Romil V. Shah
20008692

CS 520 – Intro to OS
Homework Assignment 2
Submission Date: Oct 4th, 2022

Q2. Consider modifying the Enter_Critical procedure (see Lecture 1) to assign the Id of the executing process (rather than other) so that the procedure looks like that:

```
void Enter_Critical (int process) /* 0 or 1 */
 {
   int other;

   other = 1 – process;
   flag[process] = TRUE;    /* show my interest */
   turn = process;          /* grab it! */
   while (flag [other] == TRUE && turn == process);
}
```

Will the algorithm still work (i.e., provide the critical section access satisfying all three criteria)? Why or why not?

Ans.

No, the algorithm still won't work. The reason is it violates the Mutual Exclusion.

Let us consider two processes K, R:
As the Process R enters the 'Enter_Critical ()', it achieves turn = 1;
Now, the Process K makes 'flag [0] = TRUE';
Process K executes the condition "while (flag [1] == TRUE && turn == 0)".
The condition is true, so Process K enters Critical Section.
On entering the 'Enter_Critical ()' Process K achieves turn = 0;
Process R makes "flag [1] = TRUE";
Process R executes the condition "while (flag [0] == TRUE && turn == 1)". The condition is true, so Process R enters Critical Section.
Both the processes Process K and Process R are in Critical Section.

Q3. Does the busy waiting solution work when the two processes are running on a shared-memory multiprocessor?

Ans.
Yes, the busy-waiting solution using the turn variable will work when the two processors are running on a shared memory multiprocessor.

In the given example, the 'turn' variable is used to control access to acritical region by two processes. In a multiprocessor, the difference is that the two processes are running on different CPU's. The only condition is that 'turn' itself should be a variable in shared memory.

Busy waiting solution using turn variable:
• The turn variable is initially set to 0.
The process 0 finds the turn variable as 0 and enters its critical region.
• It enters the non-critical region by making the turn variable to 1
Process 1 finds the turn variable as 1 and enters the critical region. It finishes its critical region and enters the non-critical region by making turn variable to 0
• Again, process 0 enters the critical region, finishes the work, and exit from the critical region by making turn to 1
• Currently, both processes are in non-critical region.
Now, process 0 can't enter critical section as the turn variable is 1
• As the processes running on a shared-memory multiprocessor and sharing common memory, busy waiting solution with turn variable works, but still busy waiting occurs.

Q4. When a computer is being developed, it is often first simulated by a program that runs one instruction at a time. (Even multiprocessors are simulated strictly sequentially like this.) Is it possible for a race condition to occur in such situations? (Please explain your answer.)

Ans.

When a computer is being developed, it is often first simulated by a program that runs one instruction at a time and each instruction runs in sequential manner. So, there is no possibility of race condition when it is running sequentially.

Race condition is an undesirable situation, and it comes when a device or system attempts to perform two or more operations at the same time. Because race condition occurred when there is an uncoordinated concurrent access to shared resources, so that it leads to incorrect behavior of the program, deadlocks, or lost work.

Due to race condition, redundancy and inconsistency may occur in output. But if every program is executed in a sequential manner, then no two programs go in critical section at the same time. So, in sequential execution race condition is never possible.

Q5. Suppose a queue in a semaphore is implemented not as a first-in-first-out (FIFO) queue, but as last-in-first-out (LIFO) stack. Show how this can result in starvation, that is a situation when a process is indefinitely queued in a semaphore and thus can never progress.

Ans.

A queue implemented using last-in-first-out (LIFO) stack can cause starvation because a process which is pushed last to the queue is always removed first by the scheduler.

Here, we can understand this by taking an example of Shortest Job Scheduling.
In case of Shortest Job First scheduling, if short duration processes keep on coming, then long processes may be held off indefinitely and will never be removed from the queue. This will lead to starvation.

Q6. One reason semaphores are used is for mutual exclusion—that is ensuring that only one process enters a critical session to avoid race conditions. The other reason is synchronization, that is ensuring the events happen in certain sequence. The lecture specifies three semaphores mutex, empty, and full. Which ones of these are used for mutual exclusion, and which ones—for synchronization? Explain your answer.

Ans.
Mutex Exclusion:
A mutual exclusion is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Mutual Exclusion is used only when one process at a time is being executed in their critical section.

Synchronization:
Synchronization is a process of handling resource accessibility by multiple thread requests. The main purpose of synchronization is to avoid thread interference. At times when more than one thread tries to access a shared resource, we need to ensure that resource will be used by only one thread at a time.

empty: semaphore (n)
"empty" is used to block producer if buffer is full.

mutex: semaphore (1)
"mutex" is used to lock on Buffer.

full: semaphore (0)
"full" is used to block consumer if buffer is empty.

Mutual exclusion: If process Pi is executing in its critical section, then no other processes can be executed in their critical sections.

So, mutex is used for mutual exclusion. It provides mutual exclusion.
While full and empty are used for synchronization.

Q7. Study the Bounded_Buffer problem (Section 7.2) carefully. Suppose the two wait statements of Figure 7.2.1 were reversed by the programmer [so that wait(mutex) is executed before wait(empty)]. What will happen when the buffer is full? (Note: The empty semaphore is initialized to buffer_size; the full semaphore is initialized to 0.)

Ans.

Full represents the number of occupied slots in the buffer. Then, empty semaphore is incremented by 1 because the consumer has just removed data from an occupied slot that's why it makes it empty. As, the value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

When the slot is empty, then it is going to acquire lock on the buffer, so that it cannot access the buffer until it completes its operation.

Because sem_empty is initialized with buffer_size and sem_full is initialized with 0,
Now First wait(sem_full) is executed first as compared to wait(sem_empty)
As sem_full = 0, so wait(sem_full) means we are waiting with wait (0),
As no Process can make the progress, (because both are stuck at wait(sem_full), So sem_full is never modified and both the processes will go in starvation.

Q8. The Sleeping-Barber Problem. A barbershop consists of a waiting room with n chairs and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.
Using the semaphore constructs described in the textbook (and used for solving other problems) write a program to coordinate the barber and the customers. That is, write one procedure for the Barber process and one procedure for the customer process.) You must define the customers, barbers, and mutex semaphores.
The code can be schematic (just as in the lecture). You don't have to write the real code; however, you must use the semaphore primitives wait and signal as described in the book.

Ans.

The solution to this problem includes three semaphores.

1. The customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting).
2. The barber 0 or 1 is used to tell whether the barber is idle or is working.
3. The third mutex is used to provide the mutual exclusion which is required for the process to execute.

In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room, then the upcoming customer leaves the barbershop.

When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up.

When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less than the number of chairs then he sits otherwise he leaves and releases the mutex.

If the chair is available, then customer sits in the waiting room and increments the variable waiting value and increases the customer's semaphore this wakes up the barber if he is sleeping.

At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.

Algorithm:

```
Semaphore Customers = 0;
Semaphore Barber = 0;
Mutex Seats = 1;
int FreeSeats = N;

Barber {
        while(true) {
                /* waits for a customer (sleeps). */
                down(Customers);

                /* mutex to protect the number of available seats.*/
                down(Seats);

                /* a chair gets free.*/
                FreeSeats++; /

                * bring customer for haircut.*/
                up(Barber);

                /* release the mutex on the chair.*/
                up(Seats);

                /* barber is cutting hair.*/
                        }
        }
Customer{
        While(true) {
                // protects seats so only 1 customer tries to sit in a chair if that's the case.
                down(Seats);

                if(FreeSeats > 0) {
                        // sitting down.
                        FreeSeats--;

                        // notify the barber.
                        up(Customers);

                        // release the lock
                        up(Seats);
                        // wait in the waiting room if barber is busy.
                        down(Barber);

                        // customer is having hair cut
```

```
                } else{
                        // release the lock
                        up(Seats);

                        // customer leaves
                }
        }
}
```

Q9. Artists and Viewers. In the new Center for Strange Arts (CESA) in New York City, there is a famous Weird Hall, open 24 hours a day, with a very large painting being perpetually developed by the walk-in artists. Because the artists tend to fight with one another as well as accost viewers, only one artist at a time may enter the Weird Hall, and once an artist is there, no other artists or viewers are allowed in. (They have to wait in front of the Weird Hall.) If no artist is in, the viewers may come in at any time and in any numbers; they are not queued behind the artists. Furthermore, as long as there is at least one viewer is in the Weird Hall, no artist is allowed in. Write the pseudo-code for synchronization of viewers and artists using semaphores.

Ans

Pseudo-Code:

```
define variables;
semaphore.artist=1 // to keep track
count variable. viewer = 0 // to Keep viewer count
semaphore.sem =1 // to secure viewer count critical section

// artist Code
Count
do{
        wait (artist) // to ensure only one artist can enter
        // edit the art
        Signal(artist)
}
while(true),

//viewer code
do{

        wait(sem);
        Viewer ++;
        // as soon as the first viewer enters, we block the artist

        if (viewer==1)
        wait (artist);
        //  another viewer can also enter so we signal the Semaphores.

        signal(sem);
        // after viewing the art a Person exits

        Wait (sem);
        viewer--; // a viewer exits
```

```
        If (viewer==0){
                signal (artist); // if there is no viewer left, then artist can enter.
                signal (s);
        }
}
while (true);
```