Práctica 4: Procesos e hilos.

v1.0

1 Objetivos

En esta práctica vamos a hacer varios ejercicios orientados a afianzar nuestro conocimiento del manejo del API POSIX de procesos e hilos, y cómo afecta el uso de hilos y procesos al manejo de ficheros.

Se aconseja al alumno que cree un directorio para la práctica y un subdirectorio por ejercicio. En las instrucciones se asume que el ejercicio N se hace en el subdirectorio ejercicio*N* dentro del directorio común para la práctica.

El archivo ficheros_p4.tar.gz contiene una serie de ficheros que pueden ser usados como punto de partida para el desarrollo los ejercicios de esta práctica, así como unos *makefiles* que pueden ser usados para la compilación de los distintos proyectos.

2 Ejercicios

Ejercicio 1: Creación de procesos y ejecución de programas

Desarrollar un programa `run_commands` que ejecute comandos especificados por el usuario, y espere a su terminación. Se proporciona un esqueleto de código con un *Makefile*, así como dos ficheros de entrada para comprobar el correcto funcionamiento del programa a desarrollar.

En el fichero *run_commands.c* proporcionado se incluye un código de prueba, que habrá que modificar para desarrollar la funcionalidad deseada. Este programa de partida acepta un único argumento donde se especifica un comando. El programa analiza dicho comando, y construye un array de cadenas de caracteres acabadas en `NULL` (formato de `argv`), que posteriormente se imprime por pantalla, liberando adecuadamente la memoria reservada con `malloc()`. El propósito de este programa es ilustrar el uso de la función `parse_command()` proporcionada, que deberá estudiarse y reusarse en la implementación del ejercicio.

El programa `run_commands` a desarrollar reconocerá un conjunto de opciones en la línea de comandos, que se deben procesar usando la función `getopt()`, empleada en prácticas previas. A continuación se presentan las opciones que aceptará el programa, que se recomienda implementar y probar en el orden en el que se describen:

- `-x <comando>`: Cuando el programa se invoque con esta opción, se creará un proceso hijo que ejecutará dicho comando. Para ello, el programa principal invocará la función `launch_command()` –a implementar–, que creará un proceso hijo con `fork()`, y hará que este ejecute el comando pasado como parámetro usando `execvp()`. La función devolverá el PID del proceso hijo, sin esperar a que termine su ejecución. El programa principal se encargará de esperar la terminación del proceso hijo creado. La función `launch_command()` tendrá el siguiente prototipo:

```
pid_t launch_command(char** argv);
```

- `-s <fichero>`: Esta opción permitirá al usuario indicar como argumento la ruta de un fichero con comandos a ejecutar. Este fichero será interpretado por líneas, tomando cada línea como un comando a ejecutar con la función `launch_command()`. Los comandos se ejecutarán de forma secuencial, esperando a que un comando termine antes de ejecutar el siguiente. **Sugerencia:** usar `fgets()` para leer del fichero por líneas. Consultar `man 3 fgets`

- `-b`: *(Parte opcional del ejercicio)* La opción `-b` del programa solamente tendrá efecto si se pasa junto con la opción `-s`. En este caso, los comandos del fichero indicado por la opción `-s` se ejecutarán uno tras otro sin esperar a que el comando anterior termine. Sólo cuando se hayan lanzado a ejecución todos los comandos indicados en el fichero (cada uno por parte de un proceso hijo) se esperará a que terminen todos. Asimismo, cada vez que uno de los comandos lanzados termine, el programa imprimirá por la salida estándar el número de comando que ha terminado, su PID y código de terminación –-p.ej., " `@@ Command #3 terminated (pid: 11576, status: 0)` ", como se muestra en el ejemplo de ejecución–. Para ello, el programa deberá mantener una estructura de datos –por simplicidad, array de longitud máxima prefijada– que permita asociar los PIDs de los hijos, con el número de cada comando en el orden de lanzamiento.

Ejemplo de ejecución

```
# Testing -x switch
$ ./run_commands -x ls
Makefile        run_commands    run_commands.c  run_commands.o  test1           test2

$ ./run_commands -x "echo hello"
Hello

# Testing -s switch
$ ./run_commands -s test1
@@ Running command #0: echo hello
hello
@@ Command #0 terminated (pid: 1439, status: 0)

@@ Running command #1: sleep 2
@@ Command #1 terminated (pid: 1440, status: 0)
@@ Running command #2: ls -l
total 88
-rw-r--r--@ 1 usuarioso  usuarioso   267 Oct 20 11:34 Makefile
-rwxr-xr-x  1 usuarioso  usuarioso  9960 Oct 20 11:58 run_commands
-rw-r--r--  1 usuarioso  usuarioso  4332 Oct 20 11:57 run_commands.c
-rw-r--r--  1 usuarioso  usuarioso  8984 Oct 20 11:58 run_commands.o
-rw-r--r--@ 1 usuarioso  usuarioso  31 Oct 20 11:34 test1
-rw-r--r--@ 1 usuarioso  usuarioso  41 Oct 20 11:46 test2
@@ Command #2 terminated (pid: 1443, status: 0)
@@ Running command #3: false
@@ Command #3 terminated (pid: 1444, status: 256)


# Testing -bs switch
$ ./run_commands -b -s test2
@@ Running command #0: echo one
@@ Running command #1: sleep 6
@@ Running command #2: sleep 3
@@ Running command #3: echo two
one
two
@@ Running command #4: sleep 1@@ Command #3 terminated (pid: 1457, status: 0)
@@ Command #0 terminated (pid: 1454, status: 0)
@@ Command #4 terminated (pid: 1458, status: 0)
@@ Command #2 terminated (pid: 1456, status: 0)
@@ Command #1 terminated (pid: 1455, status: 0)
```

Una vez acabado el desarrollo del programa `run_commands`, responde a las siguientes preguntas:

1. Al usar la opción `-x` del programa, el comando indicado como argumento se pasa encerrado entre comillas dobles en el caso de que este, a su vez, acepte argumentos, como por ejemplo `ls -l`. ¿Qué ocurre si el argumento de `-x` no se pasa entrecomillado? ¿Funciona correctamente el lanzamiento del programa `ls -l` si se encierra entre comillas simples en lugar de dobles? **Nota:** Para ver las diferencias prueba a ejecutar el siguiente comando: `echo $HOME`
2. ¿Es posible utilizar `execlp()` en lugar de `execvp()` para ejecutar el comando pasado como parámetro a la función `launch_command()`? En caso afirmativo, indica las posibles limitaciones derivadas del uso de `execlp()` en este contexto.
3. ¿Qué ocurre al ejecutar el comando `"echo hola > a.txt"` con `./run_commands -x`? ¿y con el comando `"cat run_commands.c | grep int"`? En caso de que los comandos no se ejecuten correctamente indica el motivo.

Ejercicio 2: Creación y paso de parámetros a hilos.

En este ejercicio vamos a usar la biblioteca de pthreads, por lo que será necesario compilar y enlazar con la opción `-pthread`.

Escribir un programa *hilos.c* que va a crear hilos cuya funcionalidad vendrá determinada por los argumentos que se le pasen en la creación. Los hilos recibirán como argumentos el puntero a una estructura que contenga dos campos: un entero, que será el número de hilo, y un caracter, que indicará si el hilo es prioritario (P) o no (N).

El programa deberá crear una variable para el argumento de cada hilo usando memoria dinámica, inicializar dicha variable con el número de hilo y su prioridad (los pares serán prioritarios y los impares no lo serán), crear los hilos y esperar a que finalicen.

Cada hilo copiará sus argumentos en variables locales, liberará la memoria dinámica reservada para los mismos, averiguará cuál es su identificador e imprimirá un mensaje con su identificador, el número de hilo y su prioridad.

El alumno debe consultar las páginas de manual de: `pthread_create` , `pthread_join` , `pthread_self` .

Probar a crear solamente una variable para el argumento de todos los hilos, dándole el valor correspondiente a cada hilo antes de la llamada a `pthread_create` . Explicar qué sucede y cuál es la razón.

Ejercicio 3: Manejo de señales.

En este ejercicio vamos a experimentar el envío de señales, haciendo que un proceso cree a un hijo, espere a una señal de un temporizador y, cuando la reciba, termine con la ejecución del hijo.

El programa principal recibirá como argumento el comando del programa que se desea que ejecute el proceso hijo. Si a su vez este comando consta de varios argumentos, estos se pasarán separados por espacios a continuación del nombre del programa a ejecutar.

El proceso padre creará un hijo, que cambiará su ejecutable con una llamada a execvp. A continuación, el padre establecerá que el manejador de la señal SIGALRM sea una función que envíe una señal SIGKILL al proceso hijo y programará una alarma para que le envíe una señal a los 5 segundos. Antes de finalizar, el padre esperará a que finalice el hijo y comprobará la causa por la que ha finalizado el hijo (finalización normal o por recepción de una señal), imprimiendo un mensaje por pantalla.

El alumno debe consultar las páginas de manual de: `sigaction` , `alarm` , `kill` , `wait` , `signal(7)` .

Para comprobar el funcionamiento correcto de nuestro programa podemos usar como argumento un ejecutable que termine en menos de 5 segundos (como ls o echo) y uno que no finalice hasta que le llegue la señal (como xeyes).

Una vez funcione el programa, modificar el padre para que ignore la señal SIGINT y comprobar que, efectivamente, lo hace.

Ejercicio 4: Manejo de ficheros con varios procesos e hilos

Se pretende crear un programa que utilice 10 procesos (el original y 9 procesos hijo) para escribir de manera concurrente un fichero “output.txt”. La idea es que cada proceso escriba una cadena de caracteres con un número decimal repetido 5 veces. Así el proceso inicial escribira 5 ceros (“00000”), el primer proceso hijo 5 unos (“11111”), el segundo 5 doses (“22222”) y así sucesivamente. De este modo el contenido del fichero al final será: 00000111112222233333444445555566666777778888899999

Un primer programador con poca experiencia en la programación de sistemas propone la siguiente implementación (fichero practica_2_5_inicial.c):

```c
int main(void)
{
    int fd1,fd2,i,pos;
    char c;
    char buffer[6];

    fd1 = open("output.txt", O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR);
    write(fd1, "00000", 5);
    for (i=1; i < 10; i++) {
        pos = lseek(fd1, 0, SEEK_CUR);
        if (fork() == 0) {
            /* Child */
            sprintf(buffer, "%d", i*11111);
            lseek(fd1, pos, SEEK_SET);
            write(fd1, buffer, 5);
            close(fd1);
            exit(0);
        } else {
            /* Parent */
            lseek(fd1, 5, SEEK_CUR);
        }
    }

    //wait for all children to finish
    while (wait(NULL) != -1);

    lseek(fd1, 0, SEEK_SET);
    printf("File contents are:\n");
    while (read(fd1, &c, 1) > 0)
        printf("%c", (char) c);
    printf("\n");
    close(fd1);
    exit(0);
}
```

Tras esta implementación el programador comprueba el funcionamiento, ejecutando 10 veces seguidas el programa con la esperanza de que no se produczcan carreras. El resultado, en la máquina del programador es:

```
$ for i in $(seq 10); do ./practica_2_5_inicial ; done
File contents are:
00000111112222233333555566666888899999
File contents are:
00000111112222255555666668888899999
File contents are:
00000111112222233333555566666888899999
File contents are:
00000111112222244444666667777799999
File contents are:
00000444447777755555666668888899999
File contents are:
00000222224444455555777778888899999
File contents are:
00000111112222244444555557777788888899999
File contents are:
00000111112222255555444448888887777799999
File contents are:
00000111112222244444555557777788888899999
File contents are:
00000111112222255555444448888887777799999
```

Al parecer el programa tiene algunos errores, puesto que se producen carreras y el resultado es incorrecto en todos los casos.

- Cuestión A - Soluciona la implementación inicial, manteniendo la escritura concurrente en el fichero. Es decir, el proceso padre escribirá los cinco ceros iniciales, el hijo uno los cinco unos, etc, sin necesidad de sincronizar los procesos. Es decir, se desea que no sea necesario imponer un orden en la ejecución de los procesos.

- Cuestión B - Proponer una solución en la que el padre escriba su número entre la escritura de los hijos, de modo que el contenido del fichero al final será el siguiente:

  00000111110000022222000003333300000444440000055555000006666600000777770000088888000009999 9

Sistemas Operativos

Template Design & Develop by HarnishDesign.

Template - Copyright © 2020 iDocs. All Rights Reserved.