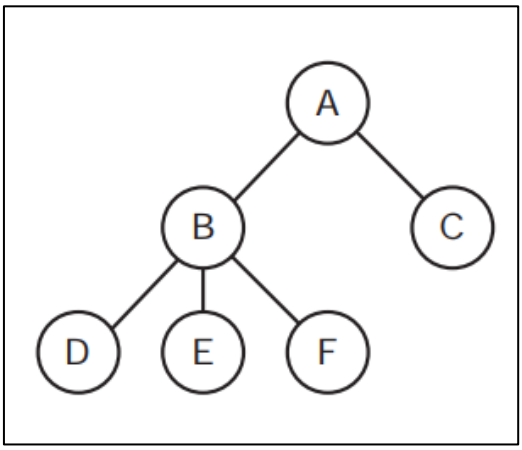
```
int execlp(const char *file, const char *arg,.../* (char  *) N
int execl(const char *path, const char *arg,.../* (char  *) NU
```

execvpe and execle: Estos dos también tienen el mismo propósito, pero su sintaxis es un poco diferente de todos los miembros anteriores de la familia exec. Los sintaxis de ambos se muestran a continuación:

```
int execvpe(const char *file, char *const argv[],char *const e

Syntax:
int execle(const char *path, const char *arg, .../*, (char *)
char * const envp[] */);
```

Un concepto clave en todos los sistemas operativos es el **proceso**. Un proceso es en esencia un programa en ejecución. Cada proceso tiene asociado un espacio de direcciones, una lista de ubicaciones de memoria que va desde algún mínimo (generalmente 0) hasta cierto valor máximo, donde el proceso puede leer y escribir información. El espacio de direcciones contiene el programa ejecutable, los datos del programa y su pila. También hay asociado a cada proceso un conjunto de recursos, que comúnmente incluye registros (el contador de programa y el apuntador de pila, entre ellos), una lista de archivos abiertos, alarmas pendientes, listas de procesos relacionados y toda la demás información necesaria para ejecutar el programa.



Si un proceso puede crear uno o más procesos aparte (conocidos como procesos hijos) y estos procesos a su vez pueden crear procesos hijos, llegamos rápidamente la estructura de árbol de procesos de la figura de la derecha. Los procesos relacionados que cooperan para realizar un cierto trabajo a menudo necesitan comunicarse entre sí y sincronizar sus actividades. A esta comunicación se le conoce como comunicación entre procesos.

En este modelo, todo el software ejecutable en la computadora, que algunas veces incluye al sistema operativo, se organiza en varios procesos secuenciales (procesos, para abreviar). Un proceso no es más que una instancia de un programa en ejecución, incluyendo los valores actuales del contador de programa, los registros y las variables. En concepto, cada proceso tiene su propia CPU virtual; en la realidad, la CPU real conmuta de un proceso a otro, pero para entender el sistema es mucho más fácil pensar en una colección de procesos que se ejecutan en (pseudo) paralelo, en vez de tratar de llevar la cuenta de cómo la CPU conmuta de programa en programa. Esta conmutación rápida de un proceso a otro se conoce como **multiprogramación**.

Una situación muy habitual dentro de un programa es la de **crear un nuevo proceso** que se encargue de una tarea concreta, descargando al proceso principal de tareas secundarias que pueden realizarse asíncronamente o en paralelo. Linux ofrece varias funciones para realizar esto: system(), fork() y exec().

La función **system()** bloquea el programa hasta que retorna, y además tiene problemas de seguridad implícitos, por lo que desaconsejo su uso más allá de programas simples y sin importancia.

La segunda manera de crear nuevos procesos es mediante **fork().** Esta función crea un proceso nuevo o "proceso hijo" que es exactamente igual que el "proceso padre". Si fork() se ejecuta con éxito devuelve:

- ⇒ Al padre: el PID del proceso hijo creado.
- ⇒ Al hijo: el valor 0.

Wait suspende la ejecución del proceso que la invoca hasta que alguno de sus procesos hijo termina. Wait regresa el pid del proceso hijo que halla terminado su ejecución y en stat_loc se almacena el valor que el proceso hijo le envía al proceso padre. La sintaxis de wait es: pid_t wait(int *stat_loc);

Exit termina la ejecución de un proceso y le devuelve el valor de status al sistema. La sintaxis de exit es: void exit(int status);

Desarrollo

Ejercicio 1

Realizar un programa que cree diez procesos hijos del mismo padre y que cada uno muestre el mensaje “Hola soy el proceso hijo N y mi pid es XXXX. El pid de mi padre es XXXX”. N es el conteo del uno al diez. El padre deberá esperar a sus hijos antes de terminar.

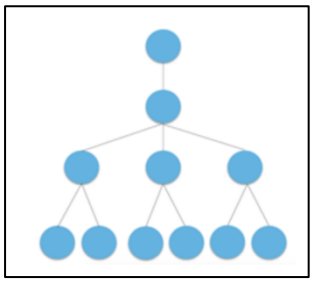
```c
int main(int argc, char const *argv[]){
    pid_t pid;
    int x;

    for(x=1;x<=10;x++){
        pid=fork();
        if(pid){
            // Instrucciones del proceso padre
            sleep(1);
        } else{
            printf("soy el hijo %d y mi PID es %d, el PID de mi padre es %d\n", x,getpid(),
getppid());
            sleep(2);
            exit(0);
        }
    }
    return 0;
}
```

```
MacBook-Pro-de-Aaron:p2 aarongarcia$ gcc index.c
MacBook-Pro-de-Aaron:p2 aarongarcia$ ./a.out
soy el hijo 1 y mi PID es 26629, el PID de mi padre es 26628
soy el hijo 2 y mi PID es 26630, el PID de mi padre es 26628
soy el hijo 3 y mi PID es 26631, el PID de mi padre es 26628
soy el hijo 4 y mi PID es 26632, el PID de mi padre es 26628
soy el hijo 5 y mi PID es 26634, el PID de mi padre es 26628
soy el hijo 6 y mi PID es 26635, el PID de mi padre es 26628
soy el hijo 7 y mi PID es 26636, el PID de mi padre es 26628
soy el hijo 8 y mi PID es 26637, el PID de mi padre es 26628
soy el hijo 9 y mi PID es 26638, el PID de mi padre es 26628
soy el hijo 10 y mi PID es 26639, el PID de mi padre es 26628
```

Ejercicio 2

Realizar un programa que cree un proceso hijo a partir de un proceso padre, el hijo creado a su vez creará tres procesos hijos más. A su vez cada uno de los tres procesos creará dos procesos más. Cada uno de los procesos creados imprimirá en pantalla el PID de su padre y su propio PID. El árbol de procesos de este programa se verá como la imagen de la derecha.
Donde el primer proceso lo nominaremos como generacion 0, hasta la ultima generacion que la nombraremos como generacion 3.



```c
int main(int argc, char const *argv[]){
    int pid,pidh;

    pid=fork(); // Primera generacion
    if(pid==-1){            //error en la llamada a fork
        perror("\nError en fork");
        exit(-1);
    }else if(pid==0){      //instrucciones del hijo
        printf("GEN_1 > Mi PID es %d y el PID de mi padre es %d\n", getpid(), getppid());
        for(int i=1; i<=3; i++){ // Generacion 2
            pid = fork();
            if(pid == -1){ // error en la llamada a fork
                perror("\nError en fork");
                exit(-1);
            }else if(pid == 0){ //instrucciones del hijo
                printf("GEN_2 > Mi PID es %d y el PID de mi padre es %d\n", getpid(), getppid());
                for(int j=0; j<2; j++){
                    pid = fork();
                    if(pid == -1){ // error en la llamada a fork
                        perror("\nError en fork");
                        exit(-1);
                    }else if(pid == 0){ //instrucciones del hijo
                        printf("GEN_3 > Mi PID es %d y el PID de mi padre es %d\n", getpid(),
getppid());
                        exit(0);
                    }else{ //instrucciones del padre
                        pidh=wait(NULL);
                    }
                }
                exit(0);
            }else{ //instrucciones del padre
                sleep(5);
                pidh=wait(NULL);
            }
        }
        exit(0);
    }else{      //instrucciones del padre
        printf("GEN_0 > Mi PID es %d\n", getpid()); // Generacion cero
```

```
        pidh=wait(NULL);
    }
    return 0;
}
```

```
[MacBook-Pro-de-Aaron:p2 aarongarcia$ gcc p2_2.c
[MacBook-Pro-de-Aaron:p2 aarongarcia$ ./a.out
 GEN_0 > Mi PID es 29885
 GEN_1 > Mi PID es 29886 y el PID de mi padre es 29885
 GEN_2 > Mi PID es 29887 y el PID de mi padre es 29886
 GEN_3 > Mi PID es 29888 y el PID de mi padre es 29887
 GEN_3 > Mi PID es 29889 y el PID de mi padre es 29887
 GEN_2 > Mi PID es 29890 y el PID de mi padre es 29886
 GEN_3 > Mi PID es 29891 y el PID de mi padre es 29890
 GEN_3 > Mi PID es 29892 y el PID de mi padre es 29890
 GEN_2 > Mi PID es 29893 y el PID de mi padre es 29886
 GEN_3 > Mi PID es 29894 y el PID de mi padre es 29893
 GEN_3 > Mi PID es 29895 y el PID de mi padre es 29893
```

Ejercicio 3

Realizar un programa que invoque a la orden ls -al /usr/bin con las funciones exec tal como se indica:
$./programa opción
Donde opción, puede ser:

- ⇒ -l, invocar a la orden ls con la función execl.
- ⇒ -v, invocar a la orden ls con la función execv.
- ⇒ -lp, invocar a la orden ls con la función execlp.

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

int main(int argc, char const *argv[]){
    if(strcmp(argv[1],"-l")==0){ // Opcion -l
        printf("Argumento -l\n");
        int ret = execl ("/bin/ls", "ls", "-la", "/usr/bin", (char *)0);
    }else if(strcmp(argv[1],"-v")==0){ // Opcion -v
```

```c
        printf("Argumento -v\n");
        char* arr[] = {"ls","-la","/usr/bin", NULL};
        execv("/bin/ls", arr);
    }else if(strcmp(argv[1],"-lp")==0){ // Opcion -lp
        printf("Argumento -lp\n");
        //int ret = execlp ("ls", "ls", "-la", "/usr/bin", (char *)0);
    }else{ //Error
        printf("Error, argumento no valido\n");
    }
    return 0;
}
```

```
[MacBook-Pro-de-Aaron:p2 aarongarcia$ gcc p3.c
[MacBook-Pro-de-Aaron:p2 aarongarcia$ ./a.out -l
Argumento -l
total 104576
drwxr-xr-x  971 root   wheel      31072 Aug 27 01:21 .
drwxr-xr-x@   9 root   wheel        288 Jul 29 15:47 ..
-rwxr-xr-x    4 root   wheel        925 Feb 22  2019 2to3-
lrwxr-xr-x    1 root   wheel         74 Aug  8 09:34 2to3-2.7 -> ../../System/L
-rwxr-xr-x    1 root   wheel      55152 Jul 29 15:59 AssetCacheLocatorUtil
-rwxr-xr-x    1 root   wheel      53552 Jul 29 15:59 AssetCacheManagerUtil
-rwxr-xr-x    1 root   wheel      48112 Jul 29 15:59 AssetCacheTetheratorUtil
-rwxr-xr-x    1 root   wheel      18320 Jul 29 15:59 BuildStrings
-rwxr-xr-x    1 root   wheel      18288 Jul 29 15:59 CpMac
-rwxr-xr-x    1 root   wheel      18288 Jul 29 15:59 DeRez
-rwxr-xr-x    1 root   wheel      18320 Jul 29 15:59 GetFileInfo
-rwxr-xr-x    1 root   wheel      69760 Jul 29 15:59 IOAccelMemory
-rwxr-xr-x    1 root   wheel      18304 Jul 29 15:59 MergePef
-rwxr-xr-x    1 root   wheel      18288 Jul 29 16:00 MvMac
-rwxr-xr-x    1 root   wheel      18304 Jul 29 15:59 ResMerger
-rwxr-xr-x    1 root   wheel      18288 Jul 29 15:59 Rez
-rwxr-xr-x    1 root   wheel      18288 Jul 29 15:59 RezDet
-rwxr-xr-x    1 root   wheel      18304 Jul 29 15:59 RezWack
-rwxr-xr-x    1 root   wheel      32384 Jul 29 16:00 SafeEjectGPU
-rwxr-xr-x    1 root   wheel      18304 Jul 29 15:59 SetFile
-rwxr-xr-x    1 root   wheel      18304 Jul 29 15:59 SplitForks
-rwxr-xr-x    1 root   wheel      18304 Jul 29 15:59 UnRezWack
-rwxr-xr-x    1 root   wheel      66608 Jul 29 15:59 a2p
-rwxr-xr-x    1 root   wheel     234128 Jul 29 15:59 a2p5.18
-rwxr-xr-x    1 root   wheel      18288 Jul 29 15:59 actool
-rwxr-xr-x    1 root   wheel      33808 Jul 29 15:59 addftinfo
-rwxr-xr-x    1 root   wheel      68464 Jul 29 15:59 afclip
-rwxr-xr-x    1 root   wheel     120848 Jul 29 15:59 afconvert
-rwxr-xr-x    1 root   wheel      55712 Jul 29 15:59 afhash
-rwxr-xr-x    1 root   wheel      98416 Jul 29 15:59 afida
-rwxr-xr-x    1 root   wheel     110208 Jul 29 15:59 afinfo
-rwxr-xr-x    1 root   wheel     162321 Feb 22  2019 afmtodit
-rwxr-xr-x    1 root   wheel      65568 Jul 29 15:59 afplay
-rwxr-xr-x    1 root   wheel      37664 Jul 29 15:59 afscexpand
-rwxr-xr-x    1 root   wheel      30096 Jul 29 15:59 agentxtrap
-rwxr-xr-x    1 root   wheel      18304 Jul 29 15:59 agvtool
-rwxr-xr-x   15 root   wheel        190 Feb 22  2019 alias
-rwxr-xr-x    1 root   wheel      41248 Jul 29 15:59 applesingle
lrwxr-xr-x    1 root   wheel         82 Aug  8 09:34 appletviewer -> /System/Li
```

Conclusiones

La sentencia fork al desarrollar esta practica fue de la que mas hubo que aprender para entender su funcionamiento, en cada ejecución, puede retornar 0 (que nos refiere al proceso hijo), menos uno (que se refiere a que hubo algún error en la creación de dicho proceso) o mas de 0 (que nos refiere al proceso padre), para poder realizar la estructura de árbol que solicita la practica, se tienen que sincronizar y matar ciertos procesos para evitar seguir creando mas y mas de estos, de tal forma que gracias a estructuras condicionales y al usar la sentencia wait() vamos anidando para lograr esto.

Referencias

[1] Bibliotecas ANSI C, función "perror". (s.f.). Recuperado 2 septiembre, 2019, de http://c.conclase.net/librerias/?ansifun=perror

[2] Wikipedia contributors. (2019, 2 septiembre). number to identify each process running on a computer. Recuperado 4 septiembre, 2019, de https://en.wikipedia.org/wiki/Process_identifier

[3] S. Operativos|| Funciones getpid(); y getppid();, G. U. S. I. M. (2015, 26 diciembre). S. Operativos|| Funciones getpid(); y getppid(); [Archivo de vídeo]. Recuperado 3 septiembre, 2019, de https://www.youtube.com/watch?v=6tzbPWeX0cU

[4] Como funciona la función fork() [Archivo de vídeo]. (2018, 11 julio). Recuperado 43 septiembre, 2019, de https://es.stackoverflow.com/questions/179414/como-funciona-la-funci%C3%B3n-fork

[5] Jesús Alberto Sánchez Tecalco, J. A. S. T. (s.f.). Árboles de procesos con fork [C] [Archivo de vídeo]. Recuperado 3 septiembre, 2019, de https://curiotecnology.blogspot.com/2012/12/arboles-de-procesos-con-fork-c.html

[6] Alex, A. H. (s.f.). Mostrar los archivos ocultos en Linux [Archivo de vídeo]. Recuperado 4 septiembre, 2019, de https://www.cambiatealinux.com/mostrar-los-archivos-ocultos-en-linux

[7] exec family of functions in C - GeeksforGeeks. (2019, 24 mayo). Recuperado 4 septiembre, 2019, de https://www.geeksforgeeks.org/exec-family-of-functions-in-c/

[8] How to use execv system call in linux? (2015, 21 agosto). Recuperado 4 septiembre, 2019, de https://stackoverflow.com/questions/32142164/how-to-use-execv-system-call-in-linux

[9] execl - execute a - Linux Man Pages (3p). (s.f.). Recuperado 4 septiembre, 2019, de https://www.systutorials.com/docs/linux/man/3p-execl/

[10] Lifka/System-Calls-Linux. (s.f.). Recuperado 5 septiembre, 2019, de https://github.com/Lifka/System-Calls-Linux/blob/master/execl.c