Review: Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
  - Deadlock $\Rightarrow$ Starvation, but not other way around

- Four conditions for deadlocks
  - Mutual exclusion
    - Only one thread at a time can use a resource
  - Hold and wait
    - Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - Resources are released only voluntarily by the threads
  - Circular wait
    - There exists a set \{T₁, ..., Tₙ\} of threads with a cyclic waiting pattern

Review: Resource Allocation Graph Examples

- Recall:
  - request edge - directed edge $T₁ \rightarrow Rₗ$
  - assignment edge - directed edge $R_j \rightarrow T_i$

- Simple Resource Allocation Graph
- Allocation Graph With Deadlock
- Allocation Graph With Cycle, but No Deadlock

Review: Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for selectively preempting resources and/or terminating tasks

- Ensure that system will never enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that might lead to deadlock

- Ignore the problem and pretend that deadlocks never occur in the system
  - used by most operating systems, including UNIX
Review: Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)

Review: Banker's Algorithm for Preventing Deadlock

- Monitor every request to see if it has the potential to lead to deadlock
  - Every thread must state a "maximum" expected allocation ahead of time
  - Keeps system in a "SAFE" state ⇒ there always exists a sequence \{T_1, T_2, ... T_n\} with T_1 able to request all its remaining resources and finish, then T_2 able to request all its remaining resources and finish, etc..
  - Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting [Maxnode]-[Allocnode] for [Requestnode]
      Grant request if result is deadlock free (conservative!)
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Goals for Today

- Tips for Programming in a Project Team
- Scheduling Policy goals
- Policy Options
- Implementation Considerations

Tips for Programming in a Project Team

- Big projects require more than one person (or long, long, long time)
  - Big OS: thousands of person-years!
- It's very hard to make software project teams work correctly
  - Doesn't seem to be as true of big construction projects
    » Consider building the Empire state building: staging iron production thousands of miles away
    » Or the Hoover dam: built towns to hold workers
- Ok to miss deadlines?
  » We make it free (slip days)
  » In reality they're very expensive: time-to-market is one of the most important things!

"You just have to get your synchronization right!"
**Big Projects**

- **What is a big project?**
  - Time/work estimation is hard
  - Programmers are eternal optimistics (it will only take two days)
    - This is why we bug you about starting the project early
    - Had a grad student who used to say he just needed “10 minutes” to fix something. Two hours later...
- **Can a project be efficiently partitioned?**
  - Partitionable task decreases in time as you add people
    - But, if you require communication:
      - Time reaches a minimum bound
      - With complex interactions, time increases!
  - Mythical person-month problem:
    - You estimate how long a project will take
    - Starts to fall behind, so you add more people
    - Project takes even more time!

**Techniques for Partitioning Tasks**

- **Functional**
  - Person A implements threads, Person B implements semaphores, Person C implements locks...
  - Problem: Lots of communication across APIs
    - If B changes the API, A may need to make changes
    - Story: Large airline company spent $200 million on a new scheduling and booking system. Two teams “working together.” After two years, went to merge software. Failed! Interfaces had changed (documented, but no one noticed). Result: would cost another $200 million to fix.
- **Task**
  - Person A designs, Person B writes code, Person C tests
    - May be difficult to find right balance, but can focus on each person’s strengths (Theory vs systems hacker)
  - Since Debugging is hard, Microsoft has two testers for each programmer
  - Most CS162 project teams are functional, but people have had success with task-based divisions

**Communication**

- More people mean more communication
  - Changes have to be propagated to more people
  - Think about person writing code for most fundamental component of system: everyone depends on them!
- Miscommunication is common
  - “Index starts at 0? I thought you said 1!”
- Who makes decisions?
  - Individual decisions are fast but trouble
  - Group decisions take time
  - Centralized decisions require a big picture view (someone who can be the “system architect”)
- Often designating someone as the system architect can be a good thing
  - Better not be clueless
  - Better have good people skills
  - Better let other people do work

**Coordination**

- More people ⇒ no one can make all meetings!
  - They miss decisions and associated discussion
  - Example from earlier class: one person missed meetings and did something group had rejected
  - Why do we limit groups to 5 people?
    - You would never be able to schedule meetings
  - Why do we require 3 or 4 people minimum?
    - You need to experience groups to get ready for real world
- People have different work styles
  - Some people work in the morning, some at night
  - How do you decide when to meet or work together?
- What about project slippage?
  - It will happen, guaranteed!
  - Another example: final project in CS152, everyone busy but not talking. One person way behind. No one knew until very end – too late!
- Hard to add people to existing group
  - Members have already figured out how to work together
How to Make it Work?

• People are human. Get over it.
  - People will make mistakes, miss meetings, miss deadlines, etc. You need to live with it and adapt
  - It is better to anticipate problems than clean up afterwards.
• Document, document, document
  - Why Document?
    » Expose decisions and communicate to others
    » Easier to spot mistakes early
    » Easier to estimate progress
  - What to document?
    » Everything (but don’t overwhelm people or no one will read)
  - Standardize!
    » One programming format: variable naming conventions, tab indents, etc.
    » Comments (Requires, effects, modifies)—javadoc?

Suggested Documents for You to Maintain

• Project objectives: goals, constraints, and priorities
• Specifications: the manual plus performance specs
  - This should be the first document generated and the last one finished
• Meeting notes
  - Document all decisions
  - You can often cut & paste for the design documents
• Schedule: What is your anticipated timing?
  - This document is critical!
• Organizational Chart
  - Who is responsible for what task?

Use Software Tools

• Source revision control software (CVS)
  - Easy to go back and see history
  - Figure out where and why a bug got introduced
  - Communicates changes to everyone (use CVS’s features)
• Use automated testing tools
  - Write scripts for non-interactive software
  - Use “expect” for interactive software
  - Microsoft rebuild the XP kernel every night with the day’s changes. Everyone is running/testing the latest software
• Use E-mail and instant messaging consistently to leave a history trail

Test Continuously

• Integration tests all the time, not at 11pm on due date!
  - Write dummy stubs with simple functionality
    » Let’s people test continuously, but more work
  - Schedule periodic integration tests
    » Get everyone in the same room, check out code, build, and test.
    » Don’t wait until it is too late!
• Testing types:
  - Unit tests: check each module in isolation (use JUnit?)
  - Daemons: subject code to exceptional cases
  - Random testing: Subject code to random timing changes
• Test early, test later, test again
  - Tendency is to test once and forget; what if something changes in some other part of the code?
Administrivia

- Midterm I coming up in < two weeks:
  - Wednesday, 10/12, 5:30 - 8:30, Here
  - Should be 2 hour exam with extra time
  - Closed book, one page of hand-written notes (both sides)
- No class on day of Midterm
  - I will post extra office hours for people who have questions about the material (or life, whatever)
- Midterm Topics
  - Topics: Everything up to that Monday, 10/10
  - History, Concurrency, Multithreading, Synchronization, Protection/Address Spaces

CPU Scheduling

- Earlier, we talked about the life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
- Question: How is the OS to decide which of several tasks to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- **Scheduling**: deciding which threads are given access to resources from moment to moment

Scheduling Assumptions

- CPU scheduling big area of research in early 70s
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is “fair” about fairness among users or programs?
    - If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system
  - USER1 USER2 USER3 USER1 USER2
  - Time

Assumption: CPU Bursts

- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Weighted toward small bursts
Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - Time to echo a keystroke in editor
    - Time to compile a program
  - Realtime Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - Minimize overhead (for example, context-switching)
    - Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - Better average response time by making system less fair

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - In early systems, FCFS meant one program scheduled until done (including I/O)
    - Now, means keep CPU until thread blocks
- Example:

<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 for the schedule is:

  - Waiting time for P₁ = 0; P₂ = 24; P₃ = 27
  - Average waiting time: \((24 + 27 + 30)/3 = 27\)
  - Average Completion time: \((24 + 27 + 30)/3 = 27\)
- Convoy effect: short process behind long process

FCFS Scheduling (Cont.)

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

<table>
<thead>
<tr>
<th>P₂</th>
<th>P₃</th>
<th>P₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>
  - Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
  - Average waiting time: \((6 + 0 + 3)/3 = 3\)
  - Average Completion time: \((3 + 6 + 30)/3 = 13\)
- In second case:
  - Average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - Safeway: Getting milk, always stuck behind cart full of small items. Upside: get to read about space aliens!

Round Robin (RR)

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don’t care who is behind you, on the other hand...
- Round Robin Scheme
  - Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue
  - \(n\) processes in ready queue and time quantum is \(q\) \(⇒\)
    - Each process gets \(1/n\) of the CPU time
    - In chunks of at most \(q\) time units
    - No process waits more than \((n-1)q\) time units
- Performance
  - \(q\) large \(⇒\) FCFS
  - \(q\) small \(⇒\) Interleaved (really small \(⇒\) hyperthreading?)
  - \(q\) must be large with respect to context switch, otherwise overhead is too high (all overhead)
Example of RR with Time Quantum = 20

- Example:
  - Process
  - Burst Time
  - \( P_1 \) 53
  - \( P_2 \) 8
  - \( P_3 \) 68
  - \( P_4 \) 24

- The Gantt chart is:

```
  0 20 28 48 68 88 108 125 145 153
P_1 P_2 P_3 P_4 P_1 P_3 P_4 P_1 P_3
```

- Waiting time for \( P_1 \): \( (68-20) + (112-88) = 72 \)
- Waiting time for \( P_2 \): \( (20-0) = 20 \)
- Waiting time for \( P_3 \): \( (28-0) + (88-48) + (125-108) = 85 \)
- Waiting time for \( P_4 \): \( (48-0) + (108-68) = 88 \)

- Average waiting time: \( (72 + 20 + 85 + 88)/4 = 66\frac{1}{4} \)
- Average completion time: \( (125 + 28 + 153 + 112)/4 = 104\frac{1}{2} \)

- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  - RR scheduler quantum of 1s
  - All jobs start at the same time

- Completion Times:
<table>
<thead>
<tr>
<th>Job #</th>
<th>FIFO</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>991</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>992</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>900</td>
<td>999</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
  - Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
- Total time for RR longer even for zero-cost switch!

Round-Robin Discussion

- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (\( \infty \))?
    - Get back FIFO
  - What if time slice too small?
    - Throughput suffers!

- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, need to balance short-job performance and long-job throughput:
    - Typical time slice today is between 10ms - 100ms
    - Typical context-switching overhead is 0.1ms - 1ms
    - Roughly 1% overhead due to context-switching

Earlier Example with Different Time Quantum

<table>
<thead>
<tr>
<th>Best FCFS:</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
<th>( P_4 )</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantum</td>
<td>0</td>
<td>8</td>
<td>32</td>
<td>85</td>
<td>153</td>
</tr>
<tr>
<td>( Q = 1 )</td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td>31\frac{1}{2}</td>
</tr>
<tr>
<td>( Q = 5 )</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
<td>62</td>
</tr>
<tr>
<td>( Q = 8 )</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61\frac{1}{2}</td>
</tr>
<tr>
<td>( Q = 10 )</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57\frac{1}{2}</td>
</tr>
<tr>
<td>( Q = 20 )</td>
<td>72</td>
<td>10</td>
<td>85</td>
<td>88</td>
<td>66\frac{1}{2}</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83\frac{1}{2}</td>
</tr>
</tbody>
</table>

| Worst FCFS | 85 | 8 | 153 | 32 | 69\frac{1}{2} |
| \( Q = 1 \) | 137 | 30 | 153 | 81 | 100\frac{1}{2} |
| \( Q = 5 \) | 135 | 28 | 153 | 82 | 99\frac{1}{2} |
| \( Q = 8 \) | 133 | 16 | 153 | 80 | 95\frac{1}{2} |
| \( Q = 10 \) | 135 | 18 | 153 | 92 | 99\frac{1}{2} |
| \( Q = 20 \) | 125 | 28 | 153 | 112 | 104\frac{1}{2} |
| Worst FCFS | 121 | 153 | 68 | 145 | 121\frac{1}{2} |
What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do.
  - Sometimes called "Shortest Time to Completion First" (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU.
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program.
  - Idea is to get short jobs out of the system.
  - Big effect on short jobs, only small effect on long ones.
  - Result is better average response time.

Discussion

- SJF/SRTF are the best you can do at minimizing average response time.
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive).
  - Since SRTF is always at least as good as SJF, focus on SRTF.
- Comparison of SRTF with FCFS and RR:
  - What if all jobs the same length?
    - SRTF becomes the same as FCFS (i.e., FCFS is best can do if all jobs the same length).
  - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones.

Example to illustrate benefits of SRTF

- Three jobs:
  - A, B: both CPU bound, run for week.
  - C: I/O bound, loop 1ms CPU, 9ms disk I/O.
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU.
- With FIFO:
  - Once A or B get in, keep CPU for two weeks.
- What about RR or SRTF?
  - Easier to see with a timeline.

SRTF Example continued:

- Disk Utilization: 9/201 ~ 4.5%.
- Disk Utilization: Approx 90%.
- Disk Utilization: 90%.
SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
    - Some systems ask the user
      » when you submit a job, have to say how long it will take
      » To stop cheating, system kills job if takes too long
    - But: Even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can’t really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
    - Optimal, so can’t do any better

SRTF Pros & Cons
- Optimal (average response time) (+)
- Hard to predict future (-)
- Unfair (-)

Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    » If program was I/O bound in past, likely in future
    » If computer behavior were random, wouldn’t help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    Let $t_{n-1}, t_{n-2}, t_{n-3}, \ldots$ be previous CPU burst lengths.
    Estimate next burst $t_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - Function $f$ could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging $t_n = \alpha t_{n-1} + (1-\alpha)t_{n-1}$ with $(0 < \alpha \leq 1)$

Multi-Level Feedback Scheduling

- Another method for exploiting past behavior
  - First used in CTSS
  - Multiple queues, each with different priority
    » Higher priority queues often considered “foreground” tasks
  - Each queue has its own scheduling algorithm
    » e.g. foreground – RR, background – FCFS
    » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Adjust each job’s priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn’t expire, push up one level (or to top)

Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    » serve all from highest priority, then next priority, etc.
  - Time slice:
    » each queue gets a certain amount of CPU time
    » e.g., 70% to highest, 20% next, 10% lowest
- Countermeasure: user action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job’s priority high
  - Of course, if everyone did this, wouldn’t work!
- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority the competitors.
  » Put in printf’s, ran much faster!
What about Fairness?

- Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
  - long running jobs may never get CPU
  - In Multics, shut down machine, found 10-year-old job
- Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting avg response time!

How to implement fairness?

- Could give each queue some fraction of the CPU
  - What if one long-running job and 100 short-running ones?
  - Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
- Could increase priority of jobs that don't get service
  - What is done in UNIX
  - This is ad hoc—what rate should you increase priorities?
  - And, as system gets overloaded, no job gets CPU time, so everyone increases in priority; Interactive jobs suffer

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example

- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

<table>
<thead>
<tr>
<th># short jobs/# long jobs</th>
<th>% of CPU each short job gets</th>
<th>% of CPU each long job gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1</td>
<td>91%</td>
<td>9%</td>
</tr>
<tr>
<td>0/2</td>
<td>N/A</td>
<td>50%</td>
</tr>
<tr>
<td>2/0</td>
<td>50%</td>
<td>N/A</td>
</tr>
<tr>
<td>10/1</td>
<td>9.9%</td>
<td>0.99%</td>
</tr>
<tr>
<td>1/10</td>
<td>50%</td>
<td>5%</td>
</tr>
</tbody>
</table>

- What if too many short jobs to give reasonable response time?
  - In UNIX, if load average is 100, hard to make progress
  - One approach: log some user out

How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
  - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data. Most flexible/general.
A Final Word on Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren’t enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    > Assuming you’re paying for worse response time in reduced productivity, customer angst, etc...
    > Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization ⇒ 100%
- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit “knee” of curve

Summary

- Suggestions for dealing with Project Partners
  - Start Early, Meet Often
  - Develop Good Organizational Plan, Document Everything, Use the right tools
  - Develop a Comprehensive Testing Plan
  - (Oh, and add 2 years to every deadline!)
- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- FCFS Scheduling: Run threads to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones
- Round-Robin Scheduling: Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: Poor when jobs are same length

Summary (2)

- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks ⇒ more tokens)
  - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness