Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation
Objectives

▪ Describe various CPU scheduling algorithms
▪ Assess CPU scheduling algorithms based on scheduling criteria
▪ Explain the issues related to multiprocessor and multicore scheduling
▪ Describe various real-time scheduling algorithms
▪ Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
▪ Apply modeling and simulations to evaluate CPU scheduling algorithms
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern
Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts
The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them

- Queue may be ordered in various ways

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.

For situations 2 and 3, however, there is a choice.
When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is nonpreemptive.

Otherwise, it is preemptive.

Under Nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.

Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.
Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in race conditions when data are shared among several processes.
- Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.
- This issue will be explored in detail in Chapter 6.
Dispatcher

- **Dispatcher** module gives control of the CPU to the process selected by the CPU scheduler; this involves:
  - Switching context
  - 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
Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced.
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
First- Come, First-Served (FCFS) Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

- Suppose that the processes arrive in the order: $P_1, P_2, P_3$
- The Gantt Chart for the schedule is:

```
  P1  | P2 | P3 |
0    | 24 | 27 |
```

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
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>( P_2 )</th>
<th>( P_3 )</th>
<th>( P_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></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
  - Consider one CPU-bound and many I/O-bound processes
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

- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
    - Could ask the user
    - Estimate

- Preemptive version called shortest.remaining.time.first
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

- SJF scheduling chart

- Average waiting time $= \frac{(3 + 16 + 9 + 0)}{4} = 7$
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

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

1. \( t_n = \text{actual length of } n^{th} \text{ CPU burst} \)
2. \( \tau_{n+1} = \text{predicted value for the next CPU burst} \)
3. \( \alpha, 0 \leq \alpha \leq 1 \)
4. Define:

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

- Commonly, \( \alpha \) set to \( \frac{1}{2} \)
Prediction of the Length of the Next CPU Burst

![Diagram showing prediction of CPU burst length](image)

<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>
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
  \[
  \tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots \\
  + (1 - \alpha) \alpha t_{n-2} + \ldots \\
  + (1 - \alpha)^{n+1} \tau_0
  \]
- Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor
Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

- Preemptive SJF Gantt Chart

- Average waiting time = \([10-1]+(1-1)+(17-2)+(5-3)\)/4 = 26/4 = 6.5
Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum \( q \)), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are \( n \) processes in the ready queue and the time quantum is \( q \), then each process gets \( 1/n \) of the CPU time in chunks of at most \( q \) time units at once. No process waits more than \((n-1)q\) time units.

- Timer interrupts every quantum to schedule next process

- Performance
  - \( q \) large \(\Rightarrow\) FIFO
  - \( q \) small \(\Rightarrow\) \( q \) must be large with respect to context switch, otherwise overhead is too high
Example of RR with Time Quantum = 4

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

- The Gantt chart is:

```
P  P  P  P  P  P  P  P  P  P
  0  4  7 10 14 18 22 26 30
```

- Typically, higher average turnaround than SJF, but **better response**
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds
Time Quantum and Context Switch Time

process time = 10

<table>
<thead>
<tr>
<th>quantum</th>
<th>context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>
Turnaround Time Varies With The Time Quantum

80% of CPU bursts should be shorter than q

<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>
Priority Scheduling

- A *priority* number (integer) is associated with each process

- The **CPU is allocated to the process with the highest priority** (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive

- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

- Problem ≡ **Starvation** – low priority processes may never execute

- Solution ≡ **Aging** – as time progresses increase the priority of the process
Example of Priority Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$P_5$</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

- Priority scheduling Gantt Chart

- Average waiting time = 8.2
Priority Scheduling w/ Round-Robin

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>$P_2$</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>$P_5$</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

- Run the process with the highest priority. Processes with the same priority run round-robin.

- Gantt Chart with time quantum = 2

```
+---+---+---+---+---+---+---+---+---+---+---+---+
| P_4 | P_2 | P_3 | P_2 | P_3 | P_2 | P_3 | P_1 | P_5 | P_1 | P_5 |
+---+---+---+---+---+---+---+---+---+---+---+---+
| 0  |    |    |    |    |    |    |    |    |    |    |    |
|    | 7  | 9  | 11 | 13 | 15 | 16 | 20 | 22 | 24 | 26 | 27 |
```
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!

- priority = 0
  - $T_0$, $T_1$, $T_2$, $T_3$, $T_4$

- priority = 1
  - $T_5$, $T_6$, $T_7$

- priority = 2
  - $T_8$, $T_9$, $T_{10}$, $T_{11}$

- priority = $n$
  - $T_x$, $T_y$, $T_z$
Multilevel Queue

- Prioritization based upon process type

highest priority

- real-time processes

system processes

interactive processes

batch processes

lowest priority
Multilevel Feedback Queue

- A process can move between the various queues.
- **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
- **Aging** can be implemented using multilevel feedback queue
Example of Multilevel Feedback Queue

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

- Scheduling
  - A new process enters queue $Q_0$ which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue $Q_1$
  - At $Q_1$ job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue $Q_2$
Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as process-contention scope (PCS) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system
Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS\n");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM\n");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
Pthread Scheduling API

/* set the scheduling algorithm to PCS or SCS */

pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* create the threads */

for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* 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)
{
    /* do some work ... */

    pthread_exit(0);

}
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing
Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- Figure
Multithreaded Multicore System

- Each core has > 1 hardware threads.
- If one thread has a memory stall, **switch** to another thread!
- Figure
Chip-multithreading (CMT) assigns each core multiple hardware threads. (Intel refers to this as hyperthreading.)

On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.
Multithreaded Multicore System

Two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU

2. How each core decides which hardware thread to run on the physical core.
Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found *pushes* task from overloaded CPU to other CPUs
- **Pull migration** – idle processors *pulls* waiting task from busy processor
Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)
- **Load balancing may affect processor affinity** as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.
If the operating system is **NUMA-aware**, it will assign memory closes to the CPU the thread is running on.
Real-Time CPU Scheduling (skipped 5.6)

- Can present obvious challenges

- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled

- **Hard real-time systems** – task must be serviced by its deadline

This section 5.6 is Skipped, you can skim through to gain more knowledge
Operating System Examples (5.7)

- Linux scheduling
- Windows scheduling
- Solaris scheduling
Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order $O(1)$ scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger $q$
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but **poor response times for interactive processes**
**Completely Fair Scheduler (CFS)**

**Scheduling classes**

- Each has specific priority
- Scheduler picks **highest priority** task in highest scheduling class
- Rather than quantum based on fixed time allotments, based on **proportion of CPU time**
- Two scheduling classes included, others can be added
  1. default
  2. real-time
Quantum calculated based on *nice value* from -20 to +19
- Lower value is higher priority
- Calculates *target latency* – interval of time during which task should run at least once
- Target latency can increase if say number of active tasks increases

CFS scheduler maintains per task *virtual run time* in variable `vruntime`
- Associated with decay factor based on priority of task – lower priority is higher decay rate
- Normal default priority yields virtual run time = actual run time

To decide next task to run, scheduler picks task with lowest virtual run time
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

![Red-black tree diagram]

When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\log N)$ operations (where $N$ is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.
Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139
Linux Scheduling (Cont.)

- Linux supports load balancing but is also NUMA-aware.
- **Scheduling domain** is a set of CPU cores that can be balanced against one another.
- Domains are organized by what they share (i.e., cache memory.) Goal is to keep threads from migrating between domains.
Windows Scheduling

- Windows uses priority-based preemptive scheduling
  - Highest-priority thread runs next
  - **Dispatcher** is scheduler
  - Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - **Real-time threads** can preempt non-real-time
  - 32-level priority scheme
  - **Variable class** is 1-15, **real-time class** is 16-31
  - Priority 0 is **memory-management thread**
  - Queue for each priority
  - If no run-able thread, runs **idle thread**
Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base
Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- **Foreground** window given 3x priority boost
- Windows 7 added **user-mode scheduling (UMS)**
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime (ConcRT)** framework
## Windows Priorities

<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>
Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin
## 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>
Solaris Scheduling

- **Interrupt threads** (highest priority)
- **Realtime (RT) threads**
- **System (SYS) threads**
- **Fair share (FSS) threads**
- **Fixed priority (FX) threads**
- **Timeshare (TS) threads**
- **Interactive (IA) threads** (lowest priority)
Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR
Algorithm Evaluation (5.8)

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>10</td>
</tr>
<tr>
<td>P₂</td>
<td>29</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
<tr>
<td>P₄</td>
<td>7</td>
</tr>
<tr>
<td>P₅</td>
<td>12</td>
</tr>
</tbody>
</table>
Deterministic Evaluation

- For each algorithm, calculate **minimum average waiting time**
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:
    ![FCS Diagram]
  - Non-preemptive SFJ is 13ms:
    ![Non-preemptive SFJ Diagram]
  - RR is 23ms:
    ![RR Diagram]
Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc.
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc.
Little’s Formula

- $n = \text{average queue length}$
- $W = \text{average waiting time in queue}$
- $\lambda = \text{average arrival rate into queue}$
- Little’s law – in steady state, processes leaving queue must equal processes arriving, thus:
  $$n = \lambda \times W$$
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds
Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems
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 for FCFS:
- Performance statistics

Simulation for SJF:
- Performance statistics

Simulation for RR (q = 14):
- Performance statistics
Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again, environments vary
End of Chapter 5