CS 370: OPERATING SYSTEMS
[CPU SCHEDULING]

Shrideep Pallickara
Computer Science
Colorado State University

COMPUTER SCIENCE DEPARTMENT — COLORADO STATE UNIVERSITY

1

Frequently asked questions from the previous class survey

- CPU scheduling
  - Does the CPU ever make scheduling decisions?
  - Who decides the time interval for each process? The CPU or the scheduler?
  - Are all nonpremeptive systems designed without a timer?
  - Does preemptive scheduling require a preemptive kernel?
  - Is waiting time part of the process' metadata?
  - Do schedulers harvest data every clock cycle to make decisions?

COLORADO STATE UNIVERSITY — Professor: SHRIDEEP PALLICKARA, COMPUTER SCIENCE DEPARTMENT — CPU SCHEDULING — L15.2

2

Topics covered in this lecture

- Scheduling Algorithms
  - SJF
  - Priority Scheduling
  - Round robin scheduling
- Multilevel feedback queues
- Lottery scheduling

3

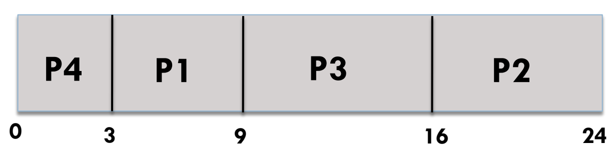
4

Shortest Job First (SJF) scheduling algorithm

- When CPU is available it is assigned to process with **smallest CPU burst**
- Moving a short process before a long process?
  - Reduction in waiting time for short process
    GREATER THAN
    Increase in waiting time for long process
- Gives us **minimum average waiting time** for a **set** of processes that arrived *simultaneously*
  - Provably Optimal

Depiction of SJF in action

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

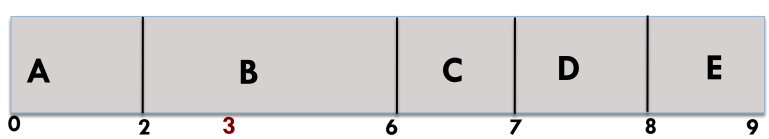


`Wait time = (3 + 16 + 9 + 0)/4 = 7`

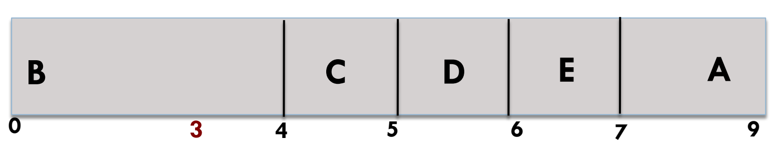
SJF is optimal ONLY when ALL the jobs are available simultaneously

- Consider 5 processes **A, B, C, D** and **E**
  - Run times are: 2, 4, 1, 1, 1
  - Arrival times are: 0,0, 3, 3, 3
- SJF will run jobs: **A, B, C, D** and **E**
  - Average wait time: (0 + 2 + 3 + 4 + 5)/5 = 2.8
  - **But** if you run **B, C, D, E** and **A** ?
    - Average wait time: (7 + 0 + 1 + 2 +3)/5 = 2.6!

Visualizing the different runs of A, B, C, D and E



Average wait time: (0 + 2 + 3 + 4 + 5)/5 = 2.8



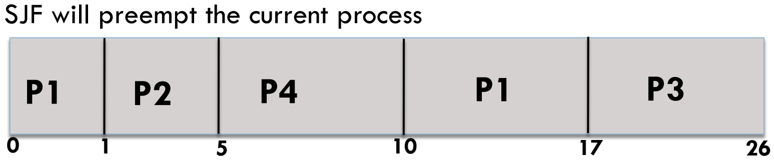
Average wait time: (7 + 0 + 1 + 2 +3)/5 = 2.6

Preemptive SJF

- What counts as "**shortest**" is the remaining time left on the task, not its original length
  - If you are a nanosecond away from finishing an hour-long task, stay on that task
    - Instead of preempting for a minute long task
- Also known, as **shortest-remaining-time-first** (SRTF)

Preemptive SJF

- A new process arrives in the ready queue
  - If it is shorter (i.e., shorter time remaining) than the currently executing process?
    - Preemptive SJF will preempt the current process



Process	Arrival	Burst
P1	0	8
P2	1	4
P3	2	9
P4	3	5

```
Wait time =
[(10-1) + (1-1) + (17-2) + (5-3)]/4
= 26/4 = 6.5
```

Characteristics of Preemptive SJF

- Can suffer from **starvation** and **frequent context switches**
  - If enough short tasks arrive, long tasks may never complete
- Analogy
  - Supermarket manager switching to SJF to reduce waiting times

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT CPU SCHEDULING L15.11

11

Does Preemptive SJF has any other downsides?

- Turns out, SJF is **pessimal** for variance in response time
- By doing the shortest tasks as quickly as possible, SJF necessarily does longer tasks *as slowly as possible*
- Fundamental **tradeoff** between reducing average response time and reducing the variance in average response time

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT CPU SCHEDULING L15.12

12

SJF IN SCHEDULERS

COMPUTER SCIENCE DEPARTMENT COLORADO STATE UNIVERSITY

13

Use of SJF in long term schedulers

- Length of the process time limit
  - Used as CPU burst estimate
- Motivate users to accurately estimate time limit
  - Lower value will give faster response times
  - Too low a value?
    - Time limit exceeded error
    - Requires resubmission!

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT CPU SCHEDULING L15.14

14

The SJF algorithm and short term schedulers

- **No way to know** the length of the next CPU burst
- So, try to **predict** it
- Processes scheduled *based on predicted* CPU bursts

15

Prediction of CPU bursts: Make estimates based on past behavior

- t_n : Length of the n^{th} CPU burst
- τ_n : Estimate for the n^{th} CPU burst
- α : Controls weight of recent and past history
- $\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$
- Burst is predicted as an exponential average of the measured lengths of previous CPU bursts

16

α controls the relative weight of recent and past history

- $\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$
- Value of t_n contains our most recent information, while τ_n stores the past history
- $\tau_{n+1} = \alpha t_n + (1-\alpha)\,\alpha t_{n-1} + \ldots + (1-\alpha)^j\,\alpha t_{n-j} + \ldots + (1-\alpha)^{n+1}\,\alpha\tau_0$
- α is less than 1, $(1-\alpha)$ is also less than one
  - **Each successive term has less weight than its predecessor**

17

The choice of α in our predictive equation

- If α=1/2
  - Recent history and past history are **equally weighted**
- With α = ½; successive estimates of τ

  $t_0/2$ $\quad t_0/4 + t_1/2$ $\quad t_0/8 + t_1/4 + t_2/2$ $\quad t_0/16 + t_1/8 + t_2/4 + t_3/2$

  - By the 3rd estimate, weight of what was observed at t_0 has dropped to 1/8.

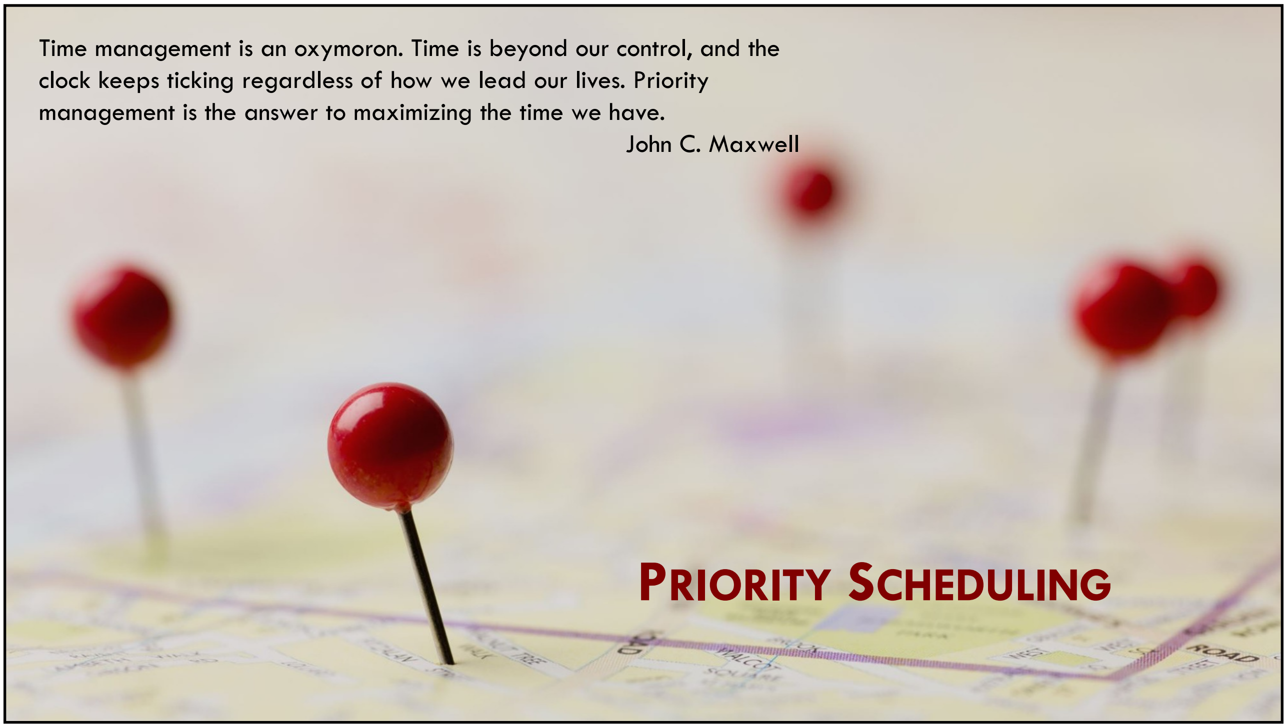
18

An example: Predicting the length of the next CPU burst

CPU burst (t_i)		**6**	**4**	**6**	**4**	**13**	**13**	**13**
"Guess" (τ_i)	10	8	6	6	5	9	11	**12**

The choice of α in our predictive equation

- $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
- If $\alpha \rightarrow 0$, $\tau_{n+1} = \tau_n$
  - Current conditions are transient
- If $\alpha=1$, $\tau_{n+1} = t_n$
  - Only most recent bursts matter
  - History is assumed to be old and irrelevant



21

Priority Scheduling

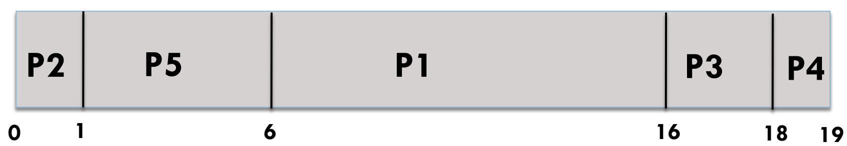
- **Priority** associated with each process
- CPU allocated to process with **highest** priority
- Can be preemptive or nonpreemptive
  - If preemptive: Preempt CPU from a lower priority process when a higher one is ready

22

Depiction of priority scheduling in action

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Here: Lower number means higher priority



`Wait time = (6 + 0 + 16 + 18 + 1)/5 = 8.2`

How priorities are set

- Internally defined priorities based on:
  - **Measured** quantities
  - Time limits, memory requirements, # of open files, ratio (averages) of I/O to CPU burst
- External priorities
  - Criteria outside the purview of the OS
  - Importance of process, $ paid for usage, politics, etc.

Issue with priority scheduling

- Can leave lower priority processes waiting indefinitely
- Perhaps apocryphal tale:
  - MIT's IBM 7094 shutdown (1973) found processes from 1967!

Coping with issues in priority scheduling: Aging

- **Gradually increase priority** of processes that wait for a long time
- Example:
  - Process starts with a priority of 127 and decrements every 15 minutes
  - Process priority becomes 0 in no more than 32 hours

Can SJF be thought of as a priority algorithm?

- Priority is **inverse** of CPU burst
- The larger the burst, the lower the priority
  - *Note*: The number we assign to represent priority levels may vary from system to system

27

28

Round-Robin Scheduling

- Similar to FCFS scheduling
  - **Preemption** to enable switch between processes
- Ready queue is implemented as **FIFO**
  - Process Entry: PCB at *tail* of queue
  - Process chosen: From *head* of the queue
- CPU scheduler goes around ready queue
  - Allocates CPU to each process *one after the other*
    - CPU-bound up to a maximum of 1 **quantum**

Round Robin: Choosing the quantum

- Context switch is **time consuming**
  - Saving and loading registers and memory maps
  - Updating tables
  - Flushing and reloading memory cache
- What if quantum is 4 ms and context switch overhead is 1 ms?
  - 20% of CPU time thrown away in administrative overhead

Round Robin: Improving efficiency by increasing quantum

- Let's say quantum is 100 ms and context-switch is 1ms
  - Now wasted time is only 1%
- But what if 50 concurrent requests come in?
  - Each with widely varying CPU requirements
  - 1st one starts immediately, 2nd one 100 ms later, …
  - The last one may have to wait for 5 seconds!
  - A shorter quantum would have given them better service

If quantum is set longer than mean CPU burst?

- **Preemption will not happen very often**
- Most processes will perform a blocking operation before quantum runs out
- Switches happens only when process blocks and cannot continue

Quantum: Summarizing the possibilities

- Too short?
  - Too *many* context switches
  - Lowers CPU efficiency
- Too long?
  - *Poor* responses to interactive requests

A round-robin analogy

- Hyperkinetic student studying for multiple exams simultaneously
  - If you switch between paragraphs of different textbooks? [Quantum is too short]
    - You won't get much done
  - If you never switch? [Quantum is too long]
    - You never get around to studying for some of the courses

Every inch of sky's got a star
Every inch of skin's got a scar
I guess that you've got everything now

Every inch of space in your head
Is filled up with the things that you read
I guess you've got everything now

And every film that you've ever seen
Fills the spaces up in your dreams
That reminds me
...
Every song that I've ever heard
Is playing at the same time, it's absurd
And it reminds me, we've got everything now

Everything Now, Arcade Fire

MULTI-LEVEL FEEDBACK QUEUES (MFQ)

Most commercial OS including Windows and MacOS, use this scheduling algorithm

COMPUTER SCIENCE DEPARTMENT — COLORADO STATE UNIVERSITY

35

MFQ is designed to achieve several simultaneous goals

- **Responsiveness**: Run short tasks quickly as in SJF
- **Low Overhead**: Minimize number of preemptions, as in FIFO
  - Minimize time spent making scheduling decisions
- **Starvation-Freedom**
  - All tasks should make progress, as in Round Robin
- **Background tasks**
  - Defer system maintenance tasks, such as defragmentation, so they do not interfere with user work
- **Fairness**

COLORADO STATE UNIVERSITY — Professor: SHRIDEEP PALLICKARA, COMPUTER SCIENCE DEPARTMENT — CPU SCHEDULING — L15.36

36

Does MFQ achieve all of these?

- As with any real system that must ***balance*** several, conflicting goals …
  - MFQ does not perfectly achieve any of these goals
- MFQ is intended to be a **reasonable compromise** in most real-world cases

MFQ

- Extension of round robin
- Instead of only a single queue, MFQ has **multiple round robin queues**
  - Each queue has a **different** *priority level* and *time quanta*

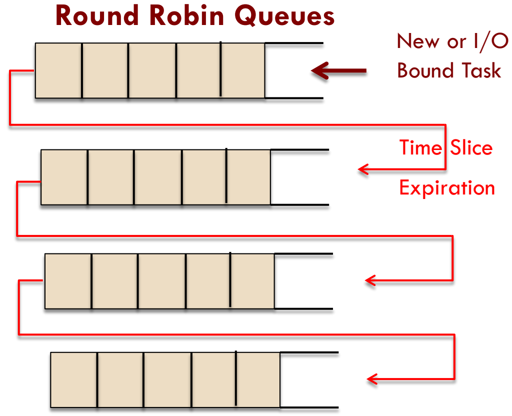
Tasks and priorities

- Tasks at a higher priority **preempt** lower priority tasks
- Tasks at the same priority level are scheduled in **round robin** fashion
- Higher priority tasks have **shorter** time quanta than lower priority tasks

MFQ: Example with 4 priority levels

Priority	Time Slice (ms)
1	10
2	20
3	40
4	60

Round Robin Queues



Task movements and priority

- Tasks are moved between priority levels to **favor short tasks** over long ones
- Every time a task uses up its time quantum?
  - It **drops** a priority level
- Every time task yields the processor because it is waiting on I/O?
  - It **stays** at the same level, or is **bumped up** a level
- If the task completes … it leaves

Impact on CPU and I/O bound processes

- A **new** CPU bound process will start as *high priority*
  - But it will quickly exhaust its time quantum and fall to the next lower priority, and then the next …
- An I/O bound process with a modest amount of computing
  - Will always be **scheduled quickly**
    - Also, keeps the disk busy
- Compute bound tasks run with a long time quantum to minimize switching overhead while sharing processor

What about starvation and fairness?

- If there are too many I/O bound tasks, the compute bound tasks may receive no time on the processor
- MFQ scheduler **monitors every process** to ensure that it is receiving its fair share
  - For e.g. at each level, maintain two queues
    - Tasks whose processes have already reached their fair share are only scheduled if other processes at the same level have also received their fair share
- Periodically, processes not receiving their fair share have their tasks increased in priority
  - Tasks that receive more than their fair share have their priority reduced

Adjusting priority addresses strategic behavior

- From a selfish point of view, a task can keep its priority high by doing a short I/O request just before its quantum expires
  - With MFQ this will be detected, and its priority reduced to its fair level

45

Lottery scheduling

- Give processes **lottery tickets** for various system resources
  - E.g., CPU time
- When a scheduling decision has to be made
  - Lottery ticket is *chosen at random*
  - Process holding **ticket gets** the resource

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT CPU SCHEDULING L15.46

46

Dealing with important processes: All processes are equal, but some processes are more equal than others

- More important processes are given **extra tickets**
  - Increase their odds of winning
- Let's say there are 100 outstanding tickets
  - 1 process holds 20 of these
  - Has 20% chance of winning each lottery
- A process holding a fraction f of tickets
  - Will get about a fraction f of the resource

Lottery Scheduling: Properties [1/2]

- Highly **responsive**
  - Chance of winning is proportional to tickets
- Cooperating processes may **exchange** tickets
  - Process **A** sends request to **B**, and then hands **B** all its tickets for a faster response
- Avoids starvation
  - Each process holds at least one ticket …. Is guaranteed to have a non-zero probability of being scheduled

Lottery Scheduling: Properties [2/2]

- Solves problems that are *difficult to handle* in other scheduling algorithms
- E.g., video server that is managing processes that feed video frames to clients
  - Clients need frames at 10, 20, and 25 frames/sec
  - Allocate processes 10, 20 and 25 tickets
    - CPU divided into approximately 10:20:25

The contents of this slide-set are based on the following references

- *Thomas Anderson and Michael Dahlin. Operating Systems Principles and Practice. 2nd Edition. ISBN: 978-0985673529.* [Chapter 7]
- *Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. Operating Systems: Three Easy Pieces. 1st edition. CreateSpace Independent Publishing Platform. ISBN-13: 978-1985086593. [Chapter 9]*
- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 6]
- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620.* [Chapter 2]