Mutex and Semaphores

Semaphore

- A **semaphore** is somewhat like an integer variable, but is special in that its operations (increment and decrement) are guaranteed to be **atomic**—you cannot be halfway through incrementing the semaphore and be interrupted and waylaid by another thread trying to do the same thing.
- By convention, when a semaphore is zero it is "locked" or "in use".
- Positive values indicate that the semaphore is available.
- A semaphore will never have a negative value.
- Semaphores which allow an arbitrary resource count are called **counting semaphores**, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** and are used to implement locks.
- Semaphores provide two operations: **wait (P)** and **signal (V)**. The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore.
- When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.
- Operations for accessing and changing value of semaphore variable.

```
P(semaphore s) {
// wait - P
while (s==0); // wait until s=0.
s = s-1;
}

V(semaphore s) {
// signal - V
s = s+1;
}
```

- P operation is called the wait, sleep or down operation.
- V operation is called signal, wake-up or up operation.
- Both operations are atomic and semaphore(s) is always initialized to 1.
- Both of the above operaitons are atomic.
- In atomic operations, in-between read, modify and update no other operation is performed that may change the variable.

- A critical section is surrounded by both operations to implement process synchronization.

```
// Some code
P(s);
// critical section

V(s);
// remainder section
```

POSIX Semaphores

Manpage

```
man sem_overview
```

POSIX semaphores allow processes and threads to synchronize their

actions.

Two operations can be performed on semaphores:

- increment the semaphore value by one (sem_post(3)) and
- decrement the semaphore value by one (sem_wait(3)).

If the value of a semaphore is currently zero, then a sem_wait(3) operation will block until the value becomes greater than zero.

POSIX semaphores come in two forms: named semaphores and unnamed

semaphores.

Mutex

- Stands for MUTual EXclusion.
- It is basically an integer value in global memory.
- They are implemented using atomic operations.
- Atomic operations are instructions within the processor that are impossible to interrupt. No one is able to access the memory targeted by the instruction until the instruction completes execution.

POSIX Mutexes

Manpage

```
sudo apt install glibc-doc
man pthread_mutex_lock
```

pthread_mutex_lock(), pthread_mutex_unlock(), pthread_mutex_init(), pthread_mutex_destroy() are basic funcitons used for POSIX implementation of mutex.

Semaphores vs. Mutexes

- A mutex is a locking mechanism that sometimes uses the same basic implementation as the binary semaphore.
- The differences between them are in how they are used.
- While a binary semaphore may be colloquially referred to as a mutex, a true mutex has a more specific use-case and definition, in that only the task that locked the mutex is supposed to unlock it.
- Mutexes have a concept of ownership the thread that locks it is supposed to unlock it.
- Mutex is SUPPOSED TO be unlocked by the thread/process that has locked it. If another thread/process unlcks it, it results in undefined behaviour and can cause the program to break unexpectedly.

(see program 6-bin_sem_vs_mutex.c)

- This constraint aims to handle some potential problems of using semaphores:

1. Priority inversion: If the mutex knows who locked it and is supposed to unlock it, it is possible to promote the priority of that task whenever a higher-priority task starts waiting on the mutex.
2. Premature task termination: Mutexes may also provide deletion safety, where the task holding the mutex cannot be accidentally deleted.
3. Termination deadlock: If a mutex-holding task terminates for any reason, the OS can release the mutex and signal waiting tasks of this condition.
4. Recursion deadlock: a task is allowed to lock a reentrant mutex multiple times as it unlocks it an equal number of times.
5. Accidental release: An error is raised on the release of the mutex if the releasing task is not its owner.

(Source: Wikipedia)

References

- Semaphores in Process Synchronization - GFG
- Semaphores

- Semaphore - Wikipedia
- Semaphores - Purdue
- Semaphores - Oracle
- Threads and Semaphores Examples
- IPC and semaphores
- Mutex unlock
- Mutex concept and Working - Low level Learning