CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2022 Lecture 8
Scheduling

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
FAQ

• What exactly is a thread? A block of code?
  – A function when called within a new thread, runs concurrently with other threads.

• Process vs thread
  – A process is isolated from other processes. Processes can run concurrently. Can have multiple threads.
  – A thread is not isolated from other threads belonging to the same process. Runs concurrently with other threads.

• What is a pthread? POSIX compliant implementation of threads.

• Java threads? Most JVMs implement threads with native, OS level threads,

• Examples of threads: Self exercise set 4
• Why use threads:
  – Parallelism if multiple cores/hyper-threading available.
  – Concurrency: quicker responses to some of the things like refreshing output, checking spelling as one types etc.

• Implicit threading: thread creating automated: compiler assisted higher level programming

• Unix signals vs interrupts: Signals are a limited form of inter-process communication. Interrupts are often initiated by hardware. In both cases, some specific routines respond.

• Hyper-threading: Requires additional hardware. Widely used

• Signals example (assume pid = 162): kill -9 162 or kill –s sigkill 162
• Pthread example: pthread_kill(ThreadID, SIGKILL );
Implicit Threading: OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```c
#pragma omp parallel
Create as many threads as there are cores
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
Splits loop task in parallel threads
```

```c
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */
    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }
    /* sequential code */
    return 0;
}
```

Compile using
gcc -fopenmp openmp.c

Self exercise 3, 4 available now.
Signal Handling

• **Signals** are used in UNIX systems to notify a process that a particular event has occurred.

• A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined

• Every signal has **default handler** that kernel runs when handling signal
  – **User-defined signal handler** can override default
  – For single-threaded, signal delivered to process
Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state.

<table>
<thead>
<tr>
<th>Mode</th>
<th>State</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>Disabled</td>
<td>--</td>
</tr>
<tr>
<td>Deferred</td>
<td>Enabled</td>
<td>Deferred</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Enabled</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

A thread’s cancellation type (mode) and state can be set.

If thread has cancellation disabled, cancellation remains pending until thread enables it.

Default type is deferred
- Cancellation only occurs when thread reaches cancellation point
  - I.e. `pthread_testcancel()`
  - Then cleanup handler is invoked

On Linux systems, thread cancellation is handled through signals.
Is complexity always good?

• Is something that is
  – More advanced
  – More complex
Generally better?
“Hyper-threading”: “simultaneous multithreading”:

- Hardware support for multiple threads in the same core (CPU)

**Performance:**

- Performance improvements are very application-dependent
- Higher energy consumption
- Not better than out-of-order execution
- Intel has dropped it in some chips
- May be enabled/disabled using firmware

---

ARM 2006

Intel 2013

Core i7-9700K 2018 8 cores, 8 threads
Forms of Parallelism

– Pipelining: instruction flows though multiple levels
– Multiple issue: Instruction level Parallelism (ILP)
  • Multiple instructions fetched at the same time
  • Static: compiler scheduling of instructions
  • Dynamic: hardware assisted scheduling of operations
    – “Superscalar” processors
    – CPU decides whether to issue 0, 1, 2, … instructions each cycle

– Thread or task level parallelism (TLP)
  • Multiple processes or threads running at the same time
Chapter 5: CPU Scheduling

• Basic Concepts
• Scheduling Criteria
• Scheduling Algorithms
• Thread Scheduling
• Multiple-Processor Scheduling
• Real-Time CPU Scheduling
• Operating Systems Examples
• Algorithm Evaluation
Diagram of Process State

Ready to Running: scheduled by scheduler
Running to Ready: scheduler picks another process, back in ready queue
Running to Waiting (Blocked) : process blocks for input/output
Waiting to Ready: Input available
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

Typical distribution of CPU bursts. Most CPU bursts are just a few ms.
CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU 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
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive. These need to be considered**
  - access to shared data by multiple processes
  - preemption while in kernel mode
  - interrupts occurring during crucial OS activities
• Dispatcher module gives control of the CPU to the process selected by the short-term 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
The Dispatcher (dentist’s office)
Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible: **Maximize**
- **Throughput** – # of processes that complete their execution per time unit: **Maximize**
- **Turnaround time** – time to execute a process from submission to completion: **Minimize**
- **Waiting time** – amount of time a process has been waiting in the ready queue: **Minimize**
- **Response time** – time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment): **Minimize**
Terms for a single process

- Command arrives
- Command begins running
- The first output of command appears
- Command finishes executing

- Time
- Wait time
- Response time
- Execution time
- Turnaround time
We will now examine several major scheduling approaches

- **Decide** which process in the ready queue is allocated the CPU
- Could be preemptive or nonpreemptive
  - preemptive: remove in middle of execution (“forced”)
- **Optimize** *measure* of interest
  - We will use Gantt charts to illustrate *schedules*
  - Bar chart with start and finish times for processes
Non-preemptive vs Preemptive scheduling

• **Non-preemptive:** Process keeps CPU until it relinquishes it when
  – *It terminates*
  – *It switches to the waiting state*
  – Used by initial versions of OSs like Windows 3.x

• **Preemptive scheduling**
  – Pick a process and let it run for a maximum of some fixed time
  – If it is still running at the end of time interval?
    • Suspend it and pick another process to run

• **A clock interrupt** at the end of the time interval to give control back of CPU back to scheduler
Scheduling Algorithms

- First-Come, First-Served (FCFS)
- Shortest-Job-First (SJF)
  - Shortest-remaining-time-first
- Priority Scheduling
- Round Robin (RR) with time quantum
- Multilevel Queue
  - Multilevel Feedback Queue
- “Completely fair”

Comparing Performance

- Average waiting time etc.
First- Come, First-Served (FCFS) Scheduling

- Process requesting CPU first, gets it first
- Managed with a FIFO queue
  - When process **enters** ready queue
    - PCB is tacked to the **tail** of the queue
  - When CPU is **free**
    - It is allocated to process at the **head** of the queue
- Simple to write and understand
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: $P_1$, $P_2$, $P_3$ but almost the same time. The Gantt Chart for the schedule is:

```
<table>
<thead>
<tr>
<th></th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>24</td>
<td>27</td>
</tr>
</tbody>
</table>
```

- Waiting time for $P_1 = $; $P_2 = $; $P_3 = $
- Average waiting time: $( + + )/ = $
- Throughput: $/ = $ per unit time

Pause for students to do the computation
First- Come, First-Served (FCFS) Scheduling

<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: \( P₁, P₂, P₃ \) but almost the same time. The Gantt Chart for the 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></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Waiting time for \( P₁ = 0; P₂ = 24; P₃ = 27 \)
- Average waiting time: \( \frac{0 + 24 + 27}{3} = 17 \)
- Throughput: \( \frac{3}{30} = 0.1 \) per unit time
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>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></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
- But note -Throughput: \( 3/30 = 0.1 \) per unit same

**Convoy effect** - short processes behind a 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

• Reduction in waiting time for short process \textit{GREATER THAN} Increase in waiting time for long process

• SJF is optimal – gives \textit{minimum average waiting time} for a given set of processes
  – The difficulty is knowing the length of the next CPU request
  – Estimate or could ask the user
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>

- All arrive at time 0.
- SJF scheduling chart: Draw it here.

- Average waiting time for $P_1, P_2, P_3, P_4 = \frac{(\quad + \quad + \quad + \quad)}{\quad} =$

Pause for students to do the computation
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>

- All arrive at time 0.
- SJF scheduling chart

- Average waiting time for $P_1, P_2, P_3, P_4 = (3 + 16 + 9 + 0) / 4 = 7$
Determining Length of Next CPU Burst

• Can only estimate the length – should be similar to the recent bursts
  – 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

Blue line: guess
Red line: actual

α = 0.5

Ex:
0.5x6 + 0.5x10 = 8
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
- $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.

If we expand the formula, substituting for $\tau_n$, we get:

$$
\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + ... \\
+ (1 - \alpha)^j \alpha t_{n-j} + ... \\
+ (1 - \alpha)^{n+1} \tau_0
$$

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

Widely used for predicting stock-market etc.
Shortest-remaining-time-first (preemptive SJF)

- Preemptive version called **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 for \( P_1, P_2, P_3, P_4 \)
  
  \[ \frac{[(10-1)+(1-1)+(17-2)+(5-3)]}{4} = \frac{26}{4} = 6.5 \text{ msec} \]
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

MIT had a low priority job waiting from 1967 to 1973 on IBM 7094!
Ex Priority Scheduling  

non-preemptive

<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  (highest)</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>

- $P_1, P_2, P_3, P_4, P_5$ all arrive at time 0.
- Priority scheduling Gantt Chart

- Average waiting time for $P_1, .. P_5$: $(6+0+16+18+1)/5 = 8.2$ msec
Round Robin (RR) with time quantum

- Each process gets a small unit of CPU time (time quantum \(q\)), usually 10-100 milliseconds. After this, the process is preempted, 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 (overhead typically in 0.5\% range)
Example of RR with Time Quantum = 4

<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>

• Arrive a time 0 in order P₁, P₂, P₃: The Gantt chart is:

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

• Waiting times: P₁:10-4 = 6, P₂:4, P₃:7, average 17/3 = 5.66 units
• Typically, higher average turnaround than SJF, but better response
• q should be large compared to context switch time
• q usually 10ms to 100ms, context switch < 10 µsec

Response time: Arrival to beginning of execution
Turnaround time: Arrival to finish of execution
Time Quantum and Context Switch Time

Much smaller quantum compared to burst: many switches
Turnaround Time Varies With The Time Quantum

Rule of thumb: 80% of CPU bursts should be shorter than $q$

Illustration
Consider $q=7$:
Turnaround times for $P_1, P_2, P_3, P_4$:
$6, 9, 10, 17$ av = 10.5
Similarly for $q = 1, ..6$ (verify yourself)

Students: Repeat for $q = 1, ..6$ at home to verify the plot.
Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm, e.g.:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation. Or
  - Time slice – 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
Real-time processes may have the highest priority.
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- 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
  - [Details at ARPACI-DUSSEAU](#)

Inventor FJ Corbató won the Touring award!
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 (no time quantum limit)

- **Scheduling**
  - A new job enters queue $Q_0$ which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue $Q_1$
  - 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 $Q_2$

Upgrading may be based on aging. Periodically processes may be moved to the top level. Variations of the scheme were used in earlier versions of Linux.
Goal: fairness in dividing processor time to tasks (Con Kolivas, Anaesthetist)

- Variable time-slice based on number and priority of the tasks in the queue.
  - Maximum execution time based on waiting processes (Q/n).
  - Fewer processes waiting, they get more time each

- Queue ordered in terms of “virtual run time”
  - execution time on CPU added to value
  - smallest value picked for using CPU
  - small values: tasks have received less time on CPU
  - I/O bound tasks (shorter CPU bursts) will have smaller values

- Balanced (red-black) tree to implement a ready queue;
  - Efficient. O(log n) insert or delete time

- Priorities (niceness) cause different decays of values: higher priority processes get to run for longer time
  - virtual run time is the weighted run-time

Scheduling schemes have continued to evolve with continuing research. A comparison.