Multilevel Feedback Queues (MLFQ)

- Multilevel feedback queues use past behavior to predict the future and assign job priorities
  
  => overcome the prediction problem in SJF

- If a process is I/O bound in the past, it is also likely to be I/O bound in the future (programs turn out not to be random.)

- To exploit this behavior, the scheduler can favor jobs that have used the least amount of CPU time, thus approximating SJF.

- This policy is adaptive because it relies on past behavior and changes in behavior result in changes to scheduling decisions.
Approximating SJF: Multilevel Feedback Queues

- Multiple queues with different priorities.
- Use Round Robin scheduling at each priority level, running the jobs in highest priority queue first.
- Once those finish, run jobs at the next highest priority queue, etc. (Can lead to starvation.)
- Round robin time slice increases exponentially at lower priorities.
Adjusting Priorities in MLFQ

• Job starts in highest priority queue.

• If job's time slices expires, drop its priority one level.

• If job's time slices does not expire (the context switch comes from an I/O request instead), then increase its priority one level, up to the top priority level.

⇒ CPU bound jobs drop like a rock in priority and I/O bound jobs stay at a high priority.

• Allow preemption when higher priority processes finish I/O
# Multilevel Feedback Queues: Example 1

3 jobs, of length 30, 20, and 10 seconds each, initial time slice 1 second, context switch time of 0 seconds, all CPU bound (no I/O), 3 queues

<table>
<thead>
<tr>
<th>Queue</th>
<th>Time Slice</th>
<th>Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Job</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Completion</th>
<th>Wait Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>MLFQ</td>
</tr>
<tr>
<td>RR</td>
<td>MLFQ</td>
</tr>
</tbody>
</table>

Average

<table>
<thead>
<tr>
<th>Queue</th>
<th>Time Slice</th>
<th>Job</th>
<th>Completion</th>
<th>Wait Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>RR</td>
<td>MLFQ</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>RR</td>
<td>MLFQ</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
<td>Average</td>
<td></td>
</tr>
</tbody>
</table>
MLFQ: Example 1

Q | TS | J
---|----|---
0 | 1  | 0,1,2
1 | 2  | 
2 | 4  | 

Q | TS | J
---|----|---
0 | 1  | 0,1,2
1 | 2  | 0,1,2
2 | 4  | 

Q | TS | J
---|----|---
0 | 1  | 
1 | 2  | 
2 | 4  | 0,1,2
# Multilevel Feedback Queues: Example 1

<table>
<thead>
<tr>
<th>Job</th>
<th>Length</th>
<th>Completion Time</th>
<th>Wait Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RR (MLFQ)</td>
<td>RR (MLFQ)</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
<td>60 (60)</td>
<td>30 (30)</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>50 (53)</td>
<td>30 (33)</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>30 (32)</td>
<td>20 (22)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Average</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>46 2/3</td>
<td>26 2/3</td>
</tr>
</tbody>
</table>

RR: `.mlfq.py --jlist 0,30,0:0,20,0:0,10,0 -Q 1 -c -S`

MLFQ: `.mlfq.py --jlist 0,30,0:0,20,0:0,10,0 -Q 1,2,4 -c -S`
3 jobs, of length 30, 20, and 10 seconds, the 10 sec job does 1 sec of I/O right after 1 second of CPU. Context switch time of 0 sec, 2 queues.

<table>
<thead>
<tr>
<th>Queue</th>
<th>Time Slice</th>
<th>Job</th>
<th>Completion RR</th>
<th>Completion MLFQ</th>
<th>Wait Time RR</th>
<th>Wait Time MLFQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Average</strong></td>
<td><strong>Average</strong></td>
<td><strong>Average</strong></td>
<td><strong>Average</strong></td>
</tr>
</tbody>
</table>

Multilevel Feedback Queues: Example 2
MLFQ: Example 2
Multilevel Feedback Queues: Example 2

<table>
<thead>
<tr>
<th>Job</th>
<th>Length</th>
<th>Completion Time</th>
<th>Wait Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RR</td>
<td>MLFQ</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>49</td>
<td>52</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>50</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>53</td>
<td>45</td>
</tr>
</tbody>
</table>

RR: "./mlfq.py --jlist 0,30,0:0,20,0:0,10,1 -i 1 -Q 2 -c -S"
MLFQ: "./mlfq.py --jlist 0,30,0:0,20,0:0,10,1 -i 1 -Q 2,4 -c -S"
Clicker Question #1

In a MLFQ, if you have two jobs: A and B. A is CPU bound and take 10 units, B is all I/O and runs forever, how long does A take to run? (0 time for context switch)

(A) 0
(B) 9
(C) 10
(D) 20
Answer on Next Slide
Improving Fairness

Since SJF is optimal, but unfair, any increase in fairness by giving long jobs a fraction of the CPU when shorter jobs are available will degrade average waiting time.

Possible solutions:

• Give each queue a fraction of the CPU time. This solution is only fair if there is an even distribution of jobs among queues.

• Adjust the priority of jobs as they do not get serviced (Unix originally did this.)
  
  - This ad hoc solution avoids starvation but average waiting time suffers when the system is overloaded because all the jobs end up with a high priority.
Summary of Scheduling:

- **FCFS:** Not fair, and average waiting time is poor.

- **Round Robin:** Fair, but average waiting time is poor.

- **SJF:** Not fair, but average waiting time is minimized assuming we can accurately predict the length of the next CPU burst. Starvation is possible.

- **Multilevel Queuing:** An implementation (approximation) of SJF.

⇒ Our modeling assumed that context switches took no time, which is unrealistic.
Clicker Question #2

Which scheduling algorithm gives the lowest average waiting time? And what scheduling algorithm is an approximation of it?

(A) Round Robin + FCFS

(B) FCFS + Round Robin

(C) SJF + Round Robin

(D) SJF + MLFQ

(E) FCFS + SJF
Answer on Next Slide
Threads

• What are threads?

• Where should we implement threads? In the kernel? In a user level threads package?

• How should we schedule threads (or processes) onto the CPU?
Processes versus Threads

- A **process** defines the address space, text, resources, etc.,

- A **thread** defines a single sequential execution stream within a process (PC, stack, registers).

- Threads extract the thread of control information from the process

- Threads are bound to a single process.

- Each process may have multiple threads of control within it.

  - The **address space of a process is shared among all its threads**

  - No system calls are required to cooperate among threads
Single and Multithreaded Processes

- Single-threaded process
- Multithreaded process
Clicker Question #3

If I fork a process A, creating process B, then I create a thread C in process B, and in C I change a global variable, does it change in A? Does it change in B?

(A) NO, NO

(B) NO, YES

(C) YES, NO

(D) YES, YES
Answer on Next Slide
Example Threaded Program

- Forking a thread can be a system call to the kernel, or a procedure call to a thread library (user code).
- Note: The example has 3 threads: main, producer and consumer. The main thread exits after creating the producer and consumer and is not shown.

```plaintext
main()
    global in, out, n, buffer[n];
    in = 0; out = 0;
    fork_thread (producer());
    fork_thread (consumer());
end

producer
    repeat
        nextp = produced item
        while in+1 mod n = out do no-op
        buffer[in] = nextp; in = (in+1) mod n

consumer
    repeat
        while in = out do no-op
        nextc = buffer[out]; out = (out+1) mod n
        consume item nextc
```

One possible memory layout:

```
static data
  heap ↓
  
thread 2  ↓ stack
  thread 1  ↓ stack

PC₁  ↓ text

Memory
```
Kernel Threads

• A kernel thread, also known as a lightweight process, is a thread that the operating system knows about.

• Switching between kernel threads of the same process requires a small context switch.
  
  – The values of registers, program counter, and stack pointer must be changed.
  
  – Memory management information does not need to be changed since the threads share an address space.

• The kernel must manage and schedule threads (as well as processes), but it can use the same process scheduling algorithms.

• Switching between kernel threads is slightly faster than switching between processes.
User-Level Threads

- A **user-level thread** is a thread that the OS does not know about.
  - Sometimes called “green threads”
  - The OS only knows about the process containing the threads.
  - The OS only schedules the process, not the threads within the process.
  - The programmer uses a thread library to manage threads (create and delete them, synchronize them, and schedule them).
User-Level Threads

Thread Ready Queue

Current Thread for each Process

Thread Ready Queue

User-Level Thread Schedule

User

Kernel Processes

Kernel

Process Ready Queue
User-Level Threads: Advantages

- There is no context switch involved when switching threads.

- User-level thread scheduling is more flexible

- A user-level code can define a problem dependent thread scheduling policy.

- Each process might use a different scheduling algorithm for its own threads.

- A thread can voluntarily give up the processor by telling the scheduler it will *yield* to other threads.

- User-level threads do not require system calls to create them or context switches to move between them

  - Thread management calls are library calls and much faster than system calls made by kernel threads

- User-level threads are typically much faster than kernel threads
User-Level Threads: Disadvantages

• Since the OS does not know about the existence of the user-level threads, it may make poor scheduling decisions:
  – It might run a process that only has idle threads.
  – If a user-level thread is waiting for I/O, the entire process will wait.
  – Solving this problem requires communication between the kernel and the user-level thread manager.

• Since the OS just knows about the process, it schedules the process the same way as other processes, regardless of the number of user threads.

• For kernel threads, the more threads a process creates, the more time slices the OS will dedicate to it.

• User level threads can not take advantage of the presence of multiple cores.
Threading Models

- Many-to-one, one-to-one, many-to-many and two-level
What gets used in practice?

- As threads became essential, OSs were designed to support them well
- All modern OSs support a one-to-one mapping
  - Linux, Windows, Mac
- Occasionally one will run into odd-ball systems that use user-level threads (for instance old Ruby code)
  - Chrome uses threads inside of multiple processes!
Clicker Question #4

In a User-Level threading library, if I do (blocking) I/O in one thread, do the other threads run? What about in a one-to-one mapping?

(A) NO, NO
(B) NO, YES
(C) YES, NO
(D) YES, YES
Answer on Next Slide
Thread Libraries

• Thread library provides programmer with API for creating and managing threads

• Two primary ways of implementing
  – Library entirely in user space
  – Kernel-level library supported by the OS
Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- WIN32 Threads: Similar to Posix, but for Windows
Java Threads

• Java threads are managed by the JVM

• Typically implemented using the threads model provided by underlying OS

• Java threads may be created by:
  – Extending Thread class
  – Implementing the Runnable interface
Examples

Pthreads:

    pthread_attr_init(&attr); /* set default attributes */

    pthread_create(&tid, &attr, sum, &param);

Win32 threads

    ThreadHandle = CreateThread(NULL, 0, Sum, &Param, 0, &ThreadID);

Java Threads:

    Sum sumObject = new Sum();

    Thread t = new Thread(new Summation(param, SumObject));

    t.start(); // start the thread
Summary

• Thread: a single execution stream within a process

• Switching between user-level threads is faster than between kernel threads since a context switch is not required.

• User-level threads may result in the kernel making poor scheduling decisions, resulting in slower process execution than if kernel threads were used.

• Many scheduling algorithms exist. Selecting an algorithm is a policy decision and should be based on characteristics of processes being run and goals of operating system (minimize response time, maximize throughput, ...).