2013-12-02 COEN 320 Real Time Systems Assignment 2 Report

Sébastien Parent-Charette [9178821]
Sammy Ibrahim [5651786]

1.1 Program & Results

The program in the priority inversion scenario demonstrates the application of priority inheritance and the priority ceiling protocols and their impact on the scheduling of 3 processes trying to use a shared semaphore.

In the first implementation we ran the priority inversion scenario using the priority inheritance protocol. The results were as expected and P1 (a higher priority process) was blocked by P3 (a lower priority process) because P3 already had a lock on the shared semaphore. P3 then inherits the priority of P1 because it is blocking a higher priority process. After the critical section of P3 is completed and the shared semaphore is released, the priority of P3 is restored and P1 locks the shared semaphore. P1 then runs its critical section until completion, releases the shared semaphore and ends. Afterwards, P2 runs to completion and eventually P3 also gets to run to completion. It runs last because it has the lowest priority of all the processes.

The second implementation for the priority inversion scenario using the priority ceiling protocol ran exactly as when using the priority inheritance protocol and produced the exact same result. The only difference is that when P1 is blocked because the shared semaphore is locked, P3 does no inherit the priority of P1 but instead receives the priority of the priority ceiling. In this case, the priority ceiling is the same as P1 and thus produces the exact same result through a different method.

In the third implementation, we ran the deadlock scenario using the priority inheritance protocol. The results were as expected and P2 locked semaphore 2 as planned. Afterwards, it is pre-empted by P1 because it has higher priority. P1 then locks semaphore 1 and afterwards attempts to lock semaphore 2 but gets blocked because it is already locked by P2. Following this, the scheduler then runs P2 which now inherited the priority of P1 and it attempts to lock semaphore 1. Since semaphore 1 is already locked by the blocked process P1, P2 also gets blocked. The system is now in deadlock and nothing happens for the reminder of the run.

In the fourth implementation, we ran the deadlock scenario using the priority ceiling protocol. The results were as expected and P2 locked semaphore 2 as in the previous implementation. Afterwards, P2 is pre-empted by P1 because of its higher priority. Then, P1 attempts to lock semaphore 1 and instead of obtaining the lock, it is suspended because other semaphores with a higher ceiling priority than P1 are locked (semaphore 2 is already locked). As P1 gets suspended, the lock attempt on semaphore 1 also triggers a priority inheritance on P2 because there is now another process of higher priority, P1, waiting on it. P2's current priority is saved and it is changed to the priority ceiling of the semaphore. The scheduler then runs P2 at its new priority and it then also obtains a lock on semaphore 1. It is able to do so because all other locked semaphores are either owned by P2 already or P2 has a higher priority than the locked semaphores' priority ceiling. P2 then completes its critical sections and consequently unlocks semaphore 1 and then semaphore 2. The unlocking of semaphore 2 triggers the restoring of P2 to its original priority that was previously stored. Also, it un-suspends process P1 and sends it the signal to lock semaphore 1. At the next tick the scheduler now runs P1 which obtains the lock on both semaphores 1 and 2. Process P1 proceeds to complete its critical sections and unlocks semaphore 2 and then semaphore 1. Afterwards, P1 runs normally to completion. Then, the schedule runs P2 and it continues until completion. P2 is executed last because it has the lowest priority.

1.2 Mutex Implementation

The mutexes in this project were used for 2 implementations as semaphores. The first one was a semaphore using the priority inheritance protocol. The important features of this implementation were adding attributes to store the id of the process that locked the mutex, the value of the original priority of the process (in case of priority inheritance) and the id of a process that was blocked by trying to lock this semaphore while it was already in a locked state. There was also a flag to indicate if priority inheritance was required during the locking step so that it can be undone during the unlock step.

The second implementation of the semaphore used the priority ceiling protocol. This implementation was very similar to the priority inheritance protocol with a few important differences. First, in the priority ceiling implementation, there is 4 new attributes that are required for every semaphore. Those are the priority ceiling of the semaphore, a flag of the current status of the semaphore (lock or unlocked), the id of a targeted semaphore by a lock attempt that resulted in a process suspend and an id field for the semaphore itself. When a lock attempt is made on the semaphore by a process a check is made to ensure that no other semaphores are currently locked (and owned by a different process) and thus prevent a possible deadlock. If a locked semaphore is found then the calling process is suspended (no longer is scheduled to run by the ThreadManager) and the locked semaphore's attributes are updated with the calling process' id and the id of the semaphore it was trying to lock. Also, the process that owns the lock on the locked semaphore has its priority changed. It inherits the priority of the priority ceiling of the locked semaphore and the "priority inheritance" flag is set. When the process that owns the lock on the locked semaphore attempts to unlock it a special check is made to see if the flag of "priority inheritance" is set. If it was, then after the semaphore is unlocked, 2 extra steps are executed. First, the priority of the process that owns the lock is restored to its original value. Then, the process that was waiting on this semaphore lock (that was suspended previously) is removed from suspension and given a signal to obtain the lock on the semaphore it originally tried to lock before suspension.

2.1 Problem Inversion Scenario - Priority Inheritance Output

```
tick=0    scheduler runs , active_p=3

P3->[0]



tick=1    scheduler runs , active_p=3

P3->[1]

.....Attempting to Lock Semaphore by ..

..... Semaphore locked by P3

tick=2    scheduler runs , active_p=2

P2->[0]



tick=3    scheduler runs , active_p=2

P2->[1]



tick=4    scheduler runs , active_p=1

P1->[0]



tick=5    scheduler runs , active_p=1

P1->[1]

.....Attempting to Lock Semaphore ..

......P1 blocked ..



tick=6    scheduler runs , active_p=3

P3->[2]
```

```
tick=7    scheduler runs , active_p=3

P3->[3]

.....Unlocking Semaphore by ..

......semaphore unlocked by P3 ..

......P1 unblocked ..


..... Semaphore locked by P1

tick=8    scheduler runs , active_p=1

P1->[2]


tick=9    scheduler runs , active_p=1

P1->[3]

.....Unlocking Semaphore ..

......semaphore unlocked by P1 ..


tick=10    scheduler runs , active_p=1

P1->[4]

.........P1 thread ends.........

tick=11    scheduler runs , active_p=2

P2->[2]


tick=12    scheduler runs , active_p=2

P2->[3]
```

```
tick=13    scheduler runs , active_p=2

P2->[4]


tick=14    scheduler runs , active_p=2

P2->[5]


tick=15    scheduler runs , active_p=2

P2->[6]

.........P2 thread ends.........

tick=16    scheduler runs , active_p=3

P3->[4]


tick=17    scheduler runs , active_p=3

P3->[5]

.........P3 thread ends.........

tick=18    scheduler runs

tick=19    scheduler runs

tick=20    scheduler runs

tick=21    scheduler runs

tick=22    scheduler runs

tick=23    scheduler runs

tick=24    scheduler runs

tick=25    scheduler runs

tick=26    scheduler runs
```

```
tick=27    scheduler runs

tick=28    scheduler runs

tick=29    scheduler runs
```

2.2 Problem Inversion Scenario - Priority Ceiling Output

```
tick=0    scheduler runs , active_p=3

P3->[0]


tick=1    scheduler runs , active_p=3

P3->[1]

.....Attempting to Lock Semaphore by ..

..... Semaphore locked by P3

tick=2    scheduler runs , active_p=2

P2->[0]


tick=3    scheduler runs , active_p=2

P2->[1]


tick=4    scheduler runs , active_p=1

P1->[0]


tick=5    scheduler runs , active_p=1

P1->[1]

.....Attempting to Lock Semaphore ..
```

```
......P1 blocked ..


tick=6    scheduler runs , active_p=3

P3->[2]


tick=7    scheduler runs , active_p=3

P3->[3]

.....Unlocking Semaphore by ..

......semaphore unlocked by P3 ..

......P1 unblocked ..


..... Semaphore locked by P1

tick=8    scheduler runs , active_p=1

P1->[2]


tick=9    scheduler runs , active_p=1

P1->[3]

.....Unlocking Semaphore ..

......semaphore unlocked by P1 ..


tick=10    scheduler runs , active_p=1

P1->[4]

.........P1 thread ends.........

tick=11    scheduler runs , active_p=2
```

```
P2->[2]


tick=12    scheduler runs , active_p=2

P2->[3]


tick=13    scheduler runs , active_p=2

P2->[4]


tick=14    scheduler runs , active_p=2

P2->[5]


tick=15    scheduler runs , active_p=2

P2->[6]

.........P2 thread ends.........

tick=16    scheduler runs , active_p=3

P3->[4]


tick=17    scheduler runs , active_p=3

P3->[5]

.........P3 thread ends.........

tick=18    scheduler runs

tick=19    scheduler runs

tick=20    scheduler runs

tick=21    scheduler runs
```

```
tick=22    scheduler runs

tick=23    scheduler runs

tick=24    scheduler runs

tick=25    scheduler runs

tick=26    scheduler runs

tick=27    scheduler runs

tick=28    scheduler runs

tick=29    scheduler runs
```

2.3 Deadlock Scenario - Priority Inheritance Output

```
tick=0-> scheduler runs (P1=0; P2=0.5), active_p = P2
P2->[0]

tick=1-> scheduler runs (P1=0; P2=0.5), active_p = P2
P2->[1]
..... P2-CS2-entry ..
..... semaphore 2 locked by P2
tick=2-> scheduler runs (P1=0.7; P2=0.5), active_p = P1
P1->[1]
..... P1-CS1-entry ..
..... semaphore 1 locked by P1
tick=3-> scheduler runs (P1=0.7; P2=0.5), active_p = P1
P1->[2]
..... P1-CS2-entry ..
......P1 attempted to lock semaphore 2, already locked by P2 ..
......P1 blocked ..

tick=4-> scheduler runs (P1=0.7; P2=0.7), active_p = P2
P2->[2]
..... P2-CS1-entry ..
......P2 attempted to lock semaphore 1, already locked by P1 ..
......P2 blocked ..

tick=5-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=6-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=7-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=8-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=9-> scheduler runs (P1=0.7; P2=0.7), active_p = P2
```

```
tick=10-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=11-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=12-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=13-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=14-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=15-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=16-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=17-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=18-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=19-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=20-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=21-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=22-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=23-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=24-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=25-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=26-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=27-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=28-> scheduler runs (P1=0.7; P2=0.7), active_p = P2

tick=29-> scheduler runs (P1=0.7; P2=0.7), active_p = P2
```

2.4 Deadlock Scenario - Priority Ceiling Output

```
tick=0-> scheduler runs (P1=0; P2=0.5), active_p = P2
P2->[0]

tick=1-> scheduler runs (P1=0; P2=0.5), active_p = P2
P2->[1]
..... P2-CS2-entry ..
..... semaphore 2 locked by P2
tick=2-> scheduler runs (P1=0.7; P2=0.5), active_p = P1
P1->[1]
```

```
..... P1-CS1-entry ..
......P1 attempted to lock semaphore 1 ..
......P1 suspended ..
......P2 inheriting PC of semaphore 2 ..

tick=3-> scheduler runs (P1=0.7; P2=1), active_p = P2
P2->[2]
..... P2-CS1-entry ..
..... semaphore 1 locked by P2
tick=4-> scheduler runs (P1=0.7; P2=1), active_p = P2
P2->[3]
..... P2-CS1-exit ..
......semaphore 1 unlocked by P2 ..

tick=5-> scheduler runs (P1=0.7; P2=1), active_p = P2
P2->[4]
..... P2-CS2-exit ..
......semaphore 2 unlocked by P2 ..
......P2 priority is restored ..
......process P1 waiting on semaphore 2 is unsuspended ..
..... semaphore 1 locked by P1
tick=6-> scheduler runs (P1=0.7; P2=0.5), active_p = P1
P1->[2]
..... P1-CS2-entry ..
..... semaphore 2 locked by P1
tick=7-> scheduler runs (P1=0.7; P2=0.5), active_p = P1
P1->[3]
..... P1-CS2-exit ..
......semaphore 2 unlocked by P1 ..

tick=8-> scheduler runs (P1=0.7; P2=0.5), active_p = P1
P1->[4]
..... P1-CS1-exit ..
......semaphore 1 unlocked by P1 ..

tick=9-> scheduler runs (P1=0.7; P2=0.5), active_p = P1
P1->[5]

tick=10-> scheduler runs (P1=0.7; P2=0.5), active_p = P1
P1->[6]
.........P1 thread ends.........
tick=11-> scheduler runs (P1=0; P2=0.5), active_p = P2
P2->[5]

tick=12-> scheduler runs (P1=0; P2=0.5), active_p = P2
P2->[6]
.........P2 thread ends.........
tick=13-> scheduler runs (P1=0; P2=0)
tick=14-> scheduler runs (P1=0; P2=0)
tick=15-> scheduler runs (P1=0; P2=0)
tick=16-> scheduler runs (P1=0; P2=0)
tick=17-> scheduler runs (P1=0; P2=0)
tick=18-> scheduler runs (P1=0; P2=0)
tick=19-> scheduler runs (P1=0; P2=0)
tick=20-> scheduler runs (P1=0; P2=0)
```

```
tick=21-> scheduler runs (P1=0; P2=0)
tick=22-> scheduler runs (P1=0; P2=0)
tick=23-> scheduler runs (P1=0; P2=0)
tick=24-> scheduler runs (P1=0; P2=0)
tick=25-> scheduler runs (P1=0; P2=0)
tick=26-> scheduler runs (P1=0; P2=0)
tick=27-> scheduler runs (P1=0; P2=0)
tick=28-> scheduler runs (P1=0; P2=0)
tick=29-> scheduler runs (P1=0; P2=0)
```