Concurrencia básica

Sistemas Operativos

Enrique Soriano

GSYC

3 de diciembre de 2018

Universidad Rey Juan Carlos

Concurrencia

- Con lo que sabemos, ya podemos ver algunos problemas de concurrencia. P. ej:
  - ¿Qué pasa si dos procesos intentan escribir un fichero *a la vez*?
  - ¿Cómo se sincronizan dos procesos usando un pipe?

Condición de carrera

Tenemos varios flujos de ejecución realizando una tarea conjunta o usando recursos compartidos:

- **Condición de carrera**: el resultado depende de cual ejecute primero. No sabemos qué flujo ejecutará primero.
- Si la operación sobre el recurso compartido no es **atómica**, dos flujos pueden interferir entre ellos $\rightarrow$ resultado de las operaciones incorrecto.
- **Operación atómica**: operación indivisible, ninguna otra operación que realice otro flujo puede interferir con ella.

Condición de carrera

- Una **condición de carrera** es el peor bug posible.
- En general, no es reproducible.
- Ocurre con cierta probabilidad (puede ser muy baja).
- Los resultados pueden ser sorprendentes y **muy difíciles de entender**.

Sincronización

- Debemos sincronizar esos flujos de ejecución para evitar la condición de carrera y forzar un resultado correcto.
- Existen distintos mecanismos para sincronizar flujos.
- Uno de los más básicos es el *cierre* o lock.

Exclusión mutua

- Un cierre proporciona **exclusión mutua**.
- **Exclusión mutua**: cuando un flujo está dentro, no puede entrar ningún otro.
- En la **región crítica** es región del programa donde se debe proporcionar exclusión mutua para evitar una colisión al usar un recurso compartido.

Cierres

Un cierre común se comporta así:

- Para usar el recurso compartido, hay que tener cogido el cierre.
- Si un cierre está libre, se puede coger.
- Si se intenta coger un cierre cogido, no se podrá: el flujo se queda bloqueado hasta que pueda cogerlo.
- Después de usar el recurso compartido, siempre se debe soltar el cierre.

Cierres lectores/escritores

Hay un tipo de cierre llamado cierre de lectores/escritores. Tiene dos tipos de operaciones de cierre:

- Los que van a leer pueden hacerlo concurrentemente (N lectores). Cogen el cierre en modo lectura.
- El que viene a escribir necesita estar solo (1 escritor). Coge el cierre en modo escritura.

Ficheros

Cuando desde varios procesos se opera con el mismo fichero, tenemos condiciones de carrera. Ejemplos:

- Cuando varios procesos quieren crear el mismo fichero, hay una condición de carrera.
- Cuando un proceso escribe en un fichero mientras que otros leen/escriben del mismo fichero, tenemos una condición de carrera. En general, no se garantiza atomicidad en las escrituras.

Open

- El modo `O_CREAT|O_EXCL` hace que la llamada open falle si existe el fichero que se quiere crear. En ese caso `errno` será `EEXIST`.
- Hay que tener cuidado con la posición (offset). Si siempre queremos escribir al final: `O_APPEND`. Si múltiples procesos añaden concurrentemente, todos los bytes escritos se escribirán al final, pero puede que los bytes del mismo `write` no terminen contiguos.

Flock

- `flock`: sirve para usar un **cierre de lectores/escritores** sobre el fichero.
  Tiene tres operaciones:
  - `LOCK_EX`: echa el cierre de escritores.
  - `LOCK_SH`: echa un cierre de lectores.
  - `LOCK_UN`: soltar el cierre que tienes.
- Se puede especificar que no sea bloqueante con `|LOCK_NB`. En ese caso, si no puedes coger el cierre la operación da error (no se bloquea).

```
int flock(int fd, int operation);
```

Threads

- Un thread (hilo) es la unidad mínima de utilización de CPU.
- Hay distintos modelos de threads.
- Por ahora, los podemos ver como procesos que comparten su memoria: TEXT, DATA, BSS.
- El estándar POSIX tiene su interfaz: pthreads. Cada sistema puede implementarla de una forma distinta.
  `man 7 pthreads`

Threads

- Los threads comparten las variables globales.
- Dogma: cada vez que toquemos un recurso compartido (p. ej. una variable compartida), nos tenemos que proteger.
- Si no hacemos eso, nos metemos en problemas.
  La programación concurrente es muy difícil.

Memoria compartida: condición de carrera

Ejemplo: suponiendo que las líneas de este programa son atómicas[1] y la variable x (inicializada a 0) está compartida entre dos flujos de control (A y B) que ejecutan este código, ¿cuáles son los posibles valores finales de la variable x?

```
1:          int i, aux;
2:          for(i=0; i<10; i++){
3:               aux = x;
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

[1] En realidad, no lo son.

Pthreads

- `pthread_create`: crea un thread, que comenzará ejecutando la función que se le pasa en su tercer parámetro. El primer parámetro es un puntero a la variable de tipo `pthread_t` que identificará al thread creado.
- `pthread_join`: espera a que muera el thread indicado en su primer parámetro.

Debemos enlazar el programa con la biblioteca:

```
gcc -c -Wall -Wshadow -g miprograma.c
gcc -o miprograma miprograma.o -lpthread
```

Memoria compartida: condición de carrera

```
int i = 0;

void *
fn(void *p)
{
    i++;  // RACE
    fprintf(stderr, "FN: i is %d\n", i);
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t  thread;

    if(pthread_create(&thread, NULL, fn, NULL)) {
        warn("error creating thread");
        return 1;
    }
    i++;   // RACE
    fprintf(stderr, "MAIN: i is %d\n", i);

    if(pthread_join(thread, NULL)  != 0){
        warn("error joining thread");
        return 1;1
    }
    return 0;
}
```

Mutex

- En pthreads tenemos cierres de distintos tipos. Veremos **mutex**.
- El tipo de datos se llama `pthread_mutex_t`.
- `pthread_mutex_init`: inicializa el cierre. Si el segundo parámetro se usa para establecer ciertos atributos, si es NULL se ponen los atributos por omisión.
- `pthread_mutex_lock`: coge el cierre.
- `pthread_mutex_unlock`: suelta el cierre.

Mutex

```
int i = 0;
pthread_mutex_t  mutex;

void *
fn(void *p)
{
    pthread_mutex_lock(&mutex);
    i++;
    fprintf(stderr, "FN: i is %d\n", i);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

Mutex

(continúa)

```
int
main(int argc, char *argv[])
{
    pthread_t  thread;
    pthread_mutex_init(&mutex);

    if(pthread_create(&thread, NULL, fn, NULL)) {
        warn("error creating thread");
        return 1;
    }

    pthread_mutex_lock(&mutex);
    i++;
    fprintf(stderr, "MAIN: i is %d\n", i);
    pthread_mutex_unlock(&mutex);

    if(pthread_join(thread, NULL)  != 0){
        warn("error joining thread");
        return 1;1
    }
    return 0;
}
```

Otros mecanismos de sincronización

Además de los cierres, existen muchos otros mecanismos de sincronización con distinta semántica:

- Semáforos
- Barreras
- Rendezvous
- Variables condición y monitores
- Canales
- ...

Si vamos a hacer programas concurrentes, tenemos que estudiar detenidamente los mecanismos ofrece el sistema en el que trabajamos y programar con **mucho cuidado**.