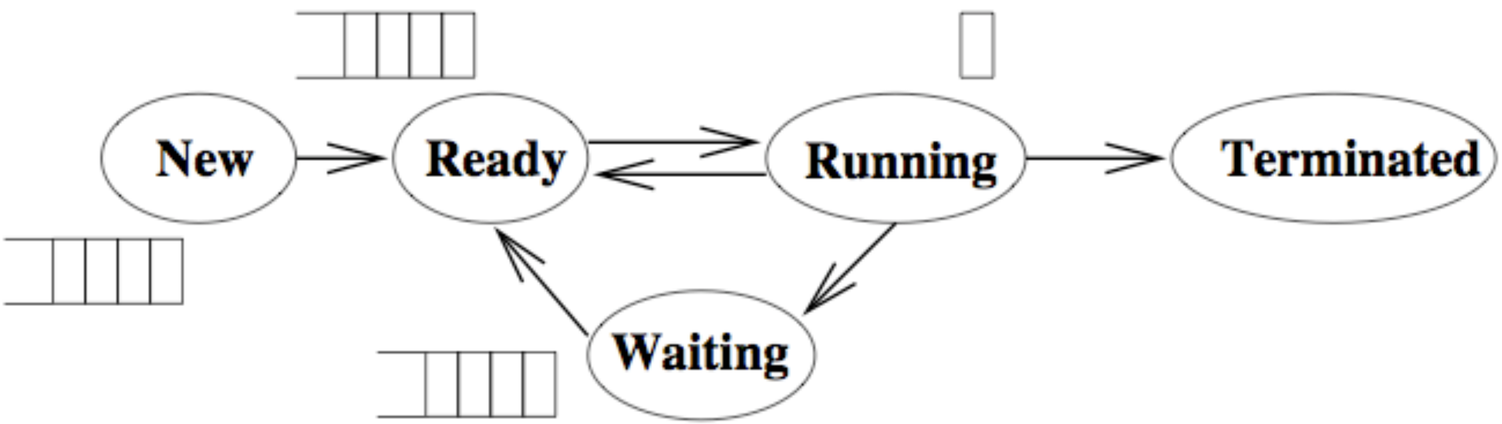
Scheduling Processes

Don Porter

Portions courtesy Emmett Witchel

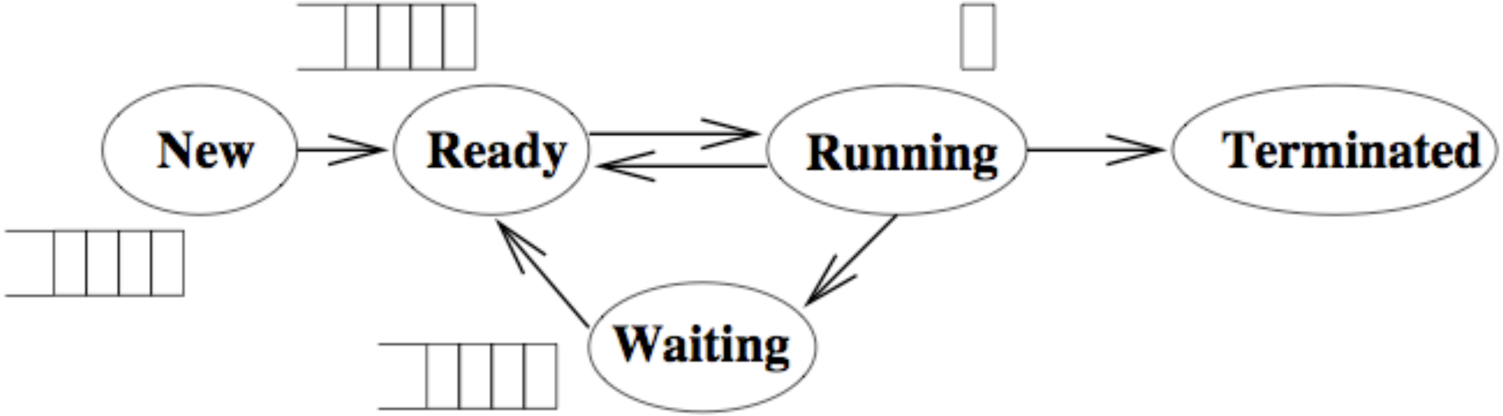
Processes (refresher)

- Each process has state, that includes its text and data, procedure call stack, etc. This state resides in memory.
- The OS also stores process metadata for each process. This state is called the Process Control Block (PCB), and it includes the PC, SP, register states, execution state, etc.
- All of the processes that the OS is currently managing reside in one and only one of these states.



Scheduling Processes

- The OS has to decide:
  - When to take a Running process back to Ready
  - Which process to select from the Ready queue to run next
- Ready Queue: Policy can be something other than First-in, First-out!



Scheduler

- The kernel runs the scheduler at least when
  - a process switches from running to waiting (blocks)
  - a process is created or terminated.
  - an interrupt occurs (e.g., timer chip)

- Non-preemptive system
  - Scheduler runs when process blocks or is created, not on hardware interrupts
- Preemptive system
  - OS makes scheduling decisions during interrupts, mostly timer, but also system calls and other hardware device interrupts

Evaluation Criteria and Policy Goals?

- **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.
- **Fairness**: ??

Scheduling Policies

- Ideal CPU scheduler
  - Maximizes CPU utilization and throughput
  - Minimizes turnaround time, waiting time, and response time
- Real CPU schedulers implement particular policy
  - 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.
- Will a fair scheduling algorithm maximize throughput? A) Yes B) No

Different Process Activity Patterns

- CPU bound
  - mp3 encoding
  - Scientific applications (matrix multiplication)
  - Compile a program or document
- I/O bound
  - Index a file system
  - Browse small web pages
- Balanced
  - Playing video
  - Moving windows around/fast window updates
- Scheduling algorithms reward I/O bound and penalize CPU bound
  - Why?

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.

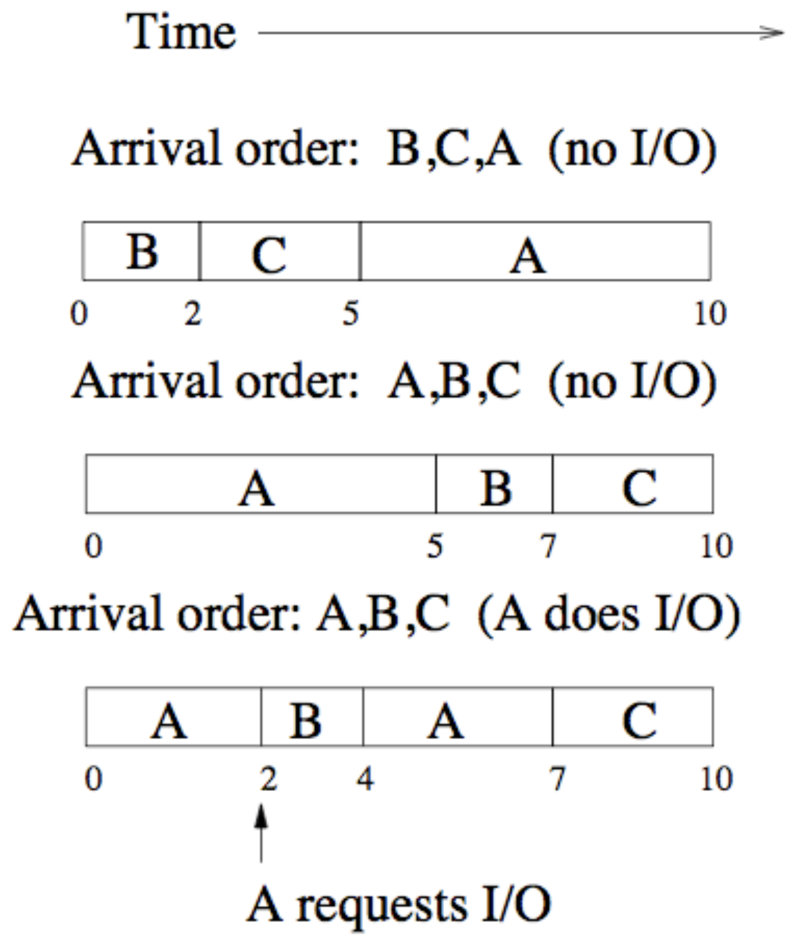
- Scheduling Algorithms to Evaluate Today:
  - FCFS: First Come, First Served
  - Round Robin: Use a time slice and preemption to alternate jobs.
  - SJF: Shortest Job First
  - Multilevel Feedback Queues: Round robin on priority queue.
  - Lottery Scheduling: Jobs get tickets and scheduler randomly picks winning ticket.

Policy 1: FCFS (First Come, First Served)

- 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 Example and Analysis

- 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.
- If the processes arrive one time unit apart, what is the average wait time in these three cases?
- Advantages:
- Disadvantages



Policy 2: Round Robin

- Run each process for its time slice (scheduling quantum)
- 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.
  - 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.
  - Max Linux time slice is 3,200ms, Why?
- Is round robin more fair than FCFS? A)Yes B)No

Round Robin Example (1)

- 5 jobs, 100 seconds each, time slice 1 second, context switch time of 0, jobs arrive at time 0,1,2,3,4

		Completion Time (Wall clock time)		Wait Time = (Compl. – Arrival – exec time)	
Job	Length	FCFS	Round Robin	FCFS	Round Robin
1	100				
2	100				
3	100				
4	100				
5	100				
Average					

Round Robin Example (1)

- 5 jobs, 100 seconds each, time slice 1 second, context switch time of 0, jobs arrive at time 0,1,2,3,4

		Completion Time (Wall clock time)		Wait Time = (Compl. – Arrival – exec time)	
Job	Length	FCFS	Round Robin	FCFS	Round Robin
1	100	100		0	
2	100	200		99	
3	100	300		198	
4	100	400		297	
5	100	500		396	
Average		300		198	

Round Robin Example (1)

- 5 jobs, 100 seconds each, time slice 1 second, context switch time of 0, jobs arrive at time 0,1,2,3,4

		Completion Time (Wall clock time)		Wait Time = (Compl. – Arrival – exec time)	
Job	Length	FCFS	Round Robin	FCFS	Round Robin
1	100	100	496	0	396
2	100	200	497	99	396
3	[illegible]	[illegible]	498	198	396
4	[illegible]	[illegible]	499	297	396
5	[illegible]	500	500	396	396
Average		300	498	198	396

Why is this better?

Round Robin Example (2)

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

		Completion Time (Wall clock time)		Wait Time = (Compl. – Arrival – exec time)	
Job	Length	FCFS	Round Robin	FCFS	Round Robin
1	50				
2	40				
3	30				
4	20				
5	10				
Average					

Round Robin Example (2)

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

		Completion Time (Wall clock time)		Wait Time = (Compl. - Arrival - exec time)	
Job	Length	FCFS	Round Robin	FCFS	Round Robin
1	50	50		0	
2	40	90		50	
3	30	120		90	
4	20	140		120	
5	10	150		140	
Average		110		80	

Round Robin Example (2)

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

		Completion Time (Wall clock time)		Wait Time = (Compl. – Arrival – exec time)	
Job	Length	FCFS	Round Robin	FCFS	Round Robin
1	50	50	150	0	100
2	40	90	140	50	100
[illegible]	[illegible]	120	120	90	90
[illegible]	[illegible]	140	90	120	70
5	10	150	50	140	40
Average		110	110	80	80

Seriously, aren't these the same?

Round Robin Example (2)

- 5 jobs, of length 50, 40, 30, 20, and 10 seconds each, time slice 1 second, context switch time of 0 seconds, arrival times 0, 1, 2, 3, 4

		Completion Time (Wall clock time)		Wait Time = (Compl. – Arrival – exec time)	
Job	Length	FCFS	Round Robin	FCFS	Round Robin
1	50	50	150	0	100
2	40	90	140	49	99
[illegible]	[illegible]	120	120	88	88
[illegible]	[illegible]	140	90	117	68
5	10	150	50	136	36
Average		110	110	78	78

Seriously, aren't these the same?

Fairness

- Was the average wait time or completion time really the right metric?
  - No!
- What should we consider for the example with equal job lengths?
  - Variance!
- What should we consider for the example with varying job lengths?
  - Is completion time proportional to required CPU cycles?

Policy 3: Shortest Job First (SJF)

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

Shortest Job First (SJF) Example

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

,		Completion Time (Wall Clock Time)			Wait Time (Compl. – Exec – Arrival)		
Job	Length	FCFS	RR	SJF	FCFS	RR	SJF
1	50						
2	40						
3	30						
4	20						
5	10						
Average							

Shortest Job First (SJF) Example

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

		Completion Time			Wait Time		
Job	Length	FCFS	RR	SJF	FCFS	RR	SJF
1	50	50	150	150	0	100	100
[illegible]	[illegible]	[illegible]	140	100	50	100	60
[illegible]	[illegible]	[illegible]	120	60	90	90	30
4	[illegible]	[illegible]	90	30	120	70	10
5	10	150	[illegible]	10	140	40	0
Average		110	110	70	80	80	40

Now that's what I'm talking about!

Shortest Job First

- Works for preemptive and non-preemptive schedulers.
- Preemptive SJF is called SRTF - shortest remaining time first.
- Advantages?
  - Free up system resources more quickly
- Disadvantages?
  - How do you know how long something will run?

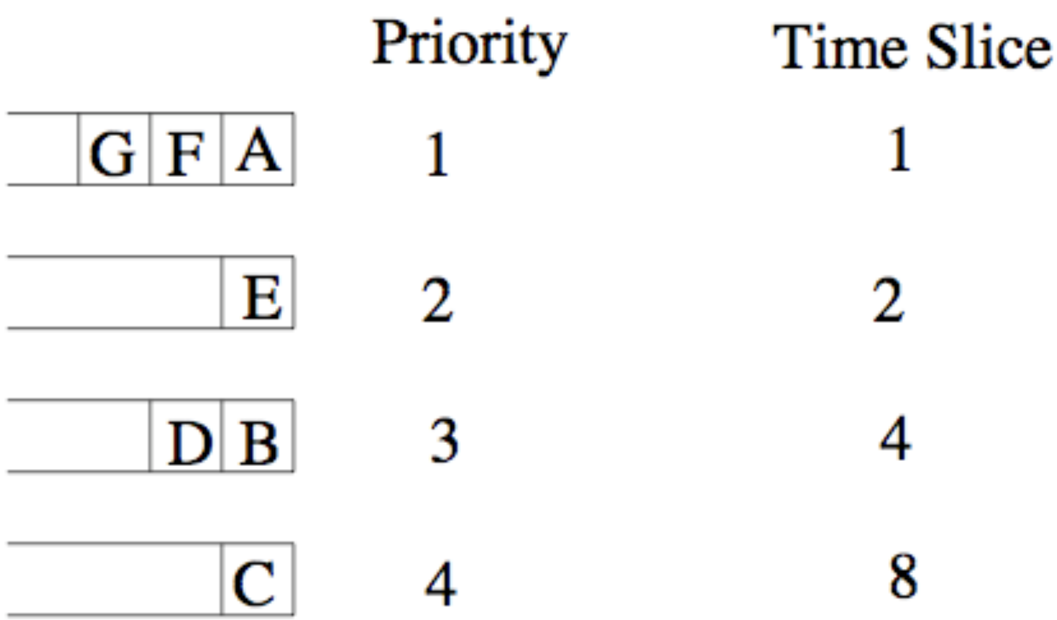
"Academic" scheduler: Useful to decide if a good idea

Idea: Use the Past to Predict the Future

- Intuition: Assign a **dynamic priority** to each task
  - Higher priority processes more likely to be scheduled
    - (if ready)
- Assign dynamic priority based on behavior during last few quanta
  - Raise dynamic priority frequently process blocks on I/O
    - Probably latency-sensitive (e.g., word processer, web server)
    - When runnable, will probably do a little work and block again on more I/O
  - Lower dynamic priority of processes that use all of their quantum
    - Probably CPU-bound
- **Adaptive:** priorities change when process changes behavior (e.g., switching from I/O to CPU-intensive)

Policy 4: Multi-Level Feedback Queues

- Approximate SJF: multiple queues with different priorities.
- OS uses Round Robin scheduling at each priority level, running the jobs in the 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.
  - *Good* for CPU-bound jobs to be lower priority (if they don't starve)

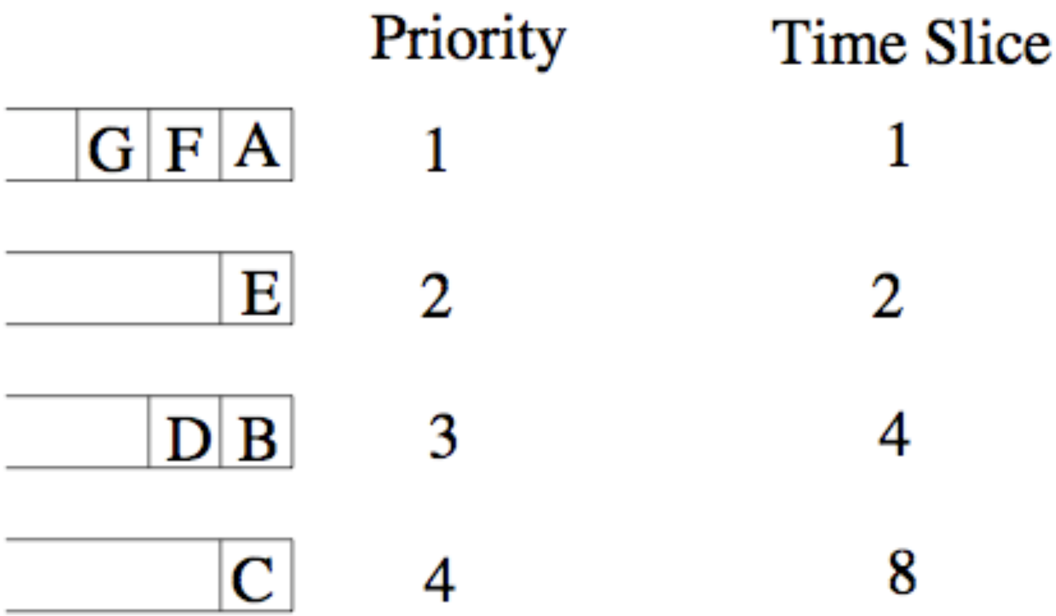


Policy 4: Multi-Level Feedback Queues

Adjust priorities as follows (details can vary):

1. Proc starts in the highest priority queue
2. If proc's time slice expires, drop its priority one level.
3. If proc's blocked with remaining time slice, increase its priority one level, up to the top priority level.

==> In practice, CPU bound procs drop like a rock in priority and I/O bound procs stay at high priority



Fairness

- SJF is optimal, but unfair
- Improving fairness means giving long jobs a fraction of the CPU when shorter jobs are available
  - Will degrade average waiting time.
- Possible solutions:
  - Give each level 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.)
    - Avoids starvation
    - Average waiting time suffers when the system is overloaded because all the jobs end up with a high priority.

Policy 5: 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

Example: Short jobs get 9 tickets, long jobs get 1 tickets each.

# short jobs / # long jobs	% of CPU each short job gets	% of CPU each long job gets
1/1	90%	10%
0/2		
2/0		
10/1		
1/10		

Lottery Scheduling Example

Example: Short jobs get 9 tickets, long jobs get 1 tickets each.

# short jobs / # long jobs	% of CPU each short job gets	% of CPU each long job gets
1/1	90%	10%
0/2	0%	50%
2/0		
10/1		
1/10		

Lottery Scheduling Example

Example: Short jobs get 9 tickets, long jobs get 1 tickets each.

# short jobs / # long jobs	% of CPU each short job gets	% of CPU each long job gets
1/1	90%	10%
0/2	0%	50%
2/0	50%	0%
10/1		
1/10		

Lottery Scheduling Example

Example: Short jobs get 9 tickets, long jobs get 1 tickets each.

# short jobs / # long jobs	% of CPU each short job gets	% of CPU each long job gets
1/1	90%	10%
0/2	0%	50%
2/0	50%	0%
10/1	9/91=~9.8%	1/91=~1%
1/10		

Lottery Scheduling Example

Example: Short jobs get 9 tickets, long jobs get 1 tickets each.

# short jobs / # long jobs	% of CPU each short job gets	% of CPU each long job gets
1/1	90%	10%
0/2	0%	50%
2/0	50%	0%
10/1	9/91=~9.8%	1/91=~1%
1/10	9/19=~47%	1/19=~5.3%

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.

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