Universidad Autónoma de Madrid

Sistemas Operativos 2022/2023

Práctica 1

Grupo 2201

Pedro José Alemañ Quiles – pedro.alemann@estudiante.uam.es

Ejercicio 1: Uso del manual.

a)

La salida al ejecutar el comando “man -k pthread > salida.txt” en la terminal es:

pthread_attr_destroy (3) - initialize and destroy thread attributes object
pthread_attr_getaffinity_np (3) - set/get CPU affinity attribute in thread attributes object
pthread_attr_getdetachstate (3) - set/get detach state attribute in thread attributes object
pthread_attr_getguardsize (3) - set/get guard size attribute in thread attributes object
pthread_attr_getinheritsched (3) - set/get inherit-scheduler attribute in thread attributes object
pthread_attr_getschedparam (3) - set/get scheduling parameter attributes in thread attributes object
pthread_attr_getschedpolicy (3) - set/get scheduling policy attribute in thread attributes object
pthread_attr_getscope (3) - set/get contention scope attribute in thread attributes object
pthread_attr_getstack (3) - set/get stack attributes in thread attributes object
pthread_attr_getstackaddr (3) - set/get stack address attribute in thread attributes object
pthread_attr_getstacksize (3) - set/get stack size attribute in thread attributes object
pthread_attr_init (3) - initialize and destroy thread attributes object
pthread_attr_setaffinity_np (3) - set/get CPU affinity attribute in thread attributes object
pthread_attr_setdetachstate (3) - set/get detach state attribute in thread attributes object
pthread_attr_setguardsize (3) - set/get guard size attribute in thread attributes object
pthread_attr_setinheritsched (3) - set/get inherit-scheduler attribute in thread attributes object
pthread_attr_setschedparam (3) - set/get scheduling parameter attributes in thread attributes object
pthread_attr_setschedpolicy (3) - set/get scheduling policy attribute in thread attributes object
pthread_attr_setscope (3) - set/get contention scope attribute in thread attributes object
pthread_attr_setstack (3) - set/get stack attributes in thread attributes object
pthread_attr_setstackaddr (3) - set/get stack address attribute in thread attributes object
pthread_attr_setstacksize (3) - set/get stack size attribute in thread attributes object
pthread_cancel (3) - send a cancellation request to a thread
pthread_cleanup_pop (3) - push and pop thread cancellation clean-up handlers
pthread_cleanup_pop_restore_np (3) - push and pop thread cancellation clean-up handlers while saving cancelability type
pthread_cleanup_push (3) - push and pop thread cancellation clean-up handlers
pthread_cleanup_push_defer_np (3) - push and pop thread cancellation clean-up handlers while saving cancelability type
pthread_create (3) - create a new thread
pthread_detach (3) - detach a thread
pthread_equal (3) - compare thread IDs
pthread_exit (3) - terminate calling thread
pthread_getaffinity_np (3) - set/get CPU affinity of a thread
pthread_getattr_default_np (3) - get or set default thread-creation attributes
pthread_getattr_np (3) - get attributes of created thread
pthread_getconcurrency (3) - set/get the concurrency level
pthread_getcpuclockid (3) - retrieve ID of a thread's CPU time clock
pthread_getname_np (3) - set/get the name of a thread
pthread_getschedparam (3) - set/get scheduling policy and parameters of a thread

pthread_join (3) - join with a terminated thread
pthread_kill (3) - send a signal to a thread
pthread_kill_other_threads_np (3) - terminate all other threads in process
pthread_mutex_consistent (3) - make a robust mutex consistent
pthread_mutex_consistent_np (3) - make a robust mutex consistent
pthread_mutexattr_getpshared (3) - get/set process-shared mutex attribute
pthread_mutexattr_getrobust (3) - get and set the robustness attribute of a mutex attributes object
pthread_mutexattr_getrobust_np (3) - get and set the robustness attribute of a mutex attributes object
pthread_mutexattr_setpshared (3) - get/set process-shared mutex attribute
pthread_mutexattr_setrobust (3) - get and set the robustness attribute of a mutex attributes object
pthread_mutexattr_setrobust_np (3) - get and set the robustness attribute of a mutex attributes object
pthread_rwlockattr_getkind_np (3) - set/get the read-write lock kind of the thread read-write lock attribute object
pthread_rwlockattr_setkind_np (3) - set/get the read-write lock kind of the thread read-write lock attribute object
pthread_self (3) - obtain ID of the calling thread
pthread_setaffinity_np (3) - set/get CPU affinity of a thread
pthread_setattr_default_np (3) - get or set default thread-creation attributes
pthread_setcancelstate (3) - set cancelability state and type
pthread_setcanceltype (3) - set cancelability state and type
pthread_setconcurrency (3) - set/get the concurrency level
pthread_setname_np (3) - set/get the name of a thread
pthread_setschedparam (3) - set/get scheduling policy and parameters of a thread
pthread_setschedprio (3) - set scheduling priority of a thread
pthread_sigmask (3) - examine and change mask of blocked signals
pthread_sigqueue (3) - queue a signal and data to a thread
pthread_spin_destroy (3) - initialize or destroy a spin lock
pthread_spin_init (3) - initialize or destroy a spin lock
pthread_spin_lock (3) - lock and unlock a spin lock
pthread_spin_trylock (3) - lock and unlock a spin lock
pthread_spin_unlock (3) - lock and unlock a spin lock
pthread_testcancel (3) - request delivery of any pending cancellation request
pthread_timedjoin_np (3) - try to join with a terminated thread
pthread_tryjoin_np (3) - try to join with a terminated thread
pthread_yield (3) - yield the processor
pthreads (7) - POSIX threads

b)

Con el comando “man man” podemos ver una tabla con los números de sección del manual seguidos por los tipos de comandos que hay en cada sección. Las ≪llamadas al sistema≫ se encuentran en la sección 2.

Por tanto, para acceder a la información acerca de la llamada al sistema write, debemos usar el comando “man 2 write”.

Ejercicio 2: Comandos y redireccionamiento.

a)

Usamos el comando “grep molino don\ quijote.txt >> aventuras.txt”. Con “grep” buscamos las líneas de don_quijote.txt que contienen ≪molino≫ y con “>>” redirigimos la salida de grep al fichero aventuras.txt, si el fichero existe se añade la salida al final pero si no lo crea.

b)

Se puede utilizar “ls | wc -l”. Con “ls” mostramos los ficheros del directorio actual redirigiendo la salida por medio de un *pipeline* a “wc” que, usando el parámetro -l, permite contar las líneas de la entrada.

c)

El comando usado es: “cat lista\ de\ la\ compra\ Elena.txt lista\ de\ la\ compra\ Pepe.txt 2> /dev/null | sort | uniq | wc -w > num\ compra.txt”.

Primero se concatenan los ficheros de texto con “cat”, en caso de que alguno no exista se ignora el mensaje de error con “2> /dev/null”. Por medio de un *pipeline* se redirige el fichero a la entrada de “sort” para ordenarlo alfabéticamente, se redirige a la entrada de “uniq” para eliminar las repeticiones, ahora se redirige a “wc” para contar, usando el parámetro -w, las palabras del fichero y, finalmente, se escribe el resultado en el fichero ‘num_compra.txt’.

Ejercicio 3: Control de errores.

a)

El mensaje que se imprime al intentar abrir un fichero que no existe es “No such file or directory”, corresponde al valor 2 de errno.

b)

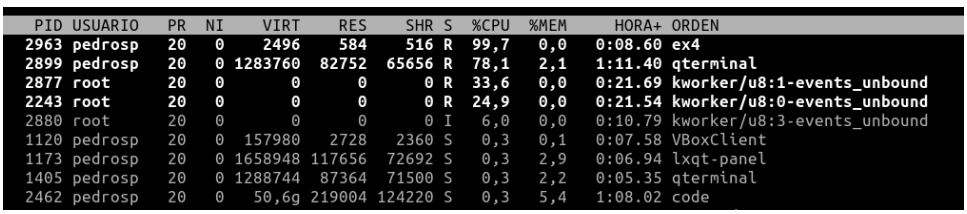
El mensaje que se imprime al intentar abrir /etc/shadow es “Permission denied”, corresponde al valor 13 de errno.

c)

Habría que guardar el valor de errno en una variable auxiliar y, una vez ejecutado printf, asignar a errno el valor de esta variable auxiliar. De esta forma, evitamos que se modifique el mensaje de error de fopen en caso de fallar printf.

Ejercicio 4: Espera activa e inactiva.

a)



PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+	ORDEN
2963	pedrosp	20	0	2496	584	516	R	99,7	0,0	0:08.60	ex4
2899	pedrosp	20	0	1283760	82752	65656	R	78,1	2,1	1:11.40	qterminal
2877	root	20	0	0	0	0	R	33,6	0,0	0:21.69	kworker/u8:1-events_unbound
2243	root	20	0	0	0	0	R	24,9	0,0	0:21.54	kworker/u8:0-events_unbound
2880	root	20	0	0	0	0	I	6,0	0,0	0:10.79	kworker/u8:3-events_unbound
1120	pedrosp	20	0	157980	2728	2360	S	0,3	0,1	0:07.58	VBoxClient
1173	pedrosp	20	0	1658948	117656	72692	S	0,3	2,9	0:06.94	lxqt-panel
1405	pedrosp	20	0	1288744	87364	71500	S	0,3	2,2	0:05.35	qterminal
2462	pedrosp	20	0	50,6g	219004	124220	S	0,3	5,4	1:08.02	code

Al utilizar clock(), se puede ver con el comando “top” que aparece el proceso del programa (ex4.c) usando prácticamente el 100% del procesador, es decir, hay otros procesos que no se pueden ejecutar porque el procesador está ocupado con este.

b)

Al utilizar sleep(), no aparece el proceso al usar “top”. Esto se debe a que el proceso es marcado como Suspendido y no se volverá a ejecutar hasta que por lo menos pase el tiempo indicado por parámetro. Así, otros procesos pueden hacer uso del procesador.

Ejercicio 5: Finalización de hilos.

a)

Puede pasar que los hilos no tengan tiempo para ejecutar la función slow_printf porque el hilo principal termina antes, eliminando los hilos.

b)

Antes de que los hilos escriban las palabras, se muestra por pantalla el mensaje de que el programa se ha ejecutado correctamente. Cuando el programa llega a pthread_exit, el hilo principal espera a que los hilos terminen. Si pasamos Valgrind podemos ver que no se liberan los recursos de los hilos, ya que para liberar hilos no “desligados” hay que utilizar pthread_join pues se espera que otro hilo reciba el retorno de estos.

c)

Se pueden desligar los hilos usando pthread_detach para cada uno y si, además, queremos que el hilo principal espere a que los hilos se ejecuten, hay que mantener pthread_exit.

Ejercicio 6: Creación de procesos.

a)

No se puede saber porque el orden de ejecución de los procesos depende del criterio usado por el planificador de procesos del sistema operativo.

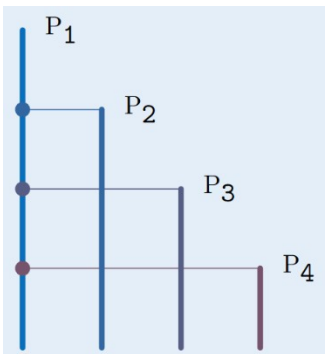
b)

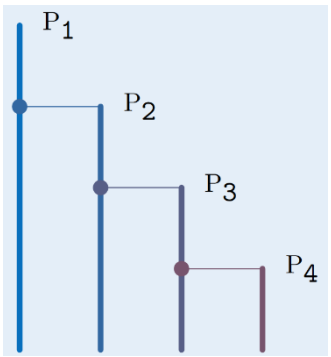
En el proceso hijo se ejecuta la instrucción:

```
printf("pid: %jd, ppid: %jd\n", (__intmax_t) getpid(), (__intmax_t) getppid());
```

Con getpid() obtenemos el PID del proceso y con getppid() el PID del proceso padre.

c)





El código da como resultado el árbol de procesos de la izquierda. Hemos llegado a esta conclusión porque en el apartado anterior comprobamos que todos los procesos hijos tienen el mismo padre.

Para conseguir el árbol de la derecha hay que hacer varias modificaciones: hacer un primer fork() fuera del bucle for y luego cambiar la posición del fork() que hay en el bucle for, habría que ponerlo en el if correspondiente al proceso hijo.

d)

Dado que no sabemos con certeza si se ejecutará primero el proceso padre o el hijo tras cada instrucción fork(), cabe la posibilidad de que en alguna de estas llamadas a la función se ejecute primero al proceso padre. De este modo, como solo se hace un "wait(NULL)", terminaría su ejecución con "exit(EXIT_SUCCESS)" antes de que finalicen algunos de sus hijos, dejándolos huérfanos.

e)

La solución más sencilla consiste en cambiar la instrucción "wait(NULL)" que hay justo después del bucle por la instrucción "while(wait(NULL) > 0)". Esta nueva instrucción ejecutará "wait(NULL)" hasta que todos los procesos hijos hayan terminado.

Ejercicio 7: Espacio de memoria.

a) Al ejecutar el código no se muestra el mensaje contenido en la variables sentence por pantalla, por tanto, el programa no es correcto. El problema es que al crear un proceso hijo con fork(), este no comparte la zona de memoria con el proceso padre. Además, tampoco se libera la memoria del array sentence.

b) Hay que liberar la memoria en ambos procesos porque, a pesar de que el proceso hijo hereda las variables creadas y la memoria reservada por el proceso padre, los procesos no comparten zona de memoria

```c
int main(void) {
    pid_t pid;
    char * sentence = calloc(sizeof(MESSAGE), 1);

    pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        strcpy(sentence, MESSAGE);
        free(sentence);
        exit(EXIT_SUCCESS);
    }
    else {
        wait(NULL);
        printf("Parent: %s\n", sentence);
        free(sentence);
        exit(EXIT_SUCCESS);
    }
}
```

Ejercicio 8: Ejecución de programas.

a) La salida es exactamente igual, esto sucede porque *argv[] es el array de argumentos de entrada del programa que vamos a ejecutar y, como sabemos, en C el primer argumento hace referencia al nombre del programa así que podemos modificarlo sin problema.

b) Al usar execl los argumentos, en vez de en un array, se pasan por separado. Además, hay que colocar la ruta completa de ls porque la función no contiene una p en su nombre, es decir, no busca el programa en la variable de entorno PATH. Quedaría así:

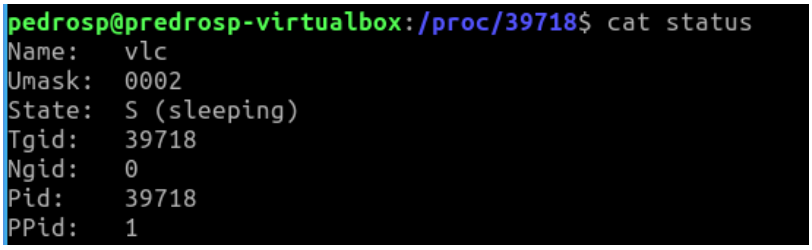
```c
execl("/usr/bin/ls", "ls", "./", (char *) NULL)
```

Ejercicio 9: Directorio de información de procesos.

Abro la carpeta de un proceso con PID 3870 ejecutando el comando cd /proc/3870, el resultado de los apartados es:

a)

Con el fichero status podemos conseguir información nombre, estado, PID, PPID, hilos (threads) abiertos, memoria virtual… Ejecuto cat status y obtengo, entre otros, el nombre del ejecutable, en este caso "vlc":



b)

El enlace simbólico al directorio actual del proceso es cwd, para conocer el path que representa este enlace simbólico podemos utilizar el comando readlink. Así, al ejecutar readlink cwd obtenemos la ruta /home/predrosp, es decir una carpeta personal de usuario del sistema.

c)

Para obtener el comando utilizado para iniciar el programa usamos el fichero cmdline, lo mostramos: cat cmdline | tr '\0' '\n'. Hemos usado tr para mostrar la salida separada con saltos de línea en vez de todo junto. En nuestro caso, hemos obtenido –started-from-file porque no iniciamos el programa mediante la terminal, en ese caso hubiésemos obtenido vlc.

d)

Para obtener las variables de entorno podemos usar el fichero environ, de nuevo tendremos que usar tr para convertir los '\0' en '\n'. Y para buscar concretamente la variable LANG usamos grep. El comando final es cat environ | tr '\0' '\n' | grep LANG y el resultado LANG=es_ES.UTF-8.

e)

Los hilos de un proceso se encuentran en la carpeta task, para listarlos usamos ls. Por tanto, el comando final es ls task.

Ejercicio 10: Visualización de descriptores de fichero.

a)

Al llegar a Stop 1, los descritpores de ficheros abiertos son los que por defecto suelen estar abiertos al iniciar un programa: STDIN_FILENO (0), STDOUT_FILENO (1) y STDERR_FILENO (2).

b)

Cuando llegamos a Stop 2, se ha llamado a open para file1 por lo que se ha añadido el descriptor 3 a la tabla de descriptores de ficheros. Antes de Stop 3 se abre el fichero file2 y, por tanto, se añade el descriptor 4 a la tabla de descriptores de ficheros.

c)

El fichero file1.txt no se ha eliminado por completo, lo que se ha borrado son las referencias al mismo, es decir, no podemos acceder a él desde un directorio, ni siquiera desde /proc. Para eliminar el fichero habrá que cerrarlo con close o finalizar el proceso. Una forma simple de recuperar los datos es usando el comando tail para copiarlo: tail -c +0 -f /proc/<PID del proceso>/fd/# > /new/path/to/file Así, volcamos la información del fichero en otro fichero nuevo, de modo que podemos recuperarla.

d)

Como dijimos en el apartado anterior, al llamar a close para file1 el fichero se elimina definitivamente y su descriptor, el 3, desaparece de la tabla (Stop 5). Al llegar a Stop 6, se ha abierto file3 con open y se ha añadido el descriptor 3 a la tabla. Por último, al llegar a Stop 7, se ha abierto file4 y se ha añadido el descriptor 5 a la tabla.

Ejercicio 11: Problemas con el buffer.

a) Se imprime el mensaje dos veces mostrándose “I am your fatherNooooooI am your father”. Se debe a que el mensaje se queda en el buffer del padre y, al hacer un fork, el hijo tiene otro buffer con el mismo mensaje dentro, debido a que es una copia. Por eso, cuando el hijo muestra su mensaje por pantalla aparece también el mensaje del padre.

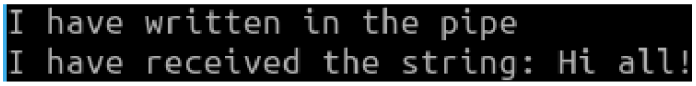
b) En este caso, al añadir “\n” en el mesaje del padre, forzamos la liberación del buffer y, por tanto, eliminamos el problema anterior.

c) De nuevo, nos encontramos con el error del apartado a), se repite el mensaje “I am your father”. Ocurre porque cuando no escribimos en la salida estándar, por ejemplo en un fichero en disco, añadir “\n” no es suficiente para liberar el buffer.

d) La forma más simple de corregir el problema es liberando manualmente el buffer de escritura, para ello podemos usar la función fflush. En el código añadimos fflush(stdout) después de cada instrucción printf.

Ejercicio 12: Ejemplo de tuberías.

a) El hijo muestra un mensaje indicando que ha escrito en la tubería mientras. Por su parte, el padre muestra otro mensaje en el que indica que ha recibido la cadena escrita por el hijo y la escribe por pantalla.



b) Se muestra lo mismo que en el apartado anterior pero el programa no finaliza, se queda esperando. El programa se queda parado porque para saber cuando finaliza el proceso de escritura es necesario cerrar el extremo de escritura, por eso el hijo espera a que termine la escritura y, a su vez, el padre espera a que termine el hijo**.**