Christopher Angelo – 2440041503
Operating System - Mid Semester Answers

Section 1 – Computer and Operating System Overview

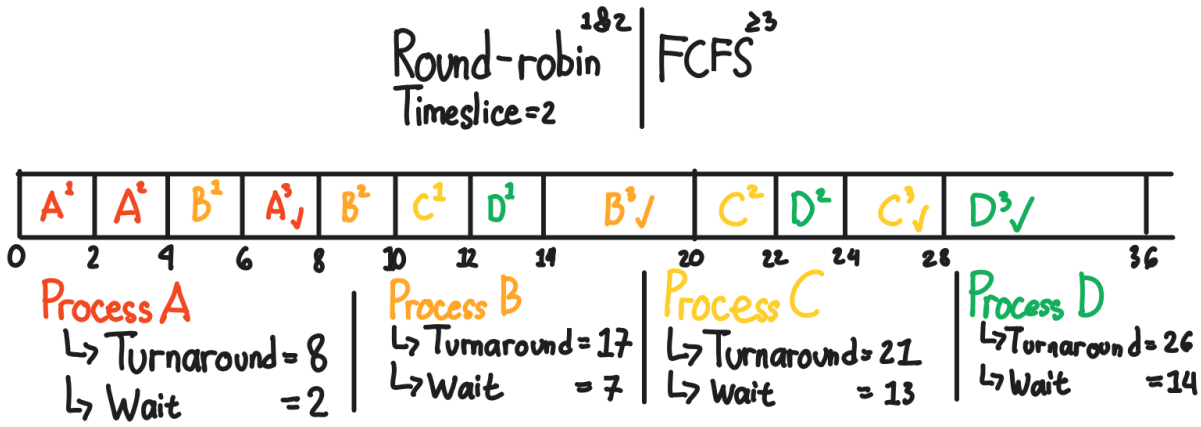
a. When a process requests a "read" instruction, the following things will happen:
   1. The process queues the instruction to the memory
   2. When the processor gets to the instruction, the instruction will be stored in the CPU's Instruction Register
   3. The Instruction Register will parse for the instruction and read the memory section pointed by the instruction
   4. The value of the pointed memory will be stored in the accumulator
   5. The processor will read through the next step in the memory.

b. In the past, computers are huge machines. A serial operating system may be deployed to these machines; a software that interfaces users, primarily programmers, directly with the hardware. These type of computers does not have a proper Read-Eval-Process-Loop such that scheduling and setup time were the concern of these computers. A sequential operating system introduces process "batching" where most of the problem of serial operating system are solved via a batch job.

c. A virtual memory system (swap in Linux) is useful in the design of an Operating System (OS) as the OS can delegate extra hard-disk space as a memory ("Virtual Memory"). While a hard drive is a lot slower than RAM, it will help the operating system in certain cases. In the case of the system running out of memory, data in the RAM can be pushed to the Virtual Memory, cleaning up some space in the RAM.

Section 2 – Process and Process Scheduling

a. Using time slice 2 round-robin for first two slices, combined with FCFS, the following will be the chart for the provided CPU burst time requests:



The average turnaround time is 18, the average wait time is 9. Using a combination of these algorithm, it is not safe to say that the algorithm combined will "improve" round-robin alone. A concern is having a process that requests long burst time; after the third round of round robin, all other tasks will be stuck waiting for the said process to be completed.

b. In Linux, a zombie state is an extra state of a process. A process gets into the zombie state when its process has ended (or dies) but their resources are still in memory. This may cause a memory / resource leak where a non-existent "phantom" process is holding the resource, wasting memory the system's memory.

c. Assuming PCB in this context means Process Control Block; A PCB is a "block" of memory that consist of the meta-information of processes. These might include PID, Process State, Priority / Nice values, File descriptors, etc. In a single user / single tasking operating system you do not need a PCB assuming every task will run sequentially without need of concurrency, without failure OR with proper failure handling and back-tracking approach. Though, this is a very edge case, thus why almost every OS deploys PCB even if it is designed to be a single user & single tasking.

Section 3 – Threads

a. *Attached in the zip file, file name "3a_mutex.c"*
b. In Linux, the system implements threading via green thread (user level thread) and processor thread (kernel level thread) to allow for code to be run concurrently. Green threads are handled by the program itself while the kernel level thread are handled by the kernel. When running, these threads are queued into a process queue where each process is given a slice of a lifetime in CPU burst time (By default using the Completely Fair Queueing scheduler).

Section 4 – Multiprocessor and Embedded system

a. Multiprocessor system can be classified into three sections
    a. **Functionally specialized** – Where there is 1 leader processor and many follower processors which follows the tasks given by the leader.
    b. **Loosely coupled processor** – Consist of many smaller systems, each with its own memory and IO
    c. **Tightly coupled multi-processor** – A modern processor; this processor shares a common memory (in this case, CPU cache) and are integrated together usually by an OS.
b. When designing a scheduler specifically for a multi-processor system, there are a couple of things that might be an issue. Since there are multiple processors in a system, one need to consider to fully utilize the amount of processor there are in the system when dispatching processes.
c. An example of an embedded computer in a transportation segment is a head-unit of a container vehicle. The head-unit may track the metrics of the vehicle (speed,

fuel amount, route taken through GPS). This system then might report back to the management where the driver may be judged.

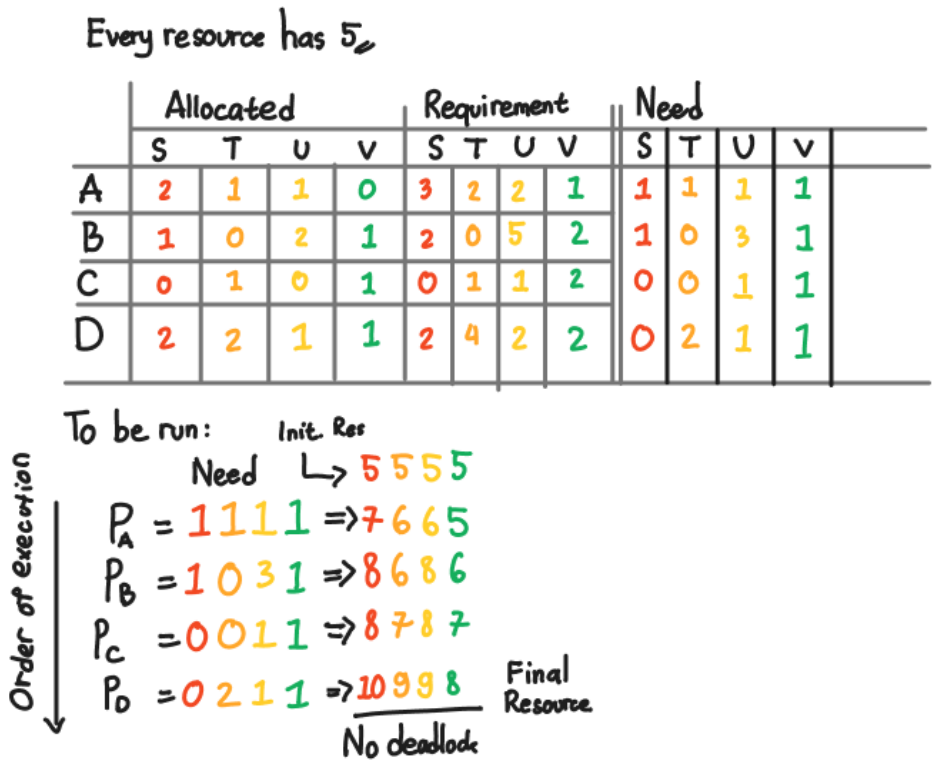
Section 5 – Process Synchronization

a. In a multi-core environment where multiple threads / processes exist at one time, a critical section is a part of a code which are being shared between them. This code may be a variable or an external resource (e.g., file descriptors) which requires to either be synchronized with each other or atomically executed. A semaphore will help this critical section to synchronize with each other by letting a variable declare whether a task has took place in another process / thread. This will better allow for parallelism.

b. *Attached in the zip file, file name "5b_signaltrap.c", ignoring SIGINT (Signal 2 – CTRL + C).*

Section 6 – Deadlock

a. The following is the banker's algorithm visualization. It is determined that there is no deadlock with 10, 9, 9, 8 as the final resource count.



b. The following is the Resource Allocation Graph for the said process snapshot. There is a deadlock detected.

c. There are multiple ways on approaching deadlock recovery:

a. **Abort all processes** – Kill all processes that are involved in the deadlock, instantly clearing the deadlock. This is the simplest but most destructive process in recovering from a deadlock and is not recommended
b. **Progressively abort deadlocked processes** – Kill all processes starting from the most-recently queued process. Doing this is better than the previous strategy as it is not as destructive as previously.
c. **Revert process state to a previous safe checkpoint** – Set-up a task to save a system process “checkpoint” that runs in an interval. When a deadlock is detected, revert the system state to a previously known safe checkpoint and restart all processes. While this is an optimal strategy, this approach requires processing power and memory to capture the said checkpoint. In addition, after restarting all processes, the process output might differ due to the order of process.