Synchronization

Real-Time Operating Systems Programming (RTOSP)
Master in Critical Computing Systems Engineering (MCCSE)

2022/23

Paulo Baltarejo Sousa and Cláudio Maia
{pbs,crr}@isep.ipp.pt

Instituto Superior de Engenharia do Porto
P.PORTO

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

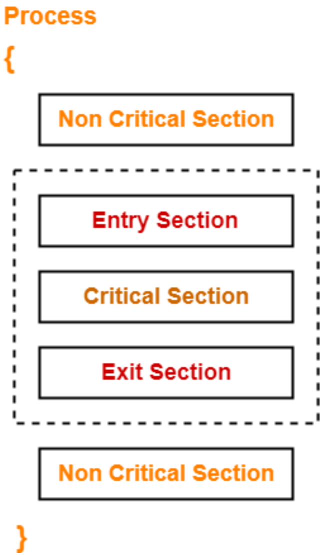
- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

Semaphores

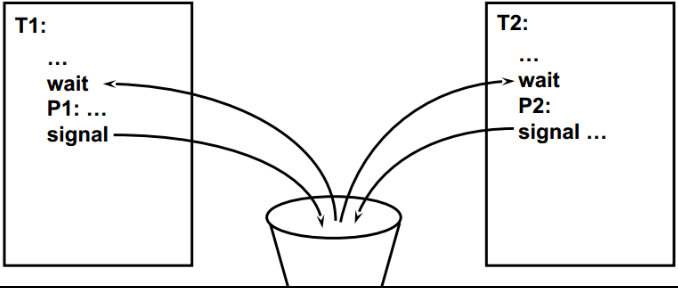
Overview (I)

- We face a **critical section problem** whenever multiple processes access the **critical section** concurrently.
    - The **critical section** refers to the segment of code where processes access shared resources, such as common variables and files, and perform write operations on them.
- To solve the **critical section problem** we design a protocol according to the rules:
    - For example: when one process has entered its critical section, no other process is allowed to execute in its critical section.



Overview (II)

- Conceptually, semaphores are integer variables that are used **to solve the critical section problem** by using two atomic operations:
  - **Wait**:
    - It decrements the value of semaphore, if it is positive.
    - If the value of semaphore is zero, then no operation is performed until a signal operation be performed.
  - **Signal**:
    - The signal operation increments the value of semaphore.



Overview (II)

- Types of Semaphores
  - **Counting** Semaphores
    - These are integer value semaphores and have an unrestricted value domain.
    - These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources.
  - **Binary** Semaphores
    - The binary semaphores are like counting semaphores but their value is restricted to 0 and 1.
    - The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0.

Process Synchronization

- Semaphores are a **synchronization mechanism** used to coordinate the activities of multiple processes in a computer system.
- They are used to enforce **mutual exclusion**, avoid race conditions and implement synchronization between processes.
- Semaphores are used to **implement critical sections**.
- By using semaphores, processes can coordinate access to shared resources, such as shared memory or I/O devices.

POSIX Semaphores

Overview (I)

- POSIX semaphores allow **processes and threads to synchronize their actions**.
- A semaphore is **an integer whose value is never allowed to fall below zero**.
- Two operations can be performed on semaphores:
  - **Increment** the semaphore value by one (`sem_post`);
  - **Decrement** the semaphore value by one (`sem_wait`).
    - If the value of a semaphore is currently zero, then a `sem_wait` operation will **block** until the value becomes greater than zero.

Named semaphores

- A named semaphore is identified by a **name** of the form `/somename`.
    - A null-terminated string of up to 251 characters consisting of an initial slash, followed by one or more characters, none of which are slashes.
- Two processes **can operate on the same named semaphore by passing the same name** to `sem_open`.
- The `sem_open` function creates a new named semaphore or opens an existing named semaphore.
- After the semaphore has been opened, it can be operated on using `sem_post` and `sem_wait`.
- When a process has finished using the semaphore, it can use `sem_close` to close the semaphore.
- When all processes have finished using the semaphore, it can be removed from the system using `sem_unlink`.

Unnamed semaphores (memory-based semaphores)

- An unnamed semaphore does not have a name.
- Instead the semaphore **is placed in a region of memory that is shared** between multiple threads (a thread-shared semaphore) or processes (a process-shared semaphore).
  - A thread-shared semaphore is placed in an area of memory shared between the threads of a process, for example, a global variable.
  - A process-shared semaphore must be placed in a shared memory region
- Before being used, an unnamed semaphore must be initialized using `sem_init`.
- It can then be operated on using `sem_post` and `sem_wait`.
- When the semaphore is no longer required, and before the memory in which it is located is deallocated, the semaphore should be destroyed using `sem_destroy`.

Semaphores API [1]

- `sem_open`: https://man7.org/linux/man-pages/man3/sem_open.3.html
- `sem_post`: https://man7.org/linux/man-pages/man3/sem_post.3.html
- `sem_wait`: https://man7.org/linux/man-pages/man3/sem_wait.3.html
  - `sem_trywait`
  - `sem_timedwait`
- `sem_close`: https://man7.org/linux/man-pages/man3/sem_close.3.html
- `sem_unlink`: https://man7.org/linux/man-pages/man3/sem_unlink.3.html

[1] https://man7.org/linux/man-pages/man7/sem_overview.7.html

Named Semaphore example (I)

```
char * name = "/my_semaphore";
int VALUE = 2;
sem_t * sem;

//If semaphore with name does not exist, then create it with VALUE
printf("Open or Create a named semaphore, %s, its value is %d\n", name,VALUE);
sem = sem_open(name, O_CREAT, 0666, VALUE);

//wait on semaphore sema and decrease it by 1
sem_wait(sema);
printf("Decrease semaphore by 1\n");

//add semaphore sema by 1
sem_post(sem);
printf("Add semaphore by 1\n");

//Before exit, you need to close semaphore and unlink it, when all processes have
//finished using the semaphore, it can be removed from the system using sem_unlink

sem_close(sem);
sem_unlink(name);
```

Named Semaphore example (II)

```
char * name = "/my_semaphore";
int VALUE = 0;
/* Create and open the semaphore */
sem_t *sem = sem_open(name, O_CREAT, 0666, VALUE);
/* Fork to create the child process */

pid_t child_pid = fork();
/* Note the child inherits a copy of the semaphore connection */

/* Child process: wait for semaphore, print "second", then exit */
if (child_pid == 0){
  sem_wait (sem);
  printf ("second\n");
  sem_close (sem);
  return 0;
}

/* Parent prints then posts to the semaphore and waits on child */
printf ("first\n");
sem_post (sem);

wait (NULL);
/* Now the child has printed and exited */
printf ("third\n");
sem_close (sem);
sem_unlink (name);
```

Semaphores API (more)

- `sem_init`: https://man7.org/linux/man-pages/man3/sem_init.3.html
- `sem_destroy`: https://man7.org/linux/man-pages/man3/sem_destroy.3.html

Unnamed Semaphore example

```
sem_t sem;
void * func(void * arg){
  //wait
  sem_wait(&sem);
  printf("Entered in the critical section\n");
  sleep5();
  printf("Leaving the critical section\n");
  //signal
  sem_post(&sem);
}
int main(){
  pthread_t t1, t2;
  sem_init(&sem, 0, 1);
  pthread_create(&t1, NULL, func, NULL);
  pthread_create(&t2, NULL, func, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  sem_destroy(&sem);
}
```

Mutexes

Overview

- A mutex is a **MUT**ual **EX**clusion mechanism, and is useful for protecting shared data structures from concurrent modifications, and implementing critical sections.
- A mutex has two possible states: **unlocked** (not owned by any thread), and **locked** (owned by one thread).
- A mutex can **never be owned by two different threads simultaneously**.
- A thread attempting to lock a mutex that is already locked by another thread **is blocked until** the owning thread unlocks the mutex first.

Mutexes *vs* Semaphores

Semaphore	**Mutex**
It is an integer	It is an object
Signaling mechanism	Locking mechanism
Multiple program threads can access a finite number of resources	Multiple program threads can access a single resource, but not at same time instant
Prefer to use for multiple resources	Prefer to use for single resource
Semaphore value **can be updated by any thread/process**	The only thread **can release the mutex which has acquired it**
Two types; Binary and counting	No types

Pthread Mutexes

PTHREAD_MUTEX

- Creating and Initializing
  - In order to create a mutex, we first need to declare a variable of type `pthread_mutex_t`, and then initialize it using `pthread_mutex_init` function or `PTHREAD_MUTEX_INITIALIZER` constant.
- Locking and unlocking
  - In order to lock a mutex, we use the function `pthread_mutex_lock`, which attempts to lock the mutex, or block the thread if the mutex is already locked by another thread.
  - After the thread did what it had to, it should free the mutex, using the `pthread_mutex_unlock` function
- Destroying
  - After we finished using a mutex, we should destroy it using the `pthread_mutex_destroy` function.

Mutexes API

- pthread_mutex_init: https://man7.org/linux/man-pages/man3/pthread_mutex_init.3p.html
- pthread_mutex_lock: https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html
- pthread_mutex_unlock: https://man7.org/linux/man-pages/man3/pthread_mutex_unlock.3p.html
- pthread_mutex_destroy: https://man7.org/linux/man-pages/man3/pthread_mutex_destroy.3p.html

Mutexes example

```c
pthread_mutex_t mut;
void * func(void * arg){
 //lock
 pthread_mutex_lock(&mut);
 printf("Entered in the critical section\n");
 sleep5();
 printf("Leaving the critical section\n");
 //unlock
 pthread_mutex_unlock(&mut);
}
int main(){
 pthread_t t1, t2;
 pthread_mutex_init(&mut, NULL);
 pthread_create(&t1, NULL, func, NULL);
 pthread_create(&t2, NULL, func, NULL);
 pthread_join(t1, NULL);
 pthread_join(t2, NULL);
 pthread_mutex_destroy(&mut);
}
```

Conditional variables

Overview

- A conditional variable in operating system programming is a special kind of **variable that is used to determine if a certain condition has been met or not**.
- It is used **to communicate between threads** when certain conditions become true.
- A conditional variable is like a queue.
- A thread stops its execution and enters the queue if the specified condition is not met.
- Once another thread makes that condition true, it sends a signal to the leading thread in the queue to continue its execution.

PTHREAD_COND (I)

- Creating and Initializing
  - Creation of a condition variable requires defining a variable of type `pthread_cond_t`, and initializing it properly.
  - Initialization may be done with either a simple use of a macro named `PTHREAD_COND_INITIALIZER` or the usage of the `pthread_cond_init` function.
- Signaling
  - In order to signal a condition variable, one should either the `pthread_cond_signal` function (to wake up a only one thread waiting on this variable), or the `pthread_cond_broadcast` function (to wake up all threads waiting on this variable).

PTHREAD_COND (II)

- Waiting
  - If one thread signals the condition variable, other threads would probably want to wait for this signal.
  - They may do using `pthread_cond_wait` function.
  - This function takes a condition variable, and a mutex (which should be locked before calling the wait function), unlocks the mutex, and waits until the condition variable is signaled, suspending the thread's execution.
  - If this signaling causes the thread to awake, the mutex is automaically locked again by the wait funciton, and the wait function returns.
- Destroying
  - After we are done using a condition variable, we should destroy it, to free any system resources it might be using. This can be done using the `pthread_cond_destroy`.
  - In order for this to work, there should be no threads waiting on this condition variable.

Conditional variables API

- pthread_cond_init: https://man7.org/linux/man-pages/man3/pthread_cond_init.3p.html
- pthread_cond_signal: https://man7.org/linux/man-pages/man3/pthread_cond_signal.3p.html
- pthread_cond_wait: https://man7.org/linux/man-pages/man3/pthread_cond_wait.3p.html
- pthread_cond_destroy: https://man7.org/linux/man-pages/man3/pthread_cond_destroy.3p.html

Conditional variables example (I)

```c
pthread_mutex_t mutexFuel ;
pthread_cond_t condFuel ; // creating the condition variable.
int fuel = 0;

void* fuelling(void* arg) {
 pthread_mutex_lock(&mutexFuel);
 fuel += 40 ;
 pthread_mutex_unlock(&mutexFuel) ;
 pthread_cond_signal(&condFuel) ;
}

void* vehicle(void* arg) {
 pthread_mutex_lock(&mutexFuel);
 while (fuel < 40) {
   pthread_cond_wait(&condFuel, &mutexFuel);
   // Equivalent to:
   // pthread_mutex_unlock(&mutexFuel);
   // wait for signal on condFuel
   // pthread_mutex_lock(&mutexFuel) ;
 }
 fuel -= 40 ;
 pthread_mutex_unlock(&mutexFuel);
}
```

Conditional variables example (II)

```
int main(int argc, char* argv[]) {
 pthread_ta[2] ;
 pthread_mutex_init(&mutexFuel, NULL) ;
 pthread_cond_init(&condFuel, NULL);

 if (pthread_create(&a[0], NULL, &fuelling, NULL) != 0) {
  perror("Failed to create thread");
 }

 if (pthread_create(&a[1], NULL, &vehicle, NULL) != 0) {
  perror("Failed to create thread");
 }

 for ( int i = 0 ; i < 2 ; i++ ) {
  if (pthread_join(a[i], NULL) != 0) {
   perror("Failed to join thread") ;
  }
 }
 pthread_mutex_destroy(&mutexFuel) ;
 pthread_cond_destroy(&condFuel) ; // destroying the threads.
 return 0 ;
}
```