Exploration: Semaphores

Introduction

We have already studied the use of Pthreads mutexes to synchronize access of multiple threads in a process to a shared resource. POSIX semaphores provide another mechanism that allows threads to synchronize their actions. However, in addition to threads, semaphores can also be used by multiple processes to synchronize their actions. Additionally, semaphores support the implementation of more complex synchronization between processes or threads that goes beyond mutual exclusion.

Basic Concepts

A semaphore is an integer whose value is always 0 or more. To synchronize actions, semaphores provide two fundamentals operations:

1. Decrement the value of the semaphore by one. This operation succeeds only if the current value of the semaphore is greater than 0. This is called **locking the semaphore**. But if the current value of the semaphore is 0, then the decrement operation blocks until the value becomes greater than 0.
2. Increment the value of the semaphore by one. This is called **unlocking the semaphore**.

Semaphore API

Semaphores can be named or unnamed. In this exploration, we will only look at **unnamed semaphores**. We

- Initialize an unnamed semaphore by calling `sem_init()`
- Destroy it by calling `sem_destroy()`
- Lock it by calling `sem_wait()`
- Unlock it by calling `sem_post()`

Let us look at these functions. Note that although semaphores can be used by multiple threads in a process or by multiple processes, to keep the explanation simple we will many times use only the term "process", or "thread", instead of repeatedly writing the whole term “process or thread.”

sem_init()

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- The `sem_init()` function (https://man7.org/linux/man-pages/man3/sem_init.3.html) initializes the unnamed semaphore pointed to by `sem`.
- The value of the argument `pshared` specifies whether this semaphore will be shared between multiple threads of a process, or between multiple processes. The value 0 means that it will shared between threads of a process, and 1 means it will be shared between multiple processes.
- The argument `value` specifies the initial value of the semaphore.
- If the function is successful, it return 0, otherwise it returns -1.

sem_destroy()

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

This function (https://man7.org/linux/man-pages/man3/sem_destroy.3.html) destroys the unnamed semaphore pointed to by `sem`. A semaphore should only be destroyed when there are no other processes or threads that are blocked on it. The behavior is undefined is a semaphore is destroyed while other processes or threads are blocked on it.

sem_wait()

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
```

This function (https://man7.org/linux/man-pages/man3/sem_wait.3.html) *tries* to decrement the semaphore `sem`.

- If the semaphore's current value is greater than 0, the function succeeds and returns immediately.
- If the semaphore's current value is 0, then any process that calls this function will block. When the value of the semaphore again become greater than 0, then one process among all the processes that are blocked on this function will be unblocked. This unblocked process will decrement the value of the semaphore and continue its execution.
- The function returns 0 on success, and -1 on error.

sem_post()

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

This function (https://man7.org/linux/man-pages/man3/sem_post.3.html) increments the semaphore `sem`. If this increment causes the value of the semaphore to become greater than 0 **and** one or more processes are currently blocked in a call to `sem_wait()` on this semaphore, then the following will happen:

- One of these processes will be unblocked
- This process will decrement the value of the semaphore and will now proceed with execution
- All other processes will remain blocked on `sem_wait()`

The function returns 0 on success, and -1 on error.

Binary Semaphores

Semaphores with an initial value of 1 are called **binary semaphores**.

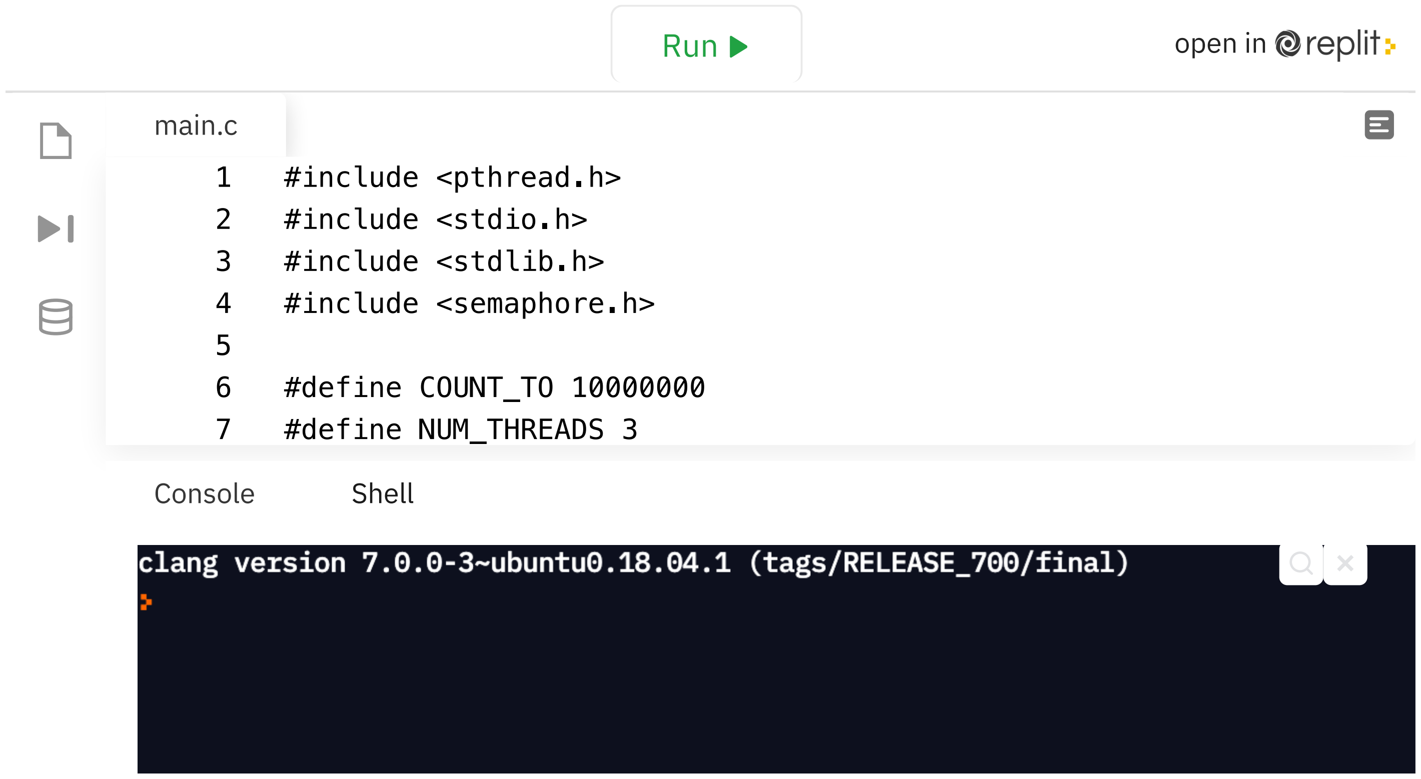
Let us consider that a semaphore `binary_sem` is initialized with the value 1 and is shared between threads of a process.

- If two threads T1 and T2 call `sem_wait()` at about the same time, one of them, say T1, will succeed in decrementing the value to 0 while T2 will be blocked at this call because the value is now 0.
- T1 can then proceed with its execution and can access or modify the shared resource.
- When T1 is done, it will call `sem_post()`. This will cause the value of `binary_sem` to increment to 1.
- Now T2 will be unblocked. It will decrement the value to 0 and proceed with its execution.
- If T1 again calls `sem_wait()`, it will be blocked until T2 calls `sem_post()`.

Note how this use of a semaphore with an initial value of 1 to enforce mutual exclusion is similar to how we used `pthread_mutex_lock` and `pthread_mutex_unlock` to implement mutual exclusion using `Pthreads` API in the previous module.

ercise

The following program implements an incorrect counter because of a race condition between the two threads when updating the shared variable `counter`. Modify the the function `perform_work` and add calls to `sem_post()` and `sem_wait()` to synchronize access to `counter` so that the program always produces the correct result.



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

#define COUNT_TO 10000000
#define NUM_THREADS 3
```

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
```

Answer

Answer (https://repl.it/@cs344/72semthreadsc#main.c)

Counting Semaphores

A semaphore with an initial value greater than 1 is called a **counting semaphore**. Counting aphores are used where we want to allow more than 1 process to simultaneously access a pool of shared resources, but still impose an upper limit on how many processes can simultaneously access this pool of shared resources. This is done by setting the initial value of the counting semaphore to the number of resources that can be simultaneously used.

Example: Sharing a Pool of Resources

In the following example, we show how threads in a process can share a pool of resources such that if all resources in the pool are being used by other threads, then a thread must wait for a resource to be released.

- We create a counting semaphore and set its initial value to POOL_SIZE, i.e., to the number of resources in the pool.
- Each thread calls `sem_wait()` on the semaphore before using the resource. If any resources are available for use, the value of the semaphore will be decremented and the thread can proceed with using the resource by calling `use_resource()`.
- When the number of threads that are currently using a resource equals POOL_SIZE, then the value of the semaphore will be 0. At this time, any thread which calls `sem_wait()` will be

blocked.

- When a thread gets done using the resource, i.e., it returns from `use_resource()` it calls `sem_post()` which will increment the value of the semaphore. If there are threads currently blocked on the semaphore, one of them will be unblocked, the semaphore value will be decremented, and the thread can proceed with calling `use_resource()`.

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
```

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- The man page for `sem_overview` (https://www.man7.org/linux/man-pages/man7/sem_overview.7.html) gives a good overview of semaphores.
- POSIX semaphores are discussed in Chapter 53 of *The Linux Programming Interface*.

  Kerrisk, M. (2010). *The Linux programming interface : a Linux and UNIX system programming handbook*. San Francisco: No Starch Press.
- **The Wikipedia article on semaphores (https://en.wikipedia.org/wiki/Semaphore_(programming))** includes examples of use of semaphores to solve some classic synchronization problems

  “Semaphore (Programming).” In Wikipedia, May 12, 2020.
- **Chapter 31 Semaphores (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf)** in the book *Operating Systems: Three Easy Pieces* discusses semaphores.

  Arpaci-Dusseau, Remzi H., and Andrea C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces, 2018.