Stephen Woodbury – 1429496 – swoodbur
HW3 - CMPS 111 – Spring 2018 – 5/9/2018 - Wednesday

1) *Multi-level feedback queues (MLFQs) are implemented in many modern time-sharing operating systems, including those derived from 4.4 BSD Unix. Briefly describe the operation of MLFQs in 4.4 BSD Unix and the motivation behind using them.*
A: What is a Multi-level feedback queue? A multi-level queue is a combination of various queues used in determining what thread/process goes next on the processor. Usually each queue inside this multi-level queue is managed by its own scheduling algorithm separate from the other queue in the multi-level queue. It is also usually the case that each queue has priority over lower priority queues. Ie, if our multi-level queue is consisted of queue A, B, and C. Then it's usually the case that A < B < C in terms of which queue holds the more important threads/processes (This is related to a Fixed priority preemptive scheduling method. An alternative would be Time slicing which is similar, except that each queue would get a slice of the CPU run time instead of scheduling always drawing from the most prioritized queue in our MLQ). A multi-level feedback queue is like a multi-level queue. It differs in that processes/threads can change the queue they're stored in as their priority changes.
What is the operation of MLFQs in 4.4 BSD Unix? 4.4 BSD Unix CPU Scheduling is based on multi-level feedback queues. Runnable threads are explicitly assigned priority which determines which queue they start off in. Threads are always scheduled from the highest priority queue (reference Fixed priority preemptive scheduling mentioned above) until that queue is exhausted, at which the schedule will begin drawing from the next most prioritized queue. If a thread comes into the MLFQ with a higher priority than the currently running thread, that thread is preempted, unless it's in kernel mode, in which case, the scheduler will wait for the running thread to go back into user mode before preempting. The priority for 4.4 BSD Unix Scheduling ranges from 0 to 127. 0 is the highest priority, 127 is the lowest priority. A thread in kernel mode will always have a priority between [0,49] while user mode threads will always have a priority between [50, 127]. Kernel mode threads cannot change their priority but a user mode thread has a say in its priority in that in can change its niceness and can state what thinks its priority should be. The actual user thread priority is determined by a formula involving the niceness of the thread, what it thinks its priority should be, its Cpu utilization, its sleep time, and the sampled mean of the sum of lengths of run queue and short term sleep queue. The Cpu utilization is calculated frequently(unless its sleeping) using it's previous value, load, and thread niceness. This priorities recent values

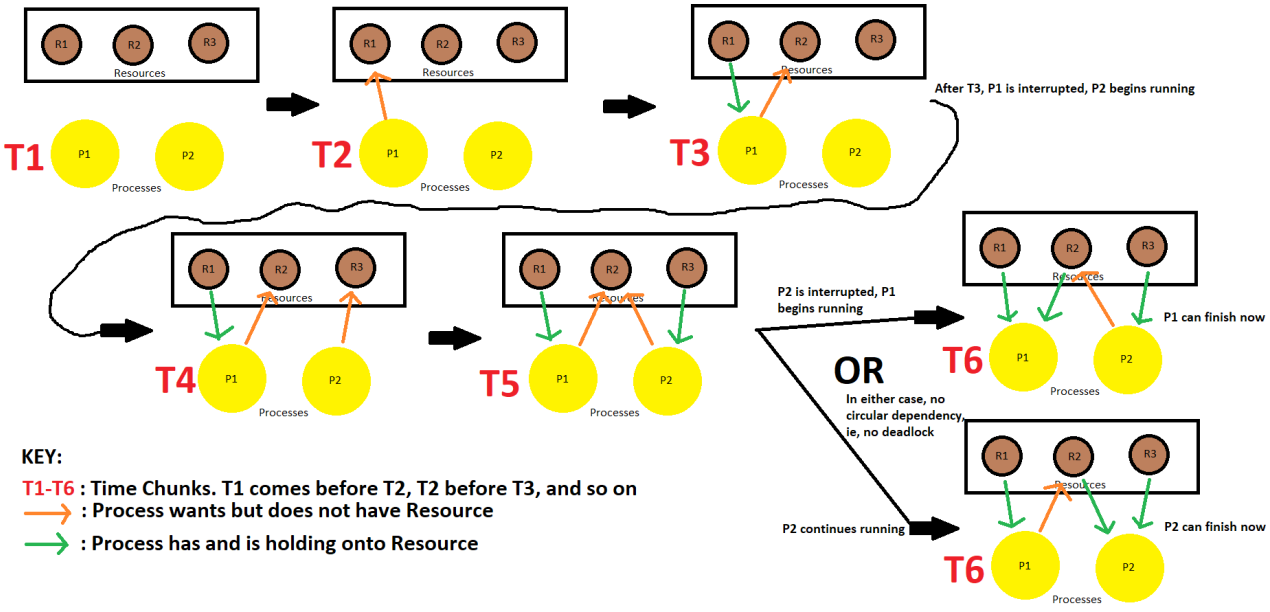
of Cpu utilization. Other things worthy of mention, the quantum is 100ms and 4.4 BSD Unix scheduling favors user-interactive processes as it raised the priority of threads that are waiting on IO.

Motivation behind using MLFQs? It's more flexible than multi-level queue scheduling. It optimizes turnaround algorithms such as SJF (SJF and related algorithms need running time of processes, which a MLFQ can learn through past behavior and prediction). MLFQ's are also great for reducing response time, which is optimal for 4.4 BSD Unix as it prioritizes user-interactive processes.

Stephen Woodbury – 1429496 – swoodbur
HW3 - CMPS 111 – Spring 2018 – 5/9/2018 - Wednesday

2) *A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer*

A: No. Let's assume our system has the perfect conditions for a deadlock to happen. That is, none of the resources allow for simultaneous use (Mutual Exclusion), resources cannot be seized from the holder(No preemption), and when a process gets a hold of a resource, it doesn't let go of it until it's done with the resource(Hold and wait). The only thing we would need now is there to be a circular wait to occur in order for there to be a deadlock. However, a circular wait can never happen, ensuring a deadlock will never happen. The worst that can happen is that one process is in the middle off collecting its resources, it collects one of its needed two resources and then gets interrupted so that the other process can run. That other process will then begin collecting its needed two resources. Of the worse things that can happen, there are two possibilities. One, the second process collects one of the remaining two resources, gets interrupted so that the first process can begin running again(which already is holding one of our resources). In this case, that thread will get its second resource, run, and then release its resources so that the second process can pick another one up to run, in which case, no deadlock. Or two, the second process picks up the last remaining resource instead of being interrupted, in which case it will finish running and release its resources so that the first process may resume running. Here too, there is no deadlock. A resource Allocation Diagram is provided below to demonstrate this.



Stephen Woodbury – 1429496 – swoodbur
HW3 - CMPS 111 – Spring 2018 – 5/9/2018 - Wednesday

3) *Explain how quantum value and the time taken to perform a context switch affect each other in a round robin process-scheduling algorithm.*
A: A quantum value is the amount of time a process is allowed to run on the CPU before being interrupted and replaced with another process. Usually it's between 10 and 100ms. The whole point of Round Robin Process Scheduling is to avoid the weakness of the First Come First Serve Scheduling algorithm, that weakness being that short processes are usually stuck behind large processes/CPU hogs. Round robin allows all processes an equal go on the CPU. Every time a process is kicked off the CPU and another is put on, a context switch occurs. If we've too short of a quantum and if our processes almost all require more time on the CPU than the quantum allows, then time and resources are wasted context switching. For example, if the quantum is only five times the length of the time it took to context switch, then our CPU will spend 20% of its time context switching. If the quantum is too long and if most of our processes require less time on the CPU than the quantum dictates, then our performance becomes more like that of a First Come First Serve CPU scheduling algorithm, completely negating the point of round robin scheduling. In short, the shorter the quantum value, the greater the percentage of time the CPU spends doing context switching.

Stephen Woodbury – 1429496 – swoodbur
HW3 - CMPS 111 – Spring 2018 – 5/9/2018 - Wednesday

4) *Five threads, A through E, arrive in alphabetic order at a scheduling queue one second apart from each other. Estimated running times are 10, 6, 2, 4, and 8 seconds, respectively. Their externally determined priorities are 3, 5, 2, 1, and 4, respectively, 5 being the highest priority. For each of the following scheduling algorithms, determine the mean turnaround time and mean waiting time. Assume thread switching is effectively instantaneous.*

(a) First Come First Served (b) Round Robin (c) Preemptive Priority Scheduling (d) Preemptive Shortest Job First

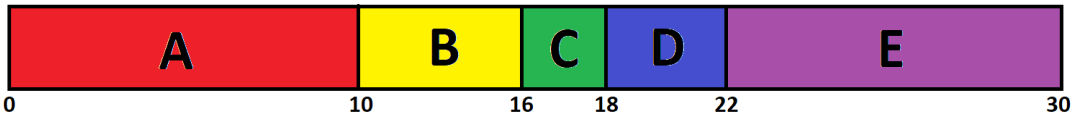
For (b), assume the system is multi-programmed with a quantum of 4 seconds. In all cases, show your work and include diagrams/charts/tables as appropriate.

A: Here is the table we'll be referring throughout all parts of this answer:

Thrd Name	Arr Time	Run Time Estimate	Priority
A	0	10	3
B	1	6	5
C	2	2	2
D	3	4	1
E	4	8	4

a) FCFS:

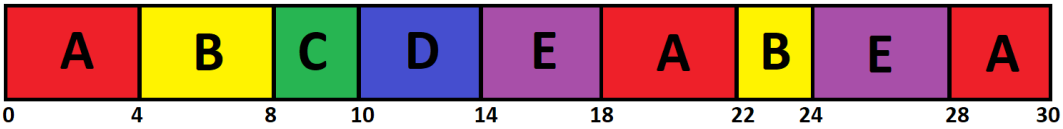
Here is a diagram of what thread runs when and for how long:



Mean Turn Around Time:
[(10-0)+(16-1)+(18-2)+(22-3)+(30-4)]/5=17.2 seconds
Mean Wait Time:
[(0-0)+(10-1)+(16-2)+(18-3)+(22-4)]/5=11.2 seconds

b) RR w/quantum=4 seconds:

Here is a diagram of what thread runs when and for how long:

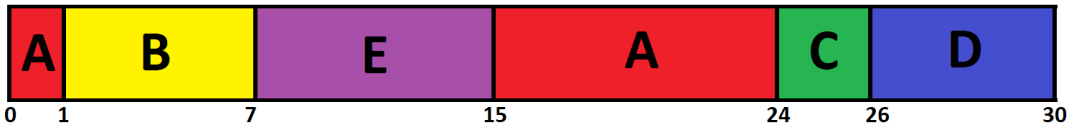


Mean Turn Around Time:
[(30-0)+(24-1)+(10-2)+(14-3)+(28-4)]/5=19.2 seconds

Mean Wait Time:
[(18-4)+(28-22)+(4-1)+(22-8)+(8-2)+(10-3)+(14-4)+(24-18)]/5
=13.2 seconds

c) Preemptive Priority:

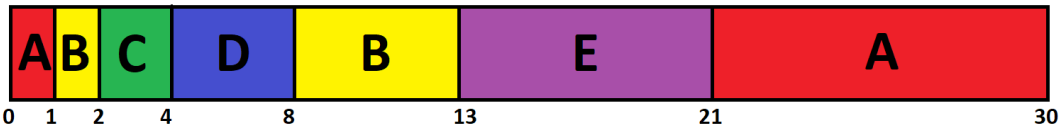
Here is a diagram of what thread runs when and for how long:



Mean Turn Around Time:
[(24-0)+(7-1)+(26-2)+(30-3)+(15-4)]/5=18.4 seconds
Mean Wait Time:
[(15-1)+0+(24-2)+(26-3)+(7-4)]/5=12.4 seconds

d) Preemptive SJF:

Here is a diagram of what thread runs when and for how long:



Mean Turn Around Time:
[(30-0)+(13-1)+(4-2)+(8-3)+(21-4)]/5=13.2 seconds
Mean Wait Time:
[(21-1)+(8-2)+0+(4-3)+(13-4)]/5=7.2 seconds

5) *If a hard real-time system has four tasks with periods of 50, 100, 200, and 250 ms (milliseconds) respectively, and the four tasks require 35, 20, 10, and X ms of CPU time respectively, calculate the largest value of X for which the system is schedulable during the period of the fourth task, and state the scheduling algorithm used. Show all your work and include charts as appropriate.*

A: Using the formula:

- If there are **m** periodic events and event **i** occurs with period P_i and requires C_i time units of CPU time to handle each event, then the load can be handled only if:

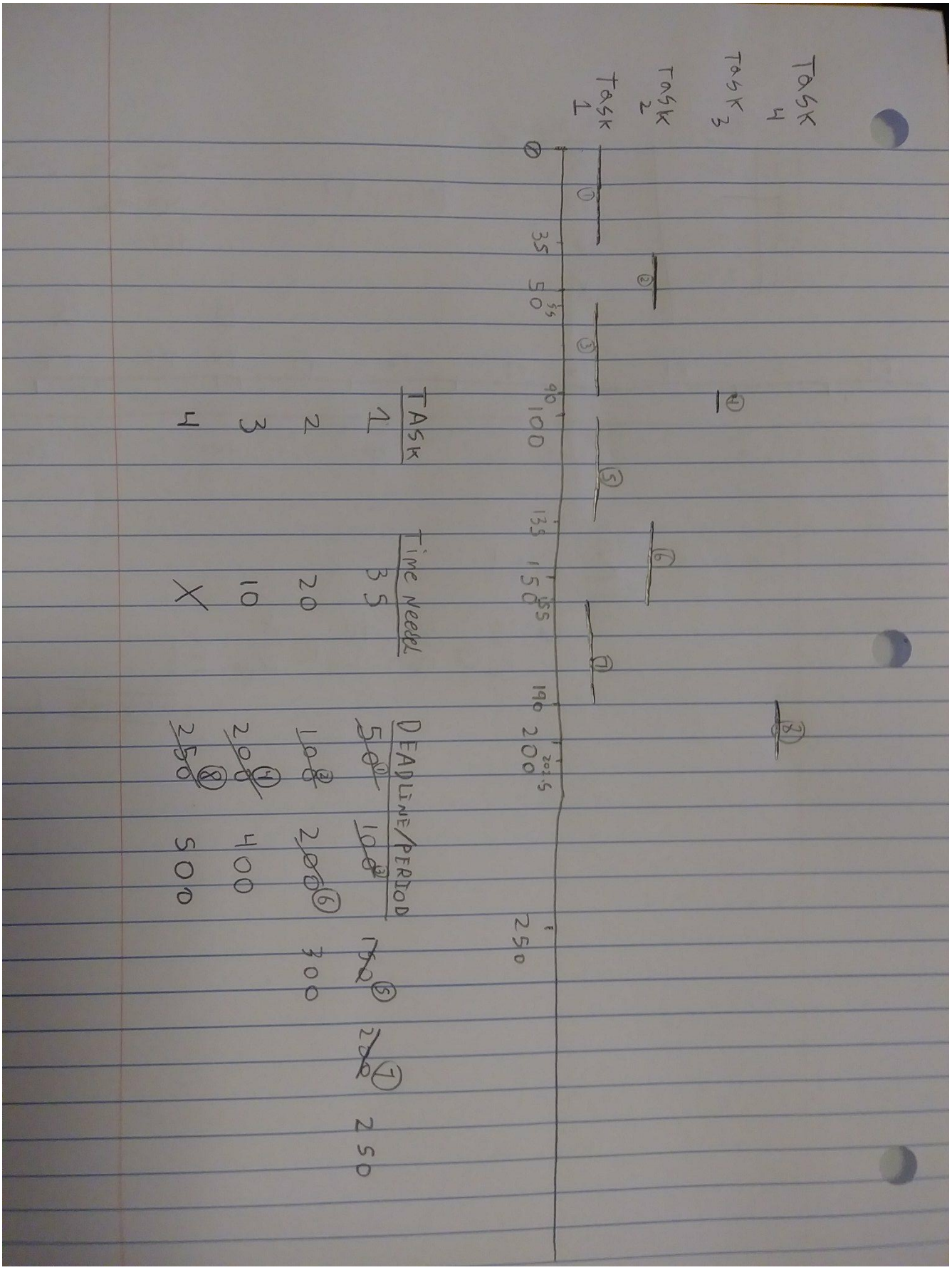
$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- **However**:
  - Satisfying this equation does ***not*** mean a schedule exists, it means one ***might*** exist if you can find a suitable scheduling algorithm
  - But ***failing*** to satisfy this equation absolutely means ***no schedule exists*** regardless of scheduling algorithm

Stephen Woodbury – 1429496 – swoodbur
HW3 - CMPS 111 – Spring 2018 – 5/9/2018 - Wednesday

We can computer the following based on the information provided:
(35/50)+(20/100)+(10/200)+(X/250) <= 1.
We find that X can be 12.5ms max. To determine if X can be 12.5 ms or not, we will user the Earliest Deadline First Algorithm. Provided is a rough sketch of what the diagram to the EDF algorithm would look like.

Stephen Woodbury – 1429496 – swoodbur
HW3 - CMPS 111 – Spring 2018 – 5/9/2018 - Wednesday



Stephen Woodbury – 1429496 – swoodbur
HW3 - CMPS 111 – Spring 2018 – 5/9/2018 - Wednesday

As we can see, 12.5ms is possible for X.
This means that the greatest that X can be is 12.5ms