Operating Systems Concepts

MLFQ & Proportional Share Scheduling

CS 4375, Fall 2025
Instructor: MD Armanuzzaman (***Arman***)
marmanuzzaman@utep.edu
September 29, 2025

Summery

- CPU Scheduling:
  - Basic concepts
  - Scheduling Criteria & Metrics
  - Different Scheduling Algorithms
    - FCFS
    - SJF
    - Priority
    - RR
  - Preemptive vs Non-preemptive Scheduling
  - Gantt Charts & Performance Comparison

Agenda

- CPU Scheduling:
    - Recap
    - Estimating CPU Burst Time
    - Multilevel Feedback Queue Scheduler
    - 4.4BSD Priority Based Scheduler
    - Proportional Share Scheduling

Comparison: FCFS

- PROS:
  - It is a fair algorithm
    - Schedule in the order that they arrive
- CONS:
  - Average response time can be lousy
    - Small requests wait behind big ones (convoy effect)
  - May lead to poor utilization of other resources
    - FCFS may result in poor overlap of CPU and I/O activity
      - E.g., a CPU-intensive job prevents an I/O intensive job from doing a small bit of computation, thus preventing it from going back and keep the I/O subsystem busy

Comparison: SJF

- PROS:
  - Provably optimal with respect to average waiting time
    - Prevents convoy effect
- CONS:
  - Can cause starvation of long jobs
  - Requires advanced knowledge of CPU burst times
    - This is not easy to predict!

Comparison: Priority

- PROS:
  - Guarantees early completion of high priority jobs
- CONS:
  - Can cause starvation of low priority jobs
  - How to decide/assign priority value?

Comparison: Round-Robin

- PROS:
  - Great for timesharing
    - No starvation
  - Does not require prior knowledge of CPU burst times
  - Generally reduces average response time
- CONS:
  - What if all jobs are almost the same length
    - Increases the turnaround time
  - How to set the “best” time quantum?
    - If too small, it increases context-switch overhead
    - If too large, response time degrades

How to Estimate CPU Burst Time?

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging
    - t_n : actual length of nth CPU burst
    - τ_n : predicted value for nth CPU burst
    - $0 \le \alpha \le 1$

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

Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts

Exponential Averaging

- If we expand the formula:

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

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

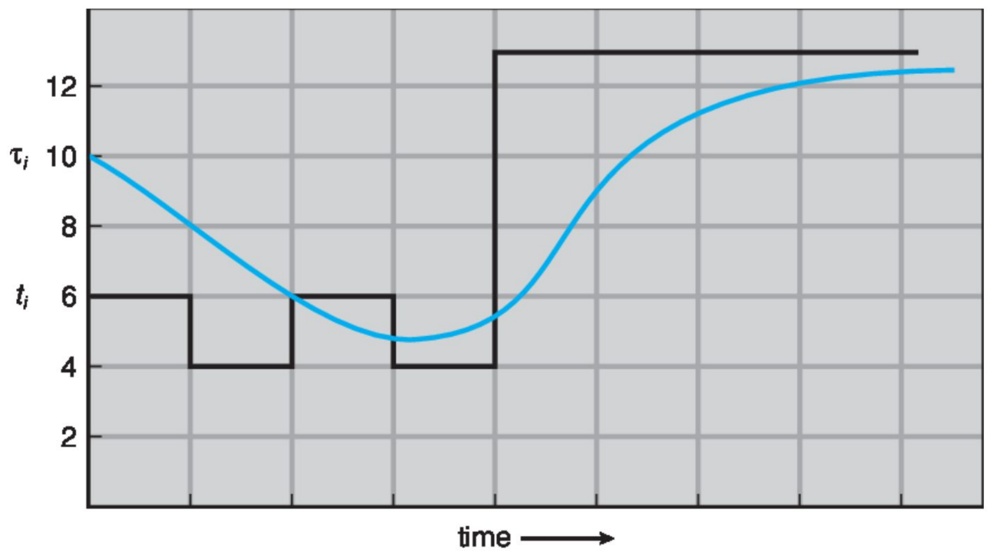
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + (1 - \alpha)2\, \tau_{n-1}$$

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + (1 - \alpha)^2\, \alpha\, t_{n-2} + (1 - \alpha)^3\, \alpha\, t_{n-3} + \ldots + (1 - \alpha)^{n+1}\, \tau_0$$

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

Exponential Averaging Example

- $\alpha = 0.5$, $\tau_0 = 10$



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

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?
  - $\alpha = 0$, $\tau_0 = 100$ ms
  - $\alpha = 0.99$, $\tau_0 = 10$ ms

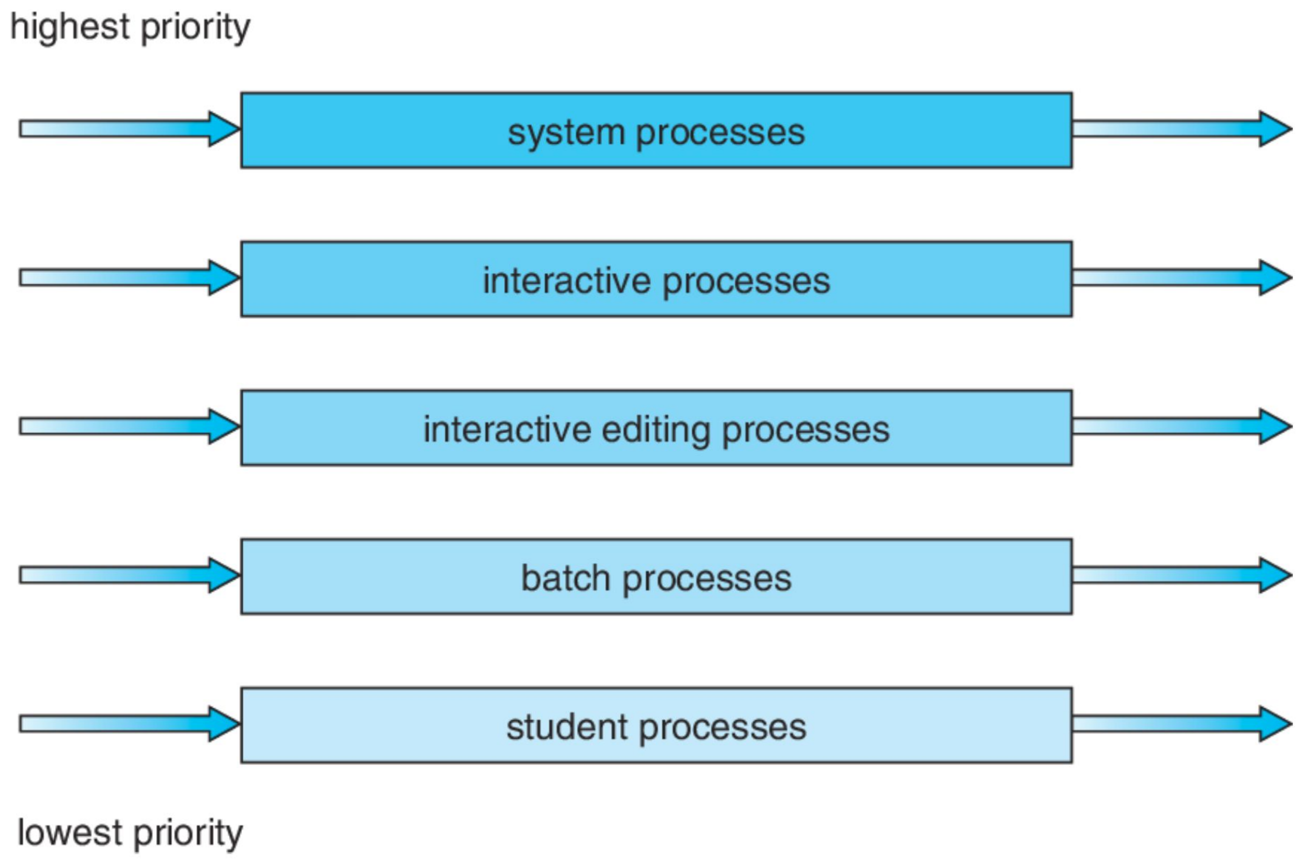
Exercise

- 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?
  - $\alpha = 0$, $\tau_0 = 100$ ms
  - $\alpha = 0.99$, $\tau_0 = 10$ ms

When $\alpha = 0$ the formula does not take recent history, and will always predict the initial 100ms for CPU burst

When $\alpha = 0.99$ the recent behavior of process is given much more weight than its past. The prediction is almost memory-less

Multilevel Queue Scheduler

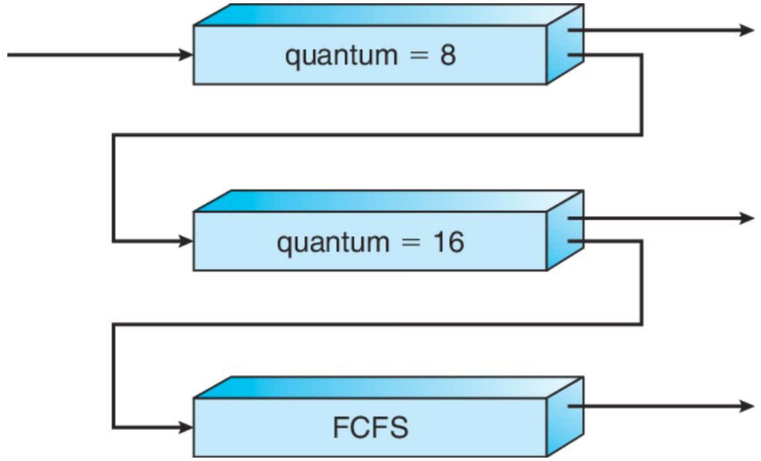


Multilevel Feedback Queue Scheduler

- 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

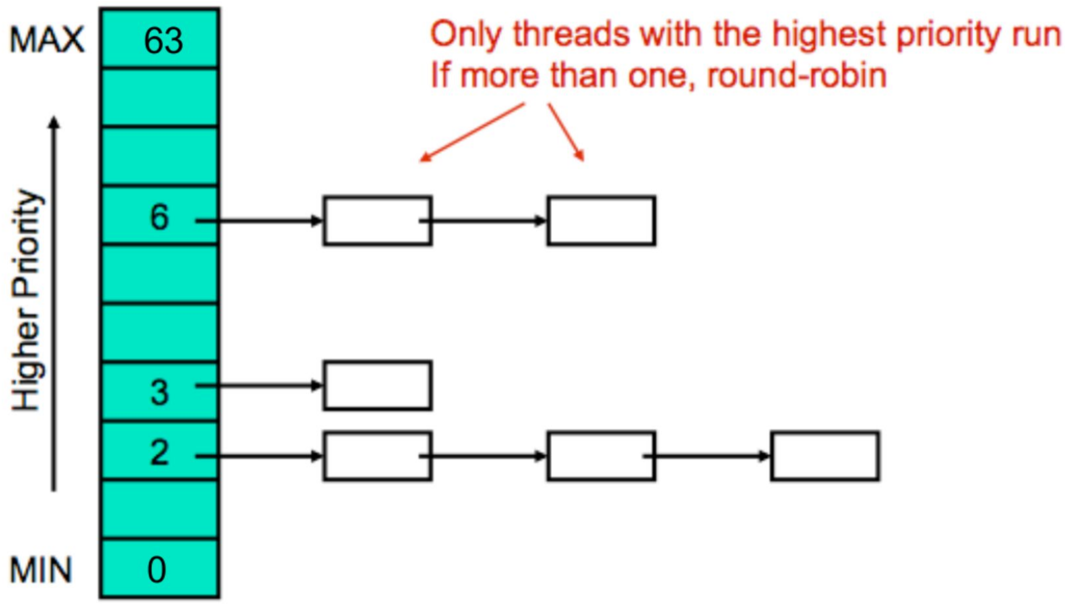
Example of Multilevel Feedback Queue Scheduler

- 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**



MLFQS: 4.4BSD Priority Based Scheduler

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


MAX
63
6
3
2
0
MIN
Higher Priority
Only threads with the highest priority run
If more than one, round-robin

MLFQS: 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)

MLFQS: “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

MLFQS: 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:

a = (2 * load_avg) / (2 * load_avg + 1)

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

MLFQS: Calculating “*load_avg*”

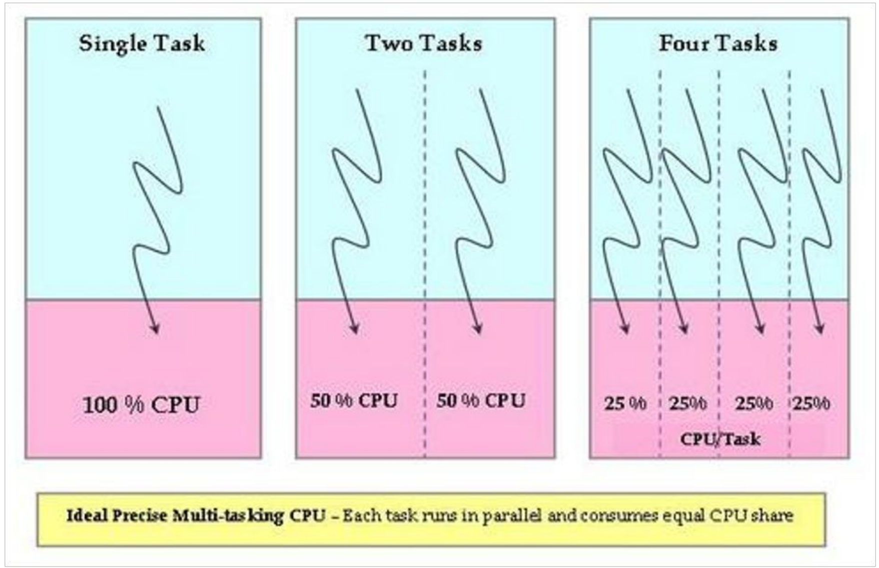
- Estimates the average number of threads ready to run over the past minute
- Like ***recent_cpu***, it is an exponentially weighted moving average
- Unlike ***priority*** and ***recent_cpu***, ***load_avg*** is system-wide, not thread-specific
- At system boot, it is initialized to 0. Once per second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

Proportional Share Scheduling

- Try to guarantee that each process gets a certain percentage of the CPU time
  - Over some period of time, sometimes called the scheduling interval
  - “Fair share” may depend on process’s priority

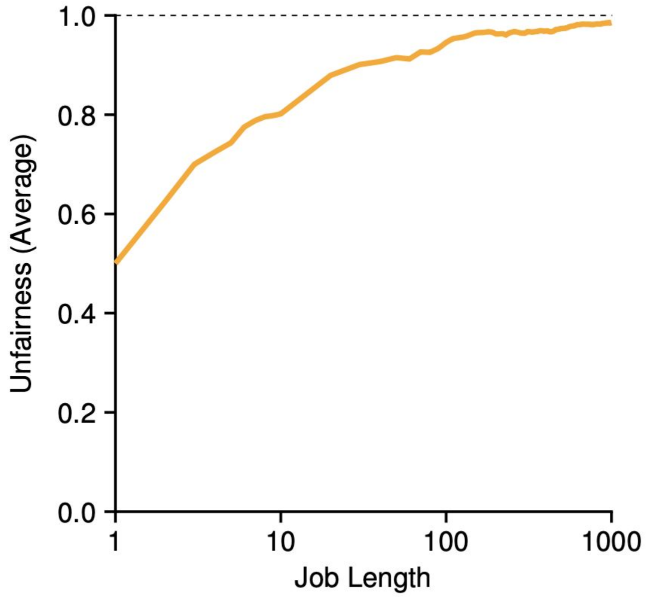

Single Task
100 % CPU
Two Tasks
50 % CPU
50 % CPU
Four Tasks
25 %
25%
25%
25%
CPU/Task
Ideal Precise Multi-tasking CPU - Each task runs in parallel and consumes equal CPU share

Lottery Scheduling

- Tickets assigned to a process represent its share
    - e.g., 10 out of 100 tickets process should get 10% of the CPU time
- Scheduling is probabilistic, using a random number generator
    - What problem does this raise?
- What about processes that do I/O?
- Ticket currency allows local authority (e.g., user) flexibility in allocating tickets
    - e.g., can do ticket inflation within local group of processes that trust each other
- Efficient implementation

Lottery Scheduling: Problems

- How To Assign Tickets?
  - Open problem
- Unfairness:
  - Two jobs with short lengths
  - Unfairness metric = A_rutime/B_runtime
- Not Deterministic



Stride Scheduling

- Deterministic fair share scheduler
- Each process has a stride which is inversely proportional to the number of tickets it has
- Every time a process runs, the scheduler increments its pass value by its stride
- Schedule the process with the lowest pass value to run next
- When a new job enters, what should be its initial pass value?

Linux Completely Fair Scheduler (CFS)

- Default scheduling policy in Linux since Linux 2.6.23 release (October 2007)
- Linux scheduler is modular and allows for other scheduling policies.
  - Type 'man sched' on a Linux system
  - https://github.com/torvalds/linux/blob/master/kernel/sched
- Goal: Divide CPU time evenly according to *vruntime*
  - Schedule the process with the lowest vruntime to run next
- *sched_latency* – scheduling interval to be divided up (typical value is 48 ms)
- *min_granularity* – smallest possible time slice (typical value is 6 ms)
- Implemented using red-black trees for efficiency

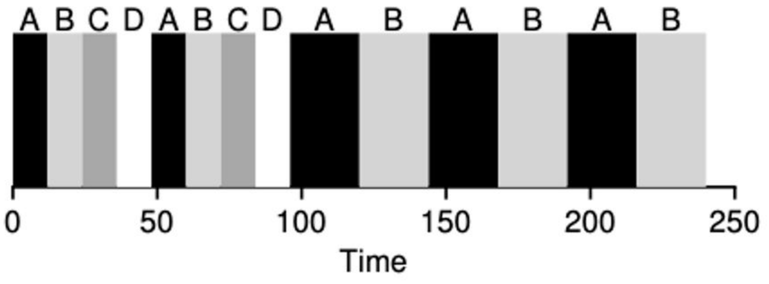


Figure 9.4: **CFS Simple Example**

CFS Weighting using Niceness

- nice and nice(2) man pages
- Example
  - $ nice –n 5 ./matmul-linux & $ ./matmul-linux

$$\mathtt{vruntime}_i = \mathtt{vruntime}_i + \frac{\mathtt{weight}_0}{\mathtt{weight}_i} \cdot \mathtt{runtime}_i$$

$$\mathtt{time\_slice}_k = \frac{\mathtt{weight}_k}{\sum_{i=0}^{n-1} \mathtt{weight}_i} \cdot \mathtt{sched\_latency}$$

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

Announcement

- Homework 1 demo
    - TA is in contact with 20 students via email
    - YOU MUST ENSURE YOU SHOW THE DEMO IN THE GIVEN TIME SLOTS
        - PENALTY OTHERWISE
- Homework 2
    - Due on Monday October 6th, 11.59 PM