Scheduling 3: Deadlock

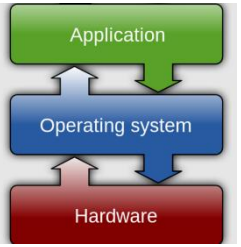
Lecture 11

Hartmut Kaiser

https://teaching.hkaiser.org/spring2026/csc4103/

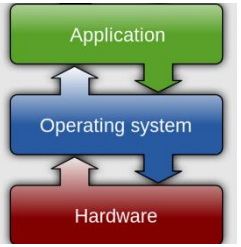
Recall: Linux O(1) Scheduler

- MLFQ-Like Scheduler with 140 Priority Levels
  - 40 for user tasks, 100 "realtime" tasks
  - All algorithms O(1) complexity – low overhead
    - Timeslices/priorities/interactivity credits all computed when job finishes time slice
- Active and expired queues at each priority
  - Once active is empty, swap them (pointers)
  - Round Robin within each queue (varying quanta)
- Timeslice depends on priority – linearly mapped onto timeslice range



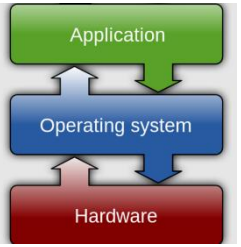
Recall: Multi-Core Scheduling

- Algorithmically, not a huge difference from single-core scheduling
- Implementation-wise, helpful to have per-core scheduling data structures
  - Cache coherence
- Affinity scheduling: once a thread is scheduled on a CPU, OS tries to reschedule it on the same CPU
  - Cache reuse



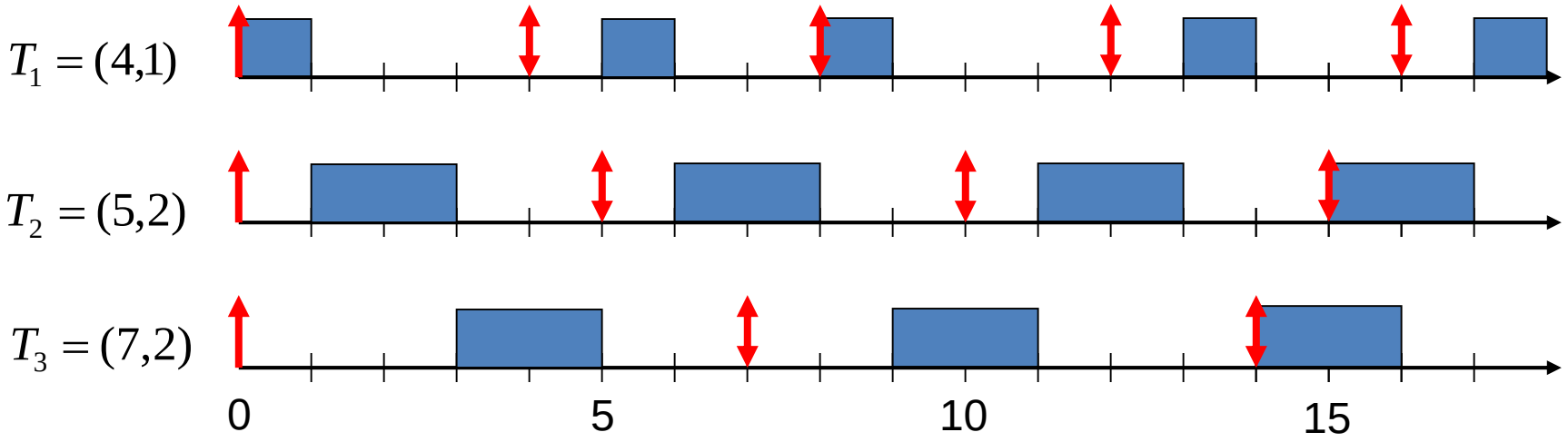
Recall: Real-Time Scheduling

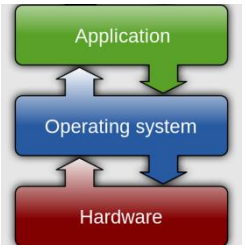
- Goal: Guaranteed Performance
  - Meet deadlines even if it means being unfair or slow
  - Limit how bad the worst case is
- Hard real-time:
  - Meet all deadlines (if possible)
  - Ideally: determine in advance if this is possible
  - Earliest Deadline First (EDF), Least Laxity First (LLF)
- Soft real-time
  - Attempt to meet deadlines with high probability
  - Constant Bandwidth Server (CBS)



Recall: Earliest Deadline First (EDF)

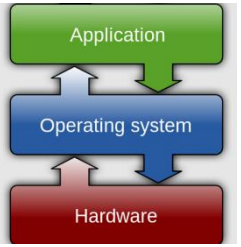
- Priority scheduling with preemption
- Prefer task with earliest deadline
  - Priority proportional to time until deadline
- Example with periodic tasks:





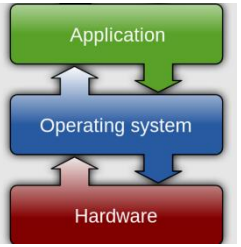
Recall: Ensuring Progress

- Starvation: thread fails to make progress for an indefinite period of time
- Causes of starvation:
  - **Scheduling policy never runs a particular thread on the CPU**
  - Threads wait for each other or are spinning in a way that will never be resolved
- Let's explore what sorts of problems we might fall into and how to avoid them…



Recall: Schedulers Prone to Starvation

- What kinds of schedulers are prone to starvation?
- Of the scheduling policies we've studied, which are prone to starvation? And can we fix them?
- How might we design scheduling policies that avoid starvation entirely?
  - Arguably more relevant now than when CPU scheduling was first developed…



Recall: Priority Inversion

- Where high priority task is blocked waiting on low priority task
- Low priority one must run for high priority to make progress
- Medium priority task can starve a high priority one

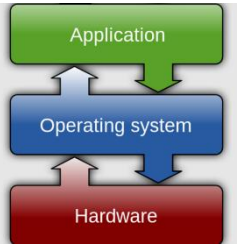
- When else might priority lead to starvation or "live lock"?

High Priority

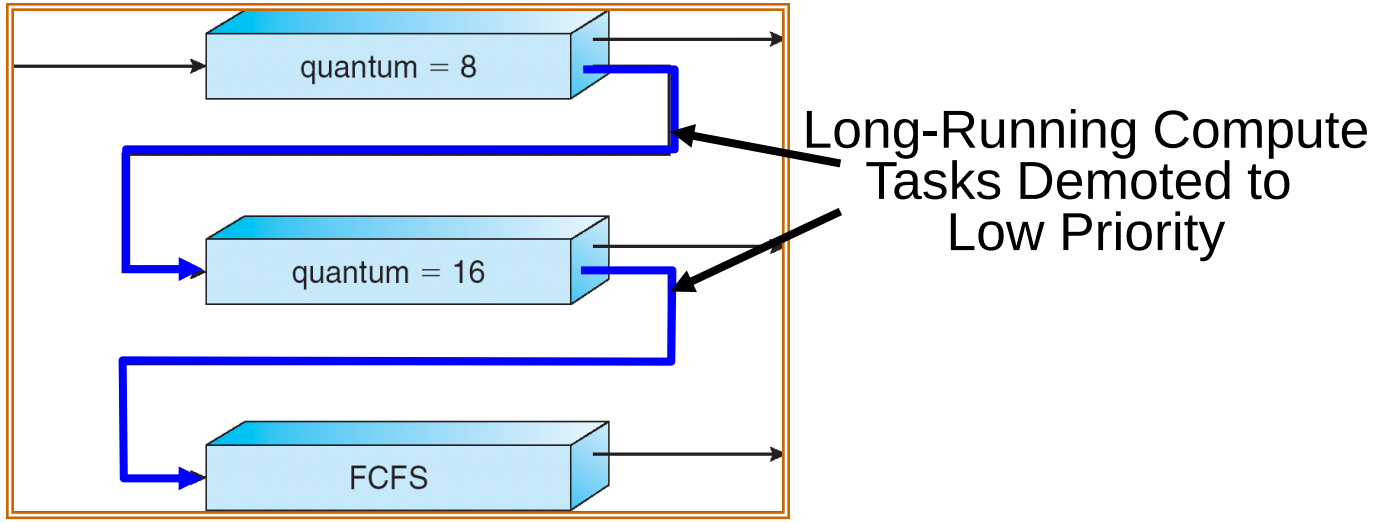
```
while (try_lock) {
…
}
```

Low Priority

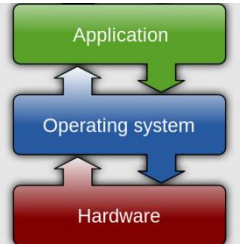
```
lock.acquire(…)
…
lock.release(…)
```



Recall: Are SRTF and MLFQ Prone to Starvation?

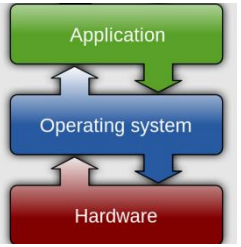


- In SRTF, long jobs are starved in favor of short ones
  - Same fundamental problem as priority scheduling
- MLFQ is an approximation of SRTF, so it suffers from the same problem



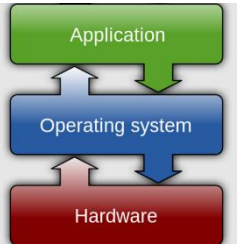
Recall: Evaluating Schedulers

- Response Time (ideally low)
  - What user sees: from keypress to character on screen
  - Or completion time for non-interactive
- Throughput (ideally high)
  - Total operations (jobs) per second
  - Overhead (e.g. context switching), artificial blocks
- Fairness
  - Fraction of resources provided to each
  - May conflict with best avg. throughput, resp. time



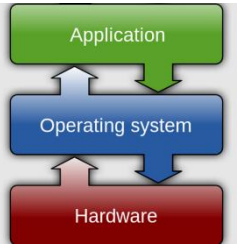
Recall: Changing Landscape of Scheduling

- Priority-based scheduling rooted in "time-sharing"
  - Allocating precious, limited resources across a diverse workload
    - CPU bound, vs interactive, vs I/O bound
- 80's brought about personal computers, workstations, and servers on networks
  - Different machines of different types for different purposes
  - Shift to fairness and avoiding extremes (starvation)
- 90's emergence of the web, rise of internet-based services, the data-center-is-the-computer
  - Server consolidation, massive clustered services, huge flashcrowds
  - It's about predictability, 95th percentile performance guarantees

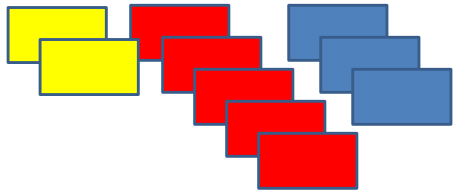


Recall: Proportional-Share Scheduling

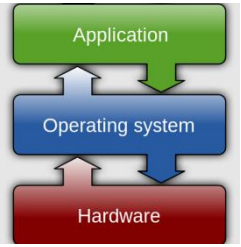
- The policies we've studied so far:
  - **Always prefer to give the CPU to a prioritized job**
  - Non-prioritized jobs may never get to run
- Instead, we can share the CPU proportionally
  - Give each job a share of the CPU according to its priority
  - Low-priority jobs get to run less often
  - But all jobs can at least make progress (no starvation)



Recall: Lottery Scheduling

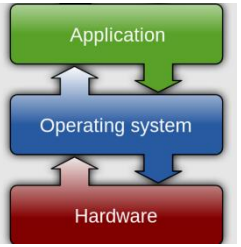


- Given a set of jobs (the mix), provide each with a share of a resource
  - e.g., 50% of the CPU for Job A, 30% for Job B, and 20% for Job C
- Idea: Give out tickets according to the proportion each should receive,
- Every quantum (tick): draw one at random, schedule that job (thread) to run

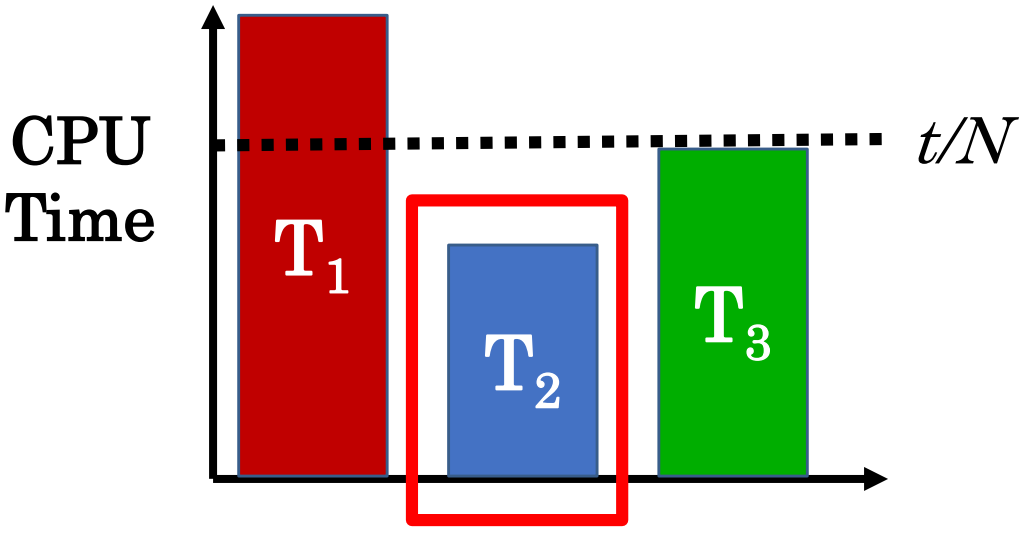


Stride Scheduling

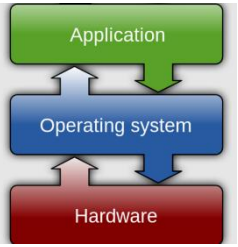
- Achieve proportional share scheduling without resorting to randomness, and overcome the "law of small numbers" problem.
- "Stride" of each job is $\frac{big\#W}{N_i}$
  - The larger your share of tickets, the smaller your stride
  - Ex: W = 10,000, A=100 tickets, B=50, C=250
  - A stride: 100, B: 200, C: 40
- Each job as a "pass" counter
- Scheduler: pick job with lowest pass, runs it, add its stride to its pass
- Low-stride jobs (lots of tickets) run more often
  - Job with twice the tickets gets to run twice as often
- Some messiness of counter wrap-around, new jobs, …



Recall: Linux Completely Fair Scheduler (CFS)

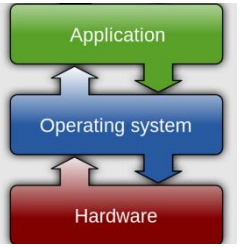


- Instead: track CPU time given to a thread so far
- Scheduling Decision:
  - "Repair" illusion of complete fairness
  - Choose thread with minimum CPU time
- Reset CPU time if thread goes to sleep and wakes back up



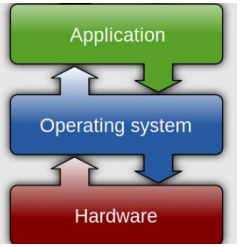
Recall: Linux CFS, Responsiveness

- In addition to fairness, we want low response time
- Constraint 1: Target Latency
  - Period of time over which every process gets service
  - Quanta = Target_Latency / n
- Target Latency: 20 ms, 4 Processes
  - Each process gets 5ms time slice
- Target Latency: 20 ms, 200 Processes
  - Each process gets 0.1ms time slice (!!!)
  - Recall Round-Robin: large context switching overhead if slice gets to small

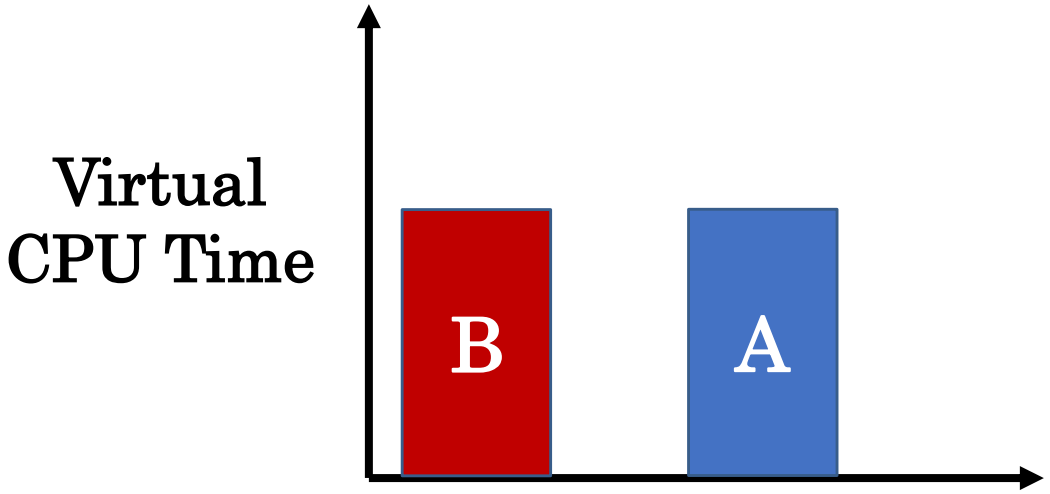


Recall: Linux CFS, Throughput

- Goal: Throughput
  - Avoid excessive overhead
- Constraint 2: Minimum Granularity
  - Minimum length of any time slice
- Target Latency 20 ms, Minimum Granularity 1 ms, 200 processes
  - Each process gets 1 ms time slice

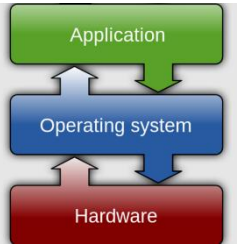


Recall: Linux CFS: Proportional Shares



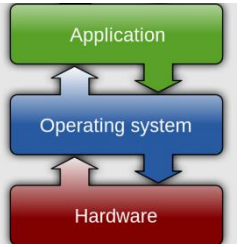
Scheduler's Decisions are based on Virtual CPU Time

- Track a thread's virtual runtime rather than its true physical runtime
- Higher weight: Virtual runtime increases more slowly
- Lower weight: Virtual runtime increases more quickly



Recall: Choosing the Right Scheduler

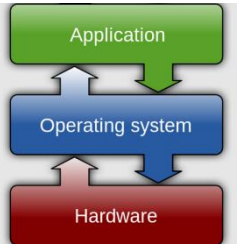
If You Care About:	Then Choose:
CPU Throughput	FCFS
Average Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness (Wait Time to Get CPU)	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority



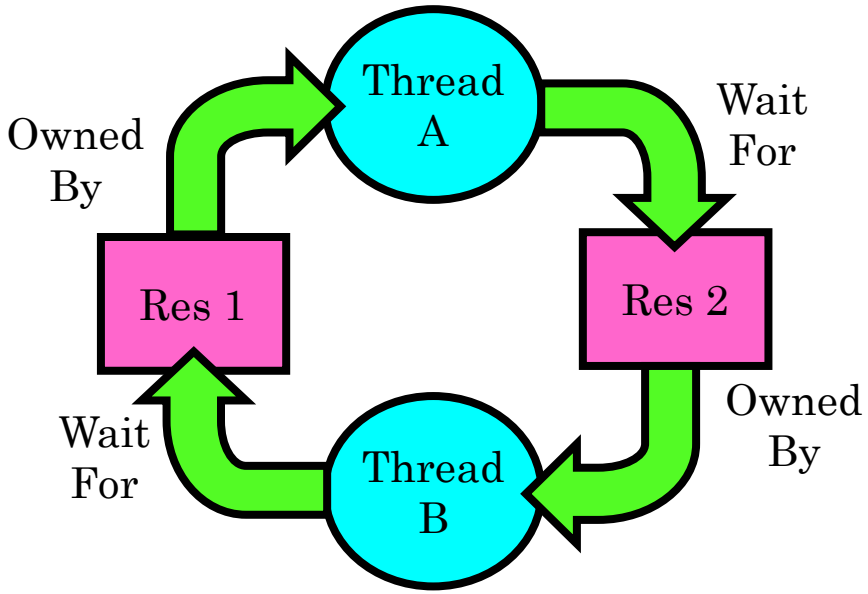
Deadlocks

Ensuring Progress

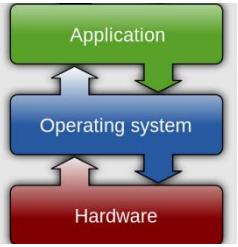
- Starvation: thread fails to make progress for an indefinite period of time
- Causes of starvation:
  - Scheduling policy never runs a particular thread on the CPU
  - Threads wait for each other or are spinning in a way that will never be resolved
- Let's explore what sorts of problems we might fall into and how to avoid them…



Deadlock: A Type of Starvation

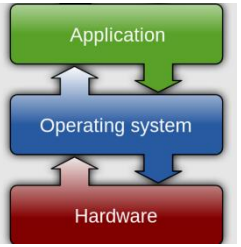


- Starvation – thread fails to make progress for an indefinite period of time
- Deadlock – starvation due to a cycle of waiting among a set of threads
  - Each thread waits for some other thread in the cycle to take some action

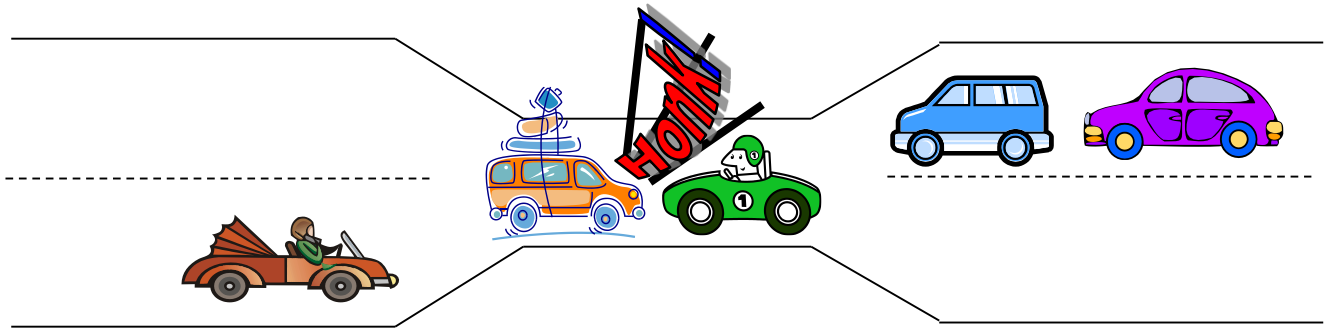


Example: Single-Lane Bridge Crossing

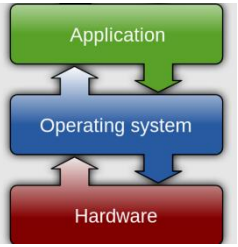
CA 140 to Yosemite National Park



Bridge Crossing Example



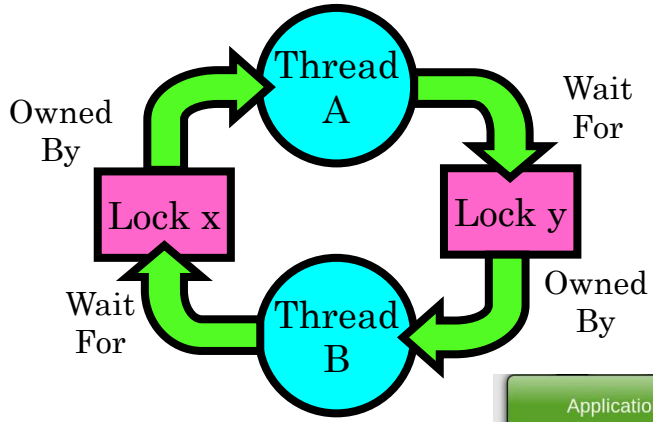
- Each segment of road can be viewed as a resource
  - Cars must own the segment under them
  - Must acquire segment that they are moving into
- Deadlock: Two cars in opposite directions meet in middle

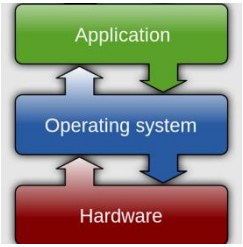


Deadlock with Locks

Thread A	Thread B
`x.Acquire();`	`y.Acquire();`
`y.Acquire();`	`x.Acquire();`
…	…
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`

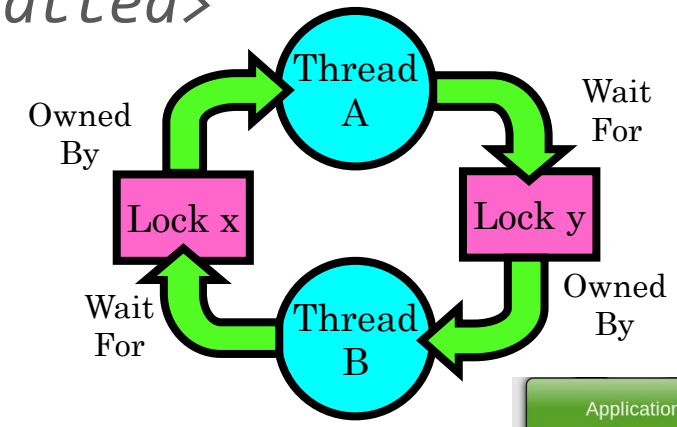
Nondeterministic Deadlock

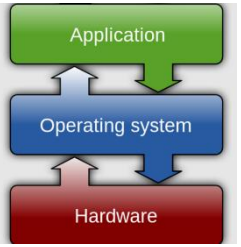




Deadlock with Locks: Unlucky Case

Thread A	Thread B
`x.Acquire();`	
	`y.Acquire();`
`y.Acquire(); <stalled>`	
	`x.Acquire(); <stalled>`
`<unreachable>`	`<unreachable>`
…	…
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`





Deadlock with Locks: "Lucky" Case

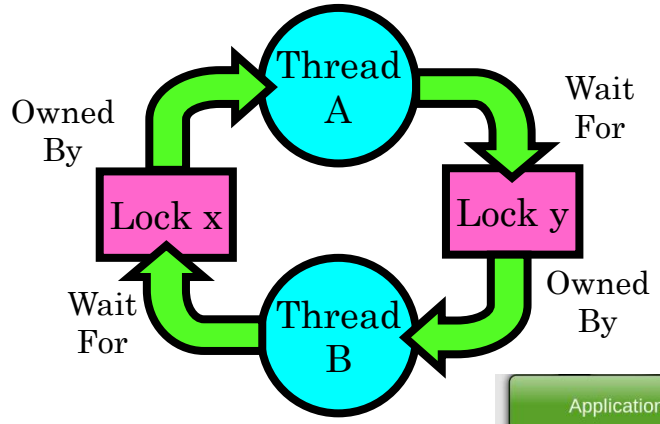
Thread A

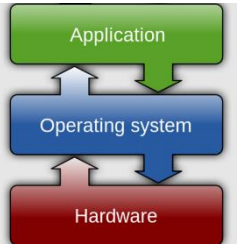
```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B

```
y.Acquire();

x.Acquire();
…
x.Release();
y.Release();
```

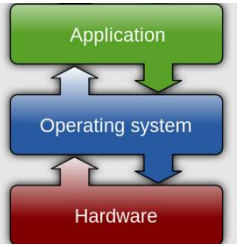




Sometimes, schedule won't trigger deadlock

Other Types of Deadlock

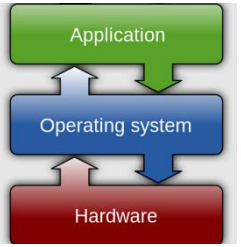
- Threads often block waiting for resources
  - Locks
  - Terminals
  - Printers
  - CD drives
  - Memory
- Threads often block waiting for other threads
  - Pipes
  - Sockets
- You can deadlock on any of these!



Deadlock with Space

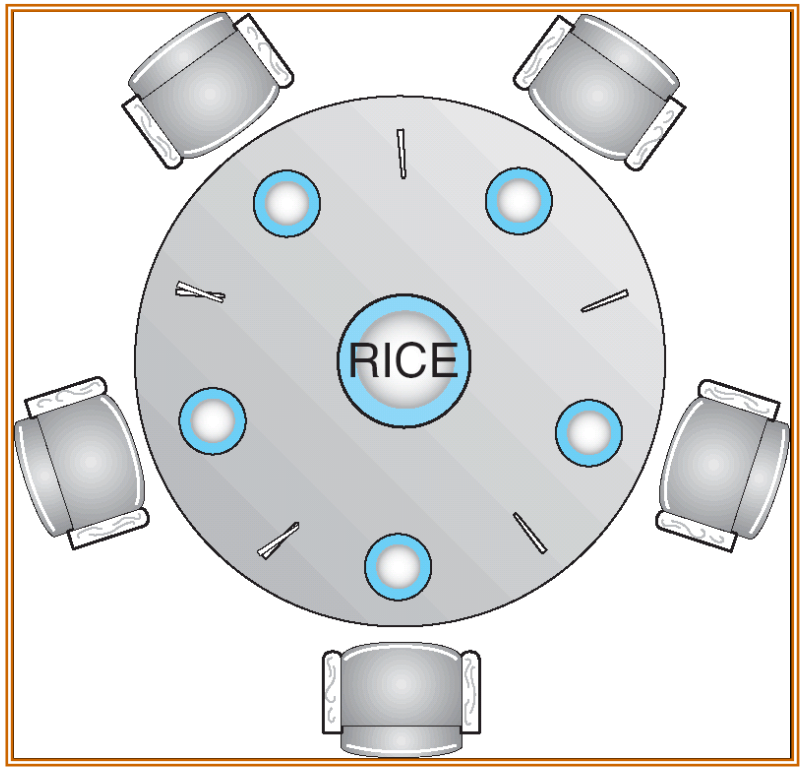
Thread A	Thread B
`AllocateOrWait(1 MB)`	`AllocateOrWait(1 MB)`
`AllocateOrWait(1 MB)`	`AllocateOrWait(1 MB)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

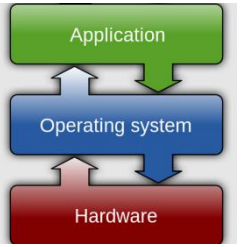
- If only 2 MB of space, we get same deadlock situation



The Dining Philosophers Problem

- Five chopsticks, five philosophers
  - Goal: Grab two chopsticks to eat
- Deadlock if they all grab chopstick to their right
- How to fix deadlock?
  - Make one of them give up a chopstick
- How to prevent deadlock?
  - Never take last chopstick if a hungry philosopher can't have two afterward
  - Alternatively, always take either two chopsticks or none

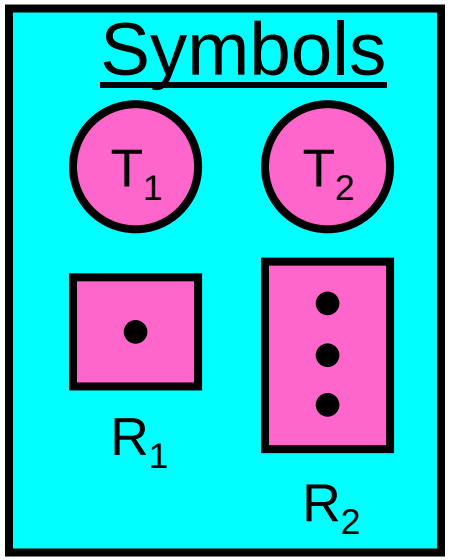


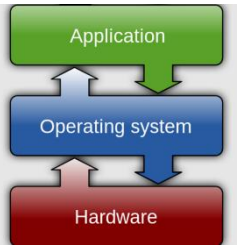


How to Detect Deadlock?

Resource-Allocation Graph

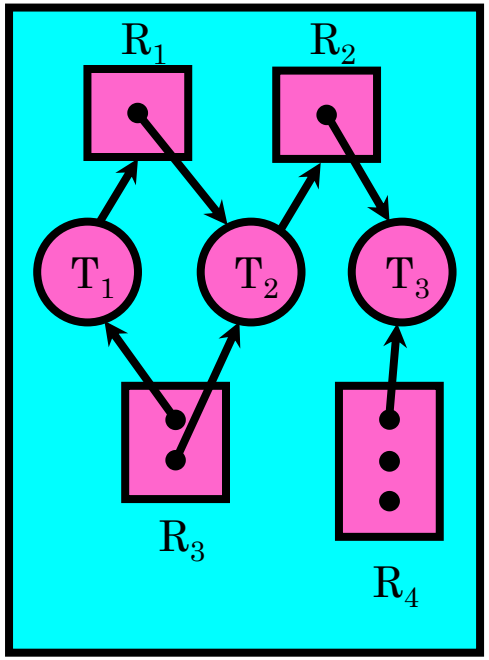
- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$
    - CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances
  - Each thread utilizes a resource as follows:
    - Request() / Use() / Release()
- Resource-Allocation Graph V:
  - 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 $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



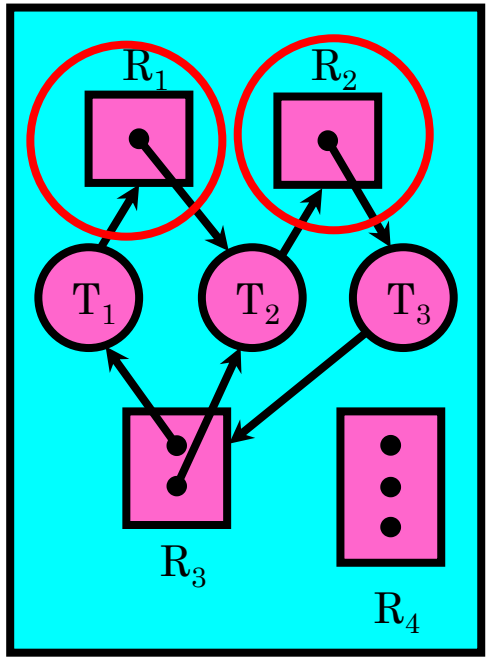


Resource-Allocation Graph Examples

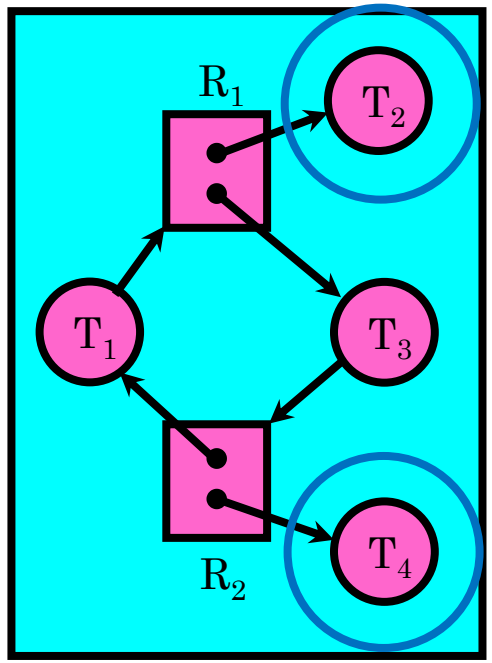
- Model: Directed Graph
- request edge
  - $T_i \rightarrow R_j$
- assignment edge
  - $R_j \rightarrow T_i$



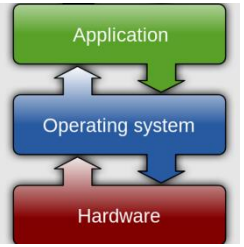
Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock



Deadlock Detection Algorithm

- Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

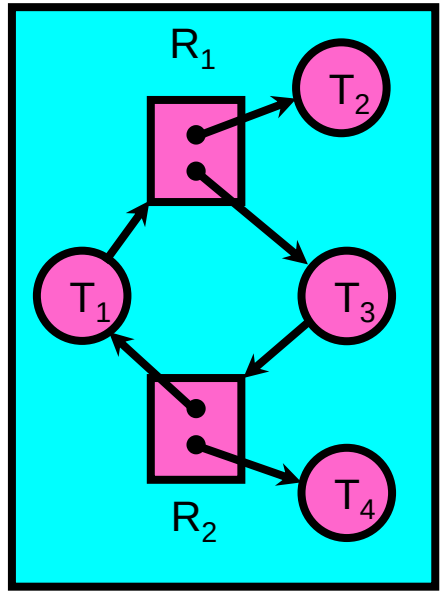
  [FreeResources]: Current free resources of each type

  [$Request_X$]: Current requests from thread X

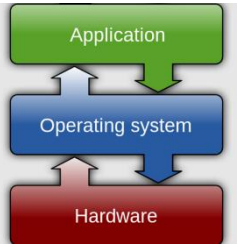
  [$Alloc_X$]: Current resources held by thread X

- See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all threads to UNFINISHED          # tasks we have not resolved yet
do {
  done = true
  Foreach thread in UNFINISHED {
    if ([Request_thread] <= [Avail]) {    # if resources are available
      remove thread from UNFINISHED       # task can terminate on its own
      [Avail] = [Avail] + [Alloc_thread]  # free resources of this task
      done = false
    }
  }
} until(done)
```

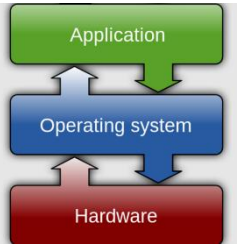


- Nodes left in UNFINISHED ⇒ deadlocked (tasks that can terminate on their own have been removed)



How Should a System Deal With Deadlock?

- Three different approaches:
  - Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
  - Deadlock prevention: write your code in a way that it isn't prone to deadlock
  - Deadlock recovery: let deadlock happen, and then figure out how to recover from it
- Modern operating systems:
  - Make sure the system isn't involved in any deadlock
  - Ignore deadlock in applications
    - "Ostrich Algorithm" or deadlock denial



Deadlock Avoidance

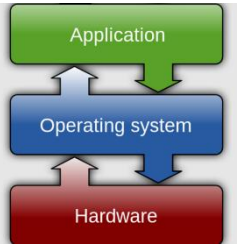
- Idea: When a thread requests a resource, OS checks if it would result in deadlock
  - If not, it grants the resource right away
  - If so, it waits for other threads to release resources

THIS DOES NOT WORK!!!!

- Example:

Blocks...

Thread A	Thread B
`x.Acquire();`	`y.Acquire();`
`y.Acquire();`	`x.Acquire();`
…	…
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`



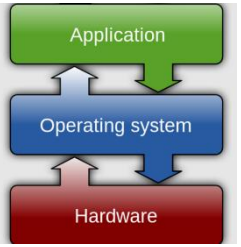
Deadlock Avoidance: Three States

- Safe state
  - System can delay resource acquisition to prevent deadlock

Deadlock avoidance: prevent system from reaching an *unsafe* state

- Unsafe state
  - No deadlock yet…
  - But threads can request resources in a pattern that **unavoidably** leads to deadlock

- Deadlocked state
  - There exists a deadlock in the system
  - Also considered "unsafe"



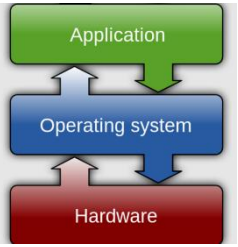
Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in an unsafe state
  - If not, it grants the resource right away
  - If so, it waits for other threads to release resources

- Example:

Wait until Thread A releases the lock

Thread A	Thread B
`x.Acquire();`	`y.Acquire();`
`y.Acquire();`	`x.Acquire();`
…	…
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`

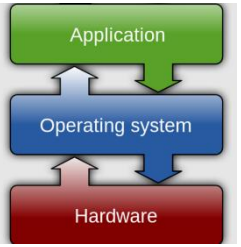


Banker's Algorithm for Avoiding Deadlock

- 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:
  - 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_{thread}]-[$Alloc_{thread}$] <= [Avail]) for ([$Request_{thread}$] <= [Avail])

      Grant request if result is deadlock free (conservative!)

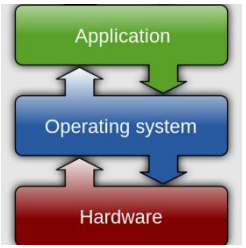


Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
  -
  -
- Ba
  -

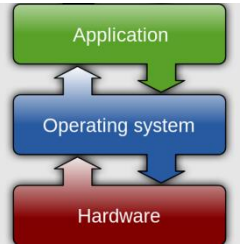
```
[Avail] = [FreeResources]
Add all threads to UNFINISHED
do {
  done = true
  Foreach thread in UNFINISHED {
    if ([Max_thread]-[Alloc_thread] <= [Avail]) {
      remove thread from UNFINISHED
      [Avail] = [Avail] + [Alloc_thread]
      done = false
    }
  }
} until(done)
```

Grant request if result is deadlock free (conservative!)



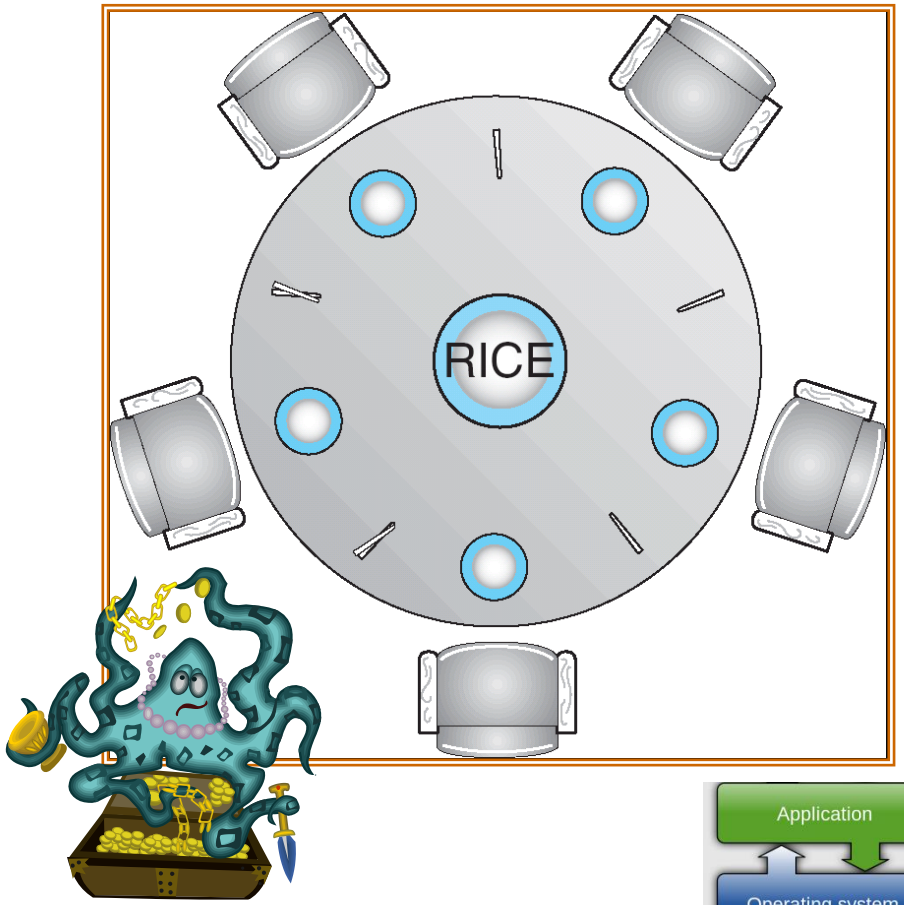
Banker's Algorithm for Avoiding Deadlock

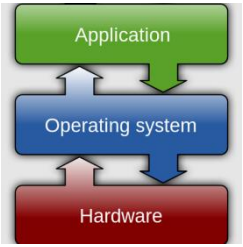
- Alternative view: Banker's Algorithm checks whether all tasks finish if:
  - Scheduler runs each task to completion one at a time, with no concurrency
    - Most conservative thing the scheduler can do—it will avoid deadlock if it's possible to do so
  - Tasks allocate resources up to maximum and hold the resources simultaneously
    - Most deadlock-prone thing the tasks can do
- If under these circumstances all tasks can proceed, then the system will not enter an 'unsafe' state



Applying Banker's Algorithm to the Dining Philosophers Problem

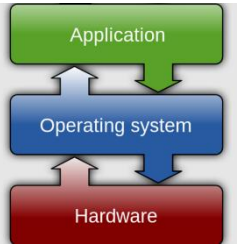
- "Safe" (won't cause deadlock) if when trying to grab chopstick either:
  - Not last chopstick
  - Is last chopstick but someone will have two afterwards (a hungry philosopher can't have two afterward)
- What if k-handed philosophers? Don't allow if:
  - It's the last one, no one would have k
  - It's 2nd to last, and no one would have k-1
  - It's 3rd to last, and no one would have k-2
  - …





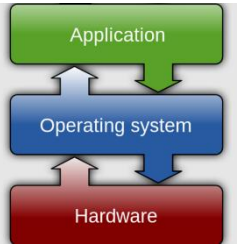
How Should a System Deal With Deadlock?

- Three different approaches:
  - Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
  - Deadlock prevention: write your code in a way that it isn't prone to deadlock
  - Deadlock recovery: let deadlock happen, and then figure out how to recover from it
- Modern operating systems:
  - Make sure the system isn't involved in any deadlock
  - Ignore deadlock in applications
    - "Ostrich Algorithm" or deadlock denial



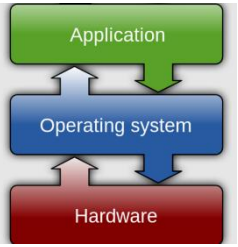
Deadlock Prevention

- Structure code in a way that it isn't prone to deadlock
- First: What must be true about our code for deadlock to happen?



Four Requirements for Deadlock

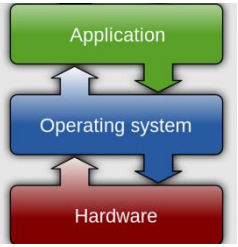
- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is 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
    - …
    - T_n is waiting for a resource that is held by T_1
- **To prevent deadlock, make sure at least one of these conditions does not hold**



Deadlock Prevention (1/4)

- Remove "Mutual Exclusion"
  - Only one thread at a time can use a resource.

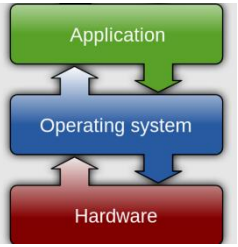
  - Infinite resources
    - Example: Virtual Memory
  - Restructure program to avoid sharing



(Virtually) Infinite Resources

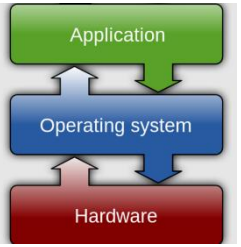
Thread A	Thread B
`AllocateOrWait(1 MB)`	`AllocateOrWait(1 MB)`
`AllocateOrWait(1 MB)`	`AllocateOrWait(1 MB)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

- With virtual memory we have "infinite" space so everything will just succeed.



Deadlock Prevention (2/4)

- Remove "Hold-and-Wait"
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - Back off and retry
    - Removes deadlock but could still lead to starvation
  - Request all resources up front
    - Reduces concurrency (parallelism?)
    - Example: Dining philosophers grab both chopsticks atomically



Request Resource Atomically

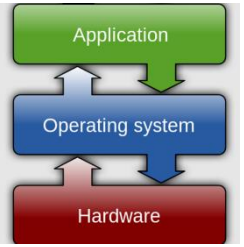
Thread A

```
x.Acqu
y.Acqu
…
y.Rele
x.Rele
```

Thread B

Consider instead:

Thread A	Thread B
`Acquire_both(x, y);`	`Acquire_both(y, x);`
…	…
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`



Request Resource Atomically

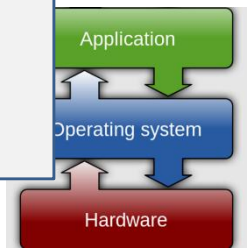
Thread A

```
x.Acqu
y.Acqu
…
y.Rele
x.Rele
```

Thread B

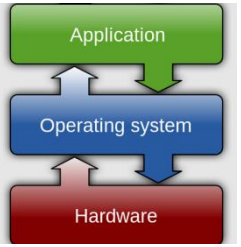
Or consider this:

Thread A	Thread B
`z.Acquire();`	`z.Acquire();`
`x.Acquire();`	`y.Acquire();`
`y.Acquire();`	`x.Acquire();`
`z.Release();`	`z.Release();`
…	…
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`



Deadlock Prevention (3/4)

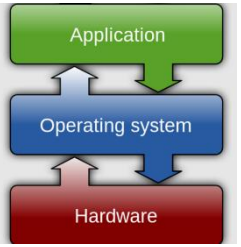
- Remove "No Preemption"
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
  - Allow OS to revoke resources it has granted
    - Example: Preemptive scheduling
  - Doesn't always work with resource semantics



Preempting Resources

Thread A	Thread B
`AllocateOrWait(1 MB)`	`AllocateOrWait(1 MB)`
`AllocateOrWait(1 MB)`	`AllocateOrWait(1 MB)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

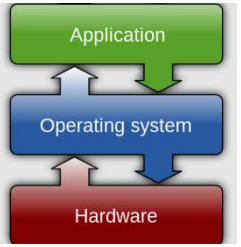
- With virtual memory we have "infinite" space so everything will just succeed.
- Alternative view: we are "pre-empting" memory when paging out to disk, and giving it back when paging back in



Deadlock Prevention (4/4)

- Remove "Circular Wait"
  - T_1 waits for T_2 and T_2 waits for T_1

  - Acquire resources in a consistent order
  - Acquire all resources atomically



Acquire Resources in a Consistent Order

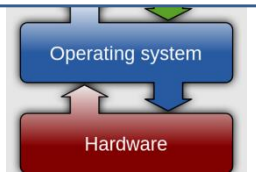
Thread A
```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B
```
y.Acquir
x.Acquir
…
x.Releas
y.Releas
```

Consider instead:

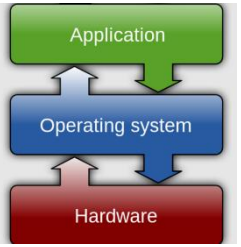
Thread A	Thread B
x.Acquire();	x.Acquire();
y.Acquire();	y.Acquire();
…	…
y.Release();	x.Release();
x.Release();	y.Release();

Does it matter in which order the locks are released?



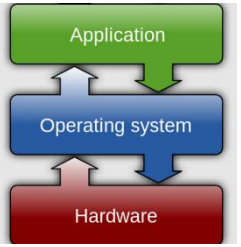
How Should a System Deal With Deadlock?

- Three different approaches:
  - Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
  - Deadlock prevention: write your code in a way that it isn't prone to deadlock
  - Deadlock recovery: let deadlock happen, and then figure out how to recover from it
- Modern operating systems:
  - Make sure the system isn't involved in any deadlock
  - Ignore deadlock in applications
    - "Ostrich Algorithm" or deadlock denial



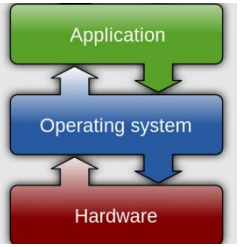
How to Deal with Deadlock?

- Terminate thread, force it to give up resources
  - Dining Philosophers Example: Remove a dining philosopher
  - In AllocateOrWait example, OS kills a process to free up some memory
  - Not always possible—killing a thread holding a lock leaves world inconsistent
- Roll back actions of deadlocked threads
  - Common techniques in databases (transactions)
  - Of course, if you restart in exactly the same way, may enter deadlock again
- Preempt resources without killing off thread
  - Temporarily take resources away from a thread
  - Doesn't always fit with semantics of computation



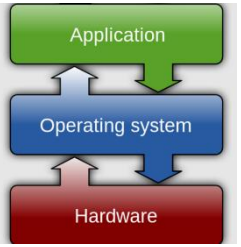
Announcements

- Assignment 2: due Monday, April 6
- Project 1: deadline extended to Monday, April 6



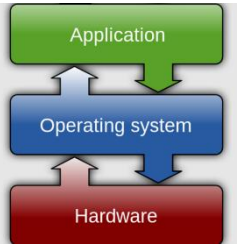
The multi-oom Test (Project 1)

- The multi-oom test is designed to stress your Project 1 implementation
- Keeps creating processes until doing so fails
  - Checks that you handle all failures properly (e.g., malloc failures)
- Exits in unclean ways
  - Checks that you properly handle exiting due to a fault, exiting with files open…
- Checks whether all memory has been released when process exits
  - Verify that every `page_alloc` has a `page_free`, every `malloc` has a corresponding `free`
- It repeats the same thing 10 times, and checks that it can spawn the same number of processes each time
  - To make sure there are no memory leaks



Conclusion

- Starvation vs. Deadlock
  - Starvation: Thread indefinitely unable to make progress
  - Deadlock: Thread(s) unable to make progress due to circular wait
- Four conditions for deadlock:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular Wait
- Three different approaches to address deadlock:
  - Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
  - Deadlock prevention: write your code in a way that it isn't prone to deadlock
  - Deadlock recovery: let deadlock happen, and then figure out how to recover from it
- Or deadlock denial: ignore the possibility of deadlock in applications



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP