Recall: 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
    » Real-time 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

Recall: Example of RR with Time Quantum = 20

- Example:

  Process | Burst Time
  --- | ---
  P₁ | 53
  P₂ | 8
  P₃ | 68
  P₄ | 24

  - The Gantt chart is:

  ![Gantt chart](chart.png)

  - Waiting time for
    
    - P₁: 
      
      \( \text{Waiting time for } P_1 = (68-20)+(112-88) = 72 \)
    
    - P₂: 
      
      \( \text{Waiting time for } P_2 = (20-0) = 20 \)
    
    - P₃: 
      
      \( \text{Waiting time for } P_3 = (28-0)+(88-48)+(125-108) = 85 \)
    
    - P₄: 
      
      \( \text{Waiting time for } P_4 = (48-0)+(108-68) = 88 \)

  - 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>999</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!

Earlier Example with Different Time Quantum

<table>
<thead>
<tr>
<th>Priority</th>
<th>Quantum</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></td>
<td>Best FCFS</td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td>31½</td>
</tr>
<tr>
<td>Q = 1</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>Q = 5</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61½</td>
<td></td>
</tr>
<tr>
<td>Q = 8</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57¼</td>
<td></td>
</tr>
<tr>
<td>Q = 10</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61½</td>
<td></td>
</tr>
<tr>
<td>Q = 20</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66½</td>
<td></td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83½</td>
<td></td>
</tr>
</tbody>
</table>

|          | Best FCFS | 85  | 32  | 153 | 32  | 89½    |
| Q = 1    | 137      | 30  | 153 | 81  | 100½ |
| Q = 5    | 135      | 28  | 153 | 82  | 99½  |
| Q = 8    | 133      | 16  | 153 | 80  | 95½  |
| Q = 10   | 135      | 18  | 153 | 92  | 99½  |
| Q = 20   | 125      | 28  | 153 | 112 | 104½ |
| Worst FCFS | 121    | 153 | 68  | 145 | 121% |

Handling Differences in Importance: Strict Priority Scheduling

- Execution Plan
  - Always execute highest-priority runnable jobs to completion
  - Each queue can be processed in RR with some time-quantum
- Problems:
  - Starvation: Lower priority jobs don’t get to run because higher priority jobs
  - Deadlock: Priority Inversion
    » Not strictly a problem with priority scheduling, but happens when low priority task has lock needed by high-priority task
    » Usually involves third, intermediate priority task that keeps running even though high-priority task should be running
- How to fix problems?
  - Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc…

Scheduling Fairness

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    » long running jobs may never get CPU
    » Urban legend: In Multics, shut down machine, found 10-year-old job ⇒ Ok, probably not…
  - 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!
Scheduling Fairness

- 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 some variants of 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 (Cont.)

- 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 jobs gets</th>
<th>% of CPU each long jobs 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?
  » 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
**How to Handle Simultaneous Mix of Diff Types of Apps?**

- Consider mix of interactive and high throughput apps:
  - How to best schedule them?
  - How to recognize one from the other?
    - Do you trust app to say that it is "interactive"?
  - Should you schedule the set of apps identically on servers, workstations, pads, and cellphones?
- For instance, is Burst Time (observed) useful to decide which application gets CPU time?
  - Short Bursts $\Rightarrow$ Interactivity $\Rightarrow$ High Priority?
- Assumptions encoded into many schedulers:
  - Apps that sleep a lot and have short bursts must be interactive apps – they should get high priority
  - Apps that compute a lot should get low(er?) priority, since they won’t notice intermittent bursts from interactive apps
- Hard to characterize apps:
  - What about apps that sleep for a long time, but then compute for a long time? 
  - Or, what about apps that must run under all circumstances (say periodically)

**Administrivia**

- Midterm 1 is Thursday (2/27)!
  - Topics: All material up to Today (emphasis on scheduling in today’s lecture)
  - You get 1 sheet of *handwritten* notes, both sides
- Review Session: Tonight after class
  - Right here from 6:30-8:00pm (after class)
- Watch Piazza for Midterm room assignments:
  - There are 5 rooms!
  - DSP midterms are separate and you should have received email from us. Let us know immediately if that is not the case!

**What if we Knew the Future?**

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has 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 to whole program or current CPU burst
  - 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
  - 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: 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 FCFS:
  - 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: ~90% but lots of wakeups!
- 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: hard to predict job’s runtime even for non-malicious users

- 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:
    \[ \tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots) \]
  - Function \( f \) could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging
    \[ \tau_n = \alpha \tau_{n-1} + (1-\alpha) t_{n-1} \]
    with \( 0 < \alpha < 1 \)
Multi-Level Feedback Scheduling

- Another method for exploiting past behavior (first use 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

Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities
  - 40 for “user tasks” (set by “nice”), 100 for “Realtime/Kernel”
  - Lower priority value \(\Rightarrow\) higher priority (for nice values)
  - Highest priority value \(\Rightarrow\) Lower priority (for realtime values)
  - All algorithms \(O(1)\)
    » Timeslices/priorities/interactivity credits all computed when job finishes time slice
    » 140-bit bit mask indicates presence or absence of job at given priority level
- Two separate priority queues: “active” and “expired”
  - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped
- Timeslice depends on priority – linearly mapped onto timeslice range
  - Like a multi-level queue (one queue per priority) with different timeslice at each level
  - Execution split into “Timeslice Granularity” chunks – round robin through priority

Countermeasure: user action that can foil intent of the OS designers
- For multi-level 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!
O(1) Scheduler Continued

- Heuristics
  - User-task priority adjusted ±5 based on heuristics
    » \( p \rightarrow \text{sleep}_{\text{avg}} = \text{sleep}_{\text{time}} - \text{run}_{\text{time}} \)
    » Higher \( \text{sleep}_{\text{avg}} \) \( \Rightarrow \) more I/O bound the task, more reward (and vice versa)
  - Interactive Credit
    » Earned when a task sleeps for a “long” time
    » Spend when a task runs for a “long” time
    » IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior
  - However, “interactive tasks” get special dispensation
    » To try to maintain interactivity
    » Placed back into active queue, unless some other task has been starved for too long…

- Real-Time Tasks
  - Always preempt non-RT tasks
  - No dynamic adjustment of priorities
  - Scheduling schemes:
    » \text{SCHED\_FIFO}: preempts other tasks, no timeslice limit
    » \text{SCHED\_RR}: preempts normal tasks, RR scheduling amongst tasks of same priority

Linux Completely Fair Scheduler (CFS)

- First appeared in 2.6.23, modified in 2.6.24
- “CFS doesn’t track sleeping time and doesn’t use heuristics to identify interactive tasks—it just makes sure every process gets a fair share of CPU within a set amount of time given the number of runnable processes on the CPU.”
- Inspired by Networking “Fair Queueing”
  - Each process given their fair share of resources
  - Models an “ideal multitasking processor” in which \( N \) processes execute simultaneously as if they truly got 1/\( N \) of the processor
    » Tries to give each process an equal fraction of the processor
  - Priorities reflected by weights such that increasing a task’s priority by 1 always gives the same fractional increase in CPU time – regardless of current priority

CFS (Continued)

- Idea: track amount of “virtual time” received by each process when it is executing
  - Take real execution time, scale by weighting factor
    » higher priority \( \Rightarrow \) real time divided by larger weight
    » Actually – multiply by sum of all weights/current weight
  - Keep virtual time advancing at same rate

- Targeted latency (\( T_L \)): period of time after which all processes get to run at least a little
  - Each process runs with quantum (\( W_p / \sum W_i \times T_L \))
  - Never smaller than “minimum granularity”

- Use of Red-Black tree to hold all runnable processes as sorted on vruntime variable
  - \( O(\log n) \) time to perform insertions/deletions
    » Cash the item at far left (item with earliest vruntime)
  - When ready to schedule, grab version with smallest vruntime (which will be item at the far left).

CFS Examples

- Suppose Targeted latency = 20ms, Minimum Granularity = 1ms
- Two CPU bound tasks with same priorities
  - Both switch with 10ms
- Nice values scale weights exponentially: \( \text{Weight}=1024/(1.25)^{\text{nice}} \)
- Two CPU bound tasks separated by nice value of 5
  - One task gets 5ms, another gets 15ms
- 40 tasks: each gets 1ms (no longer totally fair)
- One CPU bound task, one interactive task same priority
  - While interactive task sleeps, CPU bound task runs and increments vruntime
  - When interactive task wakes up, runs immediately, since it is behind on vruntime
- Group scheduling facilities (2.6.24)
  - Can give fair fractions to groups (like a user or other mechanism for grouping processes)
  - So, two users, one starts 1 process, other starts 40, each will get 50% of CPU
Real-Time Scheduling (RTS)

- Efficiency is important but **predictability** is essential:
  - We need to predict with confidence worst case response times for systems
  - In RTS, performance guarantees are:
    - Task- and/or class centric and often ensured a priori
  - In conventional systems, performance is:
    - System/throughput oriented with post-processing (... wait and see ...)
  - Real-time is about enforcing predictability, and does not equal fast computing!!!
- **Hard Real-Time**
  - Attempt to meet all deadlines
    - EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)
- **Soft Real-Time**
  - Attempt to meet deadlines with high probability
    - Minimize miss ratio / maximize completion ratio (firm real-time)
    - Important for multimedia applications
    - CBS (Constant Bandwidth Server)

Example: Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Tasks have deadlines (D) and known computation times (C)
- Example Setup:

Example: Round-Robin Scheduling Doesn’t Work

- Tasks periodic with period P and computation C in each period: \((P_i, C_i)\) for each task \(i\)
- Preemptive priority-based dynamic scheduling:
  - Each task is assigned a (current) priority based on how close the absolute deadline is (i.e. \(D_i^{t+1} = D_i^t + P_i\) for each task!)
  - The scheduler always schedules the active task with the closest absolute deadline

Earliest Deadline First (EDF)

- Scheduleable when \(\sum_{i=1}^{n} \left( \frac{C_i}{P_i} \right) \leq 1\)
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
    » Perhaps 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

Starvation vs Deadlock

• Starvation: thread waits indefinitely
  – Example, low-priority thread waiting for resources constantly in use by high-priority threads

• Deadlock: circular waiting for resources
  – Thread A owns Res 1 and is waiting for Res 2
  – Thread B owns Res 2 and is waiting for Res 1

  Deadlock → Starvation but not vice versa
  – Starvation can end (but doesn’t have to)
  – Deadlock can’t end without external intervention

Conditions for Deadlock

• Deadlock not always deterministic – Example 2 mutexes:
  Thread_A
  x.P();
  y.P();
  y.P();
  y.V();
  x.V();
  x.V();

  Thread_B
  y.P();
  x.P();
  x.V();

  = Deadlock won’t always happen with this code
    » Have to have exactly the right timing (“wrong’ timing?)
    » So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant…

• Deadlocks occur with multiple resources
  – Means you can’t decompose the problem
  – Can’t solve deadlock for each resource independently

• Example: System with 2 disk drives and two threads
  – Each thread needs 2 disk drives to function
  – Each thread gets one disk and waits for another one

Bridge Crossing Example

• Each segment of road can be viewed as a resource
  – Car must own the segment under them
  – Must acquire segment that they are moving into

• For bridge: must acquire both halves
  – Traffic only in one direction at a time
  – Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
  – If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
    – Several cars may have to be backed up
  – Starvation is possible
    – East-going traffic really fast ⇒ no one goes west
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)

Dining Lawyers Problem
- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer will grab any one they can
  - Need two chopsticks to eat
- What if all grab at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
  - Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards

Four requirements for Deadlock
- 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 thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads
    » \( T_1 \) is waiting for a resource that is held by \( T_2 \)
    » \( T_2 \) is waiting for a resource that is held by \( T_3 \)
    » \( \ldots \)
    » \( T_n \) is waiting for a resource that is held by \( T_1 \)

Resource-Allocation Graph
- System Model
  - A set of Threads \( T_1, T_2, \ldots, T_n \)
  - Resource types \( R_1, R_2, \ldots, R_m \)
    » CPU cycles, memory space, I/O devices
  - Each resource type \( R_i \) has \( W_i \) instances
  - Each thread utilizes a resource as follows:
    » Request() / Use() / Release()
- Resource-Allocation Graph
  - \( V \) is partitioned into two types:
    » \( T = \{T_1, T_2, \ldots, T_n\} \), the set threads in the system.
    » \( R = \{R_1, R_2, \ldots, R_m\} \), the set of resource types in system
  - request edge – directed edge \( T_1 \rightarrow R_j \)
  - assignment edge – directed edge \( R_j \rightarrow T_i \)
Resource-Allocation Graph Examples

- Model:
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

Simple Resource Allocation Graph

Allocation Graph With Deadlock

Allocation Graph With Cycle, but No Deadlock

Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preemting 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
  - This is not say that this is a "method", rather intentional ignorance?

Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let $[X]$ represent an m-ary vector of non-negative integers (quantities of resources of each type):
    - $[\text{FreeResources}]$: Current free resources each type
    - $[\text{Request}_X]$: Current requests from thread $X$
    - $[\text{Alloc}_X]$: Current resources held by thread $X$
  - See if tasks can eventually terminate on their own
    - $[\text{Avail}] = [\text{FreeResources}]$
    - Add all nodes to UNFINISHED
    - do {
        - done = true
        - foreach node in UNFINISHED {
            - if $([\text{Request}_\text{node}] <= [\text{Avail}])$
              - remove node from UNFINISHED
              - $[\text{Avail}] = [\text{Avail}] + [\text{Alloc}_\text{node}]$
              - done = false
          }
        } until(done)
  - Nodes left in UNFINISHED ⇒ deadlocked

What to do when detect deadlock?

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Shoot a dining lawyer
  - But, not always possible – killing a thread holding a mutex leaves world inconsistent

- Preempt resources without killing off thread
  - Take away resources from thread temporarily
  - Doesn’t always fit with semantics of computation

- Roll back actions of deadlocked threads
  - Hit the rewind button on TiVo, pretend last few minutes never happened
  - For bridge example, make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again

- Many operating systems use other options
Techniques for Preventing Deadlock

• Infinite resources
  – Include enough resources so that no one ever runs out of resources. Doesn’t have to be infinite, just large
  – Give illusion of infinite resources (e.g. virtual memory)
  – Examples:
    » Bay bridge with 12,000 lanes. Never wait!
    » Infinite disk space (not realistic yet?)

• No Sharing of resources (totally independent threads)
  – Not very realistic

• Don’t allow waiting
  – How the phone company avoids deadlock
    » Call to your Mom in Toledo, works its way through the phone lines, but if blocked get busy signal.
  – Technique used in Ethernet/some multiprocessor nets
    » Everyone speaks at once. On collision, back off and retry
  – Inefficient, since have to keep retrying
    » Consider: driving to San Francisco; when hit traffic jam, suddenly you’re transported back home and told to retry!

Techniques for Preventing Deadlock (cont’d)

• Make all threads request everything they’ll need at the beginning.
  – Problem: Predicting future is hard, tend to over-estimate resources
  – Example:
    » If need 2 chopsticks, request both at same time
    » Don’t leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time

• Force all threads to request resources in a particular order preventing any cyclic use of resources
  – Thus, preventing deadlock
    » Make tasks request disk, then memory, then…
    » Keep from deadlock on freeways around SF by requiring everyone to go clockwise

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)

Banker’s Algorithm for Preventing Deadlock

• Toward right idea:
  – State maximum (max) resource needs in advance
  – Allow particular thread to proceed if:
    (available resources - #requested) ≥ max
    remaining that might be needed by any thread

• Banker’s algorithm (less conservative):
  – Allocate resources dynamically
    » 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
      ([Max node] - [Alloc node] <= [Avail]) for ([Request node] <= [Avail])
      Grant request if result is deadlock free (conservative!)
**Banker’s Algorithm for Preventing Deadlock**

- Toward right idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:
    \[(\text{available resources} - \#\text{requested}) \geq \text{max remaining that might be needed by any thread}\]

- Banker’s algorithm (less conservative):
  - Allocate resources dynamically
    - 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 \(([\text{Max node}] - [\text{Alloc node}] \leq [\text{Avail}])\) for \(([\text{Request node}] \leq [\text{Avail}])\)
    - Grant request if result is deadlock free (conservative!)

---

**Banker’s Algorithm Example**

- Banker’s algorithm with dining lawyers
  - “Safe” (won’t cause deadlock) if when try to grab chopstick either:
    - Not last chopstick
    - Is last chopstick but someone will have two afterwards
  - What if k-handed lawyers? Don’t allow if:
    - It’s the last one, no one would have k
    - It’s 2nd to last, and no one would have k-1
    - It’s 3rd to last, and no one would have k-2
    - ...
Summary (1 of 3)

• **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

• **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 and scheduling algorithms
  – Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

Summary (2 of 3)

• **Lottery Scheduling:**
  – Give each thread a priority-dependent number of tokens (short tasks⇒more tokens)

• **Linux CFS Scheduler: Fair fraction of CPU**
  – Approximates a “ideal” multitasking processor

• **Realtime Schedulers such as EDF**
  – Guaranteed behavior by meeting deadlines
  – Realtime tasks defined by tuple of compute time and period
  – Schedulability test: is it possible to meet deadlines with proposed set of processes?

Summary (3 of 3)

• **Starvation vs. Deadlock**
  – Starvation: thread waits indefinitely
  – Deadlock: circular waiting for resources

• **Four conditions for deadlocks**
  – Mutual exclusion
  – Hold and wait
  – No preemption
  – Circular wait

• **Techniques for addressing Deadlock**
  – Allow system to enter deadlock and then recover
  – Ensure that system will never enter a deadlock
  – Ignore the problem and pretend that deadlocks never occur