Project 1 Report
Name: xiaoyu_ma Student id: 523031910469

Abstract

This report is for OS project 2, which is about using the semaphore, mutex lock and so on to implemente interprocess communictation.

The first sub project is a classic IPC problem called Stooge Farmers Problem, and the second one is a bicycle manufacturing simulation. The goal of the second sub project is to simulate a bicycle manufacturing process using semaphores and mutex locks to control access to shared resources.

1 Stooge Farmers Problem

1.1 Problem Description

Larry, Moe, and Curly are planting seeds. Larry digs the holes. Moe then places a seed in each hole. Then the Curly fills the hole up. To guarantee the correct order of farming activities, there are several synchronization constraints.
To achieve this, we use semaphore to enable mutual exclusion to the shared resource, which is the shovel. And use atomic operation and thread lock to maintain the status of each hole.
In the end, Larry, Moe and Curly will coorperate appropriately and complete the task uder the constraints.

1.2 Implementation

Here I use 3 atomic bool arrays to represent the status of each hole, which is

1. diggedHoles, This is for Larry to record if the hole is digged.
2. seededHoles, This is for Moe to record if the hole is seeded. If a hole is digged but not seeded (i.e. diggedHoles[i] = true, seededHoles[i]= false), Moe can seed it(i.e. seededHoles[i] = true).
3. filledHoles, This one is for Curly to record if the hole is filled. If a hole is seeded but not filled (i.e. seededHoles[i] = true, filledHoles[i]= false), Curly can fill it(i.e. filledHoles[i] = true).

Then, A semaphore is aquired to control the access to the shovel. There is only one shovel that needs to be used by Larry and Curly in turn. So the semaphore is initialized as a bounded semaphore with the initial value of 1. In each cycle of Larry and Curly, they should aquire the semaphore first, and release after using.

Because the constraints include that Larry shouldn't have MAX holes ahead of Curly, that is there cannot be more than MAX holes that are digged but not filled. Thus a semaphore initialized with value of MAX is applied. Each time, when Larry is about to dig a hole, he should try to obtain this semaphore at first. After Curly fills a hole, he should release this semaphore.

Note that this semaphore is the first one that is aquired by Larry, before aquiring shovel

simulation To simulate the actual situation, a random delay is applied each time when they finish their jobs.

1.3 Simulation & Annalysis

The section here discusses the simulation and analysis of the Stooge Farmers Problem. As exprimented with multiple conditions, here we take one as an example.

```
N = 10.
MAX = 3.
Larry gets the shovel.
Larry digs another hole #1.
Larry drops the shovel.
Moe plants a seed in hole #1.
Curly gets the shovel.
Curly fills a planted hole #1.
Curly drops the shovel.
Larry gets the shovel.
Larry digs another hole #2.
Larry drops the shovel.
Moe plants a seed in hole #2.
Curly gets the shovel.
Curly fills a planted hole #2.
Curly drops the shovel.
Larry gets the shovel.
Larry digs another hole #3.
Larry drops the shovel.
Moe plants a seed in hole #3.
Curly gets the shovel.
Curly fills a planted hole #3.
Curly drops the shovel.
Larry gets the shovel.
Larry digs another hole #4.
Larry drops the shovel.
Larry gets the shovel.
Larry digs another hole #5.
Larry drops the shovel.
Moe plants a seed in hole #4.
Curly gets the shovel.
Curly fills a planted hole #4.
Curly drops the shovel.
Larry gets the shovel.
Larry digs another hole #6.
Larry drops the shovel.
Moe plants a seed in hole #5.
Curly gets the shovel.
Curly fills a planted hole #5.
Curly drops the shovel.
Moe plants a seed in hole #6.
Larry gets the shovel.
Larry digs another hole #7.
Larry drops the shovel.
Curly gets the shovel.
Curly fills a planted hole #6.
Curly drops the shovel.
Larry gets the shovel.
```

```
    Larry digs another hole #8.
    Larry drops the shovel.
    Moe plants a seed in hole #7.
    Curly gets the shovel.
    Curly fills a planted hole #7.
    Curly drops the shovel.
    Moe plants a seed in hole #8.
    Larry gets the shovel.
    Larry digs another hole #9.
    Larry drops the shovel.
    Curly gets the shovel.
    Curly fills a planted hole #8.
    Curly drops the shovel.
    Larry gets the shovel.
    Larry digs another hole #10.
    Larry drops the shovel.
    Moe plants a seed in hole #9.
    Moe plants a seed in hole #10.
    Curly gets the shovel.
    Curly fills a planted hole #9.
    Curly drops the shovel.
    Curly gets the shovel.
    Curly fills a planted hole #10.
    Curly drops the shovel.
    End.
```

In the problem, Larry and Curly race the shovel. As we can see in the simulation, Larry digs a hole and drops the shovel. Only after that, Curly can gets the shovel and fill the hole.

Notice that sometimes Moe doesn't plant a seed in the hole immediately after Larry digs it. This is due to the random delay that is applied. Even with that, the order is correct. Moe only plants a seed in the hole after Larry digs it, and Curly only fills a hole after Moe plants a seed in it.

Furthermore, The MAX constraints works well. Larry never digs more than MAX holes ahead of Curly. If he does, he will be blocked by the semaphore.

1.4 Findings and Conclusions

1.4.1 Avoid starvation

In this task, The starvation happens when there is no delay after Larry or Curly finishes their job, which has to using the shovel.
In this case, Larry and Curly will keep aquiring the shovel and never release it.

To solve this problem, Two methods are applied.

1. Add a random delay after Larry and Curly finish their job. This will make sure that they will not keep aquiring the shovel forever.

2. The MAX variable that mentioned above is just a solution. This avoid Larry from keep digging holes and never release the shovel to Curly. Larry will be forced to stop when contiguously diggging MAX holes.

1.4.2 Avoid deadlock

Since we use 3 arrays here to record the status of each hole, they should be ensured with mutual exclusion access. So each time we need to modify the value, mutex lock should be aquired first.

And the semaphores mentioned above is also a solution to avoid deadlock.

1.4.3 Conclusion

In the simulation, the program fits the constraints and works well. But to be honest, when performing multithreading, special attention should be paid to the order of aquiring the semaphore and mutex lock.

2 Bicycle Manufacturing Simulation

2.1 Problem Description

In a bicycle factory, three types of workers collaborate to assemble bicycles. The factory has a shared storage box with N slots (N 3). Each slot can store either a bicycle frame or a bicycle wheel. A complete bicycle consists of 1 frame and 2 wheels.

Other constraints are listed in Project requirements File.

2.2 Implementation

In the Implementation, several semaphores and mutex locks are used to control the access to the shared storage box and other resources.

2.2.1 Mutual exclusion of Shared Storage Box

The shared storage box is a bounded buffer with N slots. Since these slots are identical, we don't need to maintain a array or other data structure to record the status of each slot.

Only 2 counters are applied to record the number of frames and wheels in the storage box.

The shared storage box has N slots in total. And the frame producer and Wheel producer produce one accessory at once.

So, a semaphore initialized with N is used to control the access to the storage box. At each iteration of a producer, it aquires the semaphore first, and then produce the accessory, place it in the storage box. Note that this semaphore will not be released here, it will be released when The assembler aquires it.

2.2.2 Mutual exclusion of the shared resource

The *box* variable is shared accross all the entities. So a mutex lock is used to ensure that only one entity can access the shared resource at a time.

The Mutex lock in Producer will be aquired after aquiring the semaphore. Otherwise, deadlock will happen.

When the lock is aquired but semaphore cannot meet the requirement, the thread stuck there, then deadlock occurs.

2.2.3 Assembler Implementation

The assembler needs to wait until there are enough frames and wheels in the storage box to commence assembly. In this phase, we need mutex lock to ensure that the assembler can access the shared resource and block other modifications.

2.2.4 Avoid starvation

If the wheel producer occupies the last available slot in the storage box, the frame producer will be blocked parmanently. Then the assembler will be blocked as well, Since it can't fetch a frame.

To avoid this, we need to ensure that the frame producer and wheel producer should not occupy all the slots in the storage box. That is, if the wheel producer has occupies the N-1 slots in the storage box, it will forced to stop to let the frame producer to place the frame in the box. And so as for frame producer, frame producer needs to spare at least 2 slots for wheel producers

2.3 Simulation & annalysis

The simulations has been done in multiple presets, Here we take one as example.

```
N = 4, M = 3
A = 1, B = 1, C = 1
Frame producer 0 produced a frame.
Frame producer 0 placed a frame. { 1 frames, 0 wheels }
Frame producer 0 produced a frame.
Frame producer 0 placed a frame. { 2 frames, 0 wheels }
Assembler 0 took a frame. { 1 frames, 0 wheels }
Wheel producer 0 produced a wheel.
Wheel producer 0 placed a wheel. { 1 frames, 1 wheels }
Wheel producer 0 produced a wheel.
Wheel producer 0 placed a wheel. { 1 frames, 2 wheels }
Assembler 0 took two wheels. { 1 frames, 0 wheels }
Wheel producer 0 produced a wheel.
Wheel producer 0 placed a wheel. { 1 frames, 1 wheels }
Assembler 0 assembled a bicycle.
Wheel producer 0 produced a wheel.
Wheel producer 0 placed a wheel. { 1 frames, 2 wheels }
Frame producer 0 produced a frame.
```

```
    Frame producer 0 placed a frame. { 2 frames, 2 wheels }
    Assembler 0 took a frame. { 1 frames, 2 wheels }
    Assembler 0 took two wheels. { 1 frames, 0 wheels }
    Assembler 0 assembled a bicycle.
    Wheel producer 0 produced a wheel.
    Wheel producer 0 placed a wheel. { 1 frames, 1 wheels }
    Wheel producer 0 produced a wheel.
    Wheel producer 0 placed a wheel. { 1 frames, 2 wheels }
    Assembler 0 took a frame. { 0 frames, 2 wheels }
    Assembler 0 took two wheels. { 0 frames, 0 wheels }
    Assembler 0 assembled a bicycle.
    End.
```

In this instance, The slots in the storage box are 4. The frame producer and wheel producer are both 1. The assembler is 1.

As we can see in the simulation, after frame producer occupies 2 of the slots, which is half of the box capacity, it gives the chance to wheel producer to place a wheel in the box.

Then the assembler can take a frame and two wheels from the storage box, and assemble a bicycle.

The sequence in assembler when waiting for accessories doesn't matter, as the both producers are designed to ramain some idle slots for each other.

In case that slots number is no smaller than 3, as the task requirement , the simulation works well.

2.4 Findings and Conclusions

This task has a lot more complicated IPC control than the first one. The main challenge is to ensure that the assembler can access the shared resource and block other modifications.

At the same time, the starvation avoidance is alse more sophisticated. We should ensure that frame producers side and wheel producers side should not occupy all the slots in the storage box, leave enough slots for the other side.