**CSCI P 536 Assignment 3 Dhaval Niphade**

**Function Prototypes**

* wait(<semaphore>);
* signal(<semaphore>);
* produced=semcreate(0);
* consumed=semcreate(1);
* sid32 produced, consumed;

**Q.1** How exactly is synchronization achieved using semaphore in our assignment?

**Answer:** In the given assignment solution, we’ve achieved synchronization through the use of semaphores along with two system procedures ‘wait’ & ‘signal’. A semaphore is merely an integer value which determines whether a process is meant to wait (essentially, sleep) or resume activity (using the signal system call).

From a broader perspective, when a system call for wait() is made, it decrements the semaphore and adds the process that called wait() to a list of waiting processes if the semaphore value is non-negative. The wait function itself however operates at a much complex level. When a system call for wait() is made the following steps occur :

* The saved interrupt mask is disabled
* If there’s a conflict in accessing the semaphore i.e. the semaphore being accessed is incorrect or no longer valid in the semaphore table, the interrupt mask is re-enabled and the system outputs an error message
* If there is no process waiting on the process list of the accessed semaphore, then a system error is returned and the interrupt mask is re-enabled
* If the count of the semaphore remains non-negative after it has been decremented, the calling process yields control of the processor and enters the waiting state. Concomitantly, a resched() call is made to switch to a process whose state is “READY” and is waiting next in the FIFO queue
* The interrupt mask is enabled

The system call signal() also operates in a similar fashion and can be considered the inverse of wait(). Again, from a broader perspective, when signal() is called, the semaphore count is incremented. At a much deeper level, it executes a similar set of instructions as the wait() system call with the difference being that it increments the semaphore count and moves the next process in line to the ready state.

**Through this use** of semaphores, we ensure that when the producer generates a new number (that’s less than count) it does not go on to generate the next one until the consumer function has completed executing its code block (in this case, printing the generated number).

*\*\*\* Proceed to page 2 for the next question\*\*\**

**Q.2** Can the above synchronization be achieved with just one semaphore? Why or why not?

**Answer**: Yes. The producer consumer problem (known as bounded – buffer problem when dealing with a queue) can be handled using a single semaphore. The use of two semaphores in the manner in which we’ve implemented them is a viable solution only when there is a single producer and consumer. In situations, where there are multiple producers or consumers, race conditions will arise. In order to circumvent this issue, we often make use of a single semaphore to implement mutual exclusion – accessing a shared resource (queue, linked list or some data structure) exclusively without the other process accessing, altering, modifying it.

The piece of code that accesses or alters the shared resource is called the critical section. When using a single semaphore, we place our wait and signal calls around this line of code. Collectively, these synchronization function calls and the line of code that performs the mutually exclusive activity form the critical section. At a given time, the critical section can only be accessed by a single process.