Sistemas Operativos
Práctica 1

Antonio Javier Casado Hernández

Escuela Politécnica Superior

Semana 1

Ejercicio 1

a) Para encontrar la lista de funciones de hilos se ejecuta el comando
`man -k pthread`
En la Figura 1 se muestra las entradas del manual que usan *pthread*.

b) Para encontrar información sobre llamadas al sistema se utiliza
`man -k "system calls"` una de esas entradas del manual es *syscalls (2) - Linux system calls*, en esa página del manual se encuentra *write(2)* y finalmente para obtener información sobre write: `man 2 write`.

Ejercicio 2

a) Se utiliza `cat 'don quijote.txt' | grep 'molino' > aventuras.txt`
El primer comando *cat* vuelca el contenido del fichero 'don quijote.txt', esto es pasado al comando *grep* que filtrará las lineas que contienen 'molino' usando pipe y finalmente la salida de *grep* se guarda en un archivo 'aventuras.txt' usando >.

b) Para obtener el número de ficheros se utiliza `ls -1 | wc -l`
El comando *ls -1*, según el manual muestra los archivos del directorio línea por línea, esto es pasado por pipe a *wc -l* que cuenta esas líneas.

c) Se utiliza
`sort "lista de la compra Elena.txt" "lista de la compra Pepse.txt" 2> /dev/null | uniq -c | wc -l > numcompra.txt`
Con *sort* se ordenan las lineas de los dos archvos.
Con *2>* se redirige el posible error a *dev/null*.
Posteriormente se pasa con pipeline a *uniq -c* que imprime todas las lineas no repetidas consecutivamente.
Se le pasa a *wc -l* que cuenta las lineas y finalmente se escribe en el archivo con >.

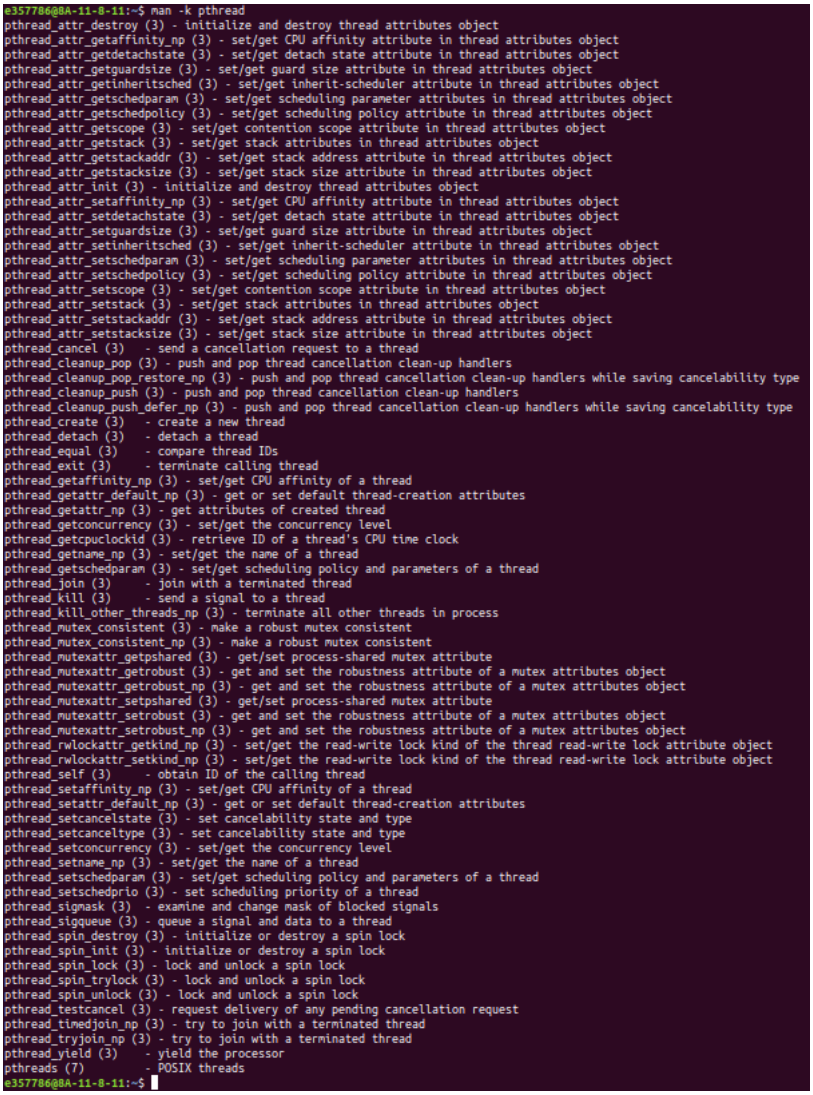

Fig. 1. Salida del comando

d) (Opcional)
Se ha utilizado `ps -A -L | awk '{print$1}' | uniq -c > hilos.txt`
ps -A -L muestra todos los procesos del sistema con -A junto con todos sus hilos con -L, *awk '{print$1}'* extrae la primera columna (el PID de cada proceso), *uniq -c* debido a que los procesos ya están ordenados, cuenta cuantos PIDs se repiten (sus hilos) y finalmente se escribe en *hilos.txt* con >.

Ejercicio 3

a) Se imprime *No such file or directory* y corresponde al valor *errno* 2

b) Devuelve *Permission denied* cuyo valor es el 13.

c) Habría que guardar *errno* en una variable de tipo int justo despues de la llamada a `fopen`

Ejercicio 4

a) Se observa que el programa se está ejecutando y consumiendo un 99% de CPU.

b) Con `sleep`, a diferencia del anterior, este programa no aparece en *top* y no está consumiendo CPU.

Ejercicio 5

a) Si se espera a los hilos, el proceso termina, haciendo terminar a sus hilos no llegando a imprimir el mensaje completo.

b) Con *pthread_ exit()* el proceso vuelve a esperar a que los hilos terminen de ejecutarse.

c)

Ejercicio 6

Código ejercicio_hilos.c

Semana 2

Ejercicio 7

a) No se puede saber porque no se sabe si el procesador ejecuta primero el padre o el hijo, en cambio si se sabe que imprimirá los padres en orden.

b) Utilizando *getpid()* y *getppid()* soluciona el problema.

```
1     /*Codigo anterior*/
2     else if(pid ==  0)
3     {
4       printf("Hijo PID=%d y mi padre=%d\n", getpid(), getppid());
5       exit(EXIT_SUCCESS);
6     }
7     /*...*/
```

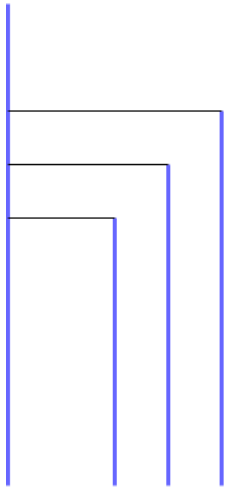

Fig. 2. Árbol de procesos de *ejemplo_fork.c*

c) En la Figura 2 se muestra el diagrama correspondiente a *ejemplo_fork.c*

d) Deja huerfanos porque wait() espera a solo un hijo.

e) Debido a que *wait()* devuelve el ID del hijo terminado, cuando el proceso padre no tenga hijos, esta función devolverá -1, entonces es cuando todos los hijos han terminado, se implementa de la siguiente manera:

```
1     /*Codigo anterior*/
2     while((wait(NULL)) > 0){
3       printf("Hijo terminado\n");
4     }
5     /*...*/
```

Ejercicio 8

Código ejercicio_arbol.c

Ejercicio 9

a) El proceso padre no imprime nada, no es correcto porque la memoria reservada por el padre no es compartida con el hijo, es decir, se duplica, haciendo así que el mensaje del padre esté vacío hasta el final.

b) Hay que liberar memoria en ambos porque al no ser mem. compartida, también se crea esa memoria para todos los hijos.

```
1       /*Codigo anterior*/
2
3       else if (pid == 0)
4       {
5         strcpy(sentence, MESSAGE);
6         free(sentence);
7         exit(EXIT_SUCCESS);
8       }
9       else
10      {
11        wait(NULL);
12        printf("Padre: %s\n", sentence);
13        free(sentence);
14        exit(EXIT_SUCCESS);
15      }
16
17      /*...*/
```

Ejercicio 10

a) Código ejercicio_shell.c.

b) Se ha utilizado `execvp()` porque no se sabe a priori el número de argumentos que el comando va a utilizar, de esta forma se puede añadir tantos como se desee.

c) Imprime *sh: 1: inexistente: not found Exited with value 127*

d) Imprime que se trata de otro tipo de señal.

e) Código ejercicio_shell_spawn.c

Semana 3

Ejercicio 11

En el manual de proc y buscando con / se puede encontrar lo deseado. En este caso se accede al proceso *self*.

a) Con el comando `ls -la en proc/self/exe` se ve que el nombre del ejecutable es */usr/bin/bash*

b) Con el comando `ls -l en proc/self/cwd` se accede al directorio.

c) Con `cat proc/self/cmdline | tr "\0' '\n'` da información sobre la linea de comandos que lo inició.

d) Con `cat proc/self/environ` muestra las variables de entorno del proceso

e) En el directorio /proc/self/task se encuentra los diferentes hilos que ejecuta el proceso, dentro de ellos toda su información.

Ejercicio 12

a) Se encuentran abiertos los descriptores 0, 1 y 2. Apuntan a dispositivos, entrada estándar de la terminal teclado por ejemplo.

b) Se han añadido 2 descriptores mas que apuntan a file1.txt y a file2.txt en el directorio donde se ejecuta el programa.

c) Sí se ha borrado FILE1 y se sigue pudiendo acceder a él (cat 3) porque sigue abierto por otro proceso, se podría copiar la info del fichero a traves del /proc/.

d) File1 ha desaparecido y han aparecido dos ficheros mas que apuntan al mismo file3.txt.
Se enumeran en orden.

Ejercicio 13

a) El mensaje se imprime 2 veces porque el buffer no se ha vaciado, por lo que el hijo hereda el buffer, y al salir de forma existosa, vacia el buffer de "Yo soy tu padre" imprimiendo este mensaje dos veces.

b) No se sigue imprimiendo 2 veces porque al incluir n se vacía el buffer.

c) Al redirigir sigue imprimiendo 2 veces porque n vacía el buffer solo cuando se imprime por pantalla.

d) El problema se corrige utilizando `fflush(stdout)`

Ejercicio 14

a) Imprime:
He recibido el string: Hola a todos!
He escrito en el pipe

b) Si no se cierra el extremo de escritura, el programa queda en espera porque se queda esperando a él mismo.

Ejercicio 15

Código ejercicio_pipes.c