Lecture 4 – Process Scheduling Algorithms

In the previous class we compared the non preemptive algorithms SJF and FCFS. We recall the performance advantages of SJF vs the fairness and simplicity advantages of FCFS. To eliminate the starvation problem for long tasks when SJF is used, we proposed the idea of prioritising tasks based on the time they have been waiting in the ready queue, so that the priority of long tasks will increase as they age in the queue.

We defined the priority calculation known as Response Ratio = (Waiting Time) / (Service Time) and the Highest Response Ratio Next (HRN) algorithm chooses the task with the largest response ratio value to run next.

So let's look at an example of how this works in practice, and then we will move on to look at preemptive scheduling.

Arrival	Job	CPU Burst
0	P1	24
0	P2	3
0	P3	3
5	P4	5
10	P5	3

So five tasks arrive into the system in the order and at the times shown with the required CPU usage times given also. So the problem is to determine the order in which a non preemptive HRN scheduler will choose to run the tasks.

At time 0, when the scheduler is making its first choice, there are only three tasks in the system, P1,P2,P3. As the waiting time for all so far is 0 then the response ratios would all be 0, so for the sake of distinguishing between them, let's assume that the arrival time is a tiny fraction more than zero to give us more meaningful response ratios.

Time	Job	Response Ratio	
0	P1	$0^+/24$	
0	P2	$0^+/3$	← Selected
0	P3	$0^+/3$	

So we see the response ratios of the two shorter tasks are slightly bigger than that of the longer task, so the shorter tasks will be chosen first. It doesn't matter which one is chosen first as the response priorities are the same for both, let's assume P2 is chosen as it is higher in the arrival ordering.

So P2 is chosen by the HRN scheduler and executes until it completes its CPU burst at time 3 (as this is non preemptive scheduling).

Time	Job	Response Ratio	
3	P1	3/24	
3	P3	3/3	Selected

At time 3, there are now only two tasks left in the ready queue, the priorities are recomputed, waiting time / service time, and we see P3 has the highest ratio. So P3 is chosen next.

P3 then completes at time 6.

Time	Job	Response Ratio	
6	P1	6/24	Selected
6	P4	1/5	

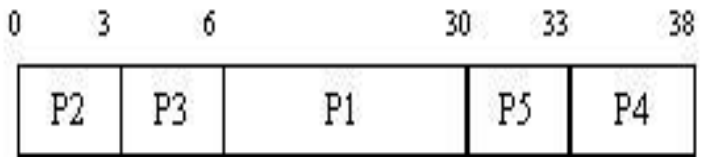
At time 6, the HRN scheduler must make another scheduling decision as the CPU becomes free once more. Now, when P2 was executing, a new process P4 arrived at time 5 and has now been waiting for 1 time unit. The long process P1 has been waiting since the start. When the priorities are recomputed we see that P1 has a higher priority and will be chosen next. So this is the idea of how to eliminate starvation, although the shorter tasks are favoured by the HRN algorithm, the longer tasks will eventually be scheduled.

So P1 executes next and completes at time 30.

Time	Job	Response Ratio	
30	P4	25/5	
30	P5	20/3	Selected

A new process has arrived into the system at time 10 while P1 was executing, so when P1 is complete, there are now two processes to choose from.

As P5 is shorter and has also been waiting for a while, its priority turns out to be better than P4. So P5 is scheduled next and then P4 is the final task scheduled. The resulting sequence of execution by the HRN scheduler is as follows:-

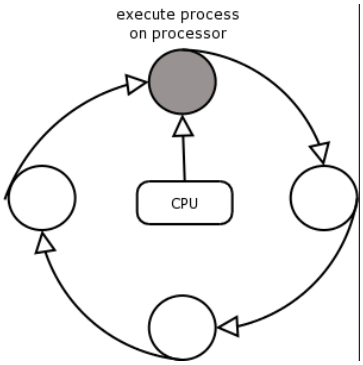


Preemptive Scheduling

Non preemptive scheduling means that when a scheduler chooses the next process to run, that process can continue executing for as long as it likes on the CPU until it is finished its burst and then voluntarily gives control of the CPU back to the scheduler.

Obviously that type of system would be completely subject to abuse in any kind of multiuser or multitasking system where a malicious or computationally intensive process might hog the cpu indefinitely. In multitasking systems, scheduling decisions are made at periodic intervals rather than when a process finishes its CPU burst. So the scheduler chooses a process and sets a hardware timer. When the hardware timer sends an interrupt, the scheduler runs again and chooses another process instead. So it reevaluates and preempts its scheduling decisions at regular intervals, known as a quantum or time slice.

A common type of algorithm used as the basis for scheduling in multitasking systems is known as Round Robin. Round Robin is chosen because it is both fair and offers good response time to all processes, which is important for achieving satisfactory interactivity performance in multitasking systems.



Round Robin is a preemptive version of FCFS, so it is fair.

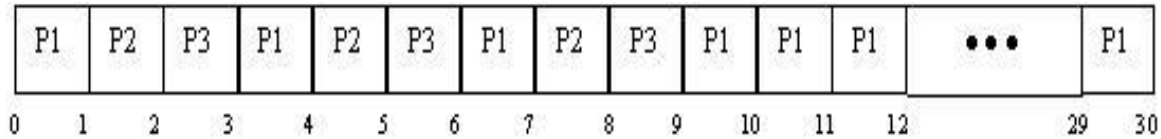
Each process executes in turn until its quantum expires forcing a task context switch to the next one in the queue.

The quantum duration can be varied, to give best results for a particular workload. If the quantum is very long, the algorithm degenerates to the performance of nonpreemptive FCFS, if the quantum is very short then there a lot of task switching overhead is introduced.

Job	CPU (Burst Time)
1	24
2	3
3	3

Let's look at how this collection of tasks would fair using a round robin scheduling algorithm.

Using Round Robin (RR) with a quantum of 1 we get the following order of execution:-



So the scheduler chooses the first task in the queue and runs it for 1 time unit. That task is put to the back of the queue, and the next task is chosen and runs for one time unit and so on. We see from the Gantt chart that at time 8, P2 finishes its 3rd quantum and that completes its cpu burst time. Similarly, P3 finishes at time 9 after its 3rd quantum and that also complete its cpu burst time.

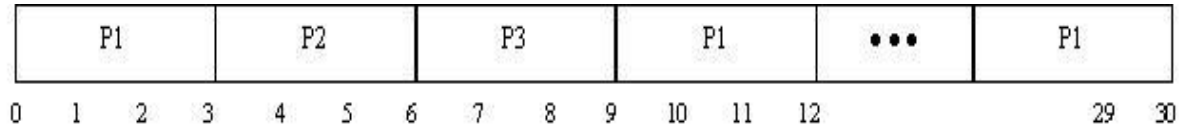
There is only one process, P1, remaining in the queue, so it is chosen to execute. When the quantum expires, the scheduler looks again at the queue and as there are no other processes, P1 is chosen again. So that is repeated for the remainder of P1's CPU burst time. In total P1 will need 24 separate time slices to complete its burst by time 30.

Looking at the performance of Round Robin with a quantum = 1, we see the following:-

Job	Waiting Time	Response Time	Turnaround Time
1	6	0	30
2	5	1	3
3	6	2	6
Average	5.666	1	15.666

If you look at the Gantt Chart for P1, it entered the queue at time 0 and left at time 30. It completed an execution burst of 24, so the total time it was waiting to execute in the queue was 6.
As P1 was the first task to be scheduled, there is no initial waiting time, so the response time is 0.
It is slightly higher for P2 as it wasn't first scheduled until time 1 and higher again for P3 as it wasn't first scheduled until time 2. In general we see that the average response time for the 3 tasks, i.e. the time between when they entered the queue and when they each received their first time slice, this average is very low and this is why round robin is a good match for an interactive system.

To see the effect of changing the quantum length on the performance, consider using Round Robin with a quantum of 3 instead. We would get the following order of execution:-



Matching the quantum length to the average length of each process's CPU burst has improved some performance criteria in this example slightly.

Job	Waiting Time	Response Time	Turnaround Time
1	6	0	30
2	3	3	6
3	6	6	9
Average	5	3	15

Two of the three processes, P2 & P3, were able to complete their CPU burst in one time slice with slightly less average waiting time and turnaround time overall. But notice that the average response time has gone up.

Round robin algorithm incurs a greater number of task switches than non preemptive algorithms. Each task switch takes a certain amount of time for the CPU to change the process environment. Too many task switches (i.e. quantum too small) means a greater proportion of CPU time is spent doing task switches instead of useful work. If the quantum is too large than RR degenerates to FCFS with poor response times.

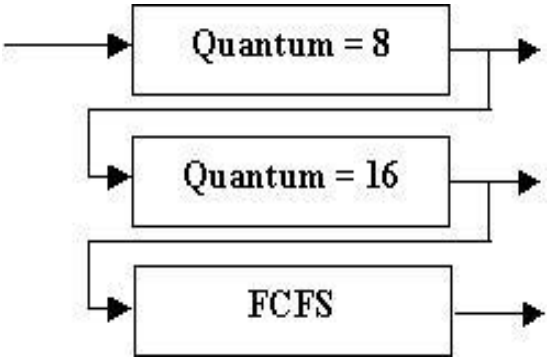
Other Scheduling Techniques

In reality, effective process scheduling is a little bit more complex than just using a revolving queue.

It is difficult to choose a time quantum to suit all jobs if there is a wide deviation in average CPU burst times.

The following scheme uses a **multilevel queueing system**, which is an adaptive approach to help reduce task switching overhead for long computational tasks, while maintaining a good response time for interactive processes.

Say we have 3 queues. Each queue is serviced using a specific algorithm. The operating system can decide what % of the CPU time should spend servicing each of the 3 queues, it might spend 50% on the top queue, 30% of its time on the middle queue and 20% on the bottom queue. The top level queue uses RR with Q=8, the middle queue uses RR with Q=16 and the bottom queue is serviced in a non preemptive FCFS manner. Processes can be moved down the queuing system depending on how they behave.



A process is initially submitted to the top level highest priority queue where it benefits from good response time. If its behaviour over time shows that it is computationally demanding then it can be demoted to a lower priority queue where it will incur less task switches due to more suitable CPU quantum length but the queue may receive less attention from the processor.

Priority Scheduling

Round Robin doesn't allow the user to tell the system which tasks are more important than others.

In a priority scheduling system, processes are assigned a numeric value which indicates their scheduling priority. We saw this earlier where the HRN algorithm computed a priority for each task automatically. But what if we wanted to influence that priority manually to allow a user or system to indicate and maintain the priority of one task over others.
Processes with the highest priority are always chosen first by the scheduler.
It is an easy scheme to implement but requires a means of calculating priorities so that lower priority tasks do not starve.

The Traditional Unix Scheduler

The traditional unix scheduler, which was originally designed for a time sharing multitasking environment, combines some of these ideas. It was designed to provide good response time for interactive user tasks while ensuring low priority background tasks don't starve and also that high priority system tasks can be done quickly.

It uses a priority scheme for process scheduling so that the highest priority processes are chosen first. It also operates round robin scheduling between processes of the same priority, and a process's priority can be changed dynamically in accordance with recent CPU usage so that it can be promoted or demoted.

A process priority is a value between 0 and 127. The lower the number, the higher the priority. Priorities 0 to 49 were for Kernel processes, and priorities from 50 to 127 for user processes.

The priority of all processes system wide is recalculated at one second intervals.
Let i represent the ith interval.

A process priority begins with an assigned base priority, used initially to place processes in bands of priority, like system or user.

Added to this base priority is a calculation from its recent CPU usage history. If a process j recently used the CPU, then its priority value is increased by adding half of its recent CPU usage time. This has the effect of decreasing its priority to the scheduler.

$$Pj(i) = Basej + CPUj(i) / 2$$

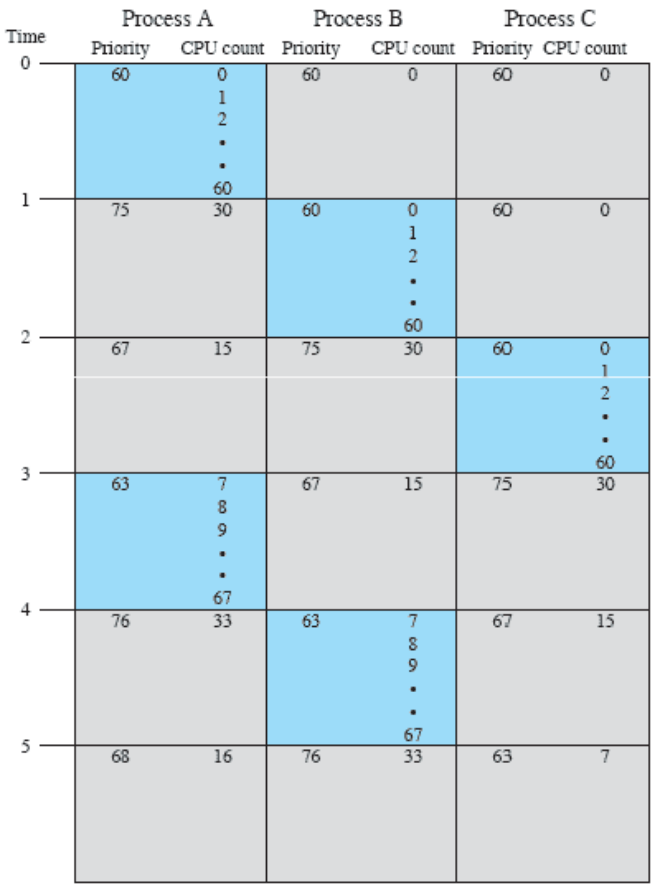
So this calculation is done for every process system wide, once a second.
The scheduler then chooses the process with the lowest priority value to run next.
While the process is running, a clock timer repeatedly interrupts the system to increment the CPU usage count of the current process.
At the end of 1 second, all process priorities are recomputed. This means recent processes will be demoted based on the calculation above and are less likely to be chosen.

To ensure processes are eventually rescheduled, the recorded CPU utilization of a process is decayed during each priority recalculation interval using the following formula:-

$$CPUj(i) = CPUj(i-1) / 2$$

So we halve the history of usage for every process after each 1 second interval and eventually it tends to 0.

Example: Traditional Unix Scheduler



Time	Process A		Process B		Process C	
	Priority	CPU count	Priority	CPU count	Priority	CPU count
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9 • • 67	67	15
5	68	16	76	33	63	7

We demonstrate the operation of traditional unix scheduling with an example.

The time line shows 1 second intervals for priority recalculation.

Within each interval the RR scheduler choses the highest priority process to run or the one at the top of the queue if priorities are equal.

A clock interrupts the system 60 times a second and updates the CPU usage of the currently running process.

The cpu usage of all processes is then halved and half of the resulting value is added to the process's base priority.

In this sequence, Process A is chosen first, but after 1 second, its new priority decreases to 75. So process B is chosen. After another 1 second interval, process A improves its priority to 67 while process B is demoted to 75. Process C is now the highest priority at 60 and is chosen next.
The blue shade shows the execution choices the scheduler makes over the first five one second intervals.

How to deal with fairness among multiple users?

In multiuser multitasking systems each user may be running a collection of their own tasks.
It is possible that one user or application may be running significantly less processes than another.
Our scheduling algorithms so far focus only on achieving fair allocation among the total set of processes, not on achieving fair allocation of CPU time among different users or different applications.

The scheme would require some alterations to the process priority calculation to include a usage of processor time for each group of processes as well as for each individual process.

For example, we could have individual process priority demoted by including a group usage value in its priority calculation.

Pj(i) = Basej + CPUj(i) / 2 + GCPUk(i) / 2

Where GCPUk(i) is the total amount of CPU usage of all processes in group k.

And we could decay GCPU in the same way over time

GCPUk(i) = GCPUk(i-1) / 2

So that a group isn't penalised forever.

Fair Share Scheduling

Process A in group one. Processes B and C in group 2.

Each group adds the same group cpu usage weighting to the priority of all its processes.

Pj(i) =
Basej + CPUj(i) / 2 +
GCPUk(i) / 2

We see here the scheduler chooses Process A first followed by process B, but Process C is not chosen next as the group to which it belongs has recently received CPU time so its priority is too low and the scheduler chooses process A again.