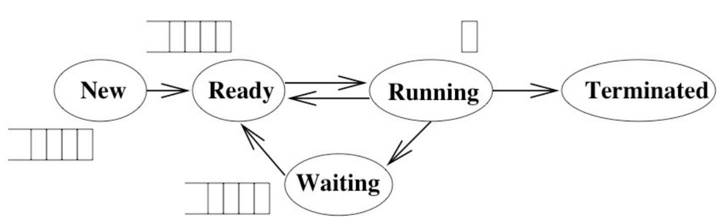
Today: Processor Scheduling

- Goals for processor scheduling
- CPU Scheduling Refresher
- CPU Scheduling in Minix

Scheduling Processes

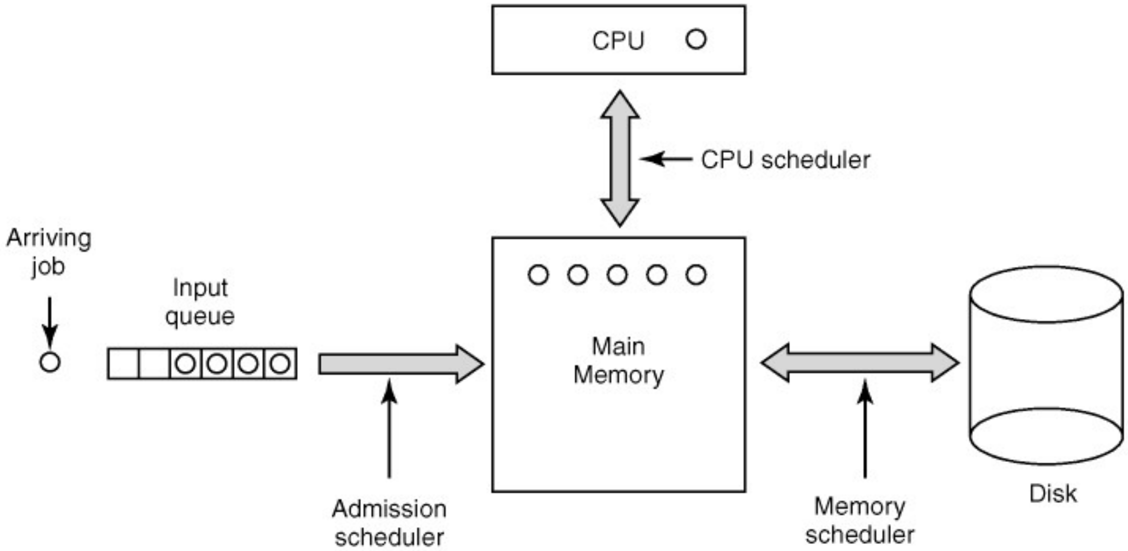
- **Multiprogramming**: running more than one process at a time enables the OS to increase system utilization and throughput by overlapping I/O and CPU activities.
- Process Execution State



- All of the processes that the OS is currently managing reside in one and only one of these state queues.

Three Level Scheduling

Long-term, short-term, memory



Scheduling Processes

- **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

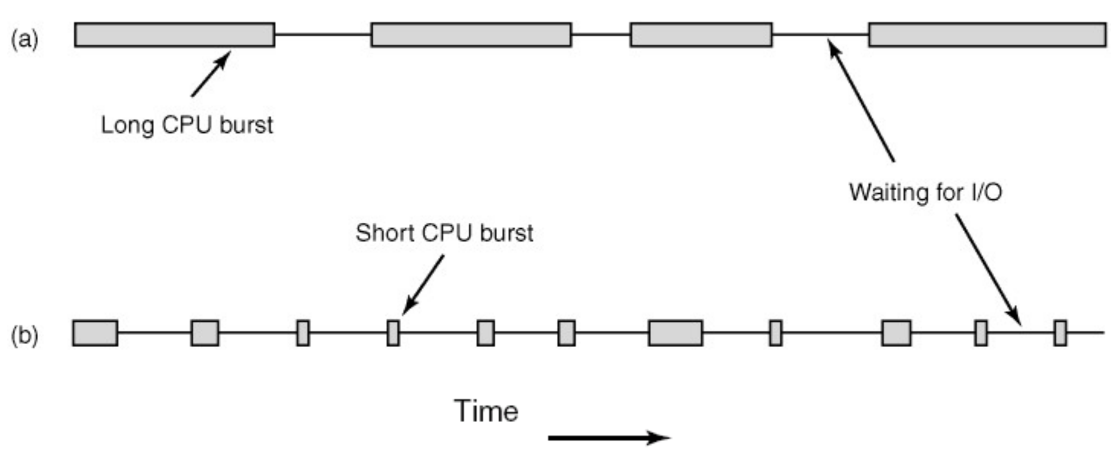
- The kernel runs the scheduler at least when
  1. a process switches from running to waiting,
  2. an interrupt occurs, or
  3. a process is created or terminated.

- **Non-preemptive system**: the scheduler must wait for one of these events

- **Preemptive system**: the scheduler can interrupt a running process

Process Behavior

Bursts of CPU usage alternate with periods of waiting for I/O.
(a) A CPU-bound process. (b) An I/O-bound 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.

Goals

All systems
Fairness — giving each process a fair share of the CPU
Policy enforcement — seeing that stated policy is carried out
Balance — keeping all parts of the system busy

Batch systems
Throughput — maximize jobs per hour
Turnaround time — minimize time between submission and termination
CPU utilization — keep the CPU busy all the time

Interactive systems
Response time — respond to requests quickly
Proportionality — meet users' expectations

Real—time systems
Meeting deadlines — avoid losing data
Predictability — avoid quality degradation in multimedia systems

Scheduling Policies

Ideally, choose a CPU scheduler that optimizes all criteria simultaneously (utilization, throughput,..), but this is not generally possible

Instead, choose a scheduling algorithm based on its ability to satisfy a policy

- Minimize average response time - provide output to the user as quickly as possible and process their input as soon as it is received.
- Minimize variance of response time - in interactive systems, predictability may be more important than a low average with a high variance.
- Maximize throughput - two components
  - minimize overhead (OS overhead, context switching)
  - efficient use of system resources (CPU, I/O devices)
- Minimize waiting time - give each process the same amount of time on the processor. This might actually increase average response time.

Scheduling Policies

Simplifying Assumptions

- One process per user
- One thread per process
- Processes are independent
- Singe processor, single core

Scheduling Algorithms: A Snapshot

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 each 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).

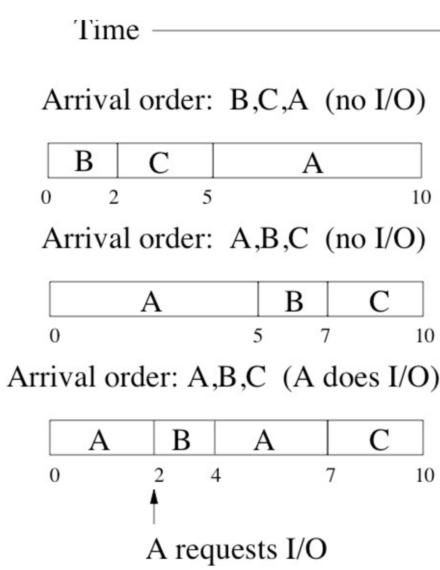
FCFS: Advantages and Disadvantages

Advantage: simple

Disadvantages:

- average wait time is highly variable as short jobs may wait behind long jobs.
- may lead to poor overlap of I/O and CPU since CPU-bound processes will force I/O bound processes to wait for the CPU, leaving the I/O devices idle

FCFS Scheduling Policy: Example



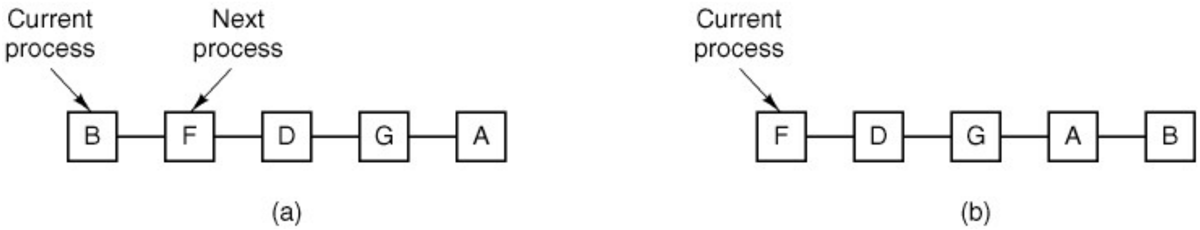
- If processes arrive 1 time unit apart, what is the average wait time in these three cases?

Round-Robin Scheduling

Round-robin scheduling.

(a) The list of runnable processes.

(b) The list of runnable processes after B uses up its quantum.



Round Robin Scheduling

- Variants of round robin are used in most time sharing systems
- 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.

  => Balance these tradeoffs by selecting a time slice where context switching is roughly 1% of the time slice.
- Today: typical time slice= 10-100 ms, context switch time= 0.1-1ms
- **Advantage:** It's fair; each job gets an equal shot at the CPU.
- **Disadvantage:** Average waiting time can be bad.

Round Robin Scheduling: Example 1

- 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					

Round Robin Scheduling: Example 1

- 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	100	496	0	396
2	100	200	497	100	397
3	100	300	498	200	398
4	100	400	499	300	399
5	100	500	500	400	400
Average		300	498	200	398

Round Robin Scheduling: Example 2

- 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					

Round Robin Scheduling: Example 2

- 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	50	150	0	100
2	40	90	140	50	100
3	30	120	120	90	90
4	20	140	90	120	70
5	10	150	50	140	40
Average		110	110	80	80

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.
- **Advantages:**
  - Provably optimal with respect to minimizing the average waiting time
  - Works for preemptive and non-preemptive schedulers
  - Preemptive SJF is called SRTF - shortest remaining time first

  => I/O bound jobs get priority over CPU bound jobs

- **Disadvantages:**
  - Impossible to predict the amount of CPU time a job has left
  - Long running CPU bound jobs can starve

SJF: 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							

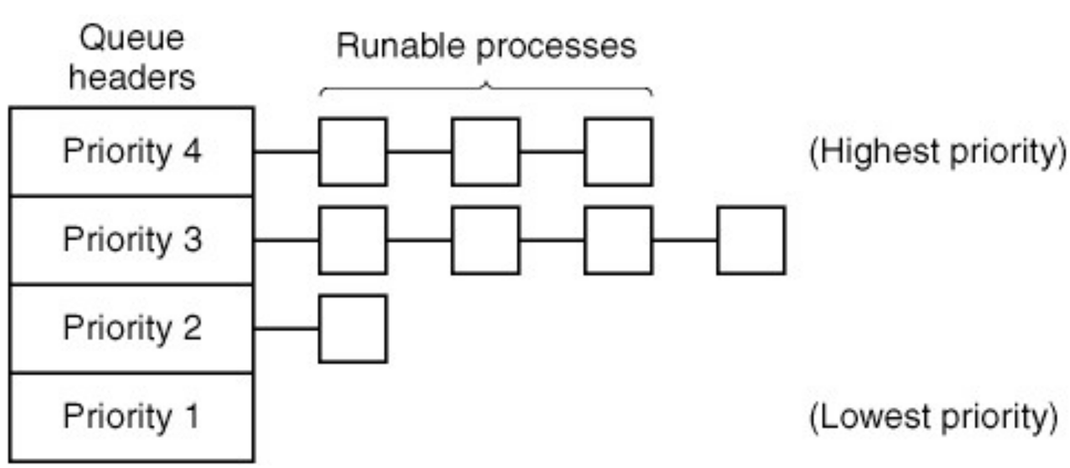
SJF: Example

- 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	50	150	150	0	100	100
2	40	90	140	100	50	100	60
3	30	120	120	60	90	90	30
4	20	140	90	30	120	70	10
5	10	150	50	10	140	40	0
Average		110	110	70	80	80	40

Priority Scheduling

A scheduling algorithm with four priority classes.



Multilevel Feedback Queues (MLFQ)

- Multilevel feedback queues use past behavior to predict the future and assign job priorities

  => overcome the prediction problem in SJF
- 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 that have used the least amount of CPU time, 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.
- Use Round Robin scheduling at each priority level, running the jobs in highest priority queue first.
- Once those finish, run jobs at the next highest priority queue, etc. (Can lead to starvation.)
- Round robin time slice increases exponentially at lower priorities.

	Priority	Time Slice
G F A	1	1
E	2	2
D B	3	4
C	4	8

Adjusting Priorities in MLFQ

- Job starts in highest priority queue.
- If job's time slices expires, drop its priority one level.
- 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.

⇒ CPU bound jobs drop like a rock in priority and I/O bound jobs stay at a high priority.

Multilevel Feedback Queues:Example 1

- 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

Queue	Time Slice	Job
1	1	
2	2	
3	4	

		Completion Time		**Wait Time**	
Job	Length	RR	MLFQ	RR	MLFQ
1	30				
2	20				
3	10				
Average					

Multilevel Feedback Queues:Example 1

- 5 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		Wait Time	
		RR	MLFQ	RR	MLFQ
1	30	60	60	30	30
2	20	50	53	30	33
3	10	30	32	20	22
Average		46 2/3	48 1/3	26	28 1/3

Queue	Time Slice	Job
1	1	$1_{1}^{1}, 2_{2}^{1}, 3_{3}^{1}$
2	2	$1_{5}^{3}, 2_{7}^{3}, 3_{9}^{3}$
3	4	$1_{13}^{7}, 2_{17}^{7}, 3_{21}^{7}$ $1_{25}^{11}, 2_{29}^{11}, 3_{32}^{10}$ ...

Multilevel Feedback Queues:Example 2

- 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 1 sec, context switch time of 0 sec, 2 queues.

Job	Length	Completion Time		Wait Time	
		RR	MLFQ	RR	MLFQ
1	30				
2	20				
3	10				
Average					

Queue	Time Slice	Job
1	1	
2	2	

Multilevel Feedback Queues:Example 2

- 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 1 sec, context switch time of 0 sec, 2 queues.

Job	Length	Completion Time		Wait Time	
		RR	MLFQ	RR	MLFQ
1	30	60	60	30	30
2	20	50	50	30	30
3	10	30	18	20	8
Average		46 2/3	45	26 2/3	25 1/3

Queue	Time Slice	Job
1	1	$1_1^1, 2_2^1, 3_3^1$ $3_6^3, 3_9^5, 3_{12}^7, 3_{15}^9, 3_{18}^{10}$
2	2	$1_5^3, 2_8^3, 1_{11}^5, 2_{14}^5, 1_{17}^7, 2_{12}^7,$ $1_{20}^9, 2_{22}^9, 1_{24}^{11}, 2_{26}^{11}, 1_{28}^{13}, 2_{30}^{13},$ $1_{32}^{15}, 2_{34}^{15}, 1_{36}^{17}, 2_{38}^{17}, 1_{40}^{19}, 2_{42}^{19},$

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		

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% (10/11)	9% (1/11)
0/2		50% (1/2)
2/0	50% (10/20)	
10/1	10% (10/101)	< 1% (1/101)
1/10	50% (10/20)	5% (1/20)

Minix: When to Schedule

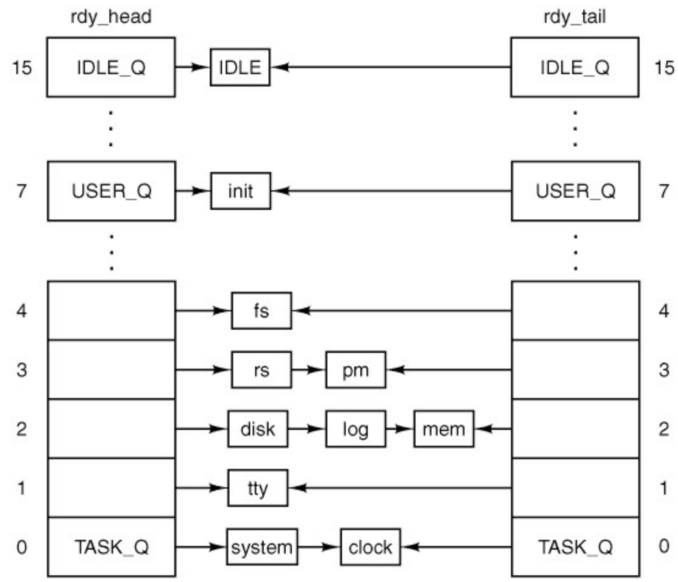
When scheduling is absolutely required:

1. When a process exits.
2. When a process blocks on I/O, or a semaphore.

When scheduling usually done (though not absolutely required)

1. When a new process is created.
2. When an I/O interrupt occurs.
3. When a clock interrupt occurs.

Scheduling in MINIX



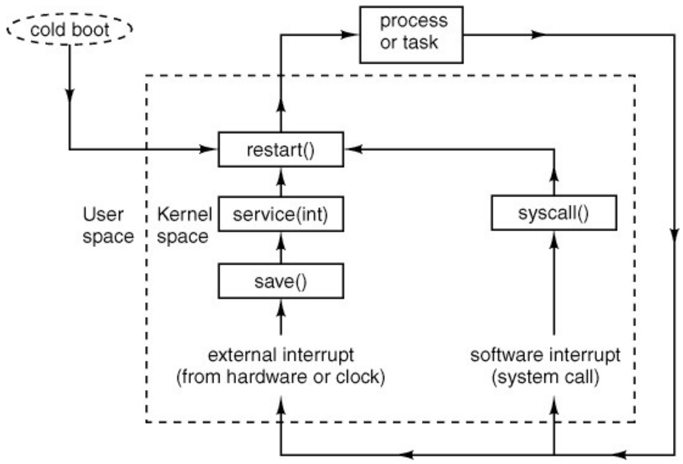
The scheduler maintains sixteen queues, one per priority level. Shown here is the initial queuing process as MINIX 3 starts up.

MLFQ Scheduling in Minix

- Priorities aligned with "**nice**" values in UNIX
  - lower number is **higher** priority, with 0 being highest
- Idle process runs at lowest priority level (level 15)
- System and clock u-kernel tasks run at highest (level 0)
- Drivers and system processes run at priority levels 1-4
- User processes run at priority 7 to 14
- Priority of kernel tasks, drivers, system proc is fixed

- User process priority changes with behavior
  - +1 is entire quantum is used
  - -1 is blocks before using quantum

Restart



Restart is the common point reached after system startup, interrupts, or system calls. The most deserving process (which may be and often is a different process from the last one interrupted) runs next. Not shown: interrupts that occur while the kernel itself is running.

Minix Scheduler Implementation

- Implementation spread across multiple files
- **main.c** - call to restart
  - rdy_heqd, rdy_tail: arrays with head/tail of each level queue
- **table.c** - initial queuing of processes during system startup
  - enqueue, dequeue: used to add / remove entries to queue
- **Function sched:** determines which process should be on which queue
  - checks in process used its entire quantum and adjusts priority +1 or -1
- **Function pick_proc:** test reach queue and find the first non-empty queue

Minix Scheduler

- **proc.c**
  - lock_send, lock_enqueu, lock_dequeue, lock_notify
    - used for basic locking and unlocking of queues

- Clock task monitors all processes
  - Quantum expires: put process at tail of the queue
  - Drivers, servers given higher quanta, but can also be pre-empted.