CSE 421/521 - Operating Systems
Spring 2018

Lecture - VII

CPU Scheduling - II

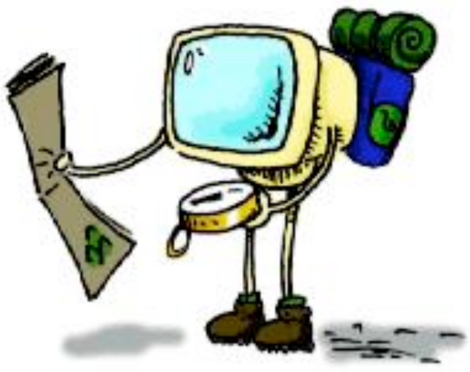
Tevfik Koşar

University at Buffalo

February 20th, 2018

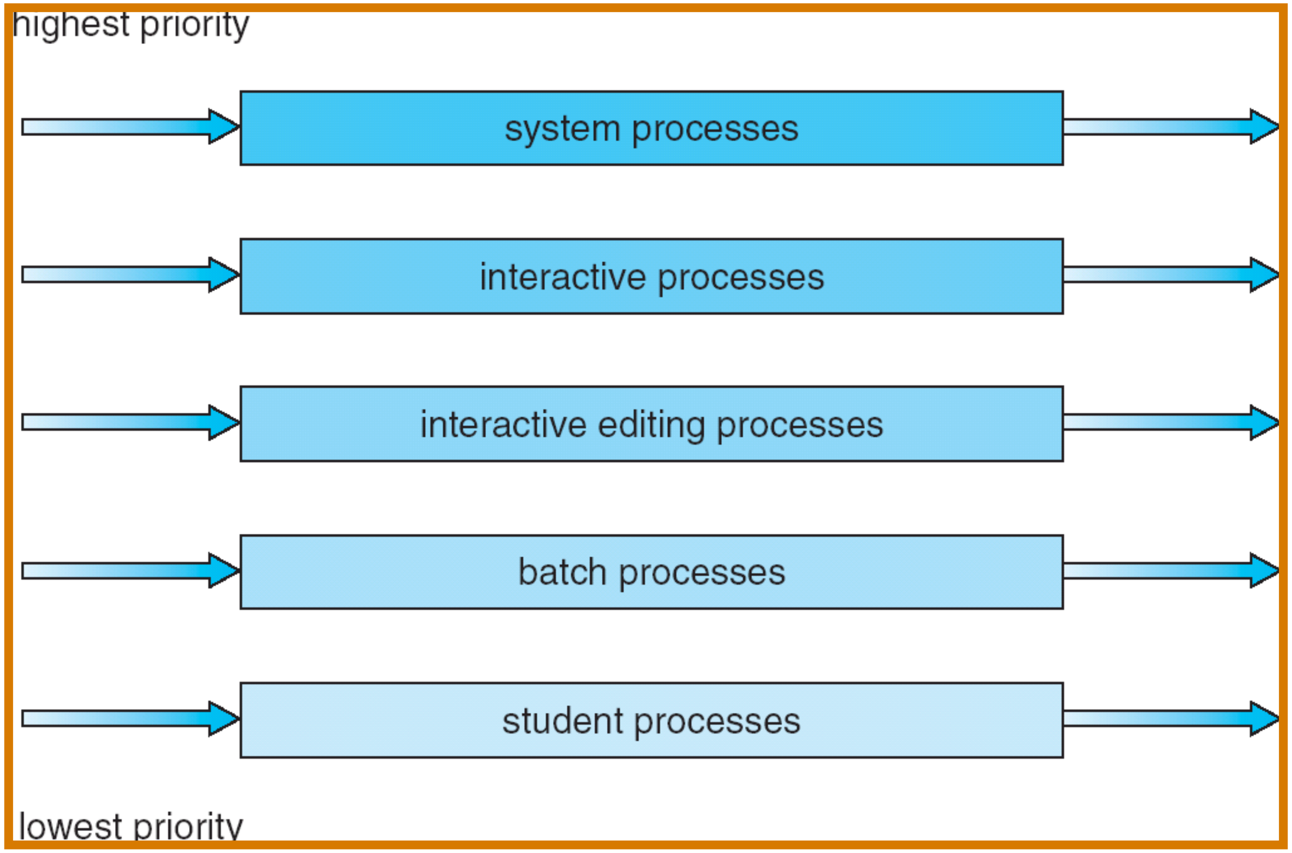
Roadmap

- CPU Scheduling
  - Multilevel Feedback Queues
  - 4.4BSD Priority Based Scheduling
  - Estimating CPU Burst Time



Multilevel Queues

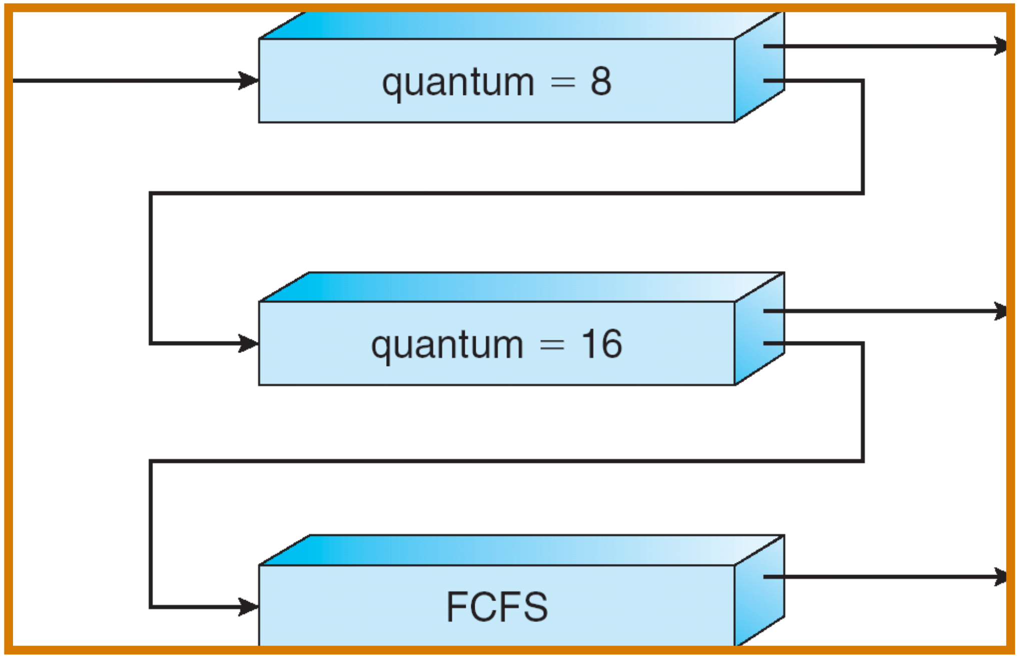
Multilevel Queue Scheduling



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 which queue a process will enter when that process needs service
  - method used to determine when to upgrade a process
  - method used to determine when to degrade a process

Example of Multilevel Feedback Queue



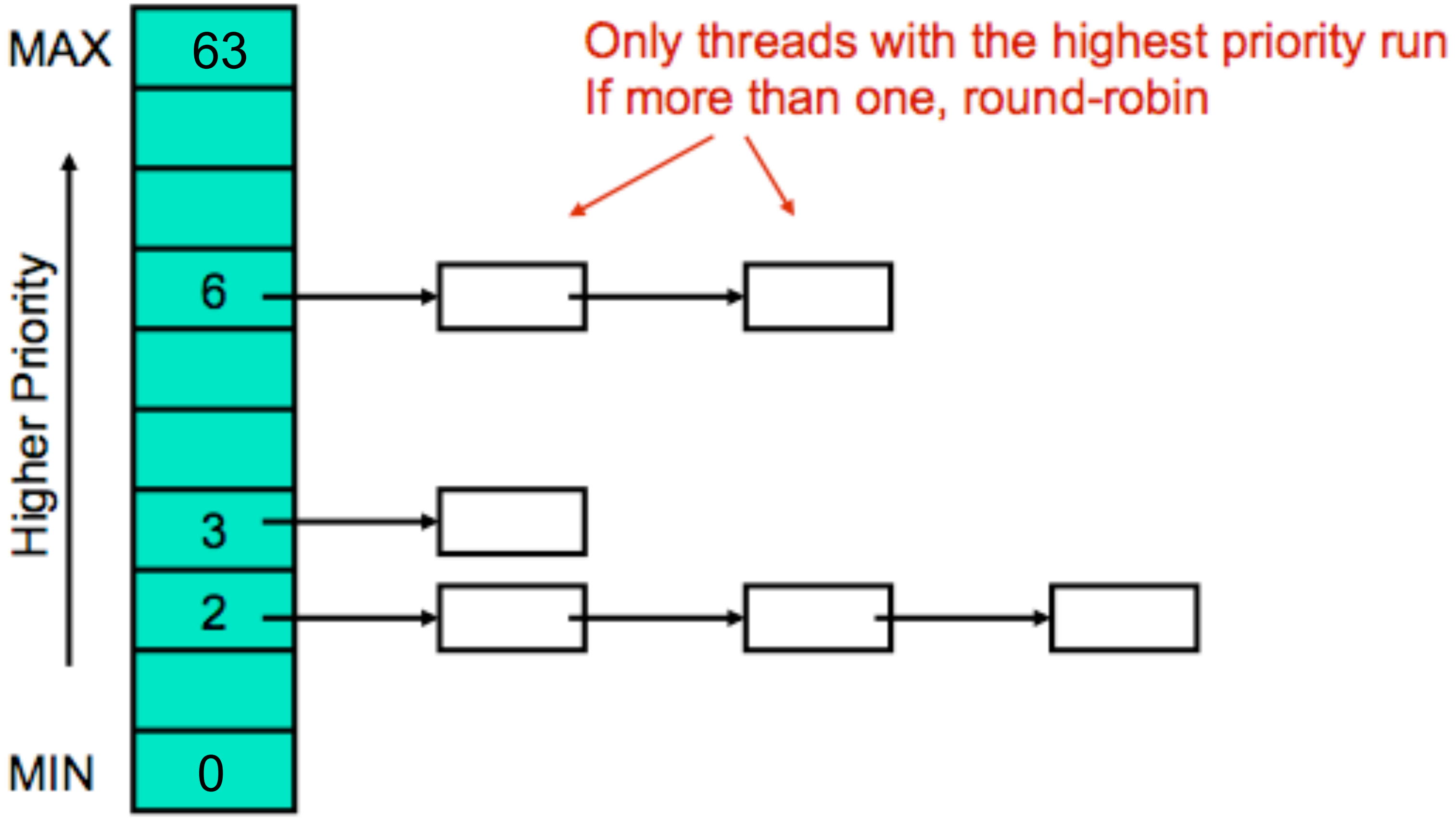
- Three queues:
  - Q_0 – RR with q = 8 ms
  - Q_1 – RR with q = 16 ms
  - Q_2 – FCFS

- 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.

4.4BSD Priority Scheduling

4.4BSD Priority Based Scheduling

4.4BSD scheduler has 64 priorities and thus 64 ready queues, numbered 0 (PRI_MIN) through 63 (PRI_MAX).



Only threads with the highest priority run
If more than one, round-robin

Calculating Priority

- NOTE: Lower numbers correspond to lower priorities in 4.4BSD, so that priority 0 is the lowest priority and priority 63 is the highest.

- Every 4 clock ticks, calculate:

  priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)

  (rounded down to the nearest integer)

- It gives a thread that has received CPU time recently lower priority for being reassigned the CPU the next time the scheduler runs.

← **Aging**

Nice Value

==> how \nice" the thread should be to other threads.

- A nice of zero does not affect thread priority.

- A positive nice, to the **maximum of 20**, decreases the priority of a thread and causes it to give up some CPU time it would otherwise receive.

A negative nice, to the **minimum of -20**, tends to take away CPU time from other threads.

Calculating recent_cpu

- An array of n elements to track the CPU time received in each of the last n seconds requires O(n) space per thread and O(n) time per calculation of a new weighted average.

- Instead, we use a exponentially weighted moving average:

  - **`recent_cpu(0) = 0`** *// or parent thread's value*
  - *at each timer interrupt,* **`recent_cpu++`** *for the running thread.*
  - *and once per second, for each thread:*

recent_cpu(t) = a * recent_cpu(t-1) + nice

where, **`a = (2*load_avg)/(2*load_avg + 1)`**

Calculating load_avg

- Estimates the average number of threads ready to run over the past minute.
- Like recent_cpu, it is an exponentially weighted moving average.
- Unlike priority and recent_cpu, load_avg is system-wide, not thread-specific.
- At system boot, it is initialized to 0. Once per second thereafter, it is updated according to the following formula:

load_avg(t) = (59/60)*load_avg(t-1) + (1/60)*ready_threads

- ready_threads: number of threads that are either running or ready to run at the time of update

How to estimate CPU burst time?

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

$$\tau_{n=1} = \alpha\, t_n + (1 - \alpha)\tau_n .$$

1. t_n = actual lenght of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define :

Examples of Exponential Averaging

- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\ t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\ t_n + (1 - \alpha)\alpha\ t_n -1 + \ldots$$
$$+(1 - \alpha\ )^j \alpha\ t_{n\ -j} + \ldots$$
$$+(1 - \alpha\ )^{n\ +1} \tau_0$$

- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor

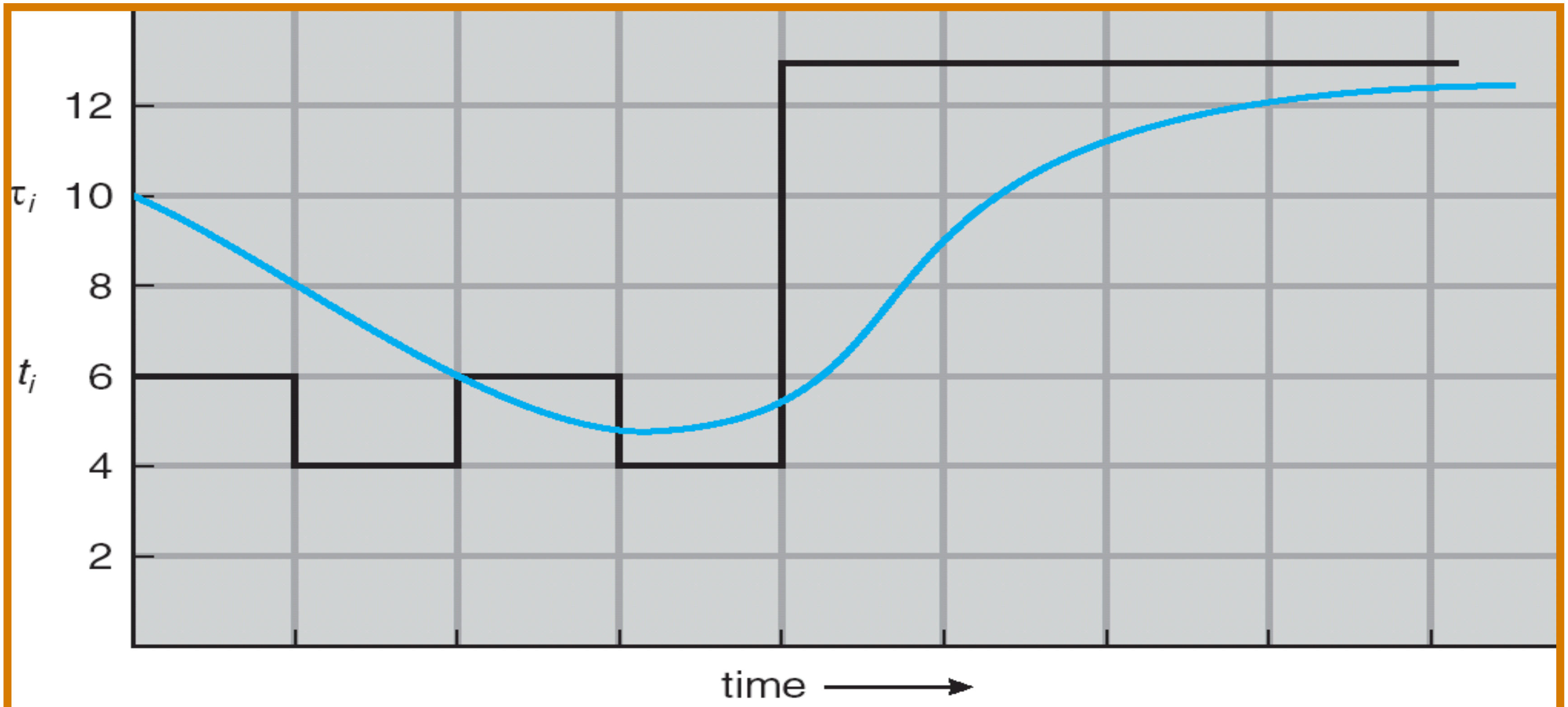
Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n .$$

1. t_n = actual lenght of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$

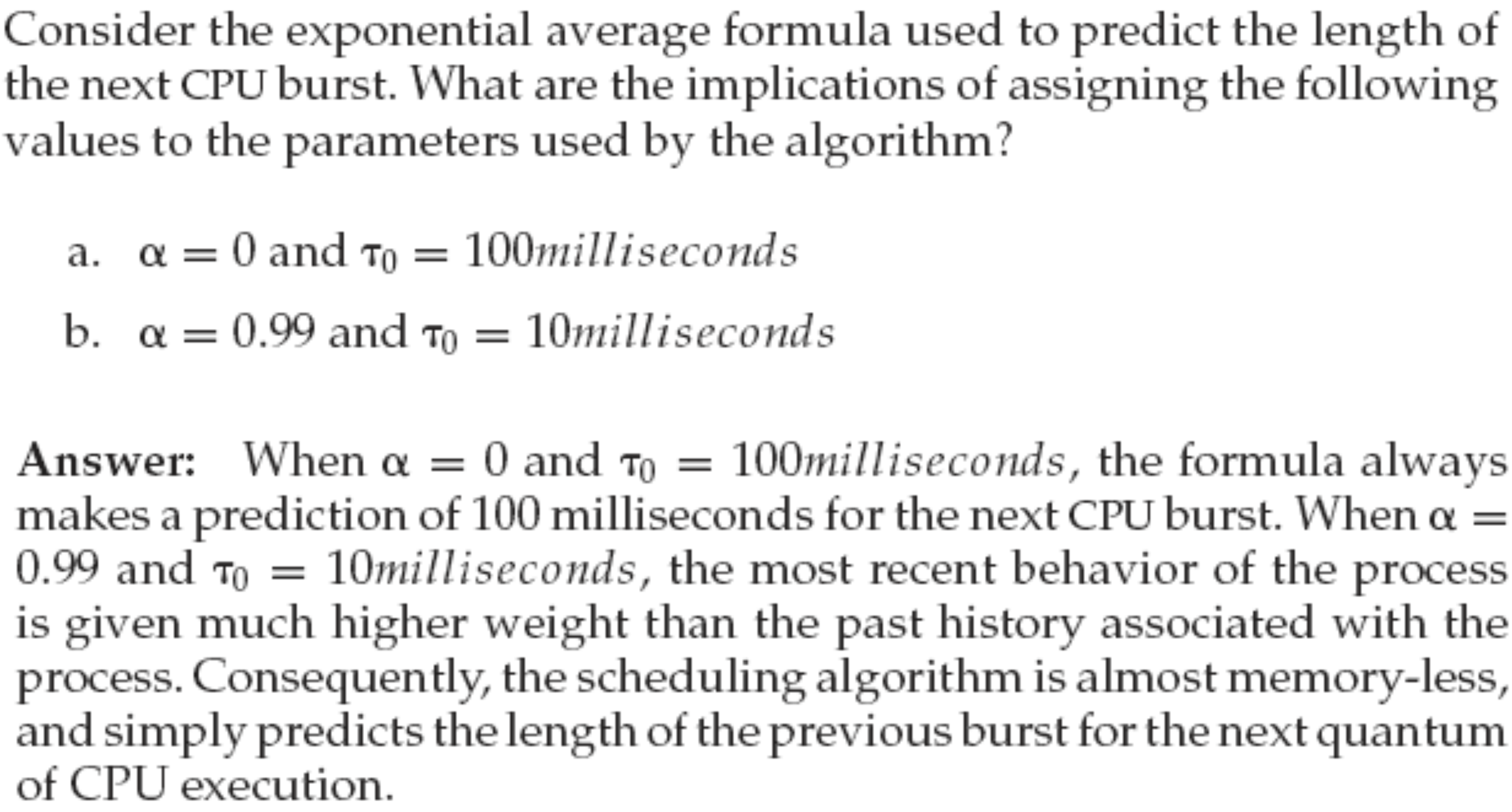
Prediction of the Length of the Next CPU Burst



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

Alpha = 1/2, T0 = 10

Exercise



Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?

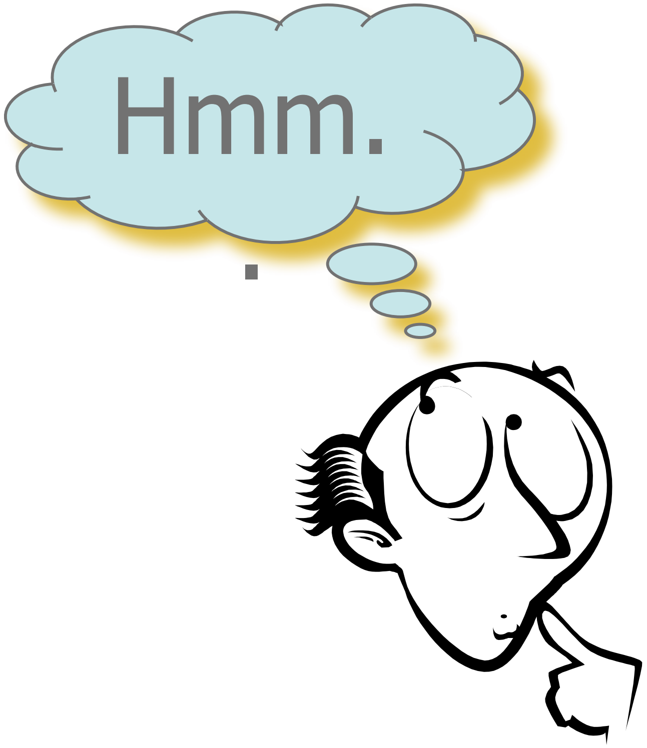
a. $\alpha = 0$ and $\tau_0 = 100 milliseconds$

b. $\alpha = 0.99$ and $\tau_0 = 10 milliseconds$

Answer: When $\alpha = 0$ and $\tau_0 = 100 milliseconds$, the formula always makes a prediction of 100 milliseconds for the next CPU burst. When $\alpha = 0.99$ and $\tau_0 = 10 milliseconds$, the most recent behavior of the process is given much higher weight than the past history associated with the process. Consequently, the scheduling algorithm is almost memory-less, and simply predicts the length of the previous burst for the next quantum of CPU execution.

Summary

- CPU Scheduling
  - Multilevel Feedback Queues
  - 4.4BSD Priority Based Scheduling
  - Estimating CPU Burst Time



- Next Lecture: Process Synchronization
- Reading Assignment: Chapter 5 from Silberschatz.

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne

- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings

- “Modern Operating Systems” book and supplementary material by A. Tanenbaum

- R. Doursat and M. Yuksel from UNR