Tema

3.3

Sistema Operativos (SSOO)

Procesos e Hilos en C

Universidad Francisco de Vitoria
UFV Madrid

Grado en Ingeniería Informática
Escuela Politécnica Superior

Índice

- Gestión de Hilos y Procesos
  - Crear procesos con *fork()*
  - Ejecutar procesos con *exec*

Procesos e Hilos: diferencias

Si se quiere que una aplicación ejecute tareas en paralelo existen dos opciones:

- Crear un nuevo proceso
- Crear un nuevo hilo (thread)

Procesos

- Un proceso de Unix es cualquier programa en ejecución, el cual es totalmente independiente de otros procesos.
- Un proceso no se puede meter en la zona de memoria de otro proceso.
- En un proceso puede haber varios hilos de ejecución.
- Un proceso es más costoso de lanzar, necesita crear una copia de todo su contenido en memoria.

Procesos e Hilos: diferencias

Hilos

- Dentro de un proceso todos los hilos comparten la misma memoria.
- Si un hilo modifica una variable, los demás hilos del mismo proceso verán dicha modificación.
- Lo anterior hace necesario el uso de semáforos o mutex (**EX**clusión **MUT**ua en español) para evitar que dos hilos modifiquen a la vez a la misma estructura de datos.
- Si un hilo “corrompe" una zona de memoria, los otros hilos del mismo proceso verán la memoria estropeada.
- Un fallo en un hilo puede hacer fallar a todos los hilos del mismo proceso.

Procesos e Hilos: diferencias

Importante

- En los hilos, al compartir la memoria y los recursos, es necesario el uso de mutex o semáforos, haciendo compleja su programación.
- Cuando se lanza un proceso, este se hace independiente de quien lo lanzó, evitando así problemas de memoria.
- Si se necesita que haya comunicación entre procesos, será necesario pensar en el uso de memorias compartidas y sus semáforos, el uso de colas de mensaje, sockets o cualquier otro mecanismo de comunicación entre procesos en Unix.

¿Cuándo uso procesos y cuándo uso hilos?

- **Procesos**: cuando hay poca comunicación entre ellos.
- **Hilos**: cuando es necesario compartir, cuando hay que actualizar datos o cuando se necesitan muchos cálculos en paralelo.

Procesos e Hilos: fork()

- fork(), en el punto donde se llame, duplica los procesos y estos empiezan a ejecutarse por separado.
  - El retorno de dicha función es, al proceso original, un identificador del proceso nuevo.
- Al proceso nuevo le devuelve 0.
  - Cada proceso puede saber si es el original o el proceso padre o el proceso nuevo (proceso hijo), de esta forma cada uno puede hacer cosas distintas.
- Un *fork()* en un *if*, permite que el proceso padre siga por la parte del *else* y proceso hijo por el *then*.
  - El proceso original podría seguir atendiendo nuevos clientes que quieran conectarse a nuestro programa por un socket.
  - El proceso hijo podrá atender a un cliente que acaba de conectarse y que es el que ha provocado que lancemos el *fork*().

Procesos e Hilos: fork()

- Recuerde que fork duplica todo el espacio de memoria. Por ello, ambos procesos tienen todas las variables repetidas, pero distintas.
- Si el proceso padre modifica la variable “counter", el proceso hijo no verá reflejado el cambio en su versión de “counter".
- Si antes del fork(), por ejemplo, se tiene un fichero abierto (un fichero normal, un socket o cualquier otra cosa), después del fork() ambos procesos tendrán abierto el mismo fichero y ambos podrán escribir en él.
- Uno de los procesos puede cerrar el fichero mientras que el otro lo puede seguir teniendo abierto.
- fork() puede devolver -1 en caso de error. Si esto ocurre, no se ha creado ningún nuevo proceso.

Procesos e Hilos: fork()

- El proceso padre puede "esperar" que el hijo termine.
- La función wait() permite “pausar” un proceso.
- wait() recibe como parámetro la dirección de un entero para que nos lo devuelva relleno.
- La función wait() deja dormido al proceso que la llama hasta que alguno de sus procesos hijo termina.
- En el entero se guardará la información de cómo ha terminado el hijo: llamando a exit(), recibe kill, está caido, otro…).

```
switch (fork())
{
    case -1:
        /* Código de error */
        ...
        break;
    case 0:
        /* Código del proceso hijo */
        ...
        break;
    default:
        ...
        /* Código del proceso original */
}
```

Procesos e Hilos: pthread_create

- **WIFEXITED(estadoHijo**)
  - Es 0 si el hijo ha terminado de una manera **anormal** (caída, con un kill, etc).
  - Distinto de 0 si ha terminado porque ha hecho una llamada a la función exit()
- **WEXITSTATUS(estadoHijo)**
  - Devuelve el valor que ha pasado el hijo a la función exit(), siempre y cuando la macro anterior indique que la salida ha sido por una llamada a exit().

Procesos e Hilos: pthread_create

Parámetros de pthread_create() :

- **pthread_t *** puntero a un identificador de thread. La función devuelve este valor relleno con lo que luego posible hacer referencia al hilo.
- **pthread_attr_t *** atributos de creación del hilo. Hay varios atributos posibles, como por ejemplo la prioridad. Un hilo de mayor prioridad se ejecutará con preferencia (tendrá más rodajas de tiempo) que otros hilos de menor prioridad. Si pasa NULL el hilo se creará con sus atributos por defecto. Si necesita un programa que cree y destruya hilos continuamente, evite el uso de NULL debido a que podría dejar memoria sin liberar al terminar un hilo.
- **void *(*)(void *)** es el tipo de una función que admite un puntero void * y que devuelve void *. Eso quiere decir que a este parámetro le podemos pasar el nombre de una función que cumpla lo antes dicho.

Procesos e Hilos: pthread_create

Parámetros de pthread_create() :

- **void *(*)(void *) e**sta función es la que se ejecutará como un hilo aparte. El hilo terminará cuando la función termine o cuando llame a la función pthread_exit(). Es común hacer que esta función cree un bucle infinito y quede suspendido en un semáforo o a la espera de una señal para hacer lo que tenga que hacer (luego volver a quedar dormido).
- **void *** es el parámetro que se le pasará a la función anterior cuando se ejecute en el hilo aparte. El programa principal puede pasar un único parámetro a la función que se ejecutará en el hilo. La función del hilo sólo tendrá que hacer el "cast" adecuado.

Llamadas exec

UNIX dispone de **exec** para lanzar a ejecución un programa, **almacenado en forma de fichero**.

Sólo existe una llamada pero las bibliotecas estándares de C disponen de varias funciones, todas comenzando por 'exec' que se diferencian por el tipo de parámetros que se pasan al programa.

- **execl:** es la forma simple y se usa cuando se conoce a priori el número de argumentos.
  - int execl (char* file, char* arg0, char* arg1, ... , 0);
  - *file:* nombre del fichero seguido de todos los argumentos, al final un puntero nulo o con valor cero.
  - Ejemplo, ejecute la siguiente instrucción: /bin/ls -l /usr/include

Llamadas exec

- Utilice: **execl ("/bin/ls", "ls", "-l", "/usr/include", 0);**
  - Argumentos: nombre del programa, comando, parámetros, contexto, puntero nulo.
- Sino conoce de ante mano el número de argumentos, se emplea la función ***execv***:
  - **execv (char* fichero, char* argv []);**
- El parámetro *argv* es una lista que representa los argumentos del programa. El útlimo valor de la lista es un nulo o cero. Adaptamos el ejemplo anterior:
  - **char* agurments [] = { "ls", "-l", "/usr/include", 0 };**
  - **execv ("/bin/ls", agurments);**

Llamadas exec

- Hasta ahora se ha escrito el nombre completo del fichero o comando a ejecutar ***"/bin/ls"*** en vez de ***"ls"***.
  - En este caso, tanto execl como execv no tienen en cuenta la variable PATH.
  - Para tener en cuenta esta variable, utilice las versiones ***execlp*** o ***execvp***.
  - ***execvp ("ls", arguments);***
  - El programa "/bin/ls“ se ejecutará si "/bin" está definida en el PATH.
- Retorno de exec: todas las llamadas exec retornan un valor diferente de NULL si el programa no se ha podido ejecutar.
- En caso de que sí se ejecute el programa, se transfiere el control a éste y la llamada *exec* nunca retorna.
- En cierta forma el programa que invoca un exec desaparece del mapa.

Sección crítica: sección de código dentro de un proceso que requiere acceso a recursos (critical section) compartidos y que no puede ser ejecutada mientras otro proceso esté en una sección de código correspondiente.

Interbloqueo: situación en la cual dos o más procesos son incapaces de actuar porque (deadlock) cada uno está esperando que alguno de los otros haga algo.

Círculo vicioso: situación en la cual dos o más procesos cambian continuamente su estado (livelock) en respuesta a cambios en los otros procesos, sin realizar ningún trabajo útil.

exclusión mutua: requisito de que cuando un proceso esté en una sección crítica que accede a recursos compartidos, ningún otro proceso pueda estar en una sección crítica que acceda a ninguno de esos recursos compartidos.

condición de carrera: situación en la cual múltiples hilos o procesos leen y escriben un dato (race condition) compartido y el resultado final depende de la coordinación relativa de sus ejecuciones.

Inanición: situación en la cual un proceso preparado para avanzar es soslayado (starvation) indefinidamente por el planificador; aunque es capaz de avanzar, nunca se le escoge.

Bibliografía

- **CARRETERO**, Jesús, GARCÍA, Félix, DE MIGUEL, Pedro, PÉREZ, Fernando. Sistemas Operativos: una visión aplicada. McGraw-Hill, 2001.
- **STALLINGS, William. Sistemas operativos: aspectos internos y principios de diseño. 5ª Edición. Editorial Pearson Educación. 2005. ISBN: 978-84-205-4462-5.**
- **TANENBAUM**, Andrew S. Sistemas operativos modernos. 3ª Edición. Editorial Prentice Hall. 2009. ISBN: 978-607- 442-046-3.

marlon.cardenas@ufv.es

Universidad Francisco de Vitoria
UFV Madrid

Grado en Ingeniería Informática
Escuela Politécnica Superior