Concurrent programming

François Trahay

Contents

Introduction

- Content of this lecture
  - discovering existing synchronization mechanisms
    * inter-process synchronization
    * intra-process synchronization

- studying classic synchronization patterns

Inter-process synchronisation

- IPC: *Inter Process Communication*
  - based on IPC objects in the OS
  - usage: usually via an entry in the filesystem
  - provides data persistence

Pipes

- Special files managed in FIFO
  - Anonymous pipes
    - `int pipe(int pipefd[2]);`
      - creates a pipe accessible by the current process
      - also accessible to future child processes
      - `pipefd[0]` for reading, `pipefd[1]` for writing
  - Named pipes
    - `int mkfifo(const char *pathname, mode_t mode);`
    - creates an entry in the filesystem accessible by any process
  - Use (almost) like a "regular" file
    - blocking reading
    - `lseek` is impossible

You have already handled pipes without necessarily realizing it: in `bash`, the sequence of commands linked by *pipes* is done via anonymous pipes created by the `bash` process.

So when we run `cmd1 | cmd2 | cmd3`, `bash` creates 2 anonymous pipes and 3 processes, then redirects (thanks to the `dup2` system call, see Lecture #11) standard input and output of processes to the different tubes.

Shared memory

- Allows you to share certain memory pages between several processes
  - Creating a zero-byte shared memory segment:
    - `int shm_open(const char *name, int oflag, mode_t mode);`
    - `name` is a key of the form `/key`
  - Changing the segment size:
    - `int ftruncate(int fd, off_t length);`
  - Mapping the segment into memory:

    * `void *mmap(void *addr, size_t length, int prot, int flags,   int fd, off_t offset);`
    * `flags` must contain `MAP_SHARED`

We will see later (during lecture 11 on I/O) another use of `mmap`.

Semaphore

- Object consisting of a value and a waiting queue
- Creating a semaphore:
  - named semaphore: `sem_t *sem_open(const char *name, int oflag,     mode_t mode, unsigned int value);`
    * `name` is a key of the form `/key`
  - anonymous semaphore: `int sem_init(sem_t *sem, int pshared, unsigned int value);`
    * if `pshared != 0`, ca be used by several processes (using a shared memory segment)
- Usage:
  - `int sem_wait(sem_t *sem);`
  - `int sem_trywait(sem_t *sem);`
  - `int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);`
  - `int sem_post(sem_t *sem);`

Intra-process synchronisation

- Based on shared objects in memory
- Possible use of IPC

Mutex

- Ensures mutual exclusion
- Type: `pthread_mutex_t`
- Initialisation:
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
  - `int pthread_mutex_init(ptread_mutex_t *m, const pthread_mutexattr_t *attr);`
- Usage:
  - `int pthread_mutex_lock(pthread_mutex_t *mutex));`
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

- Destroying a mutex:
  - `int pthread_mutex_destroy(pthread_mutex_t *mutex);`

Inter-process mutex

It is possible to synchronize threads from several processes with a `pthread_mutex_t` if it is in a shared memory area. For this, it is necessary to position the `PTHREAD_PROCESS_SHARED` attribute of the mutex with the function `int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);`

Monitors

- Allows you to wait for a condition to occur
- Consists of a mutex and a condition
- Example:

```
pthread_mutex_lock(&l);
  while(!condition) {
    pthread_cond_wait(&c, &l);
  }
  process_data();
pthread_mutex_unlock(&l);
```

```
pthread_mutex_lock(&l);
  produce_data();
  pthread_cond_signal(&c);
pthread_mutex_unlock(&l);
```

Here are the prototypes of the functions associated with the conditions:

- `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`
- `int pthread_cond_destroy(pthread_cond_t *cond);`
- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - waits for a condition to occur.
- `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
- `int pthread_cond_signal(pthread_cond_t *cond);`
  - unblocks a thread waiting for the condition
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - unblocks all threads waiting for the condition

The mutex ensures that between testing for the condition (`while (! condition)`) and wait (`pthread_cond_wait()`), no thread performs the condition.

Inter-process monitors

To synchronize multiple processes with a monitor, it is necessary to set the following attributes:

- The attribute `PTHREAD_MUTEX_SHARED` of the mutex (using `int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared)`).
- The attribute `PTHREAD_PROCESS_SHARED` of the condition (using `int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared)`).

Barrier

- Allows you to wait for a set of threads to reach *rendez-vous* point
  - Initialisation:
  - `int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *restrict attr, unsigned count);`
- Waiting:
  - `int pthread_barrier_wait(pthread_barrier_t *barrier);`
    - block until `count` threads reach `pthread_barrier_wait`
    - unblock all `count` threads

Once all the threads have reached the barrier, they are all unblocked and `pthread_barrier_wait` returns 0 except for one thread which returns `PTHREAD_BARRIER_SERIAL_THREAD`.

Inter-process barrier

To synchronize threads from multiple processes with a barrier, it is necessary to set the attribute `PTHREAD_PROCESS_SHARED` with `int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);`

Read-Write lock

- Type: `pthread_rwlock_t`
- `int pthread_rwlock_rdlock(pthread_rwlock_t* lock)`

    - Lock in read-mode
    - Possibility of several concurrent readers
- `int pthread_rwlock_wrlock(pthread_rwlock_t* lock)`
    - Lock in write-mode
    - Mutual exclusion with other writers and readers
- `int pthread_rwlock_unlock(pthread_rwlock_t* lock)`
    - Release the lock ::: notes :::

Classic synchronization patterns

- Goals
    - Being able to identify classic patterns
    - Implement these patterns with proven methods

In the literature, these problems are usually solved by using semaphores. This is because these problems have been theorized in the 1960s and 1970s by Dijkstra based on semaphores. In addition, semaphores have the advantage of being able to be used for inter-process synchronizations or intra-process.

However, modern operating systems implement many synchronization primitives which are much more efficient than semaphores. In the next slides, we will therefore rely on these mechanisms rather than semaphores.

Mutual exclusion synchronization pattern

- Allows concurrent access to a shared resource
- Principle:
    - Mutex `m` initialized
    - Primitive `mutex_lock(m)` at the start of the critical section
    - Primitive `mutex_unlock(m)` at the end of the critical section
    - Example:
        * mutex `m` initialized

```
   Prog1
mutex_lock(m)
 x=read (account)
 x = x + 10
 write (account=x)
mutex_unlock(m)
```

```
   Prog2
mutex_lock(m)
 x=read (account)
 x = x - 100
```

```
 write(account=x)
mutex_unlock(m)
```

Intra-process implementation

In a multi-threaded process, we just need to use a mutex of type `pthread_mutex_t`.

Inter-process implementation

To implement a mutual exclusion between several processes, several solutions exist: * using a `pthread_mutex_t` in a shared memory segment between processes. For this, it is necessary to set the attribute `PTHREAD_MUTEX_SHARED` in the mutex (using `pthread_mutexattr_setpshared`); * using a semaphore initialized to 1. The entry in section critical is protected by `sem_wait`, and we call `sem_post` when leaving the critical section.

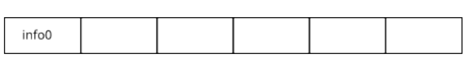
Cohort synchronization pattern

- Allows the cooperation of a group of a given maximum size
- Principle:
  - A counter initialized to `N`, and a monitor `m` to protect the counter
  - Decrement the counter at the start when needing a resource
  - Increment the counter at the end when releasing the resource

```
  Prog Vehicule
...
mutex_lock(m);
while(cpt == 0){ cond_wait(m);  }
cpt--;
mutex_unlock(m);
|...
mutex_lock(m);
cpt++;
cond_signal(m);
mutex_unlock(m);
```

Producer / Consumer synchronization pattern

- One or more threads produce data
- One or more threads consume the data produced
- Communication via a `N` blocks *buffer*
  - Executing `Produc`: produces `info0`



- Executing `Produc`: produces `info1`

info0	info1				

- Executing `Conso`: consumes `info0`

	info1				

- Executing `Produc`: produces `info2`

	info1	info2			

Implementation of a Producer / Consumer pattern

- A `available_spots` monitor initialized to `N`
- A `ready_info` monitor initialized to `0`

```
Producer:
repeat
...

mutex_lock(available_spots);
  while(available_spots<=0)
    cond_wait(available_spots);
  reserve_slot();
mutex_unlock(available_spots);

calcul(info)

mutex_lock(ready_info);
  push(info);
  cond_signal(ready_info);
mutex_unlock(ready_info);
...
endRepeat
```

```
Consumer:
repeat
 ...

    mutex_lock(ready_info);
      while(ready_info<=0)
        cond_wait(ready_info);
      extract(info)
    mutex_unlock(ready_info);

 mutex_lock(available_spots);
   free_slot();
   cond_signal(available_spots)
 mutex_unlock(available_spots);

 ...
 endRepeat
```

Inter-process Producer / Consumer

It is of course possible to implement a producer / consumer scheme between processes using conditions and mutexes. Another simpler solution is to use a pipe: since writing in a pipe being atomic, the deposit of a data boils down to writing into the pipe, and reading from the pipe extracts the data.

Reader / Writer pattern

- Allow a coherent competition between two types of process:
  - the "readers" can simultaneously access the resource

- the “writers” access the resource in mutual exclusion with other readers and writers

---

Implementation of a Reader / Writer synchronization pattern

- Use a `pthread_rwlock_t`
  - `int pthread_rwlock_rdlock(pthread_rwlock_t* lock)` to protect read operations
  - `int pthread_rwlock_wrlock(pthread_rwlock_t* lock)` to protect write operations
  - `int pthread_rwlock_unlock(pthread_rwlock_t* lock)` to release the lock

Implementation with a mutex It is possible to implement the reader / writer synchronization pattern using a mutex instead of `rwlock`: read and write operations are protected by a mutex. However, this implementation does not not allow multiple readers to work in parallel.

Implementation with a monitor The implementation of the monitor-based reader / writer is more complex. It mainly requires: * an integer `readers` which counts the number of threads reading * a boolean `writing` which indicates that a thread is writing * a `cond` condition to notify changes to these variables * a mutex `mutex` to protect concurrent access

Here is an implementation of the reader / writer using a monitor:

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

// This program simulates operations on a set of bank accounts
// Two kinds of operations are available:
// - read operation: compute the global balance (ie. the sum of all accounts)
// - write operation: transfer money from one account to another
//
// Here's an example of the program output:
//
// $ ./rw_threads_condition
// Balance: 0 (expected: 0)
// 3982358 operation, including:
//         3581969 read operations (89.945932 % )
//         400389 write operations (10.054068 % )

#define N 200
```

```c
int n_loops = 1000000;
int accounts[N];

int nb_read = 0;
int nb_write = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int readers=0;
int writing=0;

/* read all the accounts */
int read_accounts() {
  pthread_mutex_lock(&mutex);
  while(writing)
    pthread_cond_wait(&cond, &mutex);
  readers++;
  pthread_mutex_unlock(&mutex);

  nb_read++;
  int sum = 0;
  for(int i=0; i<N; i++) {
    sum += accounts[i];
  }

  pthread_mutex_lock(&mutex);
  readers--;
  if(!readers) {
    pthread_cond_signal(&cond);
  }
  pthread_mutex_unlock(&mutex);
  return sum;
}

/* transfer amount units from account src to account dest */
void transfer(int src, int dest, int amount) {
  pthread_mutex_lock(&mutex);
  while(writing || readers)
    pthread_cond_wait(&cond, &mutex);
  writing = 1;
  pthread_mutex_unlock(&mutex);

  nb_write++;
  accounts[dest] += amount;
  accounts[src] -= amount;
```

```c
  pthread_mutex_lock(&mutex);
  writing=0;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}

void* thread_function(void*arg) {
  for(int i=0; i<n_loops; i++) {

    /* randomly perform an operation
     * threshold sets the proportion of read operation.
     * here, 90% of all the operations are read operation
     * and 10% are write operations
     */
    int threshold = 90;
    int x = rand()%100;
    if(x < threshold) {
      /* read */
      int balance = read_accounts();
      if(balance != 0) {
        fprintf(stderr, "Error : balance = %d !\n", balance);
        abort();
      }
    } else {
      /* write */
      int src = rand()%N;
      int dest = rand()%N;
      int amount = rand()%100;
      transfer(src, dest, amount);
    }
  }
  return NULL;
}

int main(int argc, char**argv) {
  for(int i = 0; i<N; i++) {
    accounts[i] = 0;
  }

  int nthreads=4;
  pthread_t tid[nthreads];

  for(int i=0; i<nthreads; i++) {
    pthread_create(&tid[i], NULL, thread_function, NULL);
```

```c
  }

  for(int i=0; i<nthreads; i++) {
    pthread_join(tid[i], NULL);
  }

  int balance = read_accounts();
  printf("Balance: %d (expected: 0)\n", balance);

  int nb_op = nb_read+nb_write;
  printf("%d operation, including:\n",nb_op);
  printf("\t%d read operations (%f %% )\n", nb_read, 100.*nb_read/nb_op);
  printf("\t%d write operations (%f %% )\n", nb_write, 100.*nb_write/nb_op);

  return EXIT_SUCCESS;
}
```