CPU Scheduling

Arvind Krishnamurthy
Spring 2004

CPU Scheduling

Question: dispatcher can choose any thread on the ready queue to run; how to decide and which to choose?

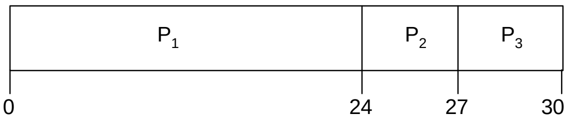
- Depends on scheduling policy goals
- ***minimize response time*** : elapsed time to do an operation (or job)
  - Response time is what the user sees: elapsed time to
    - echo a keystroke in editor
    - compile a program
    - run a large scientific problem
- ***maximize throughput*** : operations (jobs) per second
  - two parts to maximizing throughput
    - minimize overhead (for example, context switching)
    - efficient use of system resources (not only CPU, but disk, memory, etc.)
- ***fair*** : share CPU among users in some equitable way

First Come First Served

- Example:

Process	Exec. Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The schedule is:



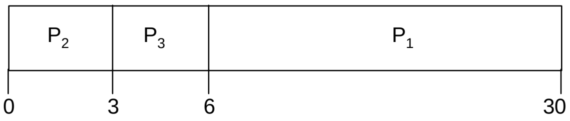
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: (0 + 24 + 27)/3 = 17
- Average response time: (24 + 27 + 30)/3 = 27

FCFS scheduling (cont'd)

Suppose that the processes arrive in the order

P_2 , P_3 , P_1 .

- The time chart for the schedule is:



  - Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Average response time: (30 + 3 + 6)/3 = 13

- FCFS Pros: simple; Cons: short jobs get stuck behind long jobs

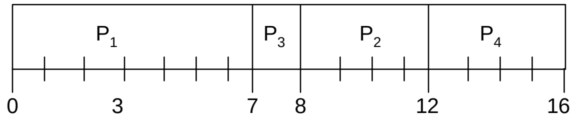
Shortest-Job-First (SJF)

- Associate with each process the length of its exec. time
  - Use these lengths to schedule the process with the shortest time
- Two schemes:
  - Non-preemptive – once given CPU it cannot be preempted until completes its quota.
  - preemptive – if a new process arrives with less work than the remaining time of currently executing process, preempt.
- SJF is optimal but unfair
  - pros: gives minimum average response time
  - cons: long-running jobs may starve if too many short jobs;
  - difficult to implement (how do you know how long job will take)

Non-preemptive SJF

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

- SJF (non-preemptive)

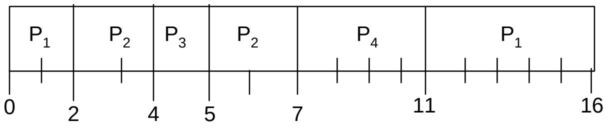


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4
- Average response time = (7 + 10 + 4 + 11)/4 = 8

Example of preemptive SJF

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

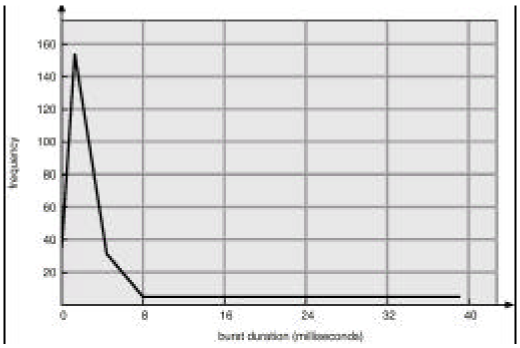
- SJF (preemptive)

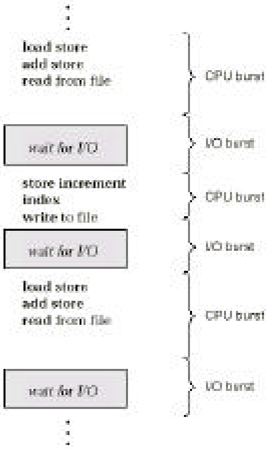


- Average waiting time = (9 + 1 + 0 + 2)/4 = 3
- Average response time = (16 + 5 + 1 + 6)/4 = 7

Alternating CPU and I/O Bursts

- CPU–I/O Burst Cycle
- CPU burst distribution





Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*). After time slice, it is moved to the end of the ready queue.

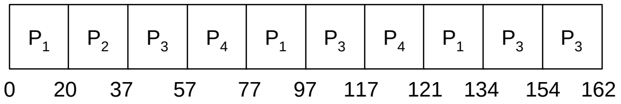
  Time Quantum = 10 - 100 milliseconds on most OS
- *n* processes in the ready queue; time quantum is *q*
  - each process gets 1/*n* of the CPU time in *q* time units at once.
  - no process waits more than (*n*-1)*q* time units.
  - each job gets equal shot at the CPU
- Performance
  - *q* large ⇒ FCFS
  - *q* too small ⇒ throughput suffers. Spend all your time context switching, not getting any real work done

---

RR with time quantum = 20

Process	Exec. Time
P_1	53
P_2	17
P_3	68
P_4	24

- The time chart is:



P_1 P_2 P_3 P_4 P_1 P_3 P_4 P_1 P_3 P_3

0 20 37 57 77 97 117 121 134 154 162

- Typically, higher average turnaround than SJF, but better fairness.

RR vs. FCFS vs. SJF

- Three tasks A, B, C
  - A and B both CPU bound (can run for a week)
  - C is I/O bound: loop 1 ms CPU followed by 10 ms disk I/O
    - running C by itself gets 90% disk utilization

  - with FIFO?
  - with RR (100 ms time slice):
    - What is the disk utilization? 10ms of disk operation every 200ms
    - How much does C have to wait after I/O completes? 190 ms
  - with RR (1 ms time slice):
    - What is the disk utilization? 10ms of disk operation every 11-12 ms
    - How much does C have to wait after I/O completes? 0 or 1 ms

Knowledge of future

- Problem: SJF or STCF require knowledge of the future
- How do you know how long program will run for?

- Option 1: ask the user
  - When you submit the job, say how long it will take
  - If your job takes more than that, jobs gets killed. (Hard to predict usage in advance.)
- Option 2:
  - Use past to predict future
  - If program was I/O bound in the past, likely to remain so
  - Favor jobs that have been at CPU least amount of time

Multilevel queue

- Ready queue is partitioned into separate queues:
  - Each with different priority
- OS does RR at each priority level
  - Run highest priority jobs first
  - Once those finish, run next highest priority etc
  - Round robin time slice increases (exponentially) at lower priorities
- Adjust each job's priority as follows:
  - Job starts in highest priority queue
  - If time slice is fully used when process is run, drop one level
  - If it is not fully used, push up one level
- CPU bound jobs drop like a rock, while short-running I/O bound jobs stay near top
- Still unfair – long running jobs may never get the CPU
- Could try to strategize!

Handling dependencies

- Scheduling = deciding who should make progress
  - Obvious: a thread's importance should increase with the importance of those that depend on it.
  - Naïve priority schemes violate this ("Priority inversion")
- Example: T1 at high priority, T2 at low
  - T2 acquires lock L. T1 tries to acquire the same lock.
- "Priority donation"
  - Thread's priority scales w/ priority of dependent threads
  - Works well with explicit dependencies

Lottery Scheduling

- Problem: this whole priority thing is really ad hoc.
  - How to ensure that processes will be equally penalized under load?
  - How to deal with priority inversion?

- Lottery scheduling
  - give each process some number of tickets
  - each scheduling event, randomly pick ticket
  - run winning process
  - to give P n% of CPU, give it ntickets * n%
- How to use?
  - Approximate priority: low-priority, give few tickets, high-priority give many
  - Approximate SJF: give short jobs more tickets, long jobs fewer. If job has at least 1, will not starve

---

Lottery Scheduling Example

- Add or delete jobs (& their tickets) affects all jobs proportionately

  short job: 10 tickets; long job: 1 ticket

#short jobs/ #long jobs	% of CPU each short job gets	% of CPU each long job gets
1 / 1	91%	9%
0 / 2	NA	50%
2 / 0	50%	NA
10 / 1	10%	1%
1 / 10	50%	5%

- Easy priority inversion:
  - Donate tickets to process you're waiting on.
  - Its CPU% scales with tickets of all waiters.

Other notes

- Client-server:
  - Server has no tickets of its own
  - Clients give server all of their tickets during RPC
  - Server's priority is sum of its active clients
  - Server can use lottery scheduling to give preferential service
- Ticket inflation: dynamic changes in priorities between trusting programs
- Currency:
  - Set up an exchange rate across groups
  - Can print more money within a group
  - Allows independent scheduling properties
- Compensation tickets
  - What happens if a thread is I/O bound and regularly blocks before its quantum expires?
  - If you complete fraction f, your tickets are inflated by 1/f