CISC2005 Principles of Operating Systems

Assignment 3

Release date: Mar. 19, 2023

Due date: April 2, 2023 23:59

No late assignment will be accepted

Every Student MUST include the following statement, together with his/her signature in the submitted homework.

I declare that the assignment submitted is original except for source material explicitly acknowledged, and that the same or related material has not been previously submitted for another course. I also acknowledge that I am aware of University policy and regulations on honesty in academic work, and of the disciplinary guidelines and procedures applicable to breaches of such policy and regulations.

Signed(Student 黄彦祯) Date 2023/03/21

Name Huang Yanzhen SID DC126732

General homework policies:

A student may discuss the problems with others. However, the work a student turns in must be created COMPLETELY by oneself ALONE. A student may not share ANY written work or pictures, nor may one copy answers from any source other than one's own brain.

Each student **MUST LIST** on the homework paper the **name of every person he/she has discussed or worked with**. If the answer includes content from any other source, the student **MUST STATE THE SOURCE**. Failure to do so is cheating and will result in sanctions. Copying answers from someone else is cheating even if one lists their name(s) on the homework.

If there is information you need to solve a problem but the information is not stated in the problem, try to find the data somewhere. If you cannot find it, state what data you need, make a reasonable estimate of its value, and justify any assumptions you make. You will be graded not only on whether your answer is correct, but also on whether you have done an intelligent analysis.

Questions:

1. Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs. (20 points)

In a single-processor system, only one process can run at a time. Disabling interrupts gives the current process full access of the CPU. If this process takes too long to execute, other process may be starved.

2. What are similarities and differences between deadlock and starvation? (20 points)

Similarities: In both the scenarios of deadlock & starvation, there are processes that hasn't been executed for a relatively long time.

Differences:

- In a deadlock, often there are 2 process blocked waiting for each other to release a resource, where none of them is executed.
- In starvation, however, there are processes being executed, however, except for some processes that has been always waiting.

3. Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```c
typedef struct {
    int available;
} lock;
```

0-not locked

1- locked - -

(available == 0) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this struct, illustrate how the following functions can be implemented using the ***test_and_set()***. (20 points)

```c
typedef struct { int available; } lock;

void init(lock *mutex) {
    // available=0 -> lock is available, available=1 -> lock is unavailable
    ________;  mutex -> available = 0
}

void acquire(lock *mutex) {
    while (________)  test-and-set (&mutex -> available)
        ; // spin-wait (do nothing)
}

void release(lock *mutex) {
    ________; mutex -> available = 0
}
```

4. Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the ***test_and_set()*** instruction. The solution should exhibit minimal busy waiting. (20 points)

```c
int guard = 0;
int semaphore value = 0;
wait() {
 while (________)      test-and-set (& guard)
     ;
 if (semaphore value == 0){
     atomically add process to a queue of processes
     waiting for the semaphore and set guard to 0;
 }
 else
 {
     ________;  semaphore value --
     guard = 0;
 }
}
signal() {
 while (________)  test-and-set (& guard)
     ;
 if (semaphore value == 0 && there is a process on the wait queue)
     wake up the first process in the queue of waiting processes
 else
     semaphore value ++
 guard = 0;
}
```

5. Here is a pseudocode for implementing a bounded buffer using the monitor. The size of the buffer is ***N***, and the variable "***count***" indicates the number of resources available in the buffer. The condition variables "***not_full***" and "***not_empty***" indicate whether the buffer is not full and not empty, respectively. The producer thread continuously inserts items into the buffer, while the consumer thread continually consumes items. Please fill in the appropriate conditional statements and use ***wait***() and ***signal***() operations to manipulate the condition variables in the blanks below to complete the code. (20 points)

```
Monitor bounded_buffer {
    Resource buffer[N];
    // Variables for indexing buffer
    int count = 0;
    // monitor invariant involves these vars
    Condition not_full; // space in buffer
    Condition not_empty; // value in buffer

    void insert(Resource item) {
 while (count==N){
       wait(not_full);
        }
        buffer[count++] = item;
        signal(not_empty)
    }

    Resource remove() {
 while (count==0){
          wait(not_empty)
        }
        item = buffer[--count];
        signal(not_full)
        return item;
    }
} // end monitor

produce() {
    while (1) {
        Resource item = produce_item();  // Produce an item.
        bounded_buffer.insert(item);
    }
}

consumer() {
    while (1) {
        Resource item = bounded_buffer.remove(item);  // Get an item.
    }
}
```