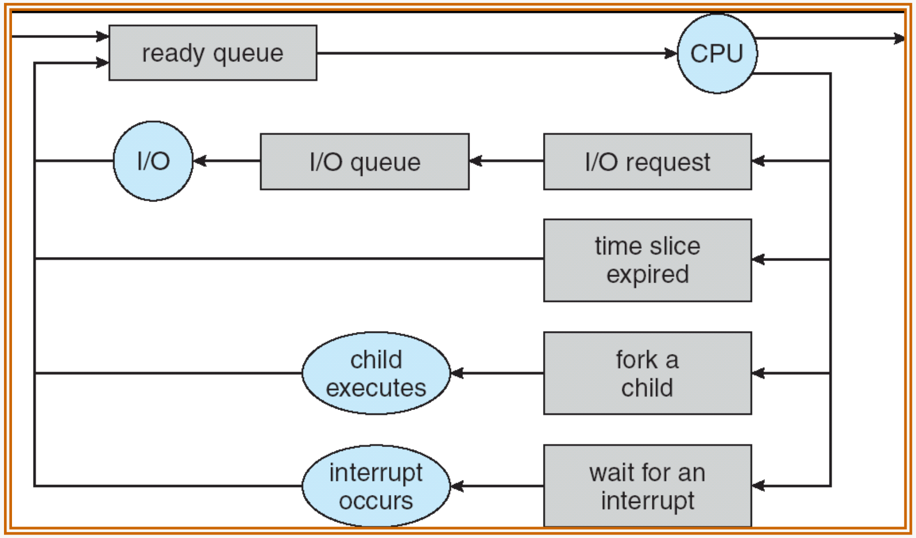
CSI 4500 Operating Systems

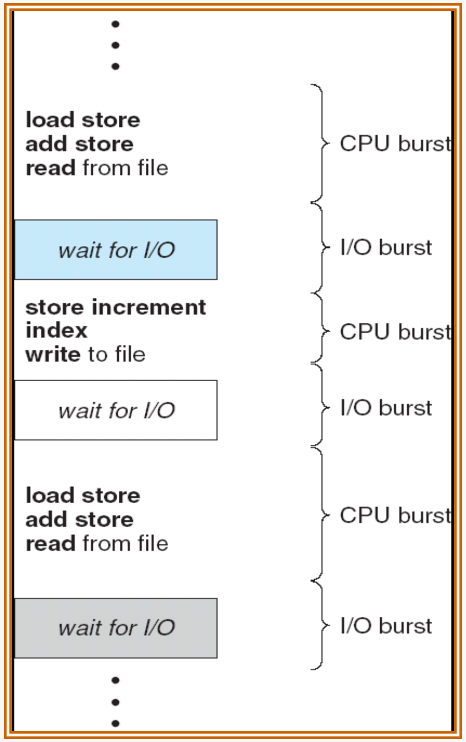
CPU Scheduling

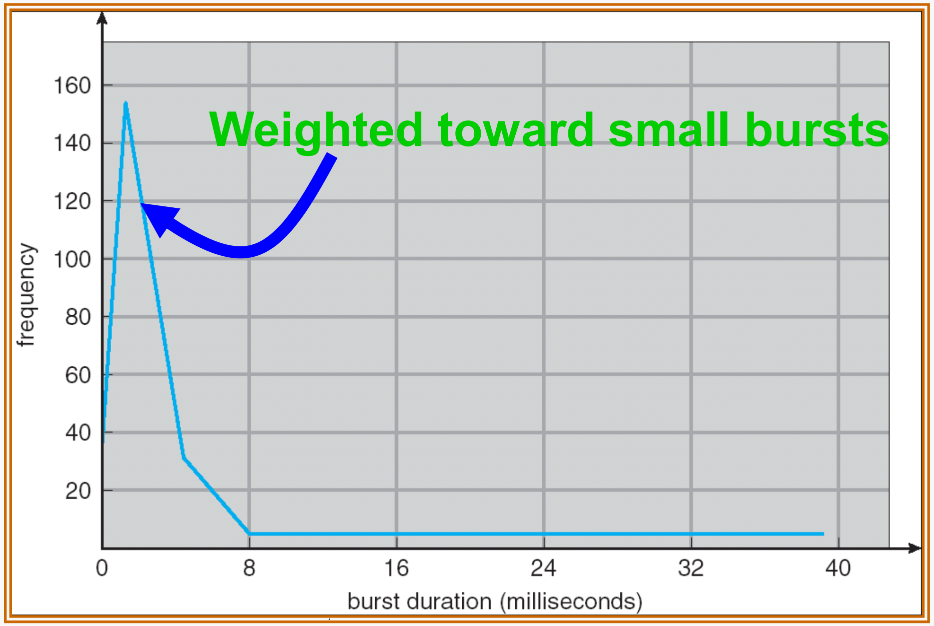
CPU Scheduling



- life-cycle of a thread shows above
  - Active threads work their way from **Ready** queue to **Running** with possible various **Waiting** queues.
- Question: How is the OS to decide which of several tasks to run next?
  - Obvious queue to worry about is ready queue
- **Scheduling:** deciding which threads are given access to resources from moment to moment.

Assumption: CPU-I/O Bursts Cycle





- Execution model: programs alternate between bursts of **CPU** and **I/O**
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Terminates

Cooperative scheduling

3. Switches from waiting to ready
4. Switches from running to ready state

Preemptive scheduling

Scheduling Criteria

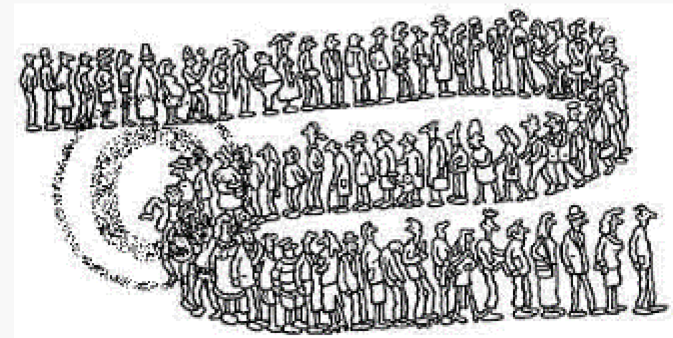
- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process, from submission till the time of completion.
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

Scheduling Policy Goals

- Minimize Response Time
  - Response time is what the user sees:
    - Time to echo a keystroke in editor
    - Time to compile a program
    - Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Throughput related to response time, but not identical:
    - Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - Minimize overhead (for example, context-switching)
    - Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - Better *average* response time by making system *less* fair

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - In early systems, FCFS meant one program scheduled until done (including I/O)
    - Now, means keep CPU until thread blocks



- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose processes arrive in the order: P_1 , P_2 , P_3
    The Gantt Chart for the schedule is:

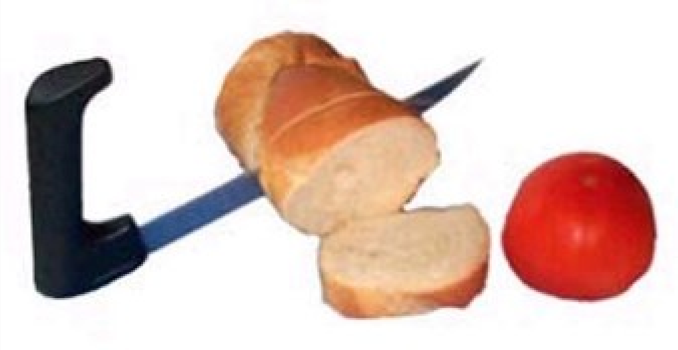
  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average Completion time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process behind long process

Round Robin (RR)

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don’t care who is behind you, on the other hand…
- Round Robin Scheme
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - n processes in ready queue and time quantum is $q \Rightarrow$
    - Each process gets $1/n$ of the CPU time
    - In chunks of at most q time units
    - No process waits more than $(n-1)q$ time units
- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ Interleaved (really small $\Rightarrow$?)
  - q must be large with respect to context switch, otherwise overhead is too high (all overhead)

Round-Robin Discussion

- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)
- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FIFO
  - What if time slice too small?
    - Throughput suffers!



- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, need to balance short-job performance and long-job throughput:
    - Typical time slice today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

Comparisons between FCFS and RR

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time
- Completion Times:

Job #	FCFS	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average response time
    - RR: 4.5
    - FCFS: 450

  Average waiting time
    - RR: 891
    - FCFS: 450
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR longer even for zero-cost switch!

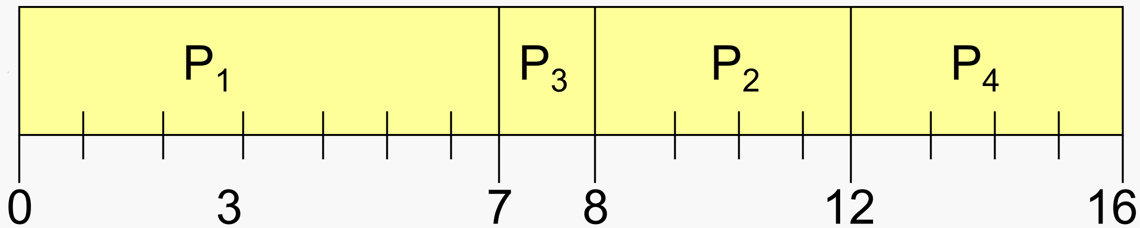
What if we Knew the Future?

- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
  - Sometimes called "Shortest Time to Completion First" (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system as soon as possible
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

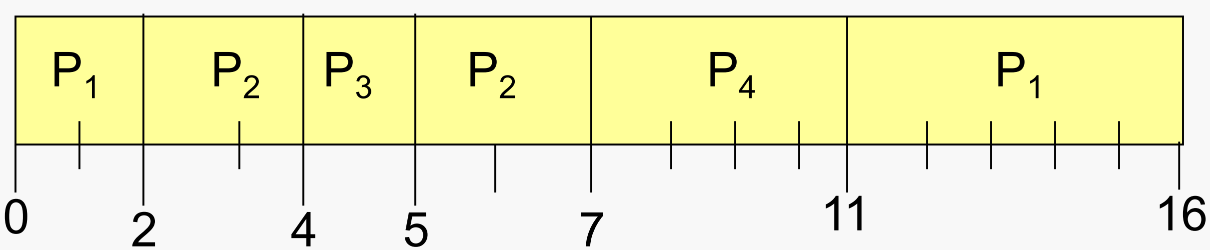


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)

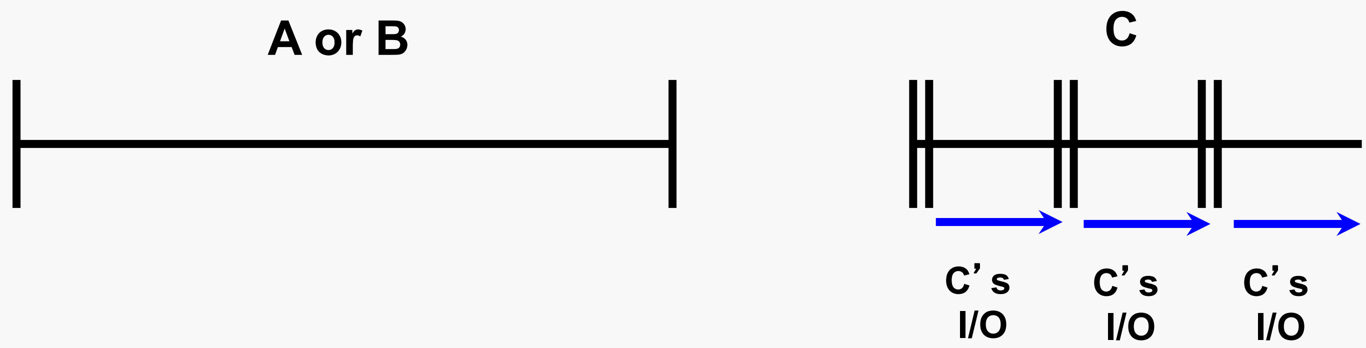


- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Discussion

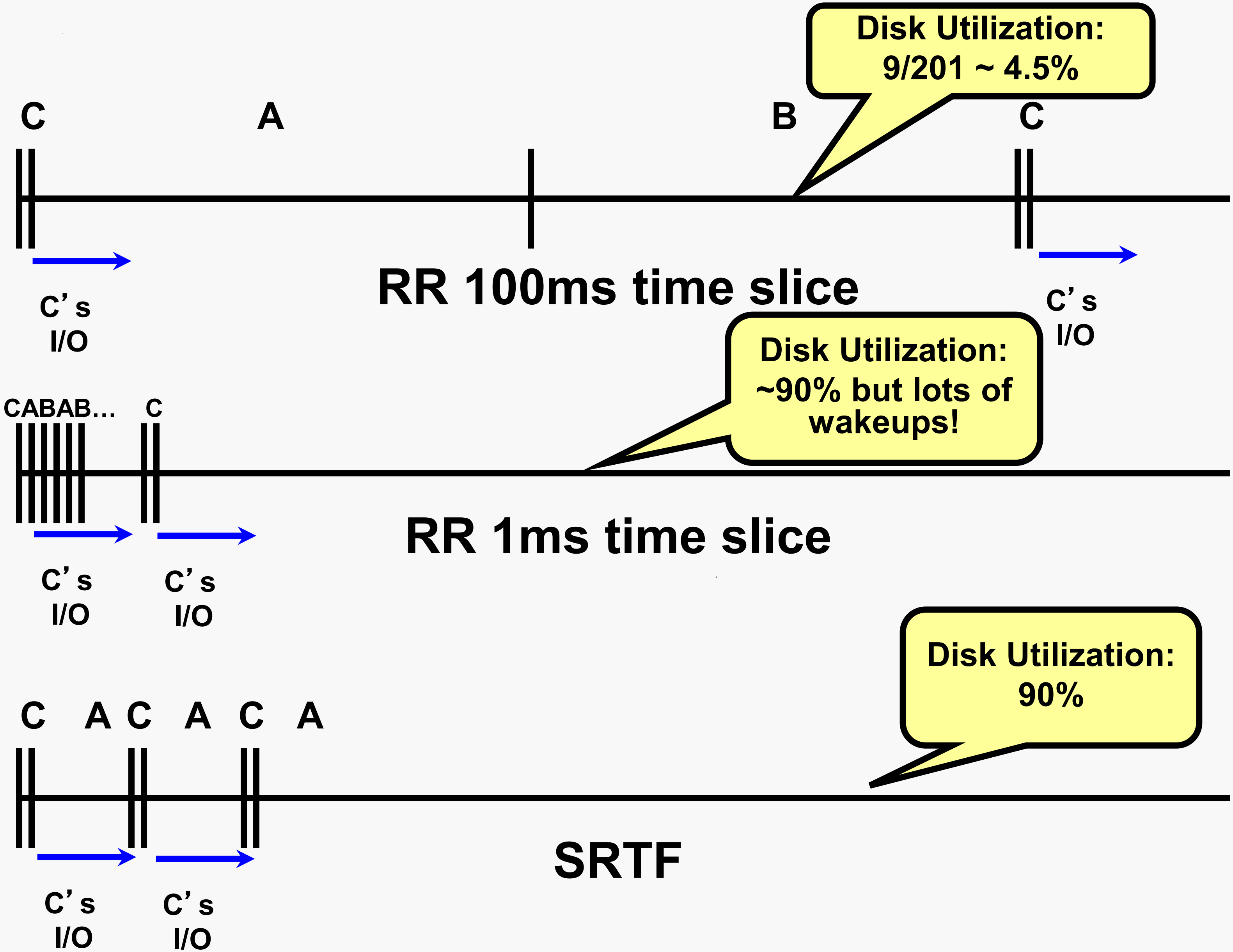
- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SRTF becomes the same as FCFS
    - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones

Example to illustrate benefits of SRTF



- Three jobs:
  - A,B: both CPU bound, run for weeks
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Example continued:



SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - When you submit a job, have to say how long it will take
    - To stop cheating, system kills job if takes too long
  - But: Even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process [giving more CPU time gradually]

Oakland UNIVERSITY

Challenges

- No universal scheduler
  - Unpredictable workloads
  - Unpredictable environment
- Building a one-size-fits-all scheduler that works well for all is challenging!
- Real scheduling algorithms are often more complex than the simple scheduling algorithms we've seen
  - FCFS
  - RR
  - SJF/SRTF

A solution

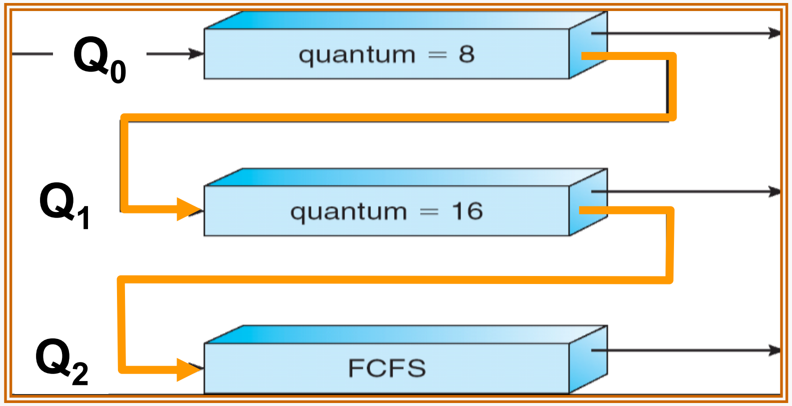
- **Multilevel Queue Scheduling**: ready queue is partitioned into multiple queues
  - Multiple queues, each with different priority
    - Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - e.g. foreground – RR, background – FCFS
    - Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Must choose scheduling algorithm to schedule between queues.
  - Fixed priority scheduling for each queue
    - serve all from highest priority, then next priority, etc.
  - **RR** between queues: Time slice:
    - each queue gets a certain amount of CPU time
    - e.g., 70% to highest, 20% next, 10% lowest

Multilevel Feedback Queue

- A process can move between the various queues
  - aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.



What about Fairness?

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - long running jobs may never get CPU
    - In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting average response time!

What about Fairness?

- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - What if one long-running job and 100 short-running ones?
    - Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don't get service
    - What is done in UNIX
    - This is ad hoc—what rate should you increase priorities?
    - And, as system gets overloaded, no job gets CPU time, so everyone increases in priority $\Rightarrow$ Interactive jobs suffer

Linux Scheduling

- Avoid starvation
- Boost interactivity
  - Fast response to user despite high load
  - Achieved by inferring interactive processes and dynamically increasing their priorities
- SMP goals
  - **Scale** well w.r.t number of processes
    - O(1) scheduling overhead
  - **Load balance**: no CPU should be idle if there is work
  - **CPU affinity**: no random bouncing of processes

Linux Scheduling Algorithm Overview

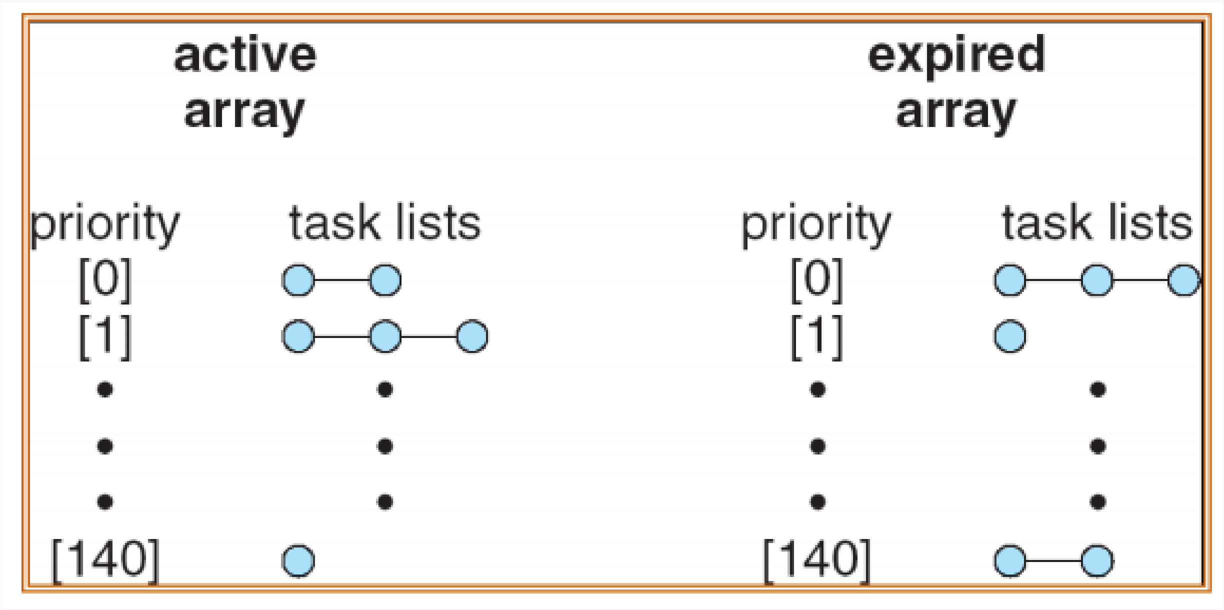
- Multilevel queue scheduler
  - Each queue associated with a priority
  - A process's priority may be adjusted dynamically
- Two classes of processes
  - Real-time processes: always schedule highest priority processes
    - FCFS (SCHED_FIFO) or RR (SCHED_RR) for processes with same priority
  - Normal processes: priority with aging
    - RR for processes with same priority (SCHED_NORMAL)
    - Aging is implemented efficiently

Priority partition

- Total 140 priorities [0, 140)
  - Smaller integer == higher priority
  - Real-time: [0, 100)
  - Normal: [100, 140)
- MAX_PRIO and MAX_RT_PRIO
  - include/linux/sched.h

Runqueue data structure

- kernel/sched.c
- struct prio_array
  - Array of priority queues
- struct runqueue
  - Two arrays, active and expired



Scheduling Algorithm

1. Find highest priority non-empty queue in rq->active; if none, simulate aging by swapping active and expired
2. next = first process on that queue
3. Adjust next’s priority
    1. Dynamically increase priority of interactive process
4. Context switch to next
5. When next used up its time slice, insert next to the right queue and call schedule again

schedule() in kernel/sched.c

Summary

- Scheduling problem
  - Given a set of processes that are ready to run
  - Which one to select *next*
- Scheduling criteria
  - CPU utilization, Throughput, Turnaround, Waiting, Response
  - Predictability: variance in any of these measures
- Scheduling algorithms
  - FCFS, SJF, SRTF, RR
  - Multilevel (Feedback-)Queue Scheduling
- The best schemes are adaptive.