Synchronization

Synchronization in POSIX

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

Introduction

Semaphores are a specific form of IPC, which means they are used to manage and synchronize access to shared resources between different processes.

- Semaphores are a very specific form of Inter-Process Communication
  - They correspond to a counter protected by a lock (i.e., a binary semaphore or lock) with a waiting queue
  - To implement a semaphore, manipulation functions must be atomic operations
    - For this reason, semaphores are normally implemented inside the kernel
  - It is possible to use synchronization strategies faster and less expensive than semaphores (e.g., mutexes)

Semaphores are a synchronization tool used to control access to a common resource in concurrent programming. They can be thought of as counters protected by locks, with an associated waiting queue.

Semaphores correspond to a counter that is protected by a lock. This can be thought of as a binary semaphore (which acts like a lock) or a general semaphore with a waiting queue.

Atomic Operations: Semaphore manipulation functions must be atomic, meaning they are performed without interruption, ensuring consistent access to the shared resource.

Binary Semaphore: A semaphore with only two states (0 and 1), often used as a simple lock.
General Semaphore: A semaphore that can have a range of values, used to control access to a resource pool.
Mutex (Mutual Exclusion): A simpler and often faster synchronization primitive compared to semaphores, used to ensure that only one thread or process accesses a resource at a time.

POSIX semaphores

- POSIX system calls are included in the header file
  - semaphore.h
- A semaphore is a type sem_t variable
  - sem_t *sem1, *sem2, ...;
- All semaphore system calls
  - Have name sem_*
  - On error, they return the value -1

System calls:
sem_init
sem_wait
sem_trywait
sem_post
sem_getvalue
sem_destroy

sem_init: Initialize a semaphore.
sem_wait: Wait (decrease) on a semaphore.
sem_trywait: Non-blocking wait on a semaphore.
sem_post: Signal (increase) a semaphore.
sem_getvalue: Get the current value of a semaphore.
sem_destroy: Destroy a semaphore.

sem_init

```
int sem_init (
   sem_t *sem,
   int pshared,
   unsigned int value
);
```

The sem_init function initializes a semaphore.

A pointer to the semaphore to be initialized.

pshared ,This parameter specifies whether the semaphore is shared between processes or threads. if non-zero it can be shared between different processes. (e.g. a parent process and it's child process)

- Initializes the semaphore counter at value **value**
- The value **pshared** identifies the semaphore type
  - If equal to `0`, the semaphore is local to the **threads of the current process**
  - Otherwise, the semaphore can be **shared between different processes** (parent that initializes the semaphore and its children)

Linux does not currently support shared semaphores

sem_wait

The sem_wait function is used to lock (decrement) the semaphore.

```
int sem_wait (
  sem_t *sem
);
```

When sem_wait is called, it attempts to decrement the semaphore's value.
If the semaphore's value is greater than 0, the decrement operation succeeds, and the function returns immediately.
If the semaphore's value is 0, the function blocks (waits) until the semaphore's value becomes greater than 0 (i.e., until another thread or process increments the semaphore).

- Standard wait
  - If the semaphore is equal to 0, it blocks the caller until it can decrease the value of the semaphore

sem_trywait

```
int sem_trywait (
  sem_t *sem
);
```

- Non-blocking wait
  - If the semaphore counter has a value greater than 0, perform the decrement, and returns 0
  - If the semaphore is equal to 0, returns -1 (instead of blocking the caller as **sem_wait** does)

sem_post

```
int sem_post (
  sem_t *sem
);
```

The sem_post function increments (unlocks) the semaphore. If there are any threads in sem_wait, one of them is woken up to proceed.

- Standard signal
  - Increments the semaphore counter, or wakes up a blocked thread if present

sem_getvalue

```
int sem_getvalue (
   sem_t *sem,
   int *valP
);
```

sem_getvalue allows obtaining the current value of the semaphore.

Better not use this function. From Linux manual: "The value of the semaphore may already have changed by the time sem_getvalue() returns"

- Allows obtaining the value of the semaphore counter
  - The value is assigned to *valP
  - If there are waiting threads
    - 0 is assigned to *valP (Linux)
    - or a negative number whose absolute value is equal to the number of processes waiting (POSIX)

The semaphore value is assigned to the integer pointed to by valp.

The value can be misleading if there are waiting threads:
On Linux, if there are waiting threads, it may return 0.
POSIX might return a negative number representing the number of waiting threads.

sem_destroy

```
int sem_destroy (
   sem_t *sem
);
```

Destroys the semaphore at the address pointed to by sem.
If there are threads currently blocked on the semaphore, destroying it can lead to undefined behavior.
Using a semaphore that has been destroyed without reinitializing it also leads to undefined results.

- Destroys the semaphore at the address pointed by sem
  - Destroying a semaphore that other threads are currently blocked on produces undefined behavior (on error, -1 is returned)
  - Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized

Esempio

Use of sem_* primitives to synchronize threads

#include <semaphore.h>: This includes the necessary header file for semaphore functions

```
#include "semaphore.h"

sem_t sem;
sem_init (&sem, 0, 0);

... create threads ...

sem_destroy (&sem);
```

Declares a semaphore variable statically.

Initializes the semaphore. The second parameter 0 indicates that the semaphore is shared between threads of the same process. The third parameter 0 sets the initial value of the semaphore.

Threads are created and will use the semaphore for synchronization.

sem_destroy(&sem);: Destroys the semaphore when it is no longer needed.

Static semaphore

```
sem_wait (&sem);
... SC ...
sem_post (&sem);
```

Static Semaphore:
Declared and initialized statically: sem_t sem; sem_init(&sem, 0, 0);
Usage includes waiting (sem_wait(&sem);) and signaling (sem_post(&sem);).
Semaphore is destroyed when no longer needed: sem_destroy(&sem);

Dynamic semaphore

```
#include "semaphore.h"

sem_t *sem;
sem = (sem_t *)
      malloc(sizeof(sem_t));
sem_init (sem, 0, 0);

... create threads ...

sem_destroy (sem);
```

allocated dynamically

used similarly as above for waiting signaling

Destroyed and memory freed when no longer needed.

```
sem_wait (sem);
... SC ...
sem_post (sem);
```

Dynamic Semaphore:
Allocated dynamically: sem_t *sem = (sem_t*)malloc(sizeof(sem_t));
sem_init(sem, 0, 0);
Used similarly for waiting and signaling.
Destroyed and memory freed when no longer needed: sem_destroy(sem);

pthread stands for POSIX (Portable Operating System Interface) threads. It is a standard for thread programming defined by the POSIX.1c standard (IEEE Std 1003.1c-1995).
pthread is a library in C/C++ used to create and manage threads in Unix-like operating systems (e.g., Linux, macOS).
The pthread library provides a set of functions for thread creation, synchronization, and management. These functions include pthread_create, pthread_join, pthread_mutex_lock, pthread_cond_wait, and many others.

Pthread mutex

A mutex (mutual exclusion) is a locking mechanism used to protect shared resources in concurrent programming.

- A **mutex** is basically a lock that we
  - Set (lock) before accessing a shared resource
    - While it is set, any other thread that tries to set it will block until we release it
  - Release (unlock) when we are done
    - If more than one thread is blocked when we unlock the mutex, then all threads blocked on the lock will be made runnable, and the first one to run will be able to set the lock

Before accessing a shared resource, the mutex must be locked.

While locked, any other thread trying to lock it will be blocked until the mutex is unlocked.
After finishing with the shared resource, the mutex must be unlocked to allow other threads to proceed.

Classical Critical Section Protocol:

Reservation code to lock the mutex.
Critical section code to access shared resource.
Release code to unlock the mutex.
Non-critical section code where no mutex is needed.

Classical Critical Section protocol

```
while (TRUE) {
  ...
  reservation code
  Critical Section
  release code
  ...
  non critical section
}
```

Pthread mutex

Mutexes are essentially binary semaphores with only two states: locked (1) or unlocked (0).

- Mutexes are essentially
  - **Binary**, **local** and **fast** semaphores
- A mutex
  - Is a variable represented by the **pthread_mutex_t** data type
  - It can be managed using the following system calls
    - pthread_mutex_init
    - pthread_mutex_lock
    - pthread_mutex_trylock
    - pthread_mutex_unlock
    - pthread_mutex_destroy

A mutex is less general than semaphores (i.e., it can assume only the two values 0 or 1)

pthread_mutex_init

Static Initialization: Using PTHREAD_MUTEX _INITIALIZER for statically allocated mutexes.

Dynamic Initialization: Using pthread_mutex_init, especially if the mutex is dynamically allocated (e.g., using malloc).

- Before we can use a mutex variable, we must first initialize it by
  - Either setting it to the constant PTHREAD_MUTEX_INITIALIZER, for statically allocated mutexes only and default attributes
  - Calling **pthread_mutex_init**, if we allocate the mutex dynamically (e.g., by calling **malloc**) or we want to set specific attributes
    - If we dinamically allocate it, then we need to call **pthread_mutex_destroy** before freeing the memory (i.e., by calling **free**)

pthread_mutex_init

The pthread_mutex_init function initializes a mutex.

```
int pthread_mutex_init (
  pthread_mutex_t *mutex,
  const pthread_mutexattr_t *attr
);
```

- A pointer to the mutex to be initialized.

- A pointer to a mutex attributes object. If you want to initialize the mutex with default attributes, you can set this to NULL.

Always include pthread.h

- Initializes the mutex referenced by **mutex** with attributes specified by **attr**
  - To initialize a mutex with the default attributes, we set **attr** to NULL
- Return value
  - The value, 0 on success
  - An error code, otherwise

pthread_mutex_lock

```
int pthread_mutex_lock (
  pthread_mutex_t *mutex
);
```

The pthread_mutex_lock function locks a mutex, i.e., it controls access to a shared resource.

- Lock a mutex, i.e., control its value and
  - Blocks the caller if the mutex is locked
  - Acquire the mutex lock if the mutex is unlocked
- Return value
  - The value 0, on success
  - An error code, otherwise

If the mutex is already locked by another thread, pthread_mutex_lock will block the calling thread until the mutex becomes available.
If the mutex is not locked, it will lock the mutex and return immediately.

pthread_mutex_trylock

can't afford to be blocked*

- If a thread can't afford to block, it can use pthread_mutex_trylock to lock the mutex conditionally
  - If the mutex is unlocked at the time **pthread_mutex_trylock** is called, then pthread_mutex_trylock will lock the mutex without blocking and return 0
  - Otherwise, **pthread_mutex_trylock** will fail, returning EBUSY without locking the mutex

If the mutex is already locked by another thread, pthread_mutex_lock will block the calling thread until the mutex becomes available.
If the mutex is not locked, it will lock the mutex and return immediately.

pthread_mutex_trylock

```
int pthread_mutex_trylock (
  pthread_mutex_t *mutex
);
```

- Similar to pthread_mutex_lock, but returns without blocking the caller if the mutex is locked
- Return value
  - The value 0, if the lock has been successfully acquired
  - **EBUSY** error if the mutex was already locked by another thread

If the mutex is not locked, pthread_mutex_trylock will lock it and return 0.
If the mutex is already locked, it will return immediately with the error code EBUSY (indicating that the mutex is busy).

pthread_mutex_unlock

```
int pthread_mutex_unlock (
  pthread_mutex_t *mutex
);
```

The pthread_mutex_unlock function releases (unlocks) a mutex that was previously locked by the calling thread.

- Release (unlock) the mutex lock (typically at the end of a sritical section)
- Return value
  - The value 0, on success
  - An error code, otherwise

This function unlocks the mutex, making it available for other threads to lock.
Typically, this function is called at the end of a critical section to allow other threads to access the shared resource.

pthread_mutex_destroy

```
int pthread_mutex_destroy (
  pthread_mutex_t *mutex
);
```

This function is used to free the memory associated with a dynamically allocated mutex.
After calling this function, the mutex cannot be used until it is reinitialized.

- Free **mutex** memory
  - Used for dynamically allocated mutexes
  - The mutex cannot be used anymore
- Return value
  - The value 0, on success
  - An error code, otherwise

Example

Use a dynamically allocated mutex to protect a CS

```
pthread_mutex_t *lock;
...
lock = malloc (1 * sizeof (pthread_mutex_t));
if (lock == NULL) ... error ...
if (pthread_mutex_init (lock, NULL) != 0) ... error ..

...
pthread_mutex_lock (lock);
CS
pthread_mutex_unlock(lock);
...

pthread_mutex_destroy (lock)
free (lock);
```

pthread_mutex_t *lock;: Declares a pointer to a mutex.

Allocates memory for the mutex.

Checks if the allocation was successful. If not, handle the error.

Initializes the mutex with default attributes.

Checks if the initialization was successful. If not, handle the error.

Locks the mutex before entering the critical section.

Unlocks the mutex after exiting the critical section.

Destroys the mutex.

Frees the dynamically allocated memory for the mutex.