

# Concurrent programming

François Trahay



CSC4508 – Operating Systems  
2022–2023

# 1 Introduction

Content of this lecture:

- discovering existing synchronization mechanisms
  - ◆ inter-process synchronization
  - ◆ intra-process synchronization
- studying classic synchronization patterns

## 2 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

## 2.1 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

## 2.2 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`

## 2.3 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`, can 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);`

## 3 Intra-process synchronisation

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

## 3.1 Mutex

- Ensures mutual exclusion
- Type: `pthread_mutex_t`
- Initialisation:
  - ◆ `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
  - ◆ `int pthread_mutex_init(pthread_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);`



## 3.2 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);
```

## 3.3 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

## 3.3.1 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

## 4 Classic synchronization patterns

### Goals

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

## 4.1 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	Prog2
<code>mutex_lock(m)</code>	<code>mutex_lock(m)</code>
<code>  x=read (account)</code>	<code>  x=read (account)</code>
<code>  x = x + 10</code>	<code>  x = x - 100</code>
<code>  write (account=x)</code>	<code>  write(account=x)</code>
<code>mutex_unlock(m)</code>	<code>mutex_unlock(m)</code>

## 4.2 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);
```

## 4.3 Producer / Consumer synchronization pattern

- One or more threads produce data
- One or more threads consume the data produced
- Communication via a  $N$  blocks *buffer*

1. Executing Produc : produces info0

info0				
-------	--	--	--	--

2. Executing Produc : produces info1

info0	info1			
-------	-------	--	--	--

3. Executing Conso : consumes info0

	info1			
--	-------	--	--	--

4. Executing Produc : produces info2

	info1	info2		
--	-------	-------	--	--

## 4.3.1 Implementation of a Producer / Consumer pattern

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

Producer:	Consumer:
repeat	repeat
...	...
<code>mutex_lock(available_spots);</code>	<code>mutex_lock(ready_info);</code>
<code>while(available_spots&lt;=0)</code>	<code>while(ready_info&lt;=0)</code>
<code>cond_wait(available_spots);</code>	<code>cond_wait(ready_info);</code>
<code>reserve_slot();</code>	<code>extract(info)</code>
<code>mutex_unlock(available_spots);</code>	<code>mutex_unlock(ready_info);</code>
<code>calcul(info)</code>	<code>mutex_lock(available_spots);</code>
	<code>free_slot();</code>
<code>mutex_lock(ready_info);</code>	<code>cond_signal(available_spots)</code>
<code>push(info);</code>	<code>mutex_unlock(available_spots);</code>
<code>cond_signal(ready_info);</code>	
<code>mutex_unlock(ready_info);</code>	...
...	<code>endRepeat</code>
<code>endRepeat</code>	



## 4.4 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

## 4.4.1 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