Chapter 5: CPU Scheduling

Zhi Wang
Florida State University
Contents

• Basic concepts
• Scheduling criteria
• Scheduling algorithms
• Thread scheduling
• Multiple-processor scheduling
• Operating systems examples
• Algorithm evaluation
Basic Concepts

- Process execution consists of a cycle of CPU execution and I/O wait
  - **CPU burst** and **I/O burst** alternate
  - CPU burst distribution varies greatly from process to process, and from computer to computer, but follows similar curves
- Maximum CPU utilization obtained with **multiprogramming**
  - CPU scheduler selects another process when current one is in I/O burst
Alternating Sequence of CPU and I/O Bursts
Histogram of CPU-burst Distribution
CPU Scheduler

- CPU scheduler selects from among the processes in ready queue, and allocates the CPU to one of them.

- CPU scheduling decisions may take place when a process:
  - switches from running to waiting state (e.g., wait for I/O)
  - switches from running to ready state (e.g., when an interrupt occurs)
  - switches from waiting to ready (e.g., at completion of I/O)
  - terminates

- Scheduling under condition 1 and 4 only is nonpreemptive
  - once the CPU has been allocated to a process, the process keeps it until terminates or waiting for I/O
  - also called cooperative scheduling

- Preemptive scheduling schedules process also in condition 2 and 3
  - preemptive scheduling needs hardware support such as a timer
  - synchronization primitives are necessary
Kernel Preemption

- Preemption also affects the OS kernel design
  - kernel states will be inconsistent if preempted when updating shared data
  - i.e., kernel is serving a system call when an interrupt happens
- Two solutions:
  - waiting either the system call to complete or I/O block
    - kernel is nonpreemptive (still a preemptive scheduling for processes!)
  - disable kernel preemption when updating shared data
    - recent Linux kernel takes this approach:
      - Linux supports SMP
      - shared data are protected by kernel synchronization
      - disable kernel preemption when in kernel synchronization
      - turned a non-preemptive SMP kernel into a preemptive kernel
Dispatcher

- **Dispatcher** module gives control of the CPU to the process selected by the short-term scheduler
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- **Dispatch latency**: the time it takes for the dispatcher to stop one process and start another running
Scheduling Criteria

- **CPU utilization**: percentage of CPU being busy
- **Throughput**: # of processes that complete execution per time unit
- **Turnaround time**: the time to execute a particular process
  - from the time of *submission* to the time of *completion*
- **Waiting time**: the total time spent waiting in the *ready queue*
- **Response time**: the time it takes from when a request was submitted until the first response is produced
  - the time it takes to *start responding*
Scheduling Algorithm Optimization Criteria

• Generally, maximize CPU utilization and throughput, and minimize turnaround time, waiting time, and response time

• Different systems optimize different values
  • in most cases, optimize average value
  • under some circumstances, optimizes minimum or maximum value
    • e.g., real-time systems
  • for interactive systems, minimize variance in the response time
Scheduling Algorithms

- First-come, first-served scheduling
- Shortest-job-first scheduling
- Priority scheduling
- Round-robin scheduling
- Multilevel queue scheduling
- Multilevel feedback queue scheduling
First-Come, First-Served (FCFS) Scheduling

- Example processes:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

- Suppose that the processes arrive in the order: P₁, P₂, P₃

- the Gantt Chart for the FCFS schedule is:

<table>
<thead>
<tr>
<th></th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

- **Waiting time** for P₁ = 0; P₂ = 24; P₃ = 27, **average waiting time**: \( \frac{0 + 24 + 27}{3} = 17 \)
FCFS Scheduling

- Suppose that the processes arrive in the order: P₂, P₃, P₁
  - the Gantt chart for the FCFS schedule is:

```
0   3   6   30
P₂  P₃  P₁
```

- Waiting time for P₁ = 6; P₂ = 0; P₃ = 3, average waiting time: \( (6 + 0 + 3)/3 = 3 \)

- **Convoy effect**: all other processes waiting until the running CPU-bound process is done
  - considering one CPU-bound process and many I/O-bound processes

- FCFS is **non-preemptive**
Shortest-Job-First Scheduling

• Associate with each process: the length of its next CPU burst
  • the process with the *smallest next CPU burst* is scheduled to run next

• SJF is **provably optimal**: it gives *minimum average waiting* time for a given set of processes
  • moving a short process before a long one decreases the overall waiting time
  • the difficulty is to know the length of the next CPU request
    • long-term scheduler can use the *user-provided processing time estimate*
    • short-term scheduler needs to approximate SFJ scheduling

• SJF can be **preemptive** or **nonpreemptive**
  • preemptive version is called *shortest-remaining-time-first*
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</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 $= \frac{(3 + 16 + 9 + 0)}{4} = 7$
Predicting Length of Next CPU Burst

- We may not know length of next CPU burst for sure, but can predict it
  - assuming it is related to the previous CPU burst

- Predict length of the next CPU bursts w/ **exponential averaging**
  \[ \tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n \]
  - \( t_n \): measured length of \( n^{th} \) CPU burst
  - \( \tau_{n+1} \): predicted length of the next CPU burst
  - \( 0 \leq \alpha \leq 1 \) (normally set to \( \frac{1}{2} \))

- \( \alpha \) determines how the history will affect prediction
  - \( \alpha = 0 \) \( \Rightarrow \) \( \tau_{n+1} = \tau_n \) \( \Rightarrow \) recent history does not count
  - \( \alpha = 1 \) \( \Rightarrow \) \( \tau_{n+1} = \alpha t_n \) \( \Rightarrow \) only the actual last CPU burst counts
  - older history carries less weight in the prediction
    \[ \tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0 \]
Prediction Length of Next CPU Burst

CPU burst \( t_i \): 6 4 6 4 13 13 13 ...

"guess" \( \tau_i \): 10 8 6 6 5 9 11 12 ...

Graph showing the comparison between CPU burst times and estimated lengths.
Shortest-Remaining-Time-First

- SJF can be **preemptive**: reschedule when a process arrives

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>P₄</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 msec
Priority Scheduling

• Priority scheduling selects the ready process with **highest priority**
  • a priority number is associated with each process, smaller integer, higher priority
  • the CPU is allocated to the process with the highest priority
  • SJF is special case of priority scheduling
    • priority is the inverse of predicted next CPU burst time
• Priority scheduling can be **preemptive** or **nonpreemptive**, similar to SJF
• **Starvation** is a problem: **low priority processes may never execute**
  • Solution: **aging** — gradually increase priority of processes that wait for a long time
Example of Priority Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P₄</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P₅</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

• Priority scheduling Gantt Chart

```
<table>
<thead>
<tr>
<th>P₂</th>
<th>P₅</th>
<th>P₁</th>
<th>P₃</th>
<th>P₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>18</td>
</tr>
</tbody>
</table>
```

• Average waiting time = 8.2 msec
Round Robin (RR)

- Round-robin scheduling selects process in a round-robin fashion
  - each process gets a small unit of CPU time (time quantum, \( q \))
    - \( q \) is too large \( \rightarrow \) FIFO, \( q \) is too small \( \rightarrow \) context switch overhead is high
    - a time quantum is generally 10 to 100 milliseconds
  - process used its quantum is preempted and put to tail of the ready queue
    - a timer interrupts every quantum to schedule next process
  - Each process gets \( 1/n \) of the CPU time if there are \( n \) processes
    - no process waits more than \( (n-1)q \) time units
  - Turnaround time is not necessary decrease if we increase the quantum
Example of Round-Robin

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

• The Gantt chart is \((q = 4)\):

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P1</th>
<th>P1</th>
<th>P1</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Wait time for \(P_1\) is 6, \(P_2\) is 4, \(P_3\) is 7, average is 5.66
Time Quantum and Context Switch

- 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 Quantum

<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>
Multilevel Queue

- Multilevel queue scheduling
  - ready queue is partitioned into separate queues
    - e.g., foreground (interactive) and background (batch) processes
  - processes are permanently assigned to a given queue
  - each queue has its own scheduling algorithm
    - e.g., interactive: RR, batch: FCFS
  - Scheduling must be done among the queues
    - fixed priority scheduling
      - possibility of starvation
    - time slice: each queue gets a certain amount of CPU time which it can schedule amongst its processes
      - e.g., 80% to foreground in RR, 20% to background in FCFS
Multilevel Queue Scheduling

- **highest priority**
  - system processes
- **interactive processes**
- **interactive editing processes**
- **batch processes**
- **student processes**

- **lowest priority**
Multilevel Feedback Queue

- Multilevel **feedback** queue scheduling uses multilevel queues
  - a process can **move between the various queues**
    - it tries to infer the **type of the processes** (interactive? batch?)
    - aging can be implemented this way
  - the goal is to give **interactive** and **I/O intensive** process **high priority**
- MLFQ schedulers are defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to assign a process a higher priority
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when it needs service
- MLFQ is the **most general** CPU-scheduling algorithm
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
- A new job enters queue $Q_0$ which is served FCFS
  - when it gains CPU, the job receives 8 milliseconds
  - if it does not finish in 8 milliseconds, the job is moved to queue $Q_1$
- In $Q_1$, the job is again served FCFS and receives 16 milliseconds
  - if it still does not complete, it is preempted and moved to queue $Q_2$
Multilevel Feedback Queues

quantum = 8

quantum = 16

FCFS
Thread Scheduling

- OS kernel schedules kernel threads
  - **system-contention scope**: competition among all threads in system
  - kernel does not aware user threads
- Thread library schedule user threads onto LWPs
  - used in many-to-one and many-to-many threading model
  - **process-contention scope**: scheduling competition within the process
  - PCS usually is based on priority set by the user
  - user thread scheduled to a LWP do not necessarily running on a CPU
    - OS kernel needs to schedule the kernel thread for LWP to a CPU
Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - `pthread_attr_set/getscope` is the API
    - `PTHREAD_SCOPE_PROCESS`: schedules threads using PCS scheduling
    - `PTHREAD_SCOPE_SYSTEM`: schedules threads using SCS scheduling
- Which scope is available can be limited by OS
  - e.g., Linux and Mac OS X only allow `PTHREAD_SCOPE_SYSTEM`
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);
    /* now join on each thread */
    for (i = 0; i < NUM THREADS; i++)
        pthread_join(tid[i], NULL);
}
```
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
  - assume processors are **identical** (homogeneous) in functionality
- Approaches to multiple-processor scheduling
  - **asymmetric multiprocessing**: only one processor makes scheduling decisions, I/O processing, and other activity
  - other processors act as dummy processing units
  - **symmetric multiprocessing** (SMP): each processor is self-scheduling
    - scheduling data structure are shared, needs to be synchronized
    - used by common operating systems
- **Processor affinity**
  - migrating process is expensive to invalidate and repopulate cache
  - solution: **let a process has an affinity for the processor currently running**
    - soft affinity and hard affinity
NUMA and CPU Scheduling
Multicore Processors

- Multicore processor has multiple processor cores on same chip
  - previous multi-processor systems have processors connected through bus
- Multicore processor may complicate scheduling due to memory stall
  - memory stall: when access memory, a process spends a significant amount of time waiting for the data to become available
- Solution: multithreaded CPU core
  - share the execute unit, but duplicate architecture states (e.g., registers) for each CPU thread
    - e.g., Intel Hyper-Threading technology
  - one thread can execute while the other in memory stall
Multithreaded Multicore System
Virtualization and Scheduling

- Virtualization may undo good scheduling efforts in the host or guests
  - Host kernel schedules multiple guests onto CPU(s)
    - in some cases, the host isn’t aware of guests, views them as processes
  - each guest does its own scheduling
    - not knowing it is running on a virtual processor
  - it can result in poor response time and performance
Linux Scheduling

- Linux kernel scheduler runs in constant time (aka. \(O(1)\) scheduler)
- Linux scheduler is preemptive, priority based
  - two priority ranges: real-time range: 0~99, nice value: 100 ~ 140
    - real-time tasks have static priorities
    - priority for other tasks is dynamic +/-5, determined by interactivity
  - these ranges are mapped into global priority; lower values, higher priority
  - higher priority gets longer quantum
  - tasks are run-able as long as there is time left in its time slice (active)
    - if no time left (expired), it is not run-able until all other tasks use their slices
      - priority is recalculated when task expired
## 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>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>lowest</td>
<td>10 ms</td>
</tr>
</tbody>
</table>
Linux Scheduling

- Kernel maintains a **per-CPU** runqueue for all the runnable tasks
  - each processor schedules itself independently
  - each runqueue has two priority arrays: active, expired
  - tasks in these two arrays are indexed by priority
    - always select the first process with the highest priority
  - when active array is empty, two arrays are exchanged
List of Runnable Tasks

<table>
<thead>
<tr>
<th>active array</th>
<th>expired array</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>priority</td>
</tr>
<tr>
<td>[0]</td>
<td>[0]</td>
</tr>
<tr>
<td>[1]</td>
<td>[1]</td>
</tr>
<tr>
<td></td>
<td>[140]</td>
</tr>
<tr>
<td>task lists</td>
<td>task lists</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[140]</td>
</tr>
</tbody>
</table>
End of Chapter 5