Lab1

Dilip kunderu

September 20 2016

1 Access to shared Buffer

Having a deferential access to shared buffer in an asynchronous fashion,

- Although two processes use the processor, the time required by each of the processes might differ. Thus, a lot of processor cycles get invested in computations which are erroneous.
- Buffer might have an overflow/underflow condition if the access is not synchronized, leading again to wasted processor cycles and/or loss of data
- Complex computations to be made by data returned by the consumer process will
  1. inconsistent state of the data
  2. busy waiting condition

2 Semaphore vs Mutex in lieu of waiting efficiency

In the case of Mutex implementation

- All processes have the same wait time as the **entire** buffer is locked when a process acquires the mutex.
- Though the uniformity is a plus, median wait time is higher in this implementation.

In the case of the counting semaphore implementation

- In the case of the most usual implementation, consumer initially waits till the buffer is full, and then starts the action.
- Unless a boundary condition is defined, an offset amounting to the buffer size would be witnessed in the output log all the time
- Though all items produced are consumed in-order, the average wait time for this implementation is higher.

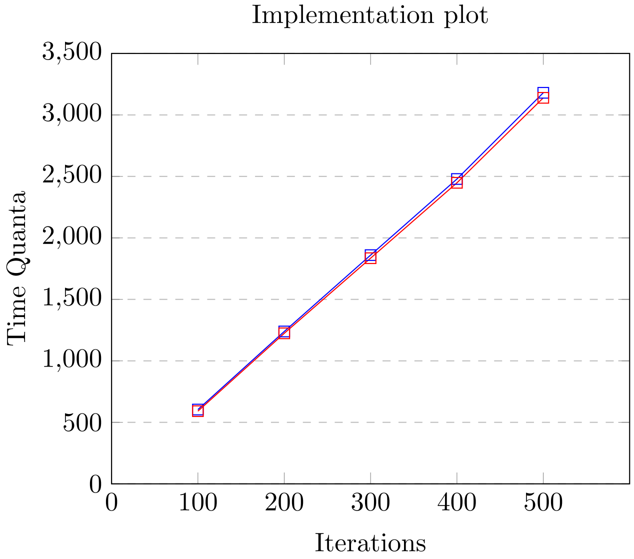
In short,

- Utilizing **Mutex** mechanisms locks the usage of the complete buffer down to only one process. In contrast, employing a general **semaphore** would allow a more granular access to the individual nodes of the buffer/ user-defined division of the buffer memory.
- **Mutex** is more like a *locking* mechanism on the buffer, and a Mutex lock can only be resumed by the process that acquired the lock initially. A (counting) **semaphore**, is slightly different, in the sense that it is more of a *signaling* mechanism, with an important differentiators being that *wait()* and *signal()* can be issued by different processes.

- Though both semaphore and mutex mechanisms solve the producer-consumer problem, wait time while deploying a counting semaphore tends to vary, in contrast to a mutex.
  While running a counting semaphore, the consumer process waits until the buffer is full, and then start with its output. In a perpetual loop, it will theoretically output all the items, in the order which they are produced, but not necessarily in the form of an *atomic* transaction ie., consuming the data immediately after the producer produces it.
  This is the prime difference that I could see between the two implementations.

3 Timing Plots for the codes

The following are the timing plots of the codes :



The plots seem to overlap because

- Only 1 producer and one consumer processes are being run
- Iterations are too little to witness variances