Operating Systems–II: CS3523
Theory Assignment - I

Sagar Jain
CS17BTECH11034

1. Consider the semaphore based solution to producer-consumer bounded buffer problem that we discussed in the class. Please answer the following:

(a) What is the minimum that the semaphores full & empty can become? Please explain how.
Answer:

There are two possible ways to use semaphores, with busy waiting or without. For this solution, I have used **without busy wait**.
In this case, the minimum value of semaphore **empty** can be **-1**. This happens when the value of semaphore **full** is **n**, i.e. (n + 1) items have been produced but none have been consumed.
As soon as n items are produced, the value of empty becomes 0. In the next iteration of the producer process, the producer sets the value of empty to -1 and must wait on it to become non-negative, as the producer process is now asleep no more items are produced.

In the same way, the minimum value for the semaphore **full** can be **-1** when the value of **empty** is **n**, i.e. there are 0 buffers that are full, the consumer process sets the value of full to -1 and sleeps till it becomes non-negative so it cannot go below -1.

(b) Similarly, what is the maximum that the semaphores full & empty can become? Please
explain how.

Answer:
The maximum value of **empty** is **n**. This happens when there are no full buffers. The producer can only reduce the value of empty and the consumer cannot increase it further since it has to wait at the statement **wait(full)** when the value of full is 0.

In the same way, the maximum value of **full** is **n**. This happens when there are on empty buffers. The consumer can only reduce the value of full and the producer cannot increase the value of full further because it cannot [ass the statement **wait(empty)** when the value of empty is 0.

(c) Please give the upper and lower bounds on the sum of full & empty in any given state.
Answer:
At any point, the value of full + empty is between n-1 and n.
wait and signal always nullify each other’s effect on the sum and in intermediary stages, the sum can equal n - 1.
(n - 1) <= empty + full <= n

2. In the class, we discussed the solution to the reader-writers problem using semaphores. We saw that this solution can cause the writer threads to starve. Please develop an alternative solution in which neither the reader nor the writers will starve. For this, you can assume that the underlying semaphore queue is fair.
Answer:

We present the following code for reader and writer to ensure neither starve,

Reader:

```
while(true) {
  wait(queue_prevent_starve);
  // to prevent starvation
  wait(mutex);
  reader_count++;
  if(reader_count == 1)
    wait(rw_mutex);
  signal(mutex);
  signal(queue_prevent_starve);

  // Read operations

  wait(mutex);
  read_count--;
  if(reader_count == 0)
      signal(rw_mutex);
  signal(mutex);
}
```

Writer:

```
while(true) {
  wait(queue_prevent_starve);
  // to prevent starvation
  wait(rw_mutex);
  signal(queue_prevent_starve);

  //  Write Operations

  signal(rw_mutex);
}
```

6.11 Does this "compare and compare-and-swap" idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

```
void lock_spinlock(int *lock) {
  while (true) {
    if (*lock == 0) {
    /* lock appears to be available */
    if (!compare_and_swap(lock, 0, 1))
    break;
    }
  }
}
```

Answer:

This **"compare and compare-and-swap"** idiom works fine for the implementation of spin-locks. We have to make sure that mutual-exclusion is guaranteed and that this is actually a spin-lock. We can see that the process calling this would, in fact, be spinning as there is a while loop which has to be escaped in order to move to further execution. Mutual exclusion is also guaranteed. Firstly, the **compare_and_swap operation is atomic**, so even if two different processes read ***lock** as 0, when they come to the condition checking the value of compare_and_swap only one will be returned false and call break; while the other will continue spinning.
We know that atomic operations block all interrupts and are bad for performance, so this implementation decreases the number of calls to **compare_and_swap** and would perform better than just compare_and_swap in my opinion.

6.12 Some semaphore implementations provide a function getValue() that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling wait() so that a process will only call wait() if the value of the semaphore is > 0, thereby preventing blocking while waiting for the semaphore. For example:

```
if (getValue(&sem) > 0)
wait(&sem);
```

Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the function getValue() in this scenario.

Answer:

We can prove that there is a potential problem in **get_value()**. The utility of get_value() would be the fact that it can avoid a process from getting blocked and call some other function if the if the condition fails.
The above cannot always be ensured. If the value of the semaphore is true (or 1 in case of integer semaphore) and if two processes execute the if condition at the same time before either can call wait, then both the processes pass the condition and call wait, but this leads to only one process passing into the critical section while the **other one must wait on sem**. This ambiguity in execution is a problem with get_value().