Jesús Castillo Benito
19/11/2024

04 - Producers & Consumers

a) Implementation of mutex with semaphores

For the implementation of this part, I only modified *sync.h*, where I added implementation of some macros that can be used to initialize the mutex and do all the necessary operations only with POSIX semaphores. The macros implemented are the following:

```
#define MUTEX_T sem_t
#define COND_T sem_t

#define MUTEX_INIT(m) sem_init(m, 0, 1)
#define MUTEX_DESTROY(m) sem_destroy(m)
#define LOCK(m) sem_wait(m)
#define UNLOCK(m) sem_post(m)

#define COND_INIT(c) sem_init(c, 0, 0)
#define COND_DESTROY(c) sem_destroy(c)
#define WAIT(c, m) do { UNLOCK(m); sem_wait(c); LOCK(m); } while (0)
#define SIGNAL(c) sem_post(c)
```

As you can see in the first 2 lines, I am using a semaphore as mutex and another semaphore as a condition variable. Then in the next two blocks, we have the implementation of the operations for the mutex and for the condition variable, respectively.

For ***initializing*** the mutex, I am using a semaphore with an initial value of 1 to make it act as if it was a mutex, indicating with a 1 that the critical section is unlocked and it is possible to go inside it.
For ***lock*** and ***unlock***, I am simply using *sem_wait* and *sem_post*, as they decrement and increment the value of the semaphore respectively. When the semaphore has a value greater than 0, it is unlocked, else it is locked.

For ***initializing*** the condition variable, I am also using a semaphore, in this case with an initial value of 0, to represent that no threads are waiting to enter the critical section.
For ***wait***, I am first trying to *unlock* the mutex, then once it is unlocked, I call sem_wait so it decreases the value of the condition variable. At this point, if the value is greater than 0, this thread will proceed immediately, else it will block until another thread increases it. Finally, I call *lock*, that reacquires the mutex, to have exclusive access to the critical section again. The do-while structure around the operations serves to ensure that they act as a single unit.

For ***signal***, I simply call sem_post, which increments the value of the semaphore, unblocking one of the threads that were waiting, if there was any.

b) Implementation of mutex with System V semaphores

To implement this functionality, I also only modified *sync.h*, adding some macros that can perform the actions as if they were real mutex.

```
#define MUTEX_T int
#define COND_T int

#define MUTEX_INIT(m) (*(m) = semget(IPC_PRIVATE, 1, IPC_CREAT | 0666),
semctl(*(m), 0, SETVAL, 1))
#define MUTEX_DESTROY(m) semctl(*(m), 0, IPC_RMID)
#define LOCK(m) do { struct sembuf sb = { 0, -1, 0 }; semop(*(m), &sb, 1);
__sync_synchronize(); } while (0)
#define UNLOCK(m) do { __sync_synchronize(); struct sembuf sb = { 0, 1, 0 };
semop(*(m), &sb, 1); } while (0)


#define COND_INIT(c) (*(c) = semget(IPC_PRIVATE, 1, IPC_CREAT | 0666),
semctl(*(c), 0, SETVAL, 0))
#define COND_DESTROY(c) semctl((*c), 0, IPC_RMID)
#define WAIT(c, m) do { UNLOCK(m); struct sembuf sb = { 0, -1, 0 }; semop(*(c),
&sb, 1); LOCK(m); } while (0)
#define SIGNAL(c) do { struct sembuf sb = { 0, 1, 0 }; semop(*(c), &sb, 1); }
while (0)
```

The implementation of these macros follows the same methodology as the POSIX semaphores, but with the operations used in System V semaphores.

For ***initializing*** the mutex, I use *semget*, which creates a new semaphore or gets the ID of an existing one. In this case, as I am using *IPC_PRIVATE,* it will create a new semaphore that will not be shared across processes. Then, the 1 stands for the number of semaphores that I want to create. Then, the flag *IPC_CREATE* | *0666* is useful to ensure that the semaphore is created if it doesn't exist and to give the semaphore the appropriate permissions. After this, we initialize the mutex with the value of 1 using *semctl* with the operation *SETVAL*.
For ***lock*** I am using *semop*, with a *sembuf struct*, that indicates that we want to decrement in 1 the value of the mutex.
For ***unlock*** I am using algo *semop* with a *sembuf struct* that indicates that we want to increase the value in this case.
For these operations, I had to add a memory barrier to ensure the order of the operations when adding optimization flags to the compiler, causing deadlocks and unexpected behaviors.

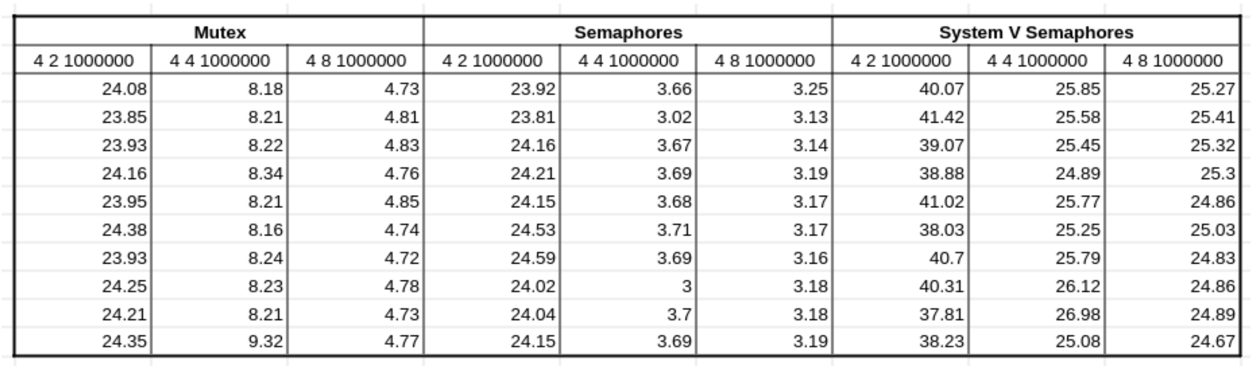
The condition variable and its operations are basically implemented the same way as the mutex, but in the initialization, instead of giving it an initial value of 1, we are going to give it an initial value of 0.

c) Comparing the performance of the implementations

To compare the different implementations, I just executed 10 times each implementation with 3 different configurations. The configurations that I used are the following:

- More producers/consumers than queue slots (4 producers/consumers, 2 queue slots and 1000000 iterations)
- Equal producers/consumers than queue slots (4 producers/consumers, 4 queue slots and 1000000 iterations)
- More queue slots than producers/consumers (4 producers/consumers, 8 queue slots and 1000000 iterations)

These are the results obtained from the tests:



Mutex			Semaphores			System V Semaphores		
4 2 1000000	4 4 1000000	4 8 1000000	4 2 1000000	4 4 1000000	4 8 1000000	4 2 1000000	4 4 1000000	4 8 1000000
24.08	8.18	4.73	23.92	3.66	3.25	40.07	25.85	25.27
23.85	8.21	4.81	23.81	3.02	3.13	41.42	25.58	25.41
23.93	8.22	4.83	24.16	3.67	3.14	39.07	25.45	25.32
24.16	8.34	4.76	24.21	3.69	3.19	38.88	24.89	25.3
23.95	8.21	4.85	24.15	3.68	3.17	41.02	25.77	24.86
24.38	8.16	4.74	24.53	3.71	3.17	38.03	25.25	25.03
23.93	8.24	4.72	24.59	3.69	3.16	40.7	25.79	24.83
24.25	8.23	4.78	24.02	3	3.18	40.31	26.12	24.86
24.21	8.21	4.73	24.04	3.7	3.18	37.81	26.98	24.89
24.35	9.32	4.77	24.15	3.69	3.19	38.23	25.08	24.67

As we can see, System V semaphores implementation is clearly worse than the other two implementations, performing very poorly in comparison. In between mutex and semaphores implementation, in the first configuration they behave with the same performance, however when we pass to the second configuration, semaphores is significantly better and in the third it is just better by a bit.

Thus said, the semaphores implementation would be the clear winner.

d) Prodcon vs Prodcon1