Outline

- Multi-Level Feedback Queue (MLFQ)
- Lottery Scheduling
- Thread Scheduling
- Scheduling in SMP Environments
- Scheduling in Linux
Multi-Level Feedback Queue (MLFQ)
Multi-Level Feedback Queue (MLFQ)

- Goals of MLFQ
- Optimize turnaround time
  - In reality, SJF does not work since OS does not know how long a process will run
- Minimize response time
  - Unfortunately, RR is really bad on optimizing turnaround time
MLFQ: Basics

- MLFQ maintains a number of queues (multi-level queue)
  - Each assigned a different priority level
  - Priority decides which process should run at a given time
How to know process type to set priority?
1. nice
2. history
How to Check Nice Values in Linux?

- % ps ax -o pid,ni,cmd
MLFQ Example

How to know process type to set priority?
1. nice
2. history

In this example, A and B are given high priority to run, while C and D may starve
MLFQ: Basic Rules

- MLFQ maintains a number of queues (multi-level queue)
  - Each assigned a different priority level
  - Priority decides which process should run at a given time

- **Rule 1:** If Priority\((A) > \text{Priority}(B)\), \(A\) runs (\(B\) doesn’t).
- **Rule 2:** If Priority\((A) = \text{Priority}(B)\), \(A\) & \(B\) run in RR.
Attempt #1: Change Priority

- Workload
  - Interactive processes (many short-run CPU bursts)
  - Long-running processes (CPU-bound)
- Each time quantum = 10ms

**Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
**Rule 4a:** If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
**Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the same priority level.
Example 1: One Single Long-Running Process

- A process enters at highest priority (time quantum = 10ms)
Example 1: One Single Long-Running Process

- A process enters at highest priority (time quantum = 10ms)
Example 1: One Single Long-Running Process

- A process enters at highest priority (time quantum = 10ms)
Example 2: Along Came a Short-Running Process

- Process A: long-running process (start at 0)
Example 2: Along Came a Short-Running Process

- Process A: long-running process (start at 0)
- Process B: short-running interactive process (start at 100)
Example 2: Along Came a Short-Running Process

- Process A: long-running process (start at 0)
- Process B: short-running interactive process (start at 100)
Example 2: Along Came a Short-Running Process

- Process A: long-running process (start at 0)
- Process B: short-running interactive process (start at 100)
Example 3: What about I/O?

- Process A: long-running process
- Process B: I/O-intensive interactive process (each CPU burst = 1ms)
Example 4: What’s the Problem?

- Process A: long-running process
- Process B + C: Interactive process
Example 4: What’s the Problem?

- Process A: long-running process
- Process B + C: Interactive process

Interactive Process B
Interactive Process C

CPU-intensive Process A starves!
Attempt #2: Priority Boost

- Simple idea: Periodically boost the priority of all processes

- **Rule 5:** After some time period $S$, move all the jobs in the system to the topmost queue.

![Diagram showing priority boosts and process interactions]

- CPU-intensive Process A proceeds!
- Interactive Process B
- Interactive Process C
Tuning MLFQ

- MLFQ scheduler is defined by many parameters:
  - Number of queues
  - Time quantum of each queue
  - How often should priority be boosted?
  - A lot more…

- The scheduler can be configured to match the requirements of a specific system
  - Challenging and requires experience
Lottery Scheduling
Lottery Scheduling

- **Goal:** Proportional share
  - One of the fair-share schedulers

- **Approach**
  - Gives processes lottery tickets
  - Whoever wins runs
  - Higher priority --> more tickets
Lottery Code

```c
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
// get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}

// 'current' is the winner: schedule it...
```
Lottery Scheduling Example

<table>
<thead>
<tr>
<th>Job A</th>
<th>Job B</th>
<th>Job C</th>
<th>Job D</th>
<th>Job E</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(1)</td>
<td>(100)</td>
<td>(200)</td>
<td>(100)</td>
</tr>
</tbody>
</table>

402 total tickets
Lottery Scheduling Example

winner = random(402)

[Block diagram showing jobs A to E with their respective quantities and 402 total tickets]
Lottery Scheduling Example

winner = 102

Job A (1)  |  Job B (1)  |  Job C (100)  |  Job D (200)  |  Job E (100)

402 total tickets
Lottery Scheduling Example

winner = 102

Is 1 > 102?

Job A (1)  |  Job B (1)  |  Job C (100)  |  Job D (200)  |  Job E (100)

402 total tickets
Lottery Scheduling Example

winner = 102

Is $2 > 102$?

Job A (1)  |  Job B (1)  |  Job C (100)  |  Job D (200)  |  Job E (100)

402 total tickets
Lottery Scheduling Example

winner = 102

Is 102 > 102?

Job A (1)  Job B (1)  Job C (100)  Job D (200)  Job E (100)

402 total tickets
Lottery Scheduling Example

winner = 102

Is 302 > 102?

Job A
(1)

Job B
(1)

Job C
(100)

Job D
(200)

Job E
(100)

402 total tickets
Lottery Scheduling Example

winner = 102

302 > 102

OS picks Job D to run!
Other Lottery Ideas

- Ticket transfers
- Ticket currencies
- Ticket inflation
- Read more in OSTEP
Thread Scheduling
When processes have also threads, we have two levels of (pseudo-)parallelism.

Scheduling in such systems differs depending on whether user-level or kernel-level threads are supported.

With **user-level threads**, in addition to the process scheduling performed by OS, we must have thread scheduling performed by the run-time thread management system of each process.

With **kernel-level** threads, the operating system can directly perform thread scheduling.
Thread scheduling with user-level threads

Possible scheduling of user-level threads with a 50 ms time quantum and threads that yield the CPU after 5 ms
Thread scheduling with kernel-level threads

Possible scheduling of kernel-level threads with a 50 ms time quantum and threads that yield the CPU after 5 ms

Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3
Thread scheduling

- With kernel-level threads, OS can exercise a finer control on the execution sequence of threads belonging to different processes.

- With user-level threads, each process can use its own customized thread scheduling algorithm.

- With kernel-level threads, a thread switch involves more overhead.
Scheduling in SMP Environments
Scheduling in SMP Environments

- Two main approaches
  - All processes are in a common ready queue *(Global Scheduling)*
  - Each CPU has its own (private) ready queue *(Partitioned Scheduling)*

- Two important dimensions
  - Processor Affinity
  - Load Balancing
Processor Affinity

- A process has an “affinity” with the processor on which it is currently running
  - Mainly due to the impact of private cache

- OS may have different mechanisms to enforce affinity
  - Soft affinity
  - Hard affinity
  - Affinity with a set of processors

- The main memory architecture can affect processor affinity

- In non-uniform memory access (NUMA) systems, a processor has faster access to some parts of the main memory
NUMA and CPU Scheduling
Load Balancing

- Dynamic Load balancing is often **unnecessary** in systems with global scheduling.

- In partitioned scheduling, some dynamic adjustment of the workload distribution (controlled migrations) may be desirable:
  - **Push** migration: A specific OS task periodically checks the load on each processor and re-adjusts the load if necessary.
  - **Pull** migration: An idle processor “pulls” a waiting task from a busy processor.

- Load balancing vs. processor affinity.
Recent trend to place multiple processor cores on same physical chip
  – Faster and consumes less power

Multiple threads per core also growing
  – Direct hardware support to threads (hardware multi-threading)
  – When a thread is subject to a memory stall (e.g. due to a cache miss), the hardware automatically switches to another thread – transparent to the operating system!
  – This is often achieved by replicating the user-visible register sets (at the hardware-level)
Quantifying Gains of Parallelism: Amdahl’s Law

- Impact of adding cores on the running time of an application with both serial and parallel components
- S is serial portion, N is the number of cores

\[ speedup \leq \frac{1}{S + \frac{(1-S)}{N}} \]

- e.g.: \( S = 0.25 \), moving from 1 to 2 cores \( \Rightarrow \) speedup \( \leq 1.6 \)
  - As \( N \) approaches infinity, speedup approaches \( 1 / S \)

- Serial portion of an application has disproportionate effect on the performance gain provided by additional cores
Scheduling in Linux
Linux Scheduling History

- 2001 2.4 O(N) scheduler, split time into epochs where each task was allowed a certain time slice, iterating through N runnable tasks and applying goodness function to determine next task.

- 2003 2.6 O(1) scheduler used multiple runqueues for each priority, it was a more efficient and scalable version of O(N), it introduced a bonus system for interactive vs. batch tasks.

- 2008 2.6.23 till now: Completely Fair Scheduler

Paper: The Linux Scheduler: A Decade of Wasted Cores
Linux Scheduling

- With version 2.5, the Linux scheduler was overhauled
  - Better support for SMP systems
  - “O(1)” scheduling algorithm

- To fix the poor performance for interactive processes, Completely Fair Scheduler (CFS) became the default Linux scheduling algorithm in release 2.6.23 till now (v4.13)

- Scheduling classes
  - Each class is assigned a specific priority range
  - Kernel can use different scheduling algorithms for different classes
Linux Scheduling (cont.)

- Standard Linux kernels implement two scheduling classes
  - A real-time scheduling class
  - A default scheduling class using CFS
- Real-time tasks have static priorities (POSIX.1b)
Linux CFS for non-real-time tasks

- CFS assigns a portion of the CPU processing time to each task
  - Proportion computed based on the nice values of tasks
  - Nice values range from -20 to +19 (mapping to priorities 100-139)

- CFS scheduler maintains per task “virtual run time” in variable vruntime
  - Computed based on the actual run time of the process and its priority
  - Scheduler selects the task with smallest vruntime value

- “Targeted latency”: interval of time during which every runnable task should ideally run at least once
  - Concept alternative to “time slice”