Logistics

- Assignment 1 due Tuesday, November 20, 5pm
- Get working on Project 1
- Assignment 2 likely will be published tomorrow
- Read Chapter 7
Outline

- Bit more on threads
- Basic scheduling concepts and criteria
- Scheduling algorithms
- Thread scheduling
- Multiple-Processor scheduling
- Real-Time CPU scheduling
- Algorithm evaluation
Regular UNIX process can be thought of as a special case of a multithreaded process

- A process that contains just one thread
Terms You Might Hear

- **Reentrant code**
  - Code that can be run by multiple threads concurrently

- **Thread-safe libraries**
  - Library code that permits multiple threads to invoke the safe function
  - Mainly concerned with variables that should be private to individual threads
  - Requires moving some global variables to local variables

- **Requirements**
  - Rely only on input data
    - or some thread-specific data
  - Must be careful about locking (later)
Thread Assignment

- How many kernel threads should a process have?
  - In an M:N model there can be significantly more user-level threads (M) than kernel-level threads (N)
  - Kernel threads are the “real” threads that can be allocated to a CPU (core) and run
  - Want to keep enough kernel threads to satisfy the desired level of concurrency in a program and activity in the system

- Suppose that all kernel threads except 1 are blocked
  - What happens if the last kernel thread blocks?
  - Does it matter if there are more user threads “ready”

- In multiprocessing, thread affinity is the notion of assigning a thread to run particular CPU
  - Help to improve the thread’s performance by keeping its execution resources (CPU, cache, memory)
Scheduler Activation

- Wouldn’t it be nice if the kernel told the application and the application had a way to get more kernel threads?
  - *Scheduler activation*
    - at thread block, the kernel tells the application via an *upcall*
    - an *upcall* is a general term for an invocation of application function from the kernel
  - User-level thread scheduler can then get a new user-level thread created

- Way of conveying information between the kernel and the user-level thread scheduler regarding the disposition of:
  - # user-level threads (increase or decrease)
  - User-level thread state (running to waiting, waiting to ready)
Why not threads?

- Threads can interfere with one another
  - Impact of more threads on caches
  - Impact of more threads on TLB
  - Bug in one thread...

- Executing multiple threads may slow them down
  - Impact of single thread vs. switching among threads

- Harder to program a multithreaded program
  - Multitasking hides context switching
  - Multithreading introduces concurrency issues
Resource Allocation

- In a multiprogramming system, we need to share resources among the running processes
  - What are the types of OS resources?
- Which process gets access to which resources?
  - To maximize performance
Resource Types

- **Memory**: Allocate portion of finite resource
  - Physical resources are limited
  - Virtual memory tries to make this appear infinite

- **I/O**: Allocate portion of finite resource and time spent with the resource
  - Store information on disk
  - A time slot to store that information

- **CPU**: Allocate time slot with resource
  - A time slot to run instructions

- We will focus on CPU resource allocation for now
Types of CPU Scheduling

- CPU resource allocation is also known as scheduling
- **Long-term scheduling** (admission): determining whether to add to the set of processes to be executed
- **Medium-term scheduling**: determining whether to add to the number of processes partially or fully in memory (degree of multiprogramming)
- **Short-term scheduling**: determining which process will be executed by the processor
- **I/O scheduling**: determining which process’s pending I/O request will be handled by an available I/O device
CPU Scheduling Views

- Single process view
  - GUI request
    - click on the mouse (*responsiveness*)
  - Scientific computation
    - long-running, but want to complete ASAP (*time to solution*)

- System view (objectives)
  - Get as many tasks done as quickly as possible
    - *throughput* objective
  - Minimize waiting time for processes
    - *response time* objective
  - Get full utilization from the CPU
    - *utilization* objective
Process Scheduling

- Process transition diagram
- Think about the OS perspective
When Scheduling Occurs

- CPU scheduling decisions may take place when:
  1. A process switches from running to waiting state
  2. A process switches from running to ready state
  3. A process switches from waiting to ready
  4. A process terminates

- Process voluntarily gives up (yields) the CPU
  - CPU scheduler kicks in and decides who to go next
  - Process gets put on the ready queue
  - It could get immediately re-scheduled to run
Scheduling Problem

- Choose the ready/running process to run at any time
  - Maximize “performance”

- Model (estimate) “performance” as a function
  - System performance of scheduling each process
    - $f(process) = y$
  - What are some choices for $f(process)$?

- Choose the process with the best $y$
  - Estimating overall performance is intractable
  - Scheduling so all tasks are completed as soon as possible is a NP-complete problem!
  - Adding in pre-emption does not help (as we will see)
Preemption

- Can we reschedule a process that is actively running (i.e., preempt its execution)?
  - If so, we have a preemptive scheduler
  - If not, we have a non-preemptive scheduler
- Suppose a process becomes ready
  - A new process is created or it is no longer waiting
- There is a currently running process
- However, it may be “better” (whatever this means) to schedule the process just put on the ready queue
  - So, we have to preempt the running process
- In what ways could the new process be better?
Basic Concepts – CPU-I/O Bursts

- Maximum CPU utilization is obtained with multiprogramming … Why?
- CPU–I/O burst cycle
  - Process execution consists of cycles of CPU execution and I/O wait
    - run instructions (CPU burst)
    - wait for I/O (I/O burst)
- CPU burst distribution (i.e., how much a CPU is used during a burst) is of main concern … Why?
- Scheduling is aided by knowing the length of these bursts
  - Hmm, do we know this? … more later
Histogram of CPU Burst Times

![Histogram of CPU Burst Times](image-url)
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler

- This involves:
  - Switching context
    - save context of running process
    - loading process context of selected process to run
  - Switching to user mode
  - Jumping to the proper location in the user program to continue execution of the program

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

- **Utilization/efficiency**
  - Keep the CPU busy 100% of the time with useful work

- **Throughput**
  - Maximize the number of jobs processed per hour.

- **Turnaround time (latency)**
  - From the time of submission to the time of completion.

- **Waiting time**
  - Sum of time spent (in ready queue) waiting to be scheduled on the CPU

- **Response time**
  - Time from submission until the first response is produced (mainly for interactive jobs)

- **Fairness**
  - Make sure each process gets a fair share of the CPU
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
Scheduling Algorithms

- Some may seem intuitively better than others
- But a lot has to do with the type of offered workload to the processor
- Best scheduling comes with best context of the tasks to be completed
  - Knowing something about the workload behavior is important
First-Come, First-Served (FCFS)

- Serve the jobs in the order they arrive
- Non-preemptive
  - Process is run until it has to wait or terminates
- Simple and easy to implement
  - When a process is ready, add it to tail of ready queue, and serve the ready queue in FCFS order
- Very fair
  - No process is starved out
  - Service order is immune to job size


FCFS

<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
  - Assume processes arrive at 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: \((0 + 24 + 27)/3 = 17\)
Reducing Waiting Time

- Suppose processes arrive in the order: P₂, P₃, P₁
  - Again, assume that they arrive at the same time
- The Gantt chart for the schedule is:

```
+---+---+---+
| P₂ | P₃ | P₁ |
+---+---+---+
| 0 | 3   | 6   |
| 30|
```

- Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
- Average waiting time: \( (6 + 0 + 3)/3 = 3 \)
- Much better than previous case … Why?
- *Convoy effect*: short processes gets placed behind long process in the scheduling order
**Shortest-Job-First (SJF)**

- Suppose we associate with each process the length of its next CPU burst
  - Hmm, do we know this?
- Use these lengths to schedule the process
  - Process with the shortest next CPU burst time goes first
- Two schemes:
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *Preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, then preempt (known as the *Shortest-Remaining-Time-First (SRTF)*)
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - So we always use it, right?
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 = \((0 + 6 + 3 + 7)/4 = 4\)
## 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 (preemptive)**

- **Average waiting time** = \((9 + 1 + 0 + 2)/4 = 3\)
Determining Length of Next CPU Burst

- Can only estimate the length (do not know for sure)
  - 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

- What to set $\alpha$ to?
  - $\alpha = 0$?
  - $\alpha = 1$?
  - Commonly, $\alpha$ set to $\frac{1}{2}$

- SRTF is the preemptive version

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: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$. 
Exponential Averaging

If we expand the formula, we get:

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

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

![Diagram of CPU burst prediction with actual burst times and guesses]

- **CPU burst ($t_i$)**: 6 4 6 4 13 13 13 ...
- "guess" ($\tau_i$): 10 8 6 6 5 9 11 12 ...
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>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

![Gantt Chart]

- Average waiting time = \[
\frac{(10-1)+(1-1)+(17-2)+5-3)}{4}
\]
  
  = \[
\frac{26}{4}
\]
  
  = 6.5 msec
Scheduling Algorithms

- First-come, First-serve (FCFS)
  - Non-preemptive
  - Does not account for waiting time (or much else)
    - Convoy problem

- Shortest Job First
  - May be preemptive
  - Optimal for minimizing waiting time (how?)

- Lots more… And what do real systems use?
Priority Scheduling

- Each process is given a certain priority “value”
- Always schedule the process with highest priority
  - Preemptive
  - Non-preemptive
- Starvation can be a problem
  - High priority processes may never execute
- Use aging to address starvation
  - Process increase priority as time progresses
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 msec
Priorities

- Note that FCFS and SJF are specialized versions of **Priority Scheduling**
  - Assigning priorities to the processes in a certain way
- What would the priority function be for FCFS?
- What would the priority function be for SJF?
**Round Robin (RR)**

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

- Approach
  - If there are $n$ processes in the ready queue and the time quantum is $q$
  - Then each process gets $l/n$ of the CPU time
  - In chunks of at most $q$ time units at once.
  - No process waits more than $(n-1)q$ time units
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:

- 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 usec
RR Time Quantum

- Round robin is virtually sharing the CPU between the processes giving each process the illusion that it is running in isolation (at $1/n$-th the CPU speed).
- Smaller the time quantum, the more realistic the illusion (note that when time quantum is of the order of job size, it degenerates to FCFS).
- But what is the drawback when time quantum gets smaller?
Time Quantum and Context Switch Time

- Process time = 10
- Quantum: 12
- Context switches: 0

- Process time = 6
- Quantum: 6
- Context switches: 1

- Process time = 4
- Quantum: 1
- Context switches: 9
Turnaround Time Varies With Time Quantum

80% of CPU bursts should be shorter than q
**RR Time Quantum**

- If time quantum size decreases, what happens to the context switches?
- But context switches are not free!
  - Saving/restoring registers
  - Switching address spaces
  - Indirect costs (cache pollution)
Scheduling Desirables

- **SJF**
  - Minimize waiting time
    - requires estimate of CPU bursts

- **Round robin**
  - Share CPU via time quanta
  - if burst turns out to be “too long”

- **Priorities**
  - Some processes are more important
  - Priorities enable composition of “importance” factors

- **No single best approach – now what?**
Round Robin with Priority

- Have a ready queue for each priority level
- Always service the non-null queue at the highest priority level
- Within each queue, you perform round-robin scheduling between those processes
- Problems?
  - With fixed priorities, processes lower in the priority level can get starved out!
  - In general, you employ a mechanism to “age” the priority of processes
Multilevel Queue

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

- Each queue has its own scheduling algorithm: foreground – RR background – FCFS

- Scheduling must be done between 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
Multilevel Queue Scheduling

- Highest priority:
  - System processes
- Interactive processes
- Interactive editing processes
- Batch processes
- Student processes

Lowest 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
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 job enters queue $Q_0$ which 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 $Q_2$
Performance for Schedulers

- Queueing Theory Analysis - uses well-established mathematical models and techniques
- Simulation - create a model of the system and simulate its performance using simulation software
- Empirical Experiments - implement and test the algorithms in a real system
Queuing Theory Analysis

Single-server Queue:
Queuing Theory Analysis

- Inputs:
  - Arrival rate - from a probability distribution (usually Poisson which implies random arrivals)
  - Service time - from a probability distribution (often exponential)
  - Scheduling discipline/algorithm
Queuing Theory Analysis

- Outputs:
  - Items waiting
  - Waiting time
  - Items queued
  - Queuing time
Queuing Theory Analysis

Single-server Queue:

Little’s Formula: $n = \lambda W$  
(n = queue length)
Simulation Analysis

- Discrete-event Simulation
  - Often uses models similar to queueing analysis
  - More detailed or more realistic parameters (e.g. trace driven)
  - Simulates events step by step and gathers statistics rather than using mathematical formulas
Empirical Experiments

- Run experiments on live system

- Properties:
  - Costly and time-consuming
  - Sometimes not possible
  - More realistic
Traditional UNIX Scheduling

- Multilevel feedback queues

- 128 priorities possible (-64 to +63)

- 1 Round Robin queue per priority

- Every scheduling event the scheduler picks the highest priority (lowest number) non-empty queue and runs jobs in round-robin
UNIX Process Scheduling

- Negative numbers reserved for processes waiting in kernel mode (just woken up by interrupt handlers) (why do they have a higher priority?)
- Time quantum = 1/10 sec (empirically found to be the longest quantum that could be used without loss of the desired response for interactive jobs such as editors)
  - Short time quantum means better interactive response
  - Long time quantum means higher overall system throughput since less context switch overhead and less processor cache flush.
- Priority dynamically adjusted to reflect
  - Resource requirement (e.g., blocked awaiting an event)
  - Resource consumption (e.g., CPU time)
**Linux Scheduler**

- Kernel 2.4 and earlier: essentially the same as the traditional UNIX scheduler

- Kernel 2.6: $O(1)$ scheduler
  - Time to select process is constant regardless of system load or the number of processors
  - Separate queue for each priority level
  - CPU affinity (keeps processes on same CPU)

- More recently (kernel 2.6.23 and up): CFS
  - Completely Fair Scheduler (runs $O(\log N)$)
    - uses red-black trees rather than runqueues
Linux Scheduling

- Two algorithms: time-sharing and real-time
  - Time-sharing (still abstracted)
    - Two queues: active and expired
    - In active, until you use your entire time slice (quantum), then expired
      - once in expired, wait for all others to finish (fairness)
    - Priority recalculation -- based on waiting vs. running time
      - from 0-10 milliseconds
      - add waiting time to value, subtract running time
      - adjust the static priority
  - Real-time
    - Soft real-time
    - Posix.1b compliant – two classes
      - FCFS and RR (highest priority process always runs first)
## 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>100</td>
<td></td>
<td>10 ms</td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lowest</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread Scheduling

- When threads are supported, it is threads that are scheduled, not processes
  - Distinction between user-level and kernel-level threads

- 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
Multiple-Processor Scheduling

- CPU scheduling is more complex with multiple CPUs
- Homogeneous processors within a multiprocessor
- Asymmetric multiprocessing
  - Only one processor accesses the system data structures, alleviating the need for data sharing
- Symmetric multiprocessing (SMP)
  - Each processor is self-scheduling, all processes in common ready queue, or each has its own private ready queue
  - Currently, most common
- Processor affinity
  - Process favors the processor on which it is currently running
    - soft affinity
    - hard affinity
Multiple-Processor Scheduling

- Need to keep all CPUs loaded for efficiency
- Load balancing attempts to balance the workload
- Push migration
  - Periodic task checks load on each processor
  - Pushes task from overloaded CPU to other CPUs
- Pull migration
  - Idle processors pulls waiting task from busy processor
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
Multithreaded Multicore System

- **thread**
  - C M C M C M C M

- **thread\_1**
  - C M C M C M C C

- **thread\_0**
  - C M C M C M C C

- **compute cycle**
- **memory stall cycle**
Summary

CPU Scheduling
- Algorithms
- Combination of algorithms
  - Multi-level Feedback Queues

Scheduling Systems
- UNIX
- Linux
Next Class

- Concurrency and synchronization