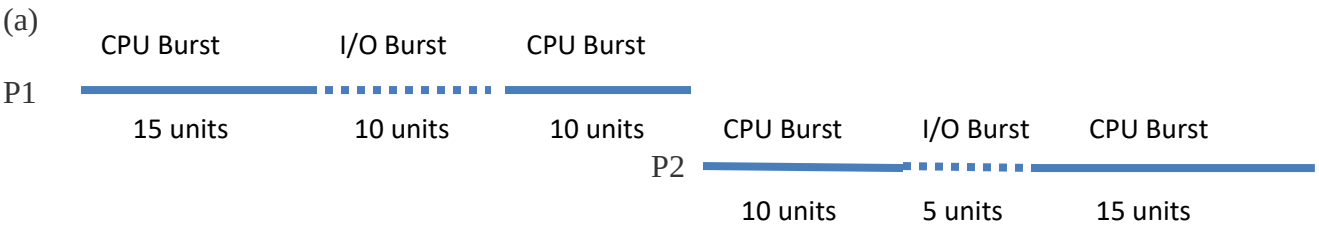
Theory Assignment 1 Solutions

COMP 346

Fall 2019

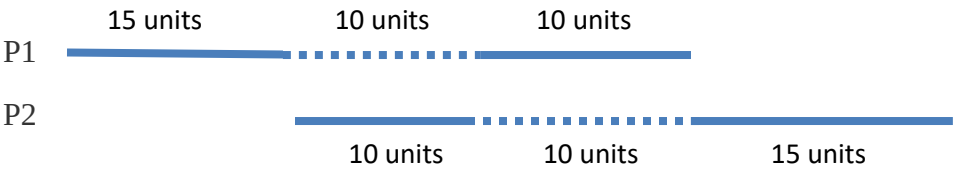
Q.1.

(a)



Minimal completion time = (15 + 10 + 10) + (10 + 5 + 15) units = 65 units

(b) Here is one possibility where P1 runs first:



Minimum completion time = (15 + 10 + 10 + 15) units = 50 units

Another possibility is that P2 runs first. It can be checked that it will give the same total execution time.

(c) Throughput for part (a) = 2/65 tasks/unit time = 0.0308 tasks/unit time

Throughput for part (b) = 2/50 tasks/unit time = 0.04 tasks/unit time

Multiprogramming definitely increases throughput by making the CPU non-idle by executing another ready-to-run process when the running process blocks.

Q.2. (a) Let us first consider a uniprocessor system. Consider one of the N processes as Pm, which has the maximum execution time. If we could execute all the remaining N-1 processes during the waiting times of Pm, and if none of them increases the waiting time of Pm, then the completion time for N processes will be equal to maximum (t1, t2,…,tN).

For a multiprocessor system, if there are at least N processors, then all the N processes could run in parallel. Then, without any other overheads, this could also give execution time equal to maximum (t1, t2,…,tN).

(b) Total execution time could be greater than the sum of all process execution times if it were not possible to overlap the waiting time of the one process with the execution times of the other processes (e.g., processes are waiting for timer events like clock interrupt) which make all processes wait simultaneously. This is compounded with context switch overhead.

Another situation is when all the processes are CPU-bound and they do not block for I/O execution. So the context switch overhead makes the total execution time greater than the sum of all process execution times.

Q.3. (ii), (iii) and (iv) should be privileged. In the case of (ii), if users could clear memory arbitrarily, one process may clear other process's memory by mistake or on purpose. In the case of (iii), if a user can turn off interrupts arbitrarily, he/she is capable of disturbing the proper functioning of the system. In the case of (iv), if a user process can switch from user mode to kernel mode arbitrarily (e.g. via an assembly instruction), a malicious user could call some privileged instructions in kernel mode directly (which is unlike the case of switching to kernel mode through a trap, for instance invoked through a system call).

In the case of (vi), all registers are in the process's context. So copying from one (general purpose) register to another is not privileged (and it happens in most assembly instructions). However, some registers are privileged (e.g. the interrupt status register, ISR), and modifying it directly will be prohibited.

Q.5. No, interrupts are not always advantageous over polling, considering that interrupts have overheads due to context saving and restoring. For example: consider the specific scenario where it is known a priori that within a bounded time a small burst of I/O or an event would occur (i.e. in synchronous I/O), then it may be advantageous to do (periodic) polling rather than interrupt. Moreover, if I/O or event is frequent then polling could be advantageous because majority of polling cycles would result a hit. Note that interrupts are more advantageous when I/O or event is asynchronous.

Q.6. (next page)

Q.7. (a) When a context switch occurs, the system needs to save the current context (process state and values of CPU registers) of the running process so that it can resume execution the next time it runs from exactly where it left. Saving of context must be atomic to ensure that the context of the process will be stored completely without creating any inconsistency. Similarly restoring the context of a ready to run process must also be atomic to ensure its correct execution. Note that context switch code executes inside the kernel, and critical section(s) of this code manipulates shared kernel variables and data-structures; hence atomic execution of these critical sections is essential for consistency of shared data.

(b) Failure to ensure atomicity of critical sections could result in inconsistencies, for instance: a process executing with some of the registers containing values from another process, or the program counter with an erroneous value, or the ready queue pointers in an inconsistent state,

etc. This would result in the process running in an unforeseen way, which could lead to its termination or damage to the system or other processes.

(c) Atomicity can be achieved by disabling interrupts at the beginning of a context switch and enabling interrupts after completing the context switch. This will guarantee that context switch occurs without interference from any other operations, and the context of the previous process is saved completely and the context of the new process is loaded exactly from the state where it left the CPU. Moreover, it will ensure consistency of the kernel data structures (e.g., ready queue, device queues, etc.) and registers which are manipulated during a context switch. Note that disabling and enabling of interrupts is done inside the kernel executing in the kernel mode and hence it is necessarily safe.

Q.6.