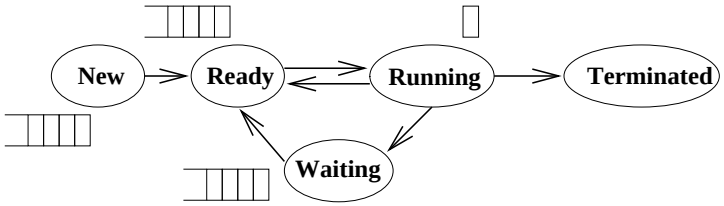
Processes — Last Time

- Each process maintains its own state, that includes its text and data, procedure call stack, etc.
- The OS also stores process state for each process. This state is called the *Process Control Block* (PCB), and it includes the PC, SP, register states, execution state, etc.
- Process Execution State



- For each of these execution states, the OS maintains a state queue. All of the processes that the OS is currently managing reside in one and only one of these state queues.

Process Scheduling: what to execute when — Today

- Scheduling criteria
- Scheduling algorithms

Scheduling Processes

Multiprocessing (concurrency) - one process on the CPU running, and one or more doing I/O enables the OS to increase system utilization and throughput by overlapping I/O and CPU activities.

Long Term Scheduling: How does the OS determine the degree of multiprogramming, i.e., the number of jobs executing at once in the primary memory?

Short Term Scheduling: How does (or should) the OS select a process from the ready queue to execute?

- Policy Goals
- Policy Options
- Implementation considerations

Short Term Scheduling

The kernel runs the scheduler at least when

1. a process switches from running to waiting,
2. an interrupt occurs, or
3. a process is created or terminated.

In a *non-preemptive* system, the scheduler must wait for one of these events, but in a *preemptive* system the scheduler can interrupt a running process.

Criteria for Comparing Scheduling Algorithms:

CPU Utilization The percentage of time that the CPU is busy.

Throughput The number of processes completing in a unit of time.

Turnaround time The length of time it takes to run a process from initialization to termination, including all the waiting time.

Waiting time The total amount of time that a process is in the ready queue.

Response time The time between when a process is ready to run and its next I/O request.

Scheduling Policies

Ideally, we would like a CPU schedule that maximizes CPU utilization and throughput while minimizing turnaround time, waiting time, and response time, but this is not generally possible. Instead, we choose a scheduling algorithm based on its ability to satisfy a policy goal:

- Minimize response time - provide output to the user as quickly as possible and process their input as soon as it is received.
- Minimize variance of average response time - in an interactive system, predictability may be more important than a low average with a high variance.
- Maximize throughput - two components
  1. minimize overhead (OS overhead, context switching)
  2. efficient use of system resources (CPU, I/O devices)
- Minimize waiting time - be *fair* by ensuring each process waits the same amount of time. This goal often increases average response time.

Scheduling Policies

Simplifying Assumptions

- One process per user
- One thread per process (more on this topic next week)
- Processes are independent

  Researchers developed these algorithms in the 70's when these assumptions were more realistic, and it is still an open problem how to relax these assumptions.

Scheduling Algorithms:

FCFS: First Come, First Served

Round Robin: Use a time slice and preemption to alternate jobs.

SJF: Shortest Job First

Multilevel Feedback Queues: Round robin on priority queue.

Lottery Scheduling: Jobs get tickets and scheduler randomly picks winning ticket.

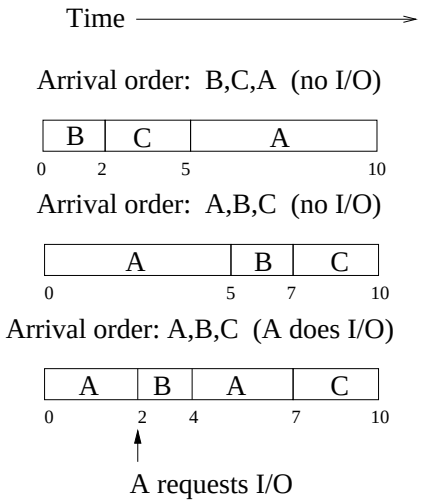
Scheduling Policies

FCFS: First-Come-First-Served (or FIFO: First-In-First-Out)

- The scheduler executes jobs to completion in arrival order.
- In early FCFS schedulers, the job did not relinquish the CPU even when it was doing I/O.
- We will assume a FCFS scheduler that runs when processes are blocked on I/O, but that is non-preemptive, i.e., the job keeps the CPU until it blocks (say on an I/O device).

FCFS Scheduling Policy

Examples:



If the processes arrive 1 time unit apart, what is the average wait time in these three cases?

Advantages:

-
-

Disadvantages:

-
-

Scheduling Policies

Round Robin: most time sharing systems use some variation of this policy.

- Add a timer and use a preemptive policy.
- After each time slice, move the running thread to the back of the queue.
- Selecting a time slice:
  - Too large - waiting time suffers, degenerates to FCFS if processes are never preempted.
  - Too small - throughput suffers because too much time is spent context switching.

  $\Rightarrow$ Balance the two by selecting a time slice where context switching is roughly 1% of the time slice.

  A typical time slice today is between 10-100 milliseconds, with a context switch time of 0.1 to 1 millisecond.

Round Robin Examples

- 5 jobs, 100 seconds each, time slice 1 second, context switch time of 0

Job	length	Completion Time		Wait Time	
		FCFS	Round Robin	FCFS	Round Robin
1	100				
2	100				
3	100				
4	100				
5	100				
Average					

- 5 jobs, of length 50, 40, 30, 20, and 10 seconds each, time slice 1 second, context switch time of 0 seconds

Job	length	Completion Time		Wait Time	
		FCFS	Round Robin	FCFS	Round Robin
1	50				
2	40				
3	30				
4	20				
5	10				
Average					

Advantages:

Disadvantages:

SJF/SRTF: Shortest Job First

- Schedule the job that has the least (expected) amount of work (CPU time) to do until its next I/O request or termination.

$\Rightarrow$ I/O bound jobs get priority over CPU bound jobs.

Example: 5 jobs, of length 50, 40, 30, 20, and 10 seconds each, time slice 1 second, context switch time of 0 seconds

Job	length	Completion Time			Wait Time		
		FCFS	RR	SJF	FCFS	RR	SJF
1	50						
2	40						
3	30						
4	20						
5	10						
Average							

- Works for preemptive and non-preemptive schedulers.
- Preemptive SJF is called SRTF - shortest remaining time first.

Advantages:

-
-

Disadvantages:

-
-

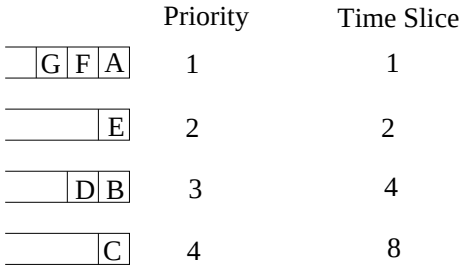
Multilevel Feedback Queues:

Using the Past to Predict the Future: Multilevel feedback queues attempt to overcome the prediction problem in SJF by using the past I/O and CPU behavior to assign process priorities.

- If a process is I/O bound in the past, it is also likely to be I/O bound in the future (programs turn out not to be random.)
- To exploit this behavior, the scheduler can favor jobs (schedule them sooner) when they use very little CPU time (absolutely or relatively), thus approximating SJF.
- This policy is **adaptive** because it relies on past behavior and changes in behavior result in changes to scheduling decisions.

Approximating SJF: Multilevel Feedback Queues

- Multiple queues with different priorities.
- OS uses Round Robin scheduling at each priority level, running the jobs in highest priority queue first.
- Once those finish, OS runs jobs out of the next highest priority queue, etc. (Can lead to starvation.)
- Round robin time slice increases exponentially at lower priorities.



Adjust priorities as follows (details can vary):

1. Job starts in highest priority queue.
2. If job's time slices expires, drop its priority one level.
3. If job's time slices does not expire (the context switch comes from an I/O request instead), then increase its priority one level, up to the top priority level.

$\Longrightarrow$ In practice, CPU bound jobs drop like a rock in priority and I/O bound jobs stay at a high priority.

Multilevel Feedback Queues

Example: 3 jobs, of length 30, 20, and 10 seconds each, initial time slice 1 second, context switch time of 0 seconds, all CPU bound (no I/O), 3 queues .

Job	length	Completion Time RR	Completion Time MLFB	Wait Time RR	Wait Time MLFB
1	30				
2	20				
3	10				
Average					

Queue	Time Slice	Job
1	1	
2	2	
3	4	

Multilevel Feedback Queues

Example: 3 jobs, of length 30, 20, and 10 seconds, the 10 sec job has 1 sec of I/0 every other sec, initial time slice 2 sec, context switch time of 0 sec, 2 queues.

Job	length	Completion Time RR	Completion Time MLFB	Wait Time RR	Wait Time MLFB
1	30				
2	20				
3	10				
Average					

Queue	Time Slice	Job
1	2	
2	4	

Improving Fairness

Since SJF is optimal, but unfair, any increase in fairness by giving long jobs a fraction of the CPU when shorter jobs are available will degrade average waiting time. Possible solutions:

- Give each queue a fraction of the CPU time. This solution is only fair if there is an even distribution of jobs among queues.
- Adjust the priority of jobs as they do not get serviced (Unix originally did this.) This ad hoc solution avoids starvation but average waiting time suffers when the system is overloaded because all the jobs end up with a high priority,.

Lottery Scheduling

- Give every job some number of lottery tickets.
- On each time slice, randomly pick a winning ticket.
- On average, CPU time is proportional to the number of tickets given to each job.
- Assign tickets by giving the most to short running jobs, and fewer to long running jobs (approximating SJF). To avoid starvation, every job gets at least one ticket.
- Degrades gracefully as load changes. Adding or deleting a job affects all jobs proportionately, independent of the number of tickets a job has.

Lottery Scheduling

Example: Short jobs get 10 tickets, long jobs get 1 ticket each.

# short jobs / # long jobs	% of CPU each short job gets	% of CPU each long job gets
1/1	91%	9%
0/2		
2/0		
10/1		
1/10		

Summary of Scheduling Algorithms:

FCFS: Not fair, and average waiting time is poor.

Round Robin: Fair, but average waiting time is poor.

SJF: Not fair, but average waiting time is minimized assuming we can accurately predict the length of the next CPU burst. Starvation is possible.

Multilevel Queuing: An implementation (approximation) of SJF.

Lottery Scheduling: Fairer with a low average waiting time, but less predictable.

$\Longrightarrow$ Our modeling assumed that context switches took no time, which is unrealistic.

Next Time:

- Threads: multiple coordinating processes