Outline

• What is scheduling?
  – What are our scheduling goals?
• What resources should we schedule?
• Example scheduling algorithms and their implications
What Is Scheduling?

• An operating system often has choices about what to do next

• In particular:
  – For a resource that can serve one client at a time
  – When there are multiple potential clients
  – Who gets to use the resource next?
  – And for how long?

• Making those decisions is scheduling
OS Scheduling Examples

• What job to run next on an idle core?
  – How long should we let it run?
• In what order to handle a set of block requests for a disk drive?
• If multiple messages are to be sent over the network, in what order should they be sent?
How Do We Decide How To Schedule?

• Generally, we choose goals we wish to achieve
• And design a scheduling algorithm that is likely to achieve those goals
• Different scheduling algorithms try to optimize different quantities
• So changing our scheduling algorithm can drastically change system behavior
The Process Queue

• The OS typically keeps a queue of processes that are ready to run
  – Ordered by whichever one should run next
  – Which depends on the scheduling algorithm used

• When time comes to schedule a new process, grab the first one on the process queue

• Processes that are not ready to run either:
  – Aren’t in that queue
  – Or are at the end
  – Or are ignored by scheduler
Potential Scheduling Goals

• Maximize throughput
  – Get as much work done as possible

• Minimize average waiting time
  – Try to avoid delaying too many for too long

• Ensure some degree of fairness
  – E.g., minimize worst case waiting time

• Meet explicit priority goals
  – Scheduled items tagged with a relative priority

• Real time scheduling
  – Scheduled items tagged with a deadline to be met
Different Kinds of Systems, Different Scheduling Goals

• **Time sharing**
  – Fast response time to interactive programs
  – Each user gets an equal share of the CPU

• **Batch**
  – Maximize total system throughput
  – Delays of individual processes are unimportant

• **Real-time**
  – Critical operations must happen on time
  – Non-critical operations may not happen at all
Preemptive Vs. Non-Preemptive Scheduling

- When we schedule a piece of work, we could let it use the resource until it finishes.
- Or we could use virtualization techniques to interrupt it part way through
  - Allowing other pieces of work to run instead.
- If scheduled work always runs to completion, the scheduler is non-preemptive.
- If the scheduler temporarily halts running jobs to run something else, it’s preemptive.
Pros and Cons of Non-Preemptive Scheduling

+ Low scheduling overhead
+ Tends to produce high throughput
+ Conceptually very simple
  − Poor response time for processes
  − Bugs can cause machine to freeze up
    − If process contains infinite loop, e.g.
  − Not good fairness (by most definitions)
  − May make real time and priority scheduling difficult
Pros and Cons of Pre-emptive Scheduling

+ Can give good response time
+ Can produce very fair usage
+ Works well with real-time and priority scheduling

− More complex
− Requires ability to cleanly halt process and save its state
− May not get good throughput
Scheduling: Policy and Mechanism

• The scheduler will move jobs into and out of a processor (*dispatching*)
  – Requiring various mechanics to do so

• How dispatching is done should not depend on the policy used to decide who to dispatch

• Desirable to separate the choice of who runs (policy) from the dispatching mechanism
  – Also desirable that OS process queue structure not be policy-dependent
Scheduling the CPU

yield (or preemption)

ready queue → dispatcher → context switcher → CPU

resource granted → resource manager → resource request

new process
Scheduling and Performance

• How you schedule important system activities has a major effect on performance

• Performance has different aspects
  – You may not be able to optimize for all of them

• Scheduling performance has very different characteristic under light vs. heavy load

• Important to understand the performance basics regarding scheduling
General Comments on Performance

• Performance goals should be quantitative and measurable
  – If we want “goodness” we must be able to quantify it
  – You cannot optimize what you do not measure

• Metrics ... the way & units in which we measure
  – Choose a characteristic to be measured
    • It must correlate well with goodness/badness of service
  – Find a unit to quantify that characteristic
    • It must a unit that can actually be measured
  – Define a process for measuring the characteristic

• That’s enough for now
  – But actually measuring performance is complex
How Should We Quantify Scheduler Performance?

- Candidate metric: throughput (processes/second)
  - But different processes need different run times
  - Process completion time not controlled by scheduler

- Candidate metric: delay (milliseconds)
  - But specifically what delays should we measure?
    - Time to finish a job (turnaround time)?
    - Time to get some response?
  - Some delays are not the scheduler's fault
    - Time to complete a service request
    - Time to wait for a busy resource
Fairness as a Scheduling Metric

• Maybe we want to make sure all processes are treated fairly
• In what dimension?
  – Fairness in delay? Which one?
  – Fairness in time spent processing?
• Many metrics can be used in Jain’s fairness equation:

\[ J(x_1, x_2, \ldots, x_n) = \frac{\left( \sum_{i=1}^{n} x_i \right)^2}{n \cdot \sum_{i=1}^{n} x_i^2} \]
Other Scheduling Metrics

• Mean time to completion (seconds)
  – For a particular job mix (benchmark)
• Throughput (operations per second)
  – For a particular activity or job mix (benchmark)
• Mean response time (milliseconds)
  – Time spent on the ready queue
• Overall “goodness”
  – Requires a customer specific weighting function
  – Often stated in Service Level Agreements
An Example – Measuring CPU Scheduling

• Process execution can be divided into phases
  – Time spent running
    • The process controls how long it needs to run
  – Time spent waiting for resources or completions
    • Resource managers control how long these take
  – Time spent waiting to be run
    • This time is controlled by the scheduler

• Proposed metric:
  – Time that “ready” processes spend waiting for the CPU
Typical Throughput vs. Load Curve

Throughput vs. offered load graph showing the relationship between throughput and offered load. The graph includes an ideal line and a typical curve, with the maximum possible capacity marked as a peak on the ideal line.
Why Don’t We Achieve Ideal Throughput?

• Scheduling is not free
  – It takes time to dispatch a process (overhead)
  – More dispatches means more overhead (lost time)
  – Less time (per second) is available to run processes

• How to minimize the performance gap
  – Reduce the overhead per dispatch
  – Minimize the number of dispatches (per second)

• This phenomenon is seen in many areas besides process scheduling
Typical Response Time vs. Load Curve

Delay (response time) vs. offered load

- Ideal
- Typical
Why Does Response Time Explode?

• Real systems have finite limits
  – Such as queue size
• When limits exceeded, requests are typically dropped
  – Which is an infinite response time, for them
  – There may be automatic retries (e.g., TCP), but they could be dropped, too
• If load arrives a lot faster than it is serviced, lots of stuff gets dropped
• Unless careful, overheads during heavy load explode
• Effects like receive livelock can also hurt in this case
Graceful Degradation

• When is a system “overloaded”?
  – When it is no longer able to meet service goals

• What can we do when overloaded?
  – Continue service, but with degraded performance
  – Maintain performance by rejecting work
  – Resume normal service when load drops to normal

• What should we not do when overloaded?
  – Allow throughput to drop to zero (i.e., stop doing work)
  – Allow response time to grow without limit
Non-Preemptive Scheduling

• Consider in the context of CPU scheduling
• Scheduled process runs until it yields CPU
• Works well for simple systems
  – Small numbers of processes
  – With natural producer consumer relationships
• Good for maximizing throughput
• Depends on each process to voluntarily yield
  – A piggy process can starve others
  – A buggy process can lock up the entire system
Non-Preemptive Scheduling Algorithms

• First come first served
• Shortest job next
  – We won’t cover this in detail
• Real time schedulers
First Come First Served

• The simplest of all scheduling algorithms
• Run first process on ready queue
  – Until it completes or yields
• Then run next process on queue
  – Until it completes or yields
• Highly variable delays
  – Depends on process implementations
• All processes will eventually be served
First Come First Served Example

<table>
<thead>
<tr>
<th>Dispatch Order</th>
<th>0, 1, 2, 3, 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>Duration</td>
</tr>
<tr>
<td>0</td>
<td>350</td>
</tr>
<tr>
<td>1</td>
<td>125</td>
</tr>
<tr>
<td>2</td>
<td>475</td>
</tr>
<tr>
<td>3</td>
<td>250</td>
</tr>
<tr>
<td>4</td>
<td>75</td>
</tr>
<tr>
<td>Total</td>
<td>1275</td>
</tr>
<tr>
<td>Average wait</td>
<td>595</td>
</tr>
</tbody>
</table>

Note: Average is worse than total/5 because four other processes had to wait for the slow-poke who ran first.
When Would First Come First Served Work Well?

- FCFS scheduling is very simple
- It may deliver very poor response time
- Thus it makes the most sense:
  1. In batch systems, where response time is not important
  2. In embedded (e.g. telephone or set-top box) systems where computations are brief and/or exist in natural producer/consumer relationships
Real Time Schedulers

• For certain systems, some things must happen at particular times
  – E.g., industrial control systems
  – If you don’t rivet the widget before the conveyer belt moves, you have a worthless widget

• These systems must schedule on the basis of real-time deadlines

• Can be either hard or soft
Hard Real Time Schedulers

• The system absolutely must meet its deadlines
• By definition, system fails if a deadline is not met
  – E.g., controlling a nuclear power plant . . .
• How can we ensure no missed deadlines?
• Typically by very, very careful analysis
  – Make sure no possible schedule causes a deadline to be missed
  – By working it out ahead of time
  – Then scheduler rigorously follows deadlines
Ensuring Hard Deadlines

• Must have deep understanding of the code used in each job
  – You know exactly how long it will take
• Vital to avoid non-deterministic timings
  – Even if the non-deterministic mechanism usually speeds things up
  – You’re screwed if it ever slows them down
• Typically means you do things like turn off interrupts
• And scheduler is non-preemptive
• Typically you set up a pre-defined schedule
  – No run time decisions
Soft Real Time Schedulers

• Highly desirable to meet your deadlines
• But some (or any) of them can occasionally be missed
• Goal of scheduler is to avoid missing deadlines
  – With the understanding that you might
• May have different classes of deadlines
  – Some “harder” than others
• Need not require quite as much analysis
Soft Real Time Schedulers and Non-Preemption

• Not as vital that tasks run to completion to meet their deadline
  – Also not as predictable, since you probably did less careful analysis

• In particular, a new task with an earlier deadline might arrive

• If you don’t pre-empt, you might not be able to meet that deadline
What If You Don’t Meet a Deadline?

• Depends on the particular type of system
• Might just drop the job whose deadline you missed
• Might allow system to fall behind
• Might drop some other job in the future
• At any rate, it will be well defined in each particular system
What Algorithms Do You Use For Soft Real Time?

• Most common is Earliest Deadline First
• Each job has a deadline associated with it
  – Based on a common clock
• Keep the job queue sorted by those deadlines
• Whenever one job completes, pick the first one off the queue
• Perhaps prune the queue to remove jobs whose deadlines were missed
• Minimizes total lateness
Example of a Soft Real Time Scheduler

• A video playing device
• Frames arrive
  – From disk or network or wherever
• Ideally, each frame should be rendered “on time”
  – To achieve highest user-perceived quality
• If you can’t render a frame on time, might be better to skip it entirely
  – Rather than fall further behind
Preemptive Scheduling

• Again in the context of CPU scheduling
• A thread or process is chosen to run
• It runs until either it yields
• Or the OS decides to interrupt it
• At which point some other process/thread runs
• Typically, the interrupted process/thread is restarted later
Implications of Forcing Preemption

• A process can be forced to yield at any time
  – If a higher priority process becomes ready
    • Perhaps as a result of an I/O completion interrupt
  – If running process’s priority is lowered
    • Perhaps as a result of having run for too long
• Interrupted process might not be in a “clean” state
  – Which could complicate saving and restoring its state
• Enables enforced “fair share” scheduling
• Introduces gratuitous context switches
  – Not required by the dynamics of processes
• Creates potential resource sharing problems
Implementing Preemption

• Need a way to get control away from process
  – E.g., process makes a sys call, or clock interrupt
• Consult scheduler before returning to process
  – Has any ready process had its priority raised?
  – Has any process been awakened?
  – Has current process had its priority lowered?
• Scheduler finds highest priority ready process
  – If current process, return as usual
  – If not, yield on behalf of current process and switch to higher priority process
Clock Interrupts

- Modern processors contain a clock
- A peripheral device
  - With limited powers
- Can generate an interrupt at a fixed time interval
- Which temporarily halts any running process
- Good way to ensure that runaway process doesn’t keep control forever
- Key technology for preemptive scheduling
Round Robin Scheduling Algorithm

• Goal - fair share scheduling
  – All processes offered equal shares of CPU
  – All processes experience similar queue delays

• All processes are assigned a nominal time slice
  – Usually the same sized slice for all

• Each process is scheduled in turn
  – Runs until it blocks, or its time slice expires
  – Then put at the end of the process queue

• Then the next process is run

• Eventually, each process reaches front of queue
Properties of Round Robin Scheduling

• All processes get relatively quick chance to do some computation
  – At the cost of not finishing any process as quickly
  – A big win for interactive processes

• Far more context switches
  – Which can be expensive

• Runaway processes do relatively little harm
  – Only take $1/n^{th}$ of the overall cycles
Round Robin and I/O Interrupts

• Processes get halted by round robin scheduling if their time slice expires
• If they block for I/O (or anything else) on their own, the scheduler doesn’t halt them
• Thus, some percentage of the time round robin acts no differently than FIFO
  – When I/O occurs in a process and it blocks
**Round Robin Example**

Assume a 50 msec time slice (or *quantum*)

<table>
<thead>
<tr>
<th>Process</th>
<th>Length</th>
<th>1st</th>
<th>2nd</th>
<th>3d</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
<th>8th</th>
<th>Finish</th>
<th>Switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>350</td>
<td>0</td>
<td>250</td>
<td>475</td>
<td>650</td>
<td>800</td>
<td>950</td>
<td>1050</td>
<td></td>
<td>1100</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>125</td>
<td>50</td>
<td>300</td>
<td>525</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>525</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>475</td>
<td>100</td>
<td>350</td>
<td>550</td>
<td>700</td>
<td>850</td>
<td>1000</td>
<td>1100</td>
<td>1250</td>
<td>1275</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>250</td>
<td>150</td>
<td>400</td>
<td>600</td>
<td>750</td>
<td>900</td>
<td></td>
<td></td>
<td></td>
<td>900</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>75</td>
<td>200</td>
<td>450</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>475</td>
<td>2</td>
</tr>
</tbody>
</table>

Average waiting time: 100 msec

First process completed: 475 msec
Comparing Example to Non-Preemptive Examples

• Context switches: 27 vs. 5 (for both FIFO and SJF)
  – Clearly more expensive
• First job completed: 475 msec vs.
  – 75 (shortest job first)
  – 350 (FIFO)
  – Clearly takes longer to complete some process
• Average waiting time: 100 msec vs.
  – 350 (shortest job first)
  – 595 (FIFO)
  – For first opportunity to compute
  – Clearly more responsive
Choosing a Time Slice

• Performance of a preemptive scheduler depends heavily on how long time slice is
• Long time slices avoid too many context switches
  – Which waste cycles
  – So better throughput and utilization
• Short time slices provide better response time to processes
• How to balance?
Costs of a Context Switch

• Entering the OS
  – Taking interrupt, saving registers, calling scheduler

• Cycles to choose who to run
  – The scheduler/dispatcher does work to choose

• Moving OS context to the new process
  – Switch stack, non-resident process description

• Switching process address spaces
  – Map-out old process, map-in new process

• Losing instruction and data caches
  – Greatly slowing down the next hundred instructions
Characterizing Costs of a Time Slice Choice

• What % of CPU use does a process get?
• Depends on workload
  – More processes in queue = fewer slices/second
• CPU share = time_slice * slices/second
  – 2% = 20ms/sec = 2ms/slice * 10 slices/sec
  – 2% = 20ms/sec = 5ms/slice * 4 slices/sec
• Natural rescheduling interval
  – When a typical process blocks for resources or I/O
  – Ideally, fair-share would be based on this period
  – Time-slice ends only if process runs too long
Multi-Level Feedback Queue (MLFQ) Scheduling

- One time slice length may not fit all processes
- Create multiple ready queues
  - Short quantum (foreground) tasks that finish quickly
    - Short but frequent time slices, optimize response time
  - Long quantum (background) tasks that run longer
    - Longer but infrequent time slices, minimize overhead
  - Different queues may get different shares of the CPU
- Finds balance between good response time and good turnaround time
How Do I Know What Queue To Put New Process Into?

• If it’s in the wrong queue, its scheduling discipline causes it problems
• Start all processes in short quantum queue
  – Move downwards if too many time-slice ends
  – Move back upwards if too few time slice ends
  – Processes dynamically find the right queue
• If you also have real time tasks, you know what belongs there
  – Start them in real time queue and don’t move them
Multiple Queue Scheduling

- **real time queue**
  - \( \#\text{yield} = \infty \)
  - \( \text{ts}_{\text{max}} = \infty \)
  - \( \#\text{tse} = \infty \)

- **short quantum queue**
  - \( \#\text{yield} = \infty \)
  - \( \text{ts}_{\text{max}} = 500\text{us} \)
  - \( \#\text{tse} = 10 \)

- **medium quantum queue**
  - \( \#\text{yield} = 10 \)
  - \( \text{ts}_{\text{max}} = 2\text{ms} \)
  - \( \#\text{tse} = 50 \)

- **long quantum queue**
  - \( \#\text{yield} = 20 \)
  - \( \text{ts}_{\text{max}} = 5\text{ms} \)
  - \( \#\text{tse} = \infty \)
Priority Scheduling Algorithm

• Sometimes processes aren’t all equally important
• We might want to preferentially run the more important processes first
• How would our scheduling algorithm work then?
• Assign each job a priority number
• Run according to priority number
Priority and Preemption

• If non-preemptive, priority scheduling is just about ordering processes
• Much like shortest job first, but ordered by priority instead
• But what if scheduling is preemptive?
• In that case, when new process is created, it might preempt running process
  – If its priority is higher
Priority Scheduling Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Priority</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>350</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>125</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>475</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>250</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>75</td>
</tr>
</tbody>
</table>

Process 3’s priority is lower than running process

Process 4’s priority is higher than running process

Process 4 completes

So we go back to process 2
Problems With Priority Scheduling

- Possible starvation
- Can a low priority process ever run?
- If not, is that really the effect we wanted?
- May make more sense to adjust priorities
  - Processes that have run for a long time have priority temporarily lowered
  - Processes that have not been able to run have priority temporarily raised
Hard Priorities Vs. Soft Priorities

• What does a priority mean?
• That the higher priority has absolute precedence over the lower?
  – Hard priorities
  – That’s what the example showed
• That the higher priority should get a larger share of the resource than the lower?
  – Soft priorities
Priority Scheduling in Linux

• Each process in Linux has a priority
  – Called a *nice* value
  – A soft priority describing share of CPU that a process should get

• Commands can be run to change process priorities

• Anyone can request lower priority for his processes

• Only privileged user can request higher
Priority Scheduling in Windows

• 32 different priority levels
  – Half for regular tasks, half for soft real time
  – Real time scheduling requires special privileges
  – Using a multi-queue approach

• Users can choose from 5 of these priority levels

• Kernel adjusts priorities based on process behavior
  – Goal of improving responsiveness