Design a Synchronization Mechanism for the prototype OS

CSCE 351 – PROJECT III

Endi Xu

12/13/2015

Contents

Introduction

Project Goal

The goal of this project is to design and implement a semaphore type to support race-free execution of a classic IPC problem. (1)

Problem Description

The bear and the honey bee problem is what needs to be solved. The following is the description. There are 10 honey bees and a bear with a shared pot. The pot is empty at the beginning. Its capacity is 30 units of honey. Every honey bee repeatedly gathers one unit of honey and puts it into the pot each time. In a bee's lifetime, it can totally gather 15-unit honey. The bear sleeps until the pot is full. When the pot is full, bear will wake up and eat all the honey. It needs to be noted that when the bear is eating the honey, the bee could not add any honey to the pot. Each bee and the bear can be treated as a thread. They can be blocked and released by the semaphore which is designed in this project.

Project Management

This project took two weeks to complete. All the resources that are used in this project is provided by the professor on the Blackboard.

The following table is the timeline of this project.

Date	Things Finished
11/29/2015 ~ 11/30/2015	Read the handout of the project
12/1/2015 ~ 12/4/2015	Design the logic of the code
12/5/2015 ~ 12/7/2015	Write the source code
12/8/2015 ~ 12/12/2015	Debugging
12/13/2015	Write the report and finish the README file

Table 1 Timeline

Analysis

Main Idea

This project is based on the project II. The source code of the project 2 has been provided by the professor. In this project, only a few changes need to be made from the source code in project 2. A new header file of semaphore is added. The thread scheduler is revised. The main function is changed. The rest of the code is remained the same.

Data Structure

In this project, a new data structure is created which is the semaphore. In the semaphore structure, it contains 3 variables, which is a semaphore value, a number that shows waiting threads and a block queue which holds the block threads. The queue data structure has already been in created in project II. It can be used directly here.

Algorithm

The program has two types of thread. The bear thread and bee thread. As mentioned in problem description, the bear thread only being processed when the pot is full. On the other hand, the bee thread can't interrupt the bear when bear thread is processing. This means that the bear thread have to be blocked when the pot is not full and the bee thread have to be blocked when the pot is full. In this program, the semaphore is used to block and release the threads.

The following two figures shows the pseudocode of the bear process and the bee process:

```
process bear {
        i = 0;
        while (i < 5) {
                down(pot_full);         //Lock the pot_full. Bear won't be re-waked up
                portions = 0;           //Eat all honey in the pot
                up(mutex);              //Release the mutex. Bee thread will not be blocked
                i++                     //Get eating time
        }
}
```

Figure 1 Pseudocode of Bear Process

```
process honeybee {
        while (true) {
                down(mutex);            //Lock the mutex. The bee won't be interrupted
                portion++;              //Add honey to the pot
                if (portions == H)
                        up (pot_full);  //If the pot is full, let bearwake up
                else
                        up (mutex);     //If the pot is not full, let other Bee fill the pot
        }
}
```

Figure 2 Pseudocode of Bee Process

From figure 1 and figure 2, the semaphore "pot_full" will block the bear until the pot is full. The semaphore "mutex" will block the bee when the pot is full or there is another bee filling the pot. Through this way, the honey bee and bear problem can be solved.

Also, a semaphore must be designed to block and release threads. The semaphore must be able to block running threads through semaphore down. And release blocked threads through semaphore up. The following figure shows how the semaphore up/down works.

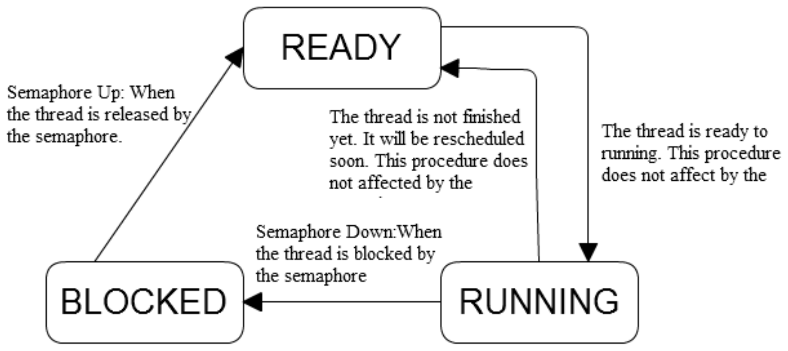
READY
Semaphore Up: When the thread is released by the semaphore.
The thread is not finished yet. It will be rescheduled soon. This procedure does not affected by the
The thread is ready to running. This procedure does not affect by the
Semaphore Down:When the thread is blocked by the semaphore
BLOCKED
RUNNING

Figure 3 Thread State

Strategy

In order to make the above algorithm works, the thread scheduler needs to be revised first. The thread scheduler used in this project is the same one used in project 2. However, in project 2, the thread scheduler does not consider the situation of blocked. So in thread scheduler, a new if statement needs to be added to check if the thread is blocked or not. If the state is block, the state should be kept until the semaphore changes the state.

In the semaphore file, when a semaphore is created, all the value are set to 0 or NULL. When down a semaphore, the state of the current thread is set to BLOCKED and enqueue it to the block queue of the semaphore. When up a semaphore, dequeue the thread from the block queue and set the state to READY. After all the processes are finished, the semaphores can be destroyed through delete function. In this function, the "free()" provided by the C library can be used.

Table 2 shows the function included in the semaphore file.

your_structure_type* mysem_create(int value)	// It returns the starting address a semaphore variable. You can use malloc() to allocate memory space and initialize the internal data structure of the semaphore based on provided parameter.
void mysem_up(*your_structure_type** sem)	// It performs the semaphore's *increment* operation. Increment the value of the semaphore. If one or more threads are sleeping on the semaphore, the one that has been sleeping the longest is allowed to complete its *down* operation. **As such, after an *up* on a semaphore with threads sleeping on it, the semaphore value is still 0 but there is one fewer sleeping threads.**
void mysem_down(*your_structure_type** sem)	// It performs the semaphore's *decrement* operation. Check to see if the semaphore value is greater than 0. If so, it decrements the value and just continues. If the value is 0, the thread is put to sleep without completing the down for the moment. **As such, the semaphore value never falls below 0.**
void mysem_delete(*your_structure_type** sem)	// It deletes the memory space of a semaphore.
int mysem_waitCount(*your_structure_type* sem*)	// Return the number of threads sleeping on the semaphore.
int mysem_value(*your_structure_type* sem*)	// Return the current value of the semaphore.

Table 2 Semaphore (2)

In the main file, the bear and bee processes are created through the algorithm provided above. Then, the same as in project 2, create a bear and 10 bee threads in the OS_primitive() function. In the main function, only the OS_primitive() function will be called.

Work Accomplished

In this project, the program works as required. Each thread can be started and ended correctly. The bear threads can be blocked when bee threads are processing. The bear thread won't be interrupt by any bee thread when it is processing. The semaphore also can be destroyed after finishing.

Conclusion

Things helped to understand the class materials

Semaphore is an important knowledge in CSCE351 class. In the homework 2, the semaphore is being used. However, at that time, the semaphore is just used through the "semaphore.h" in the C library. In this project, a self-designed semaphore type is created. It can help greatly on understanding how the semaphore performs.

Self-Evaluation

This project is all done by myself. The main idea of the program is collected from the course materials provided on the Blackboard.

Improvement

This project is very helpful. It helps me understand how the semaphore works and how to schedule threads well. However, it still can be improved. The description of the function "mysem_up" and "mysem_down" is too brief. It is hard to understand. If more details can be provided. It would be better.

Reference

1. Document Title: Project Description
   Source: https://my.unl.edu/bbcswebdav/pid-3245145-dt-content-rid-27966254_1/courses/CSCE351001.1158/description%281%29.pdf
2. Table 2: Semaphore
   Source: https://my.unl.edu/bbcswebdav/pid-3075697-dt-content-rid-26645932_1/xid-26645932_1