Introducción a Sistemas Operativos: Concurrencia

Clips xxx
Francisco J Ballesteros

1. Threads y procesos

Hemos utilizado `fork` para crear procesos, lo que hace que los procesos hijo *no* compartan recursos con el padre: tienen su propia copia de los segmentos de memoria, descriptores de fichero, etc. Esa es la abstracción, aunque en realidad, el segmento de texto se suele compartir (dado que es de sólo lectura). El resto de segmentos se comportan como si no se compartiesen, pero UNIX hace que el padre y el hijo los compartan tras degradar sus permisos a sólo-lectura. Cuando un proceso intenta escribir en una de las páginas que "no comparten", UNIX comprueba que, en efecto, el proceso puede escribir en ella y hace una copia de la página que comparten padre e hijo. Como en este punto cada proceso tiene su propia copia de la página, los permisos vuelven a dejarse como lectura-escritura, y el proceso puede completar su escritura pensando que siempre ha tenido su propia copia. A esto se lo conoce como **copy on write**.

Un proceso para el kernel es, principalmente, el flujo de control (su juego de registros ya sea en el procesador o salvado en su pila de kernel y su pila, tanto la de usuario como la de kernel). El resto de recursos puede o no compartirlos con el proceso que lo ha creado.

A la vista de esto, resulta posible crear un nuevo proceso que comparta la memoria con el padre. A esto se lo suele denominar **thread**. El nombre procede de los tiempos en que el kernel no sabía nada de threads y éstos eran *procesos ligeros de usuario* o *corutinas* creadas por la librería de C (u otra similar) sin ayuda del kernel. Desde hace años, los threads son procesos como cualquier otro y el kernel los planifica como a cualquier otro proceso. Lo único que sucede es que algunos procesos comparten segmentos de memoria (y otros recursos) con otros procesos. Eso es todo. Si piensas en los threads como en procesos todo irá bien.

Resulta más cómodo (y es más portable) crear un proceso que comparte los recursos con el padre utilizando la *librería de pthreads* que está instalada en prácticamente todos los sistemas UNIX que utilizando la llamada al sistema involucrada (que suele además variar mucho de unos UNIX a otros).

El siguiente programa crea tres threads que imprimen 5 mensajes cada uno.

`[thr.c]`:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <err.h>
#include <string.h>
```

```
static void*
tmain(void *a)
{
    int i;
    char *arg, *sts;

    arg = a;
    for(i = 0; i < 5; i++) {
        write(2, arg, strlen(arg));
    }
    asprintf(&sts, "end %s", strchr(arg, 't'));
    free(arg);
    return sts;
}


int
main(int argc, char* argv[])
{
    int i;
    pthread_t thr[3];
    void *sts[3];
    char tabs[10], *a;

    for(i = 0; i < 3; i++) {
        memset(tabs, '\t', sizeof tabs);
        tabs[i] = 0;
        asprintf(&a, "%st %d\n", tabs, i);
        if(pthread_create(thr+i, NULL, tmain, a) != 0) {
            err(1, "thread");
        }
    }

    for(i = 0; i < 3; i++) {
        pthread_join(thr[i], sts+i);
        printf("join %d: %s\n", i, sts[i]);
        free(sts[i]);
    }
    exit(0);
}
```

El programa crea un nuevo thread utilizando código como

```
pthread_t thr;
...
if (pthread_create(&thr, NULL, func, funcarg) != 0) {
    err(1, "thread");
}
```

La llamada crea un proceso que comparte los recursos con el que hace la llamada y arregla las cosas para que el nuevo flujo de control ejecute `func(funcarg)`. Donde `func` es el *programa principal* o el punto de entrada del nuevo thread y ha de tener una cabecera como

```
void *func(void *arg)
```

El argumento que pasamos al final a `pthread_create` es el argumento con que se llamará a dicho punto

de entrada. Además, la función principal del thread devuelve un `void*` que hace las veces de *exit status* para el thread.

El primer argumento es un puntero a un **tid** o identificador de thread que utiliza la librería de threads para identificar al thread en cuestión. Utilizarás este valor en sucesivas llamadas que se refieran al thread que has creado. Es similar al *pid* de un proceso. Piensa que aunque cada thread tiene un proceso, la librería mantiene más información sobre cada thread y desea utilizar sus propios identificadores.

El programa puede después llamar a

```
void *sts;
...
pthread_join(thr, &sts);
```

para (1) esperar a que el thread termine y (2) obtener su estatus. Dicho de otro modo, `pthread_join` hace las veces de *wait(2)*.

Cuando no deseamos llamar a `pthread_join` para un thread, debemos informar a la librería *pthread* de tal cosa (¡Igual que sucedía con `fork` y `wait`!). Una buena forma de hacerlo es hacer que la función principal del thread llame a

```
pthread_t me;
me = pthread_self();
pthread_detach(me);
```

Pero observa que este código ejecuta en el nuevo thread, no en el código del proceso padre. La función `pthread_self` es como *getpid(2)*, pero devuelve el *thread id* y no el *process id*.

Si ahora vuelves a leer el programa seguramente entiendas las partes que antes te parecían oscuras. Cuando lo ejecutamos, podemos ver una salida parecida a esta:

```
unix$ thr
        t 2
        t 2
        t 2
        t 2
    t 1
t 0
    t 1
t 0
t 0
t 0
t 0
    t 1
    t 1
    t 1
        t 2
join 0: end t 0
join 1: end t 1
join 2: end t 2
unix$
```

Siendo procesos distintos... ¡No sabemos en qué orden van a ejecutar! Luego la salida seguramente difiera si repetimos la ejecución. Pero podemos ver que los tres threads ejecutan concurrentemente que que el programa principal puede esperar correctamente a que terminen y recuperar el estatus de salida de cada uno de ellos.

Seguramente resultará instructivo que ahora intentes leer de nuevo el programa trazando mentalmente cómo ha podido producir la salida que hemos visto.