PRINCIPLES OF OPERATING SYSTEMS

LECTURE 10
Principles of Operating Systems

CPU SCHEDULING ALGORITHMS
(PRIORITY, ROUND ROBIN, MULTILEVEL QUEUE)

Priority Scheduling

- A priority value (integer) is associated with each process. Can be based on
  - Cost to user
  - Importance to user
  - Aging
  - %CPU time used in last X hours.
- CPU is allocated to process with the highest priority.
  - Preemptive
  - Nonpreemptive

Priority Scheduling (cont.)

- SJN is a priority scheme where the 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.

Round Robin (RR)

- Each process gets a small unit of CPU time
    - Time quantum usually 10-100 milliseconds.
    - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- *n* processes, time quantum = *q*
    - Each process gets 1/*n* CPU time in chunks of at most *q* time units at a time.
    - No process waits more than $(n-1)q$ time units.
    - Performance
        - Time slice *q* too large – response time poor
        - Time slice (∞)? -- reduces to FIFO behavior
        - Time slice *q* too small - Overhead of context switch is too expensive. Throughput poor

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  - The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 28 48 68 88 108 112 125 145 153

  - Waiting time
    - $P_1=(68-20)+(112-88)=72$
    - $P_2=(20-0)=20$
    - $P_3=(28-0)+(88-48)+(125-108)=85$
    - $P_4=(48-0)+(108-68)=88$
  - Average waiting time = $(72+20+85+88)/4=66\frac{1}{4}$
  - Average completion time = $(125+28+153+112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Comparisons between FCFS and Round Robin

- 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 #	FIFO	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 is much worse under RR!
    - Bad when all jobs same length
- 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!

Earlier Example with Different Time Quantum

Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]

0 8 32 85 153

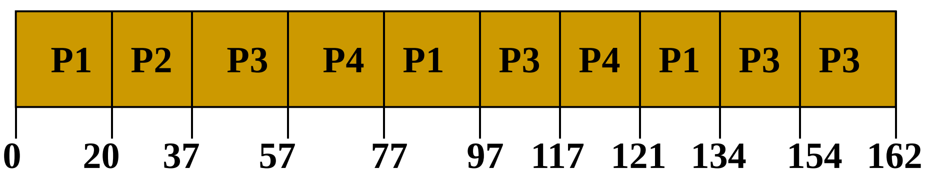
	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Round Robin Example

Time Quantum = 20

Process	Burst Time
P1	53
P2	17
P3	68
P4	24

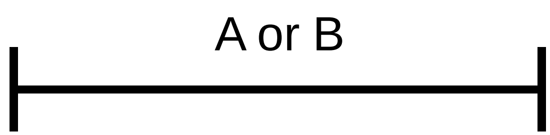
Gantt Chart for Schedule

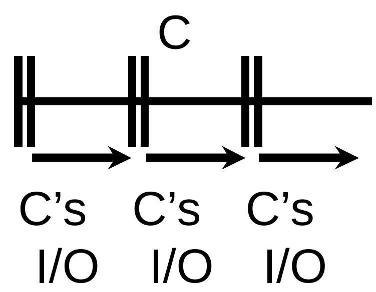


Typically, higher average turnaround time than SRTF, but better response

- Initially, UNIX timeslice (q) = 1 sec
  - Worked OK when UNIX was used by few (1-2) 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
  - q must be large wrt context switch, o/w overhead is too high
  - Typical time slice today is between 10ms – 100ms
  - Typical context switching overhead is 0.1 – 1 ms
  - Roughly 1% overhead due to context switching
- Another Heuristic - 70 – 80% of jobs block within timeslice

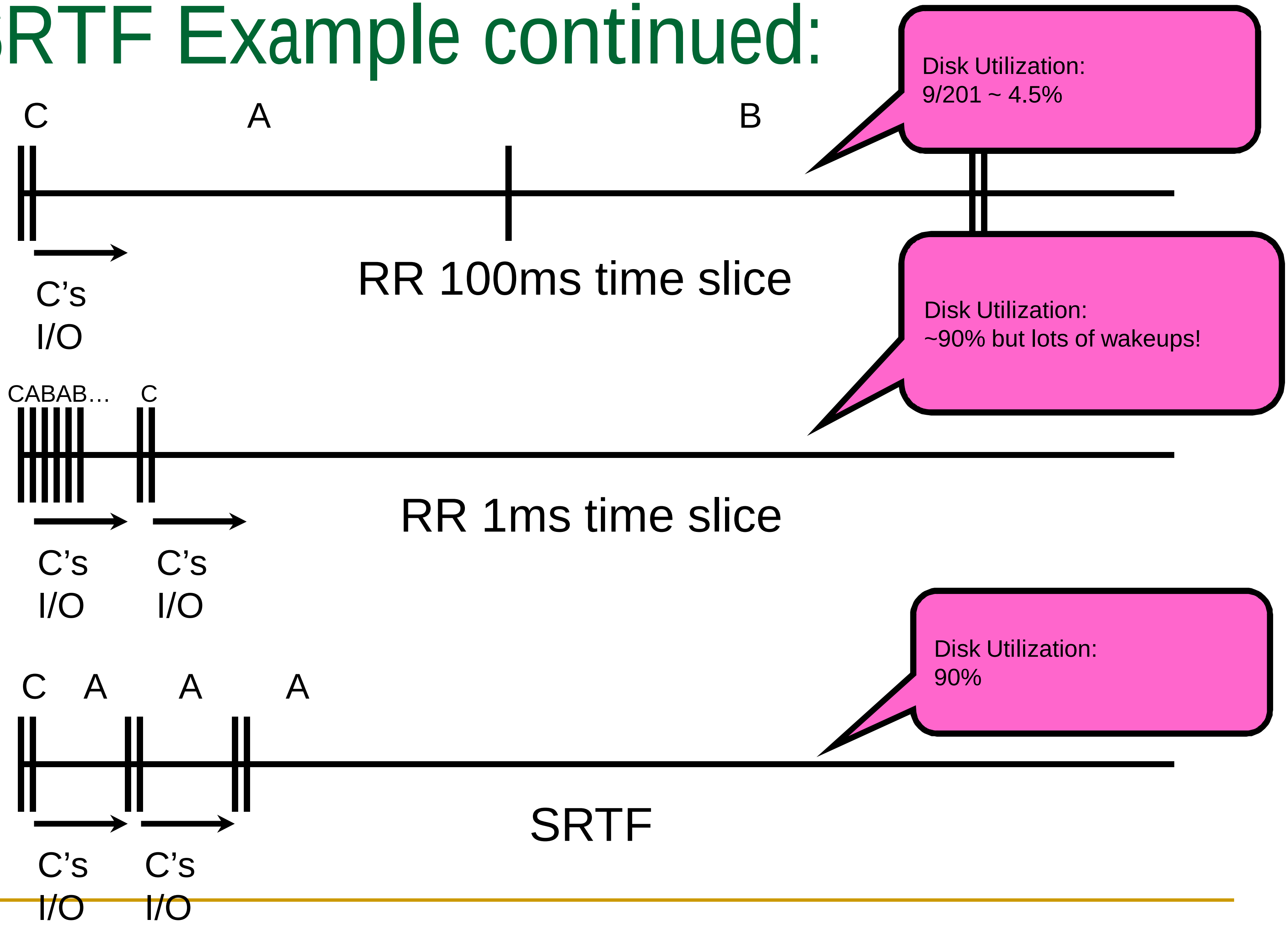
Example to illustrate benefits of SRTF





- Three jobs:
  - A,B: both CPU bound, run for week
    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 two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

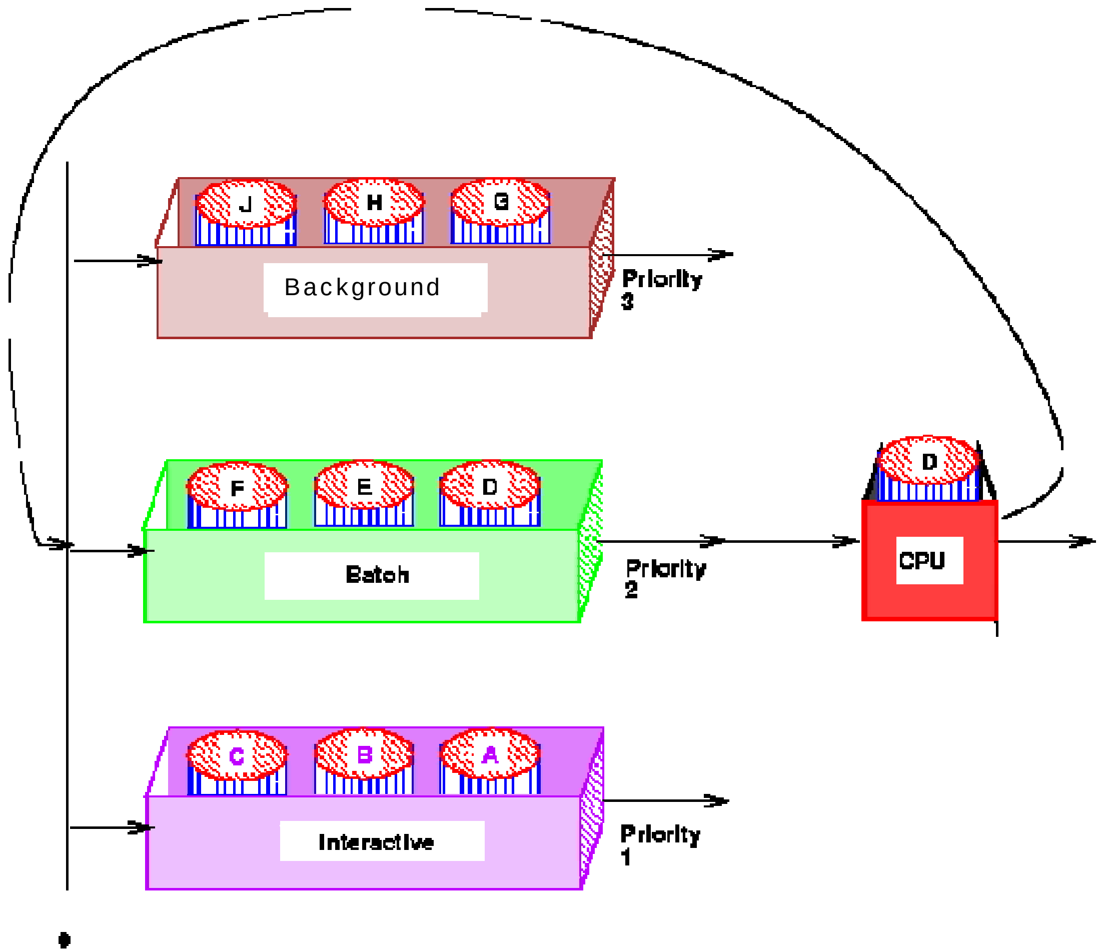
SRTF Example continued:



Multilevel Queue

- Another method for exploiting past behavior
- Ready Queue partitioned into separate queues
  - Each queue has a priority; Higher priority queues often considered “foreground” tasks
  - Eg. system processes, foreground (interactive), background (batch), ….
- Each queue has its own scheduling algorithm
  - Example: foreground (RR), background(FCFS)
  - Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Processes assigned to one queue permanently.
- Scheduling must be done between the queues
  - Fixed priority - serve all from foreground, then from background.
  - Time slice - Each queue gets some CPU time that it schedules - e.g. 80% foreground(RR), 20% background (FCFS)

Multilevel Queues



Scheduling 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 avg response time!
- 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

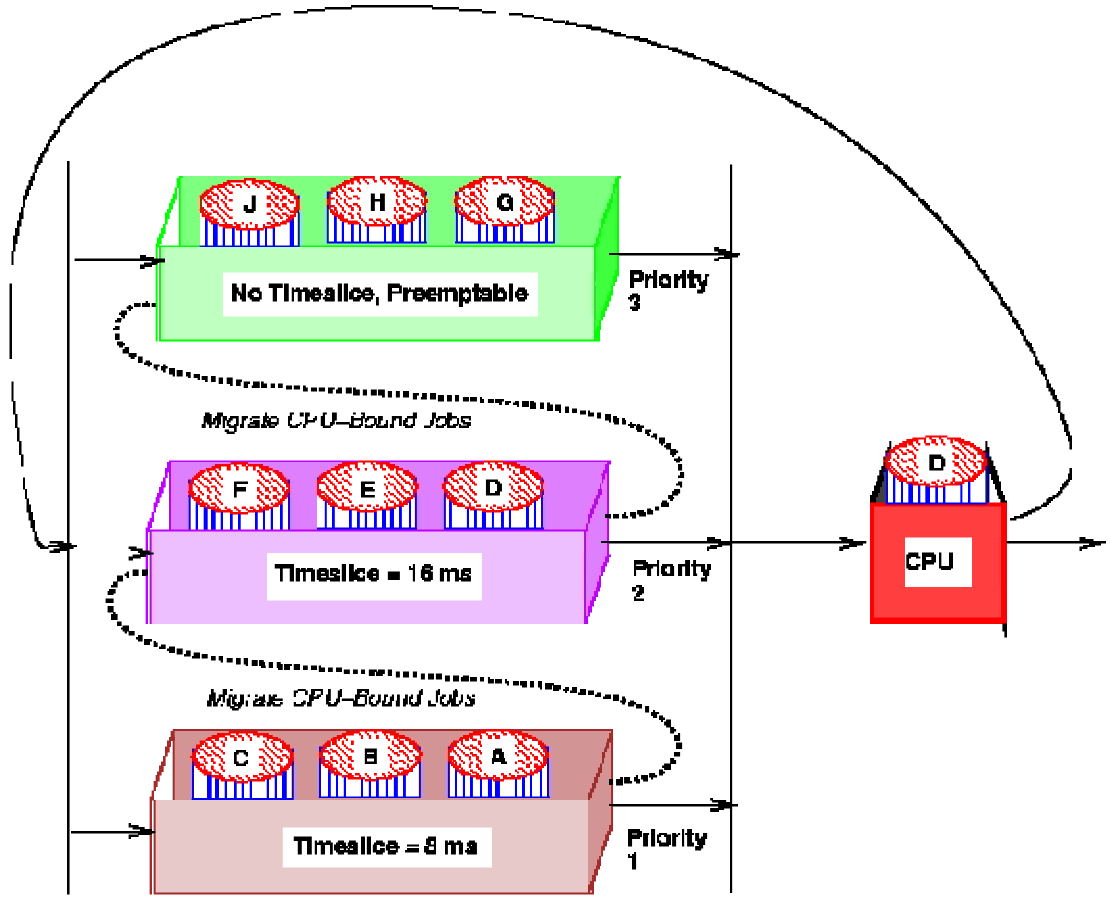
Multilevel Feedback Queue

- Multilevel Queue with priorities
- A process can *move* between the queues.
  - Aging can be implemented this way.
  - Adjust each job's priority as follows (details vary)
    - Job starts in highest priority queue
    - If timeout expires, drop one level
    - If timeout doesn't expire, push up one level (or to top)
- Parameters for a multilevel feedback queue scheduler:
  - number of queues.
  - scheduling algorithm 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.

Multilevel Feedback Queues

- Example: Three Queues -
  - Q0 - time quantum 8 milliseconds (RR)
  - Q1 - time quantum 16 milliseconds (RR)
  - Q2 - FCFS
- Scheduling
  - New job enters Q0 - When it gains CPU, it receives 8 milliseconds. If job does not finish, move it to Q1.
  - At Q1, when job gains CPU, it receives 16 more milliseconds. If job does not complete, it is preempted and moved to queue Q2.
- Countermeasure: user action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - Of course, if everyone did this, wouldn't work!

Multilevel Feedback Queues



•

Multiple-Processor Scheduling

- CPU scheduling becomes more complex when multiple CPUs are available.
  - Have one ready queue accessed by each CPU.
    - Self scheduled - each CPU dispatches a job from ready Q
    - Master-Slave - one CPU schedules the other CPUs
- Homogeneous processors within multiprocessor.
    - Permits Load Sharing
- Asymmetric multiprocessing
    - only 1 CPU runs kernel, others run user programs
    - alleviates need for data sharing

Real-Time Scheduling

- Hard Real-time Computing -
  - required to complete a critical task within a guaranteed amount of time.
- Soft Real-time Computing -
  - requires that critical processes receive priority over less fortunate ones.
- Types of real-time Schedulers
  - Periodic Schedulers - Fixed Arrival Rate
    - **E.g. Rate monotonic (RM).** Tasks are periodic. Policy is shortest-period-first, so it always runs the ready task with shortest period.
  - Aperiodic Schedulers - Variable Arrival Rate
    - **E.g. Earliest deadline (EDF).** This algorithm schedules the task with closer deadline first