Chapter 5: CPU Scheduling

肖 卿 俊

办公室：九龙湖校区计算机楼 212 室
电邮：csqjxiao@seu.edu.cn
主页：https://csqjxiao.github.io/PersonalPage
电话：025-52091022
Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming

- A Fact: Process execution consists of an alternating sequence of CPU execution and I/O wait, called CPU–I/O Burst Cycle
CPU-I/O Burst Cycle

- Process execution repeats the CPU burst and I/O burst cycle.

- When a process begins an I/O burst, another process can use the CPU for a CPU burst.
CPU-bound and I/O-bound

- A process is **CPU-bound** if it generates I/O requests infrequently, using more of its time doing computation.

- A process is **I/O-bound** if it spends more of its time to do I/O than it spends doing computation.
A CPU-bound process might have a few very long CPU bursts.

An I/O-bound process typically has many short CPU bursts.
CPU Scheduler

- When the CPU is idle, the OS must select another process to run.

- This selection process is carried out by the short-term scheduler (or CPU scheduler).

- The CPU scheduler selects a process from the ready queue and allocates the CPU to it.

- The ready queue does not have to be a FIFO one. There are many ways to organize the ready queue.
CPU Scheduling Occurs

- new
  - converting to process
  - admitted
- ready
  - scheduler dispatch
  - interrupt
- running
  - reclaim resource
  - destroy process
  - exit
- waiting
  - waiting for CPU
  - waiting for I/O or event completion
- I/O or event wait
- terminated
  - exit

Waiting for I/O or event
Circumstances that scheduling may take place

1. A process switches from the **running** state to the **waiting** state (e.g., doing for I/O)

2. A process switches from the **running** state to the **ready** state (e.g., an **interrupt** occurs)

3. A process switches from the **waiting** state to the **ready** state (e.g., I/O completion)

4. A process **terminates**
Non-preemptive vs. Preemptive

**Non-preemptive scheduling**: scheduling occurs when a process voluntarily leave the CPU resource. It either enters the waiting state (case 1) or terminates (case 4).
- Simple, but very efficient with less context switch
- 对应以前提及过的多道系统（multi-programmed OS）

**Preemptive scheduling** ( 抢占式调度): scheduling occurs in all possible cases.
- What if the running process is in critical section modifying some shared data? There is a possibility of race condition. Mutual exclusion of accessing critical section may be violated.
- The kernel must pay special attention to this situation and, hence, is more complex
Dispatcher

- Dispatcher module (分配器) gives control of the CPU to the process selected by the short-term scheduler (调度器); this involves:
  - switching execution context (save & reload)
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- Dispatch latency – time it takes for the dispatcher to stop one process and start another running.
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling
Scheduling: Policy and Mechanism

- Scheduling policy answers the question:
  - Which process/thread, among all those ready to run, should be given the chance to run next? In what order do the processes/threads get to run? For how long?

- Mechanisms are the tools for supporting the process/thread abstractions and affect how the scheduling policy can be implemented (this is review)
  - How the process or thread is represented to the system - process or thread control blocks.
  - What happens on a context switch.
  - When do we get the chance to make these scheduling decisions (timer interrupts, thread operations that yield or block, user program system calls)
Separation of Policy and Mechanism

- “Why and What” vs. “How”
- Objectives and strategies vs. data structures
  - hardware and software implementation issues.
- Process abstraction vs. Process machinery
CPU Scheduling Policy

The CPU scheduler makes a sequence of “moves” that determines the interleaving of processes

- Programs use process synchronization to prevent “bad moves”
- ...but otherwise scheduling choices appear (to the program) to be nondeterministic.

The scheduler’s moves are dictated by a scheduling policy
Scheduling Criteria

Before presenting detailed scheduling policies, we discuss how to evaluate the “goodness” of a scheduling policy.

There are many criteria for comparing different scheduling policies. Here are five common ones:

- CPU utilization (CPU利用率)
- Throughput (执行任务的吞吐量)
- Turnaround time (进程的周转时间)
- Waiting time (进程的等待时间)
- Response time (对用户的响应时间)
CPU Utilization

- We want to keep the CPU as busy as possible.

- CPU utilization ranges from 0 to 100 percent. Normally 40% is **lightly** loaded and 90% or higher is **heavily** loaded.

- You can bring up a CPU usage meter to see CPU utilization on your system.
Throughput

- The number of processes completed per time unit is called *throughput*.

- Higher throughput means more jobs get done.

- However, this criteria is affected by the characteristics of processes.
  - For long processes, this rate may be one job per hour, and, for short jobs, this rate may be 10 per minute.
Turnaround Time

- The time period between job submission to completion is the **turnaround time**.

- From a user’s point of view, turnaround time is more important than CPU utilization and throughput.

- Turnaround time is the sum of
  - waiting time before entering the system
  - waiting time in the ready queue
  - waiting time in all other events (e.g., I/O)
  - time the process actually running on the CPU
CPU Scheduling Occurs

- new
  - converting to process
  - admitted

- ready
  - scheduler dispatch
  - interrupt
  - waiting for CPU
  - I/O or event completion

- running
  - reclaim resource
  - destroy process

- waiting
  - waiting for I/O or event

- terminated
  - exit

I/O or event wait

Operating System Concepts
Waiting Time

Waiting time is the sum of the periods that a process spends waiting in the ready queue.

Why only ready queue?

- CPU scheduling algorithms do not affect the amount of time during which a process is waiting for I/O and other events.
- However, CPU scheduling algorithms do affect the time that a process stays in the ready queue.
Response Time

- The time from the submission of a request (in an interactive system) to the first response is called response time. It does not include the time that it takes to output the response.

- For example, in front of your workstation, you perhaps care more about the time between hitting the Return key and getting your first output (e.g., response time) than the time from hitting the Return key to the completion of your program (e.g., turnaround time).
Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling
We will discuss a number of scheduling algorithms (or scheduling policies):

- First-Come, First-Served (FCFS)
- Round-Robin
- Lottery Scheduling (with demonstration code)
- Shortest-Job-First (SJF)
- Priority
- Multilevel Queue
- Multilevel Feedback Queue
First-Come, First-Served (FCFS) Scheduling

- The process that requests the CPU first is allocated the CPU first.

- This can easily be implemented using a queue.

- FCFS is not preemptive. Once a process has the CPU, it will occupy the CPU until the process completes or voluntarily enters the wait state.
Suppose that the processes arrive in the order: $P_1, P_2, P_3$

The Gantt Chart for the schedule is:

**Waiting time?**

**Average waiting time?**
FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order $P_2, P_3, P_1$.

- The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong></td>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- **Convoy effect** short process behind long process
FCFS Problems

- It is easy to have the *convoy effect*: all the processes wait for the one big process to get off the CPU.

- Consider a CPU-bound process running with many I/O-bound process.

- It is in favor of long processes and may not be fair to those short ones. What if your 1-minute job is behind a 10-hour job?

- It is troublesome for time-sharing systems, where each user needs to get a share of the CPU at regular intervals.
Round Robin (RR)

- RR is similar to FCFS, except that each process is assigned a **time quantum**.

- All processes in the ready queue is a **FIFO** list.

- When CPU is free, the scheduler picks the **first** and lets it run for **one time quantum (or slice)**

- If that process uses CPU for less than one time quantum, it is moved to the of the **waiting** list.

- Otherwise, when one time quantum is up, that process is **preempted** by the scheduler and moved to the tail of the ready queue, a **FIFO** list.
## Example of RR with Time Quantum = 20

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>53</td>
</tr>
<tr>
<td>$P_2$</td>
<td>17</td>
</tr>
<tr>
<td>$P_3$</td>
<td>68</td>
</tr>
<tr>
<td>$P_4$</td>
<td>24</td>
</tr>
</tbody>
</table>

The Gantt chart is:

```
0  20  37  57  77  97  117  121  134  154  162
P_1 P_2 P_3 P_4 P_1 P_3 P_4 P_1 P_3 P_3
```
RR Scheduling: Some Issues

- If time quantum is **too large** (i.e., larger than all the CPU bursts), RR reduces to FCFS
- If time quantum is **too small** (smaller than all the CPU bursts), RR becomes processor sharing
- Context switching may affect RR’s performance
  - Shorter time quantum means more context switches
- Turnaround time also depends on the size of time quantum.
- In general, 80% of the CPU bursts should be shorter than the time quantum.
Context switching may affect RR’s performance

- Shorter time quantum means more context switches
Turnaround Time Varies With The Time Quantum

When time quantum = 1,
Turnaround of \(P_1\) = 15
Turnaround of \(P_2\) = 9
Turnaround of \(P_3\) = 3
Turnaround of \(P_4\) = 17
Average Turnaround = 11

If using SJF (P3, P2, P1, P4)
Turnaround of \(P_1\) = 10
Turnaround of \(P_2\) = 4
Turnaround of \(P_3\) = 1
Turnaround of \(P_4\) = 17
Average Turnaround = 8

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>6</td>
</tr>
<tr>
<td>(P_2)</td>
<td>3</td>
</tr>
<tr>
<td>(P_3)</td>
<td>1</td>
</tr>
<tr>
<td>(P_4)</td>
<td>7</td>
</tr>
</tbody>
</table>
Summary on Round-Robin

- RR is awful in turnaround time. Is it?

- Any policy that is fair, performs poorly on performance metrics such as turnaround time.
  - Tradeoff between fairness and performance.

- Typically, higher average turnaround time than SJF, but shorter response time.
In practice, many variations or RR

- Weighted round robin (WRR): allow each requestor a share of the common resource which is relative to its predefined weight.

- Deficit round robin (DRR): generalize the RR Scheduling to achieve Weighted Max-Min fairness. Easier to implement than WFQ
  [http://www.mathcs.emory.edu/~cheung/Courses/558/Syllabus/11-Fairness/DRR.html](http://www.mathcs.emory.edu/~cheung/Courses/558/Syllabus/11-Fairness/DRR.html)

- Weighted fair queueing (WFQ): a dynamic process that divides bandwidth (or resources).
Lottery Scheduling

- RR gives a roughly equal share of CPU to all ready processes

- Lottery scheduler is a proportional-share scheduler (fair-share scheduler)

- Instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time
Lottery Scheduling (cont.)

- Basic idea
  - Every so often, hold a lottery to determine which process should get to run next;
  - Processes that should run more often should be given more chances to win the lottery.

- Tickets
  - are used to represent the share of a resource that a process (or user or whatever) should receive.

List-based lottery (winner = 15)
Suppose:

- Two jobs competing against one another, each with the same number of tickets and the same run time.

An unfairness metric $U$:

- The time the first job completes divided by the time that the second job completes.
- With a perfect fair scheduler, two jobs should finish at roughly the same time, i.e., $U=1$. 
A Simple Unfairness Metric

Lottery scheduling is probabilistic.

Only as the jobs run for a significant number of time slices does the lottery scheduler approach the desired outcome.

Figure 9.2: Lottery Fairness Study
Lottery Scheduling: Summary

Lottery scheduling has not achieved wide-spread adoption as CPU schedulers.

- Ticket assignment is a hard problem.

However, it is useful in domains where this problem is relatively easy to solve.

- VMWare: You might like to assign one-quarter of your CPU cycles to the Windows VM and the rest to your base Linux installation.
Lottery Demonstration Code

```c
int gtickets = 0; // global ticket count
struct node_t {
    int tickets;
    struct node_t *next;
};
struct node_t *head = NULL;
void insert(int tickets) {
    struct node_t *tmp = malloc(sizeof(struct node_t));
    assert(tmp != NULL);
    tmp->tickets = tickets;
    tmp->next = head;
    head = tmp;
    gtickets += tickets;
}
int main(int argc, char *argv[]) {
    …..
    // populate list with some number of jobs
    insert(50); insert(100); insert(25);
    for (int i = 0; i < loops; i++) {
        int counter = 0;
        int winner = random() % gtickets;
        struct node_t *current = head;
        while (current) {
            counter = counter + current->tickets;
            if (counter > winner) break;
            current = current->next;
        }
        printf("winner: %d %d
\n", winner, current->tickets);
    }
}
```

Operating System Concepts
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

- When a process must be selected from the ready queue, the process with the smallest next CPU burst is selected.

- Thus, the processes in the ready queue are sorted in CPU burst length, to avoid the convoy effect.
Shortest-Job-First (SJF) Scheduling (Cont.)

- SJF can be non-preemptive or preemptive.
  - **non-preemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - **preemptive** – if a new process arrives (or enters the ready queue) with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF).

- SJF is optimal – gives minimum average waiting time for a given set of processes.
SJF is provably optimal

- Every time we make a short job before a long job, we reduce average waiting time.
- We may switch out of order jobs until all jobs are in order.
- If the jobs are sorted, job switching is impossible.
### Example of Non-Preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

**SJF (non-preemptive)**

**Average waiting time** = \((0 + 6 + 3 + 7)/4 = 4\)
### Example of Preemptive SJF

**Process** | **Arrival Time** | **Burst Time** | **SJF (preemptive)**
--- | --- | --- | ---
$P_1$ | 0.0 | 7 | $P_1$<br>0<br>2<br>4<br>5<br>7<br>11<br>16
$P_2$ | 2.0 | 4 | $P_3$<br>4<br>1<br>2

Average waiting time $= (9 + 1 + 0 + 2)/4 = 3$
But How Do We Know the Next CPU Burst?

Without a good answer to this question, SJF cannot be used for CPU scheduling.

We try to predict the next CPU burst!

Can be done by using the length of previous CPU bursts, using exponential averaging.

1. Let $t_n$ be the actual length of $n^{th}$ CPU burst
2. Let $\tau_{n+1}$ be the predicted value for the next CPU burst
3. Given $\alpha$, $0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$. 
Two Extreme Examples of Exponential Averaging

\[ \tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n. \]

- **When** \( \alpha = 0 \),
  - \[ \tau_{n+1} = \tau_n \]
  - Recent history does not count.

- **When** \( \alpha = 1 \),
  - \[ \tau_{n+1} = t_n \]
  - Only the actual last CPU burst counts.
Expansion of Exponential Averaging Formula

- If we expand the formula, we get:

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

- Then, \( \tau_{n+1} \) is a linear combination of \( \tau_0, t_1, t_2, \ldots, t_n \)

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

- Exponential averaging is just one time series prediction tool, and there are many others.
An Example of Predicting the Length of the Next CPU Burst

Assume $\alpha = 0.5$

<table>
<thead>
<tr>
<th>CPU burst ($t_i$)</th>
<th>6</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;guess&quot; ($\tau_i$)</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>

What are predicted values?
An Example of Predicting the Length of the Next CPU Burst

Assume $\alpha = 0.5$

<table>
<thead>
<tr>
<th>CPU burst $(t_i)$</th>
<th>6</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;guess&quot; $(\tau_i)$</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Question: How to train the model parameter $\alpha$?
RNN (Recurrent Neural Networks) for Time-Series Prediction

The loss function $L$ of all time steps is defined based on the loss function of every time step

$$
L(\hat{y}, y) = \sum_{t=1}^{T_y} L(\hat{y}^{<t>}, y^{<t>})
$$

Look into an RNN neural cell

For each time step $t$, the cell state $a^t$ and the cell output $y^t$ are expressed as follows:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

and

$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

where $W_{ax}$, $W_{aa}$, $W_{ya}$, $b_a$, $b_y$ are coefficients that are shared temporally and $g_1, g_2$ are activation functions.
SJF Problems

- It is difficult to estimate the next burst time value accurately.

- SJF is in favor of short jobs. As a result, some long jobs may not have a chance to run at all. This is called *starvation*. 
Priority Scheduling

- Each process has a priority.
- Priority may be determined internally or externally:
  - **internal priority**: determined by time limits, memory requirement, # of files, and so on.
  - **external priority**: not controlled by the OS (e.g., importance of the process)
- The scheduler always picks the process (in ready queue) with the highest priority to run.
- FCFS and SJF are special cases of priority scheduling. (Why?)
Priority Scheduling (Cont.)

- Priority scheduling can be **non-preemptive** or **preemptive**.

- With preemptive priority scheduling, if the newly arrived process has a higher priority than the running one, the latter is preempted.

- **Indefinite block** (or **starvation**) may occur: a low priority process may never have a chance to run.
Aging

- Aging is a technique to overcome the starvation problem.

- **Aging**: gradually increases the priority of processes that wait in the system for a long time.

- **Example**: If 0 is the highest (resp., lowest) priority, then we could decrease (resp., increase) the priority of a waiting process by 1 every fixed period (e.g., every minute).
## A Short Recap

<table>
<thead>
<tr>
<th></th>
<th>Average Turnaround Time</th>
<th>Response Time</th>
<th>Fairness</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS</td>
<td>Bad, Convoy effect</td>
<td>Bad, convoy effect</td>
<td>Bad</td>
</tr>
<tr>
<td>Round Robin</td>
<td>Bad, change with time quantum</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Lottery</td>
<td>Bad, any policy that seeks fairness is bad on performance</td>
<td>Probabilistic, so no guarantee on the worst case</td>
<td>Better and more flexible, but ticket assignment is hard</td>
</tr>
<tr>
<td>SJF</td>
<td>Provably optimal</td>
<td>Bad</td>
<td>Bad, essentially a priority scheduler that favors short jobs</td>
</tr>
<tr>
<td>Priority Scheduling (I/O bound &gt; CPU bound)</td>
<td>Could be good, if higher priority is given to processes with shorter CPU bursts</td>
<td>Bad, a low-priority process may not be executed after a long time</td>
<td>Bad, have starvation problem, can be mitigated by aging</td>
</tr>
</tbody>
</table>

**Can we combine the advantages of SJF and Round Robin?**
Multilevel Queue

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)

- Each process is assigned permanently to one queue based on some properties of the process (e.g., memory usage, priority, process type)

- Each queue has its own scheduling algorithm,
  - foreground – RR
  - background – FCFS
A process $P$ can run only if all queues above the queue that contains $P$ are empty.

When a process is running and a process in a higher priority queue comes in, the running process is preempted.
Scheduling must be done between the queues.

- Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.

- Lottery Scheduling – each queue gets a certain amount of CPU time which it can schedule amongst its processes, i.e., 80% to foreground in RR, 20% to background in FCFS
Multilevel Feedback Queue

- **Multilevel queue with feedback scheduling** is similar to multilevel queue; however, it allows processes to move between queues.
  - Aging can be implemented by this way

- **Basic Idea:** Processes with shorter (longer) CPU bursts are given higher (lower) priority.

- If a process uses more (less) CPU time, it is moved to a queue of lower (higher) priority. As a result, I/O-bound (CPU-bound) processes will be in higher (lower) priority queues.
  - Example: if a process didn’t finish (or finish) in its allocated time quantum, it will be demoted to a lower priority queue (or promoted to a higher priority queue).
Example of Multilevel Feedback Queue

Three queues:
- $Q_0$ – RR with time quantum 8 milliseconds
- $Q_1$ – RR with time quantum 16 milliseconds
- $Q_2$ – FCFS

An example of demotion to low-priority queue
- A new job enters $Q_0$ which is served by RR. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to $Q_1$.
- 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 $Q_2$. 
Multilevel Feedback Queues

- quantum = 8
- quantum = 16
- FCFS
Processes in queue $i$ have time quantum $2^i$.

When a process’ behavior changes, it may be placed (i.e., promoted or demoted) into a different queue.

Thus, when an I/O-bound process starts to use more CPU, it may be demoted to a lower queue.
Multilevel Feedback Queue (Cont.)

Multilevel-feedback-queue scheduler defined by the following parameters:

- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling
Solaris 2 Scheduling

<table>
<thead>
<tr>
<th>Global Priority</th>
<th>Scheduling Order</th>
<th>Class-Specific Priorities</th>
<th>Scheduler Classes</th>
<th>Run Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>First</td>
<td>Real Time</td>
<td>Kernel threads of real-time LWPs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>System</td>
<td>Kernel service threads</td>
<td></td>
</tr>
<tr>
<td>Lowest</td>
<td>Last</td>
<td>Interactive and time sharing</td>
<td>Kernel threads of interactive and time-sharing LWPs</td>
<td></td>
</tr>
</tbody>
</table>
### Solaris Dispatch Table

<table>
<thead>
<tr>
<th>Priority</th>
<th>Time Quantum</th>
<th>Time Quantum Expired</th>
<th>Return from Sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>160</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>15</td>
<td>160</td>
<td>5</td>
<td>51</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>10</td>
<td>52</td>
</tr>
<tr>
<td>25</td>
<td>120</td>
<td>15</td>
<td>52</td>
</tr>
<tr>
<td>30</td>
<td>80</td>
<td>20</td>
<td>53</td>
</tr>
<tr>
<td>35</td>
<td>80</td>
<td>25</td>
<td>54</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>30</td>
<td>55</td>
</tr>
<tr>
<td>45</td>
<td>40</td>
<td>35</td>
<td>56</td>
</tr>
<tr>
<td>50</td>
<td>40</td>
<td>40</td>
<td>58</td>
</tr>
<tr>
<td>55</td>
<td>40</td>
<td>45</td>
<td>58</td>
</tr>
<tr>
<td>59</td>
<td>20</td>
<td>49</td>
<td>59</td>
</tr>
</tbody>
</table>
## Windows 2000(XP) Priorities

### Thread priority level

<table>
<thead>
<tr>
<th></th>
<th>real-time</th>
<th>high</th>
<th>above normal</th>
<th>normal</th>
<th>below normal</th>
<th>idle priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>time-critical</td>
<td>31</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>highest</td>
<td>26</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>above normal</td>
<td>25</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>normal</td>
<td>24</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>below normal</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>lowest</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>idle</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Process priority class

https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85)
Linux Scheduling

Two algorithms: time-sharing and real-time

Time-sharing

- Prioritized credit-based – process with most credits is scheduled next
- Credit subtracted when timer interrupt occurs
- When credit = 0, another process chosen
- When all runnable processes have credit = 0, recrediting occurs
  - Based on factors including priority and history
Linux Scheduling (Cont.)

- **Real-time**
  - Posix.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first
  - Soft real-time

- Each CPU has a runqueue made up of 140 priority lists that are serviced in FIFO order. Tasks that are scheduled to execute are added to the end of their respective runqueue's priority list
The Relationship Between Priorities and Time-slice length

<table>
<thead>
<tr>
<th>numeric priority</th>
<th>relative priority</th>
<th>time quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>highest</td>
<td>200 ms</td>
</tr>
<tr>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>lowest</td>
<td>10 ms</td>
</tr>
</tbody>
</table>

The first 100 priority lists of the runqueue are reserved for real-time tasks, and the last 40 are used for user tasks (MAX_RT_PRIO=100 and MAX_PRIO=140).

http://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS/CSCI560_Proj_main/index.html
List of Tasks Indexed According to Priorities

In addition to the CPU's runqueue, which is called the active runqueue, there's also an expired runqueue.

<table>
<thead>
<tr>
<th>active array</th>
<th>expired array</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>task lists</td>
</tr>
<tr>
<td>[0]</td>
<td>[0]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[140]</td>
<td>[140]</td>
</tr>
</tbody>
</table>

When a task on the active runqueue uses all of its time slice, it's moved to the expired runqueue. During the move, its time slice is recalculated (and so is its priority).
If no tasks exist on the active runqueue for a given priority, the pointers for the active and expired runqueues are swapped, thus making the expired priority list the active one.
Scheduler Policy

- Each process has an associated scheduling policy and a static scheduling priority
  - `SCHED_FIFO` - A First-In, First-Out real-time process
  - `SCHED_RR` - A Round Robin real-time process
  - `SCHED_NORMAL` - A conventional, time-shared process (used to be called `SCHED_OTHER`) for normal tasks
  - `SCHED_BATCH` - for "batch" style execution of processes; for computing-intensive tasks
  - `SCHED_IDLE` - for running very low priority background job

[http://linux.die.net/man/2/sched_setscheduler](http://linux.die.net/man/2/sched_setscheduler)
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling
Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP

- Global Scheduling – How the kernel decides which kernel thread to run next
Pthread Scheduling API

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);

    /* set the scheduling algorithm to PROCESS or SYSTEM*/
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);

    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i],&attr,runner,NULL);
```
Pthread Scheduling API

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param) {
    printf("I am a thread\n");
    pthread_exit(0);
}

SCHED_OTHER is the standard Linux time-sharing scheduler that is intended for all processes that do not require the special real-time mechanisms.
http://linux.die.net/man/2/sched_setscheduler
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling
Scheduling Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each scheduling algorithm for that workload.

- Queuing models

- Simulations

- Implementation
Evaluation of CPU Schedulers by Simulation

Actual process execution

- CPU 10
- I/O 213
- CPU 12
- I/O 112
- CPU 2
- I/O 147
- CPU 173

Trace tape

Simulation

- FCFS
- SJF
- RR (Q = 14)

Performance statistics for FCFS
Performance statistics for SJF
Performance statistics for RR (Q = 14)
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling
Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available.

- *Homogeneous processors* within a multiprocessor system.
  - Load sharing

- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.