Starvation and Deadlock, Memory and Segments

4 January 2026
Lecture 10

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

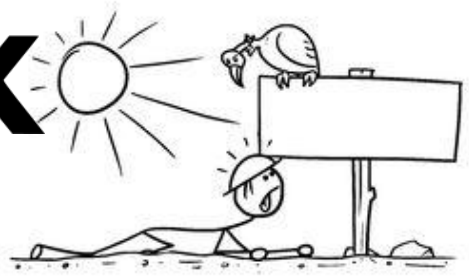
Scheduler	First Come First Serve	Round Robin (RR) • Scheduling quantum	Priority scheduling • Priority inversion
Starvation	Measurements • Response time • Average wait time • Average completion time	Shortest Job First	Shortest Remaining Time First
Lottery Scheduling	Multi-level feedback	O(1)	CFS • Virtual Runtime

Topics for Today

- Starvation and Deadlock
- Address Spaces and Segmentation
  - Loading and Translating
  - Segments
  - Fragmentation

Today's concepts

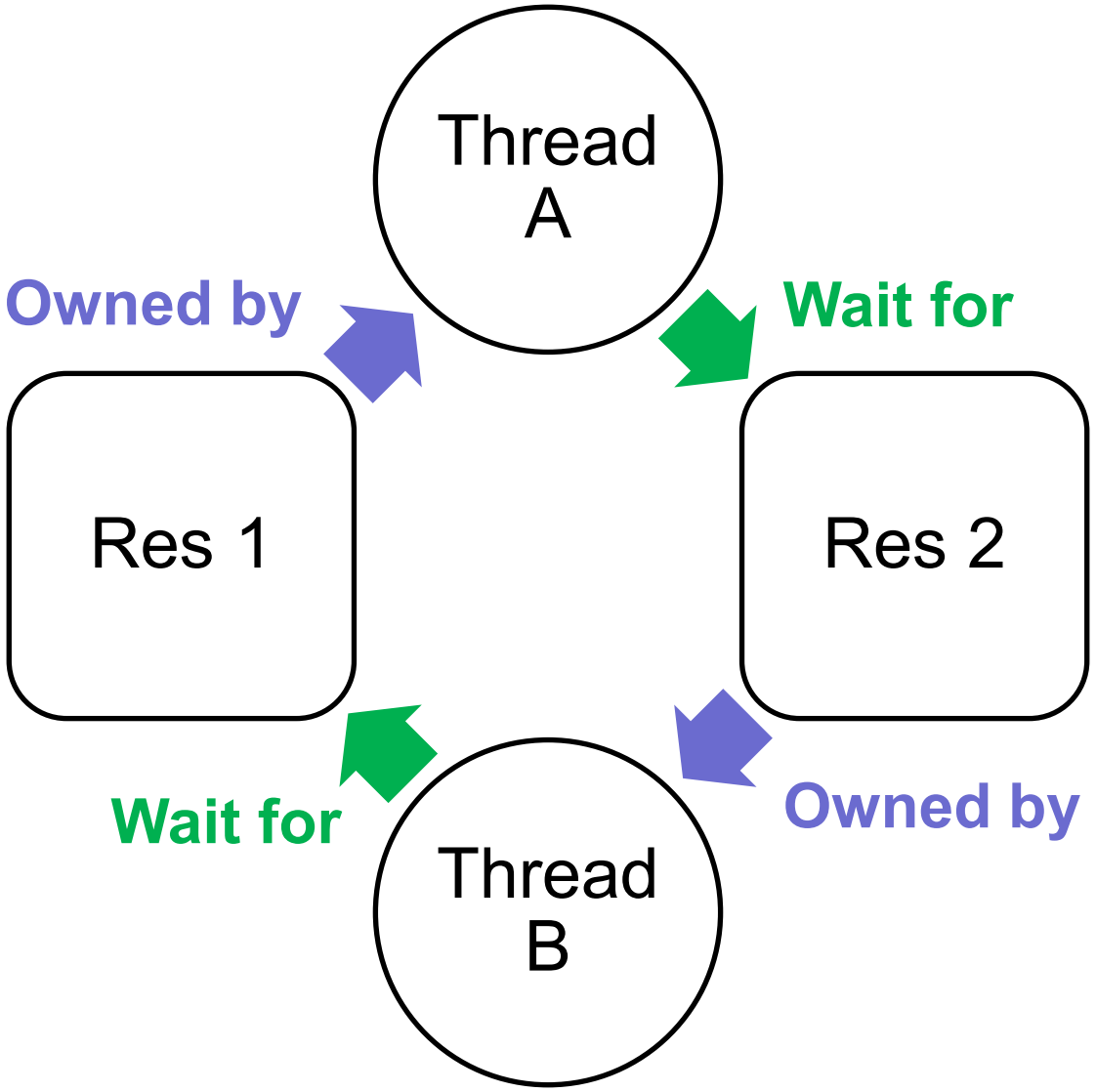
Starvation vs DEADLOCK



- 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 $\Rightarrow$ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

Image Credit: http://mcs109.bu.edu/site/files/deadlock/citydeadlock.jpg

DEADLOCK Graph



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

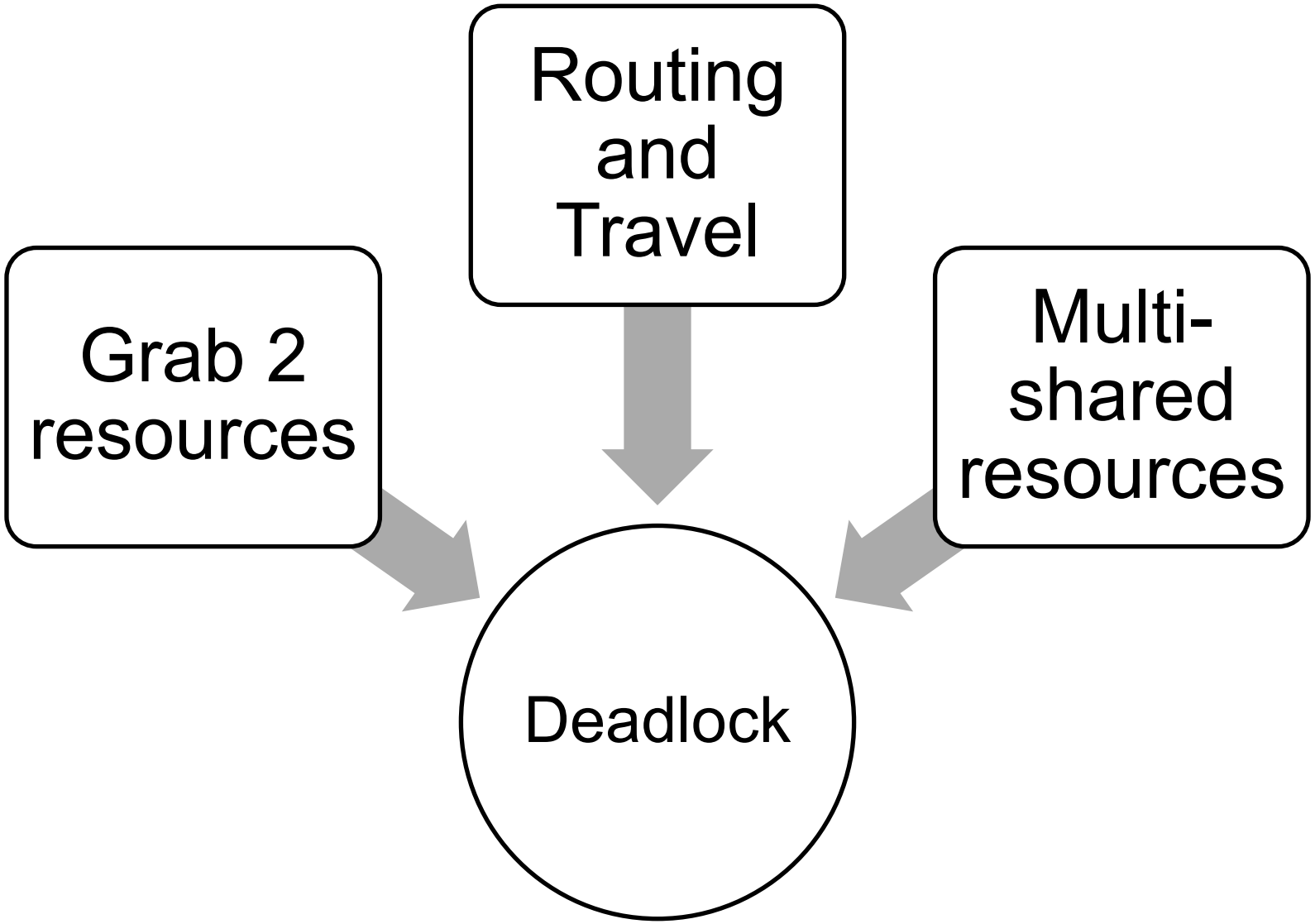
Occasional Deadlock

- Deadlock not always deterministic: Example of two mutexes:

Thread A	Thread B
`x.P();`	`y.P();`
`y.P();`	`x.P();`
`y.V();`	`x.V();`
`x.V();`	`y.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 humus making machine…
- Deadlocks occur with multiple resources
  - Means you can't decompose the problem
  - Can't solve deadlock for each resource independently
- Example: System with two disk drives and two threads
  - Each thread needs two disk drives to function
  - Each thread gets one disk and waits for another one

Deadlock Examples



One Lane Bridge Crossing

By Qrpnut (Own work) [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

One Lane Bridge Crossing

Each segment of road is a resource

- Car must own the segment under it
- Must acquire segment that it is 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

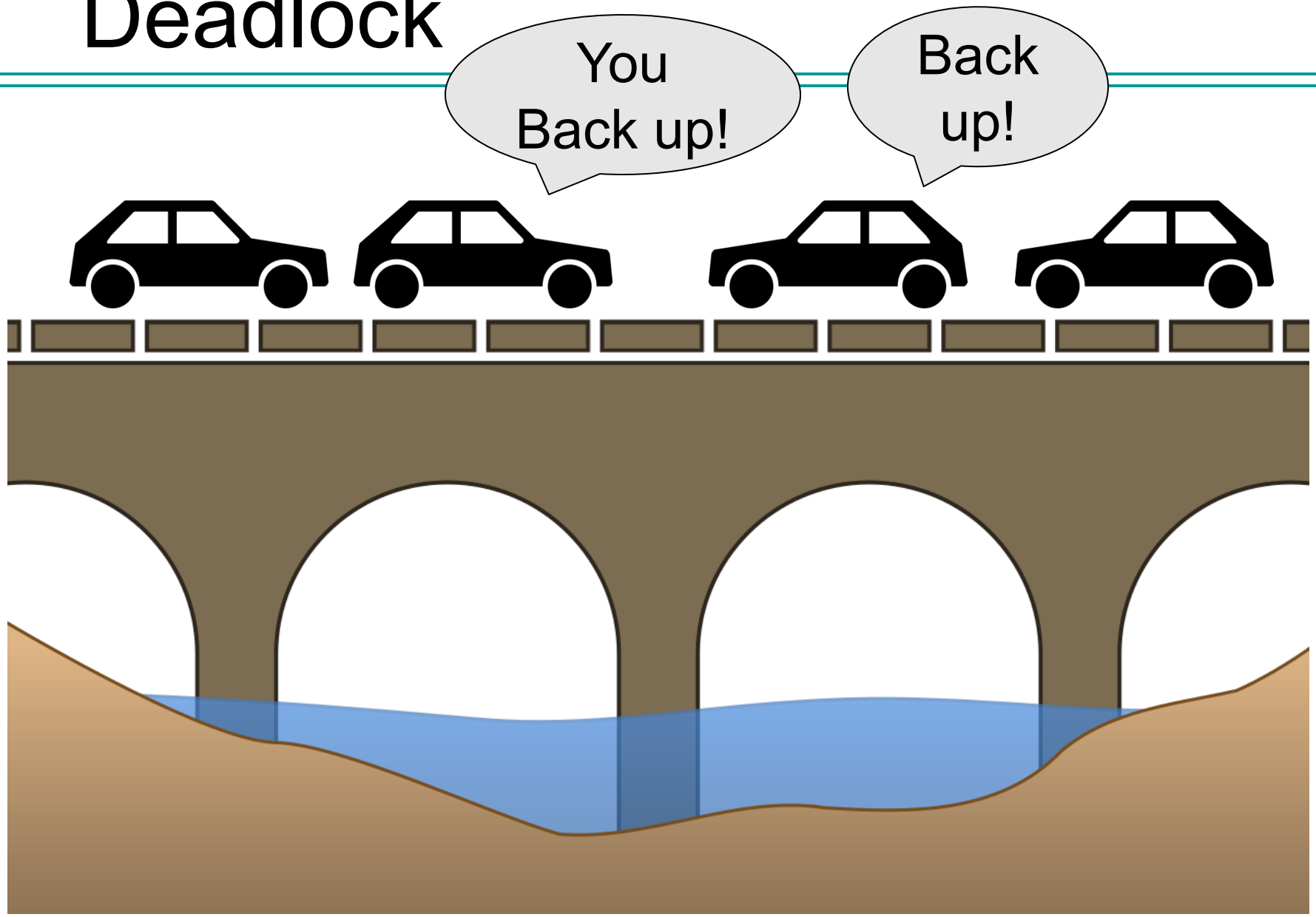
If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)

- Several cars may have to be backed up

Starvation is possible

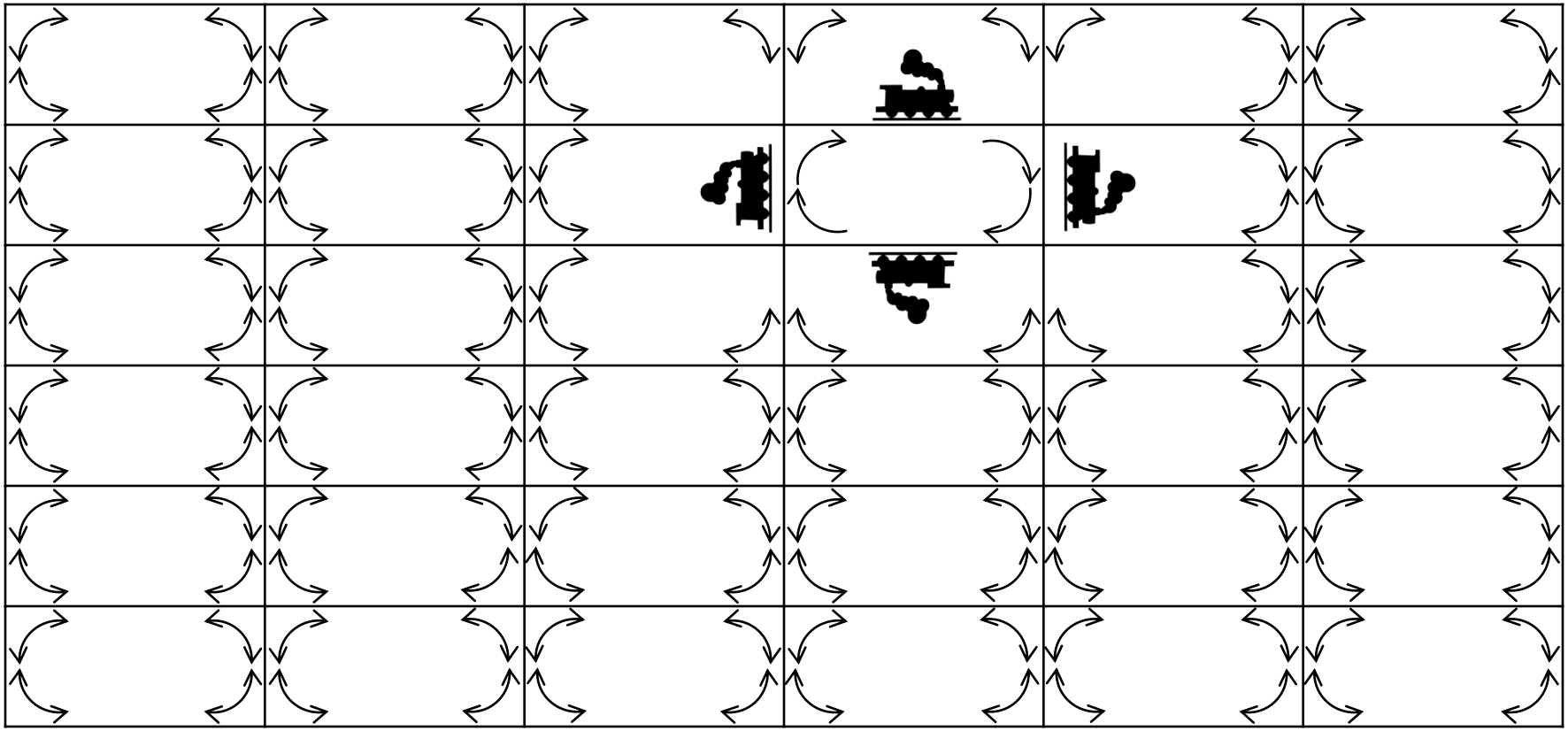
- East-going traffic really fast ⇒ no one goes west

Deadlock



Trains (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains

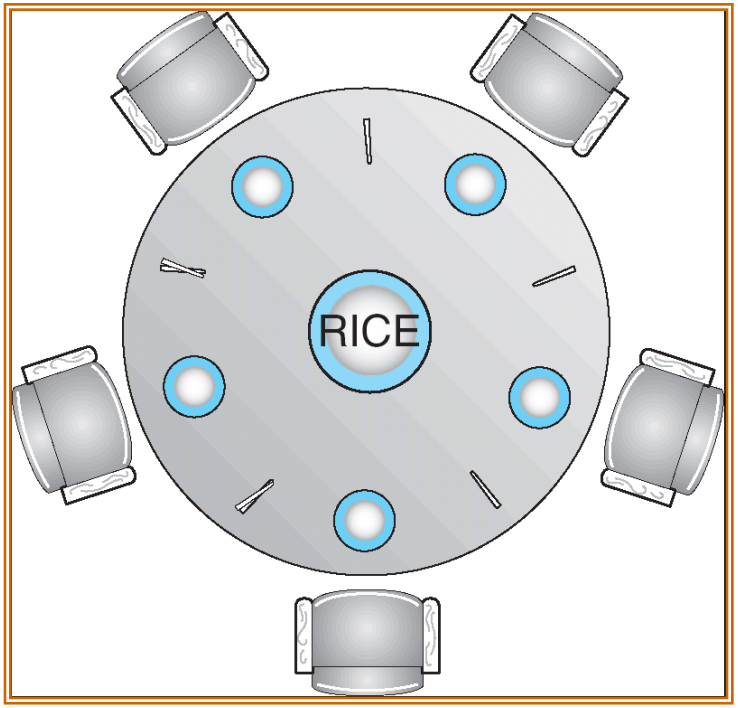


Trains (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - How can we prevent circular dependency?

Dining Students Problem

- Five chopsticks/Five students (really cheap restaurant)
  - Free-for all: Student 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 (!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?

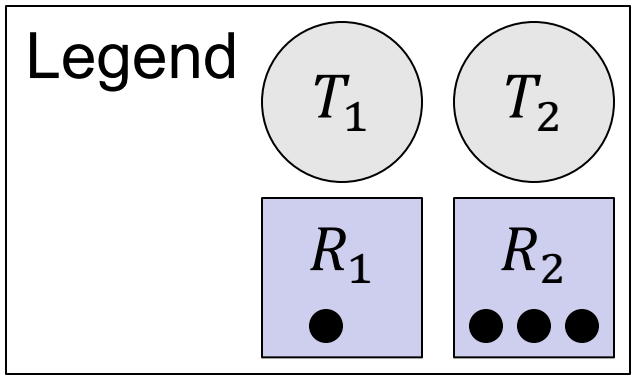


Picturing deadlock can help us understand it better…

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()`

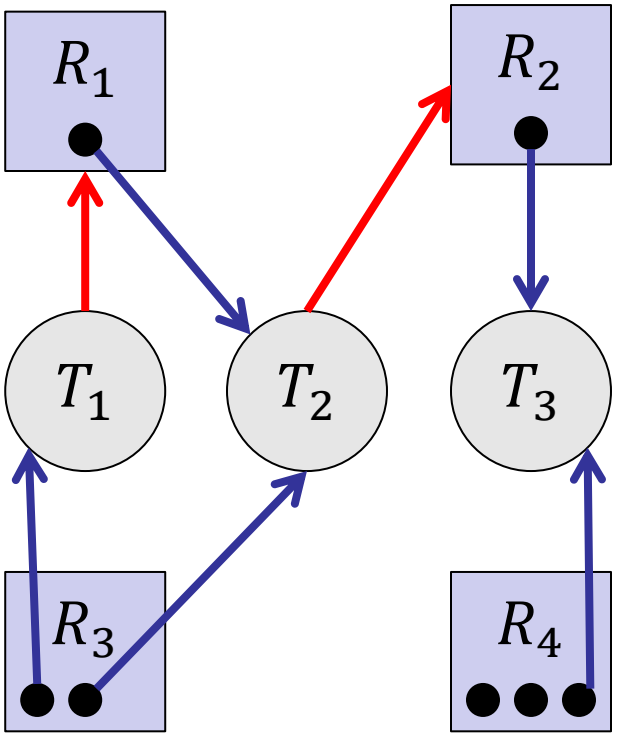


- Resource-Allocation Graph:
  - V is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$ the set of threads in the system.
    - $R = \{R_1, R_2, \ldots, R_n\}$ the set of resource types in system
  - Request Edge – Directed edge $T_i \rightarrow R_j$
  - Assignment Edge – Directed edge $R_j \rightarrow T_i$

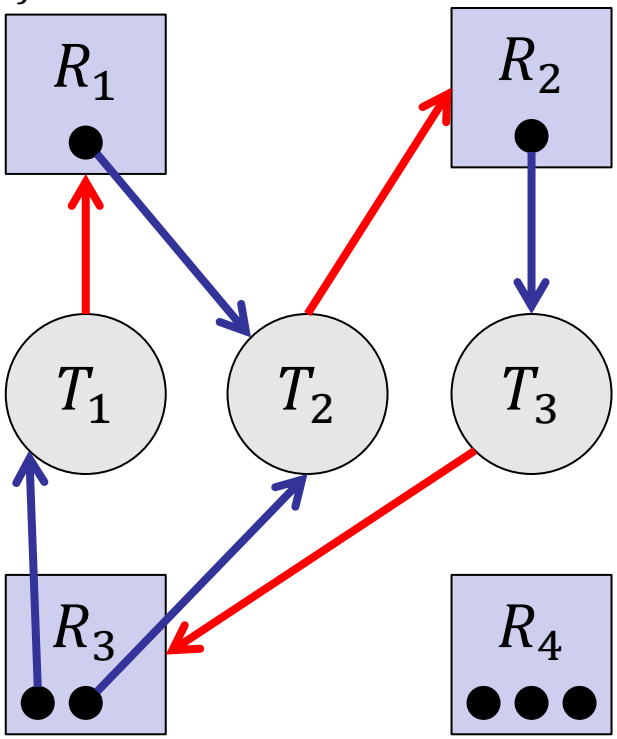
Resource Allocation Graph Examples

Request Edge – Directed edge $T_i \rightarrow R_j$

Assignment Edge – Directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph

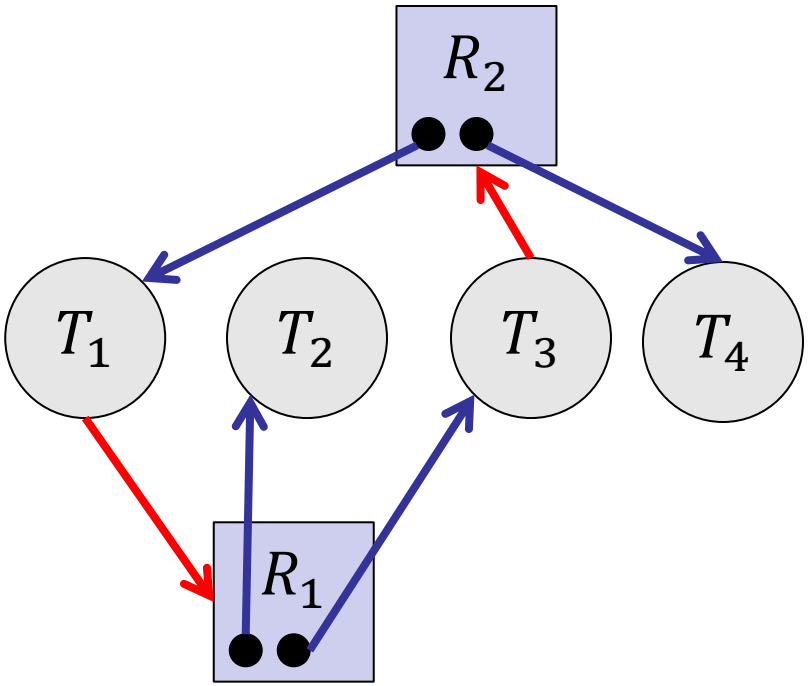


Allocation Graph with Deadlock

Resource Allocation Graph Examples

Request Edge – Directed edge $T_1 \rightarrow R_j$

Assignment Edge – Directed edge $R_j \rightarrow T_i$



Allocation Graph with Cycle
But No Deadlock

Methods for Handling Deadlock

Allow system to enter deadlock and then recover

- Requires deadlock detection algorithm
- Some technique for forcibly 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

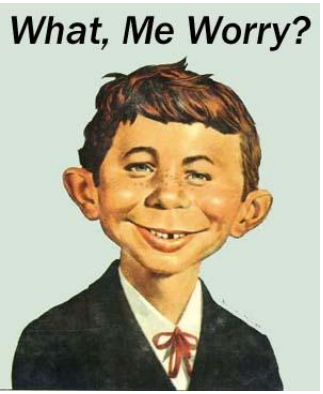


Image sources: http://www.besthackingtricks.com/wp-content/uploads/2015/08/Recover.jpeg, https://s-media-cache-ak0.pinimg.com/originals/d4/c6/ea/d4c6ea0e2d387adadf284ac846b05090.jpg

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):

    `[FreeResources]:` Current free resources each type

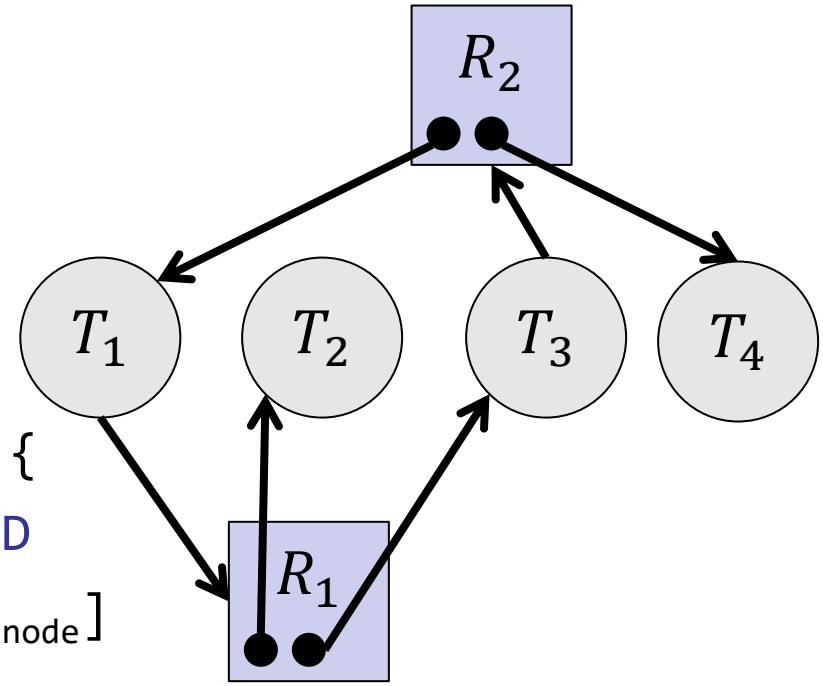
    $[\texttt{Request}_X]$: Current requests from thread X

    $[\texttt{Alloc}_X]$: Current resources held by thread X

- See if tasks can eventually terminate on their own

Deadlock Detection Algorithm

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   foreach node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_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!
- Expel a dining student (or tell him there's free beer elsewhere)
- Not always possible – killing a thread holding a mutex leaves the world inconsistent

Preempt resources

Without killing off thread

- Take away resources from thread temporarily
- Doesn't always fit with semantics of computation

Image source: http://www.aperfectworld.org/05202.html

What to do when detect deadlock?

Roll back

Undo 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)
- If you restart in exactly the same way, may reenter deadlock once again

None of the above

- Many operating systems use other options

Image source: Walmart

Techniques for Preventing Deadlock

- 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 (ex. virtual memory)
- Examples:
  - Ayalon Fwy with 12,000 lanes. Never wait!
  - Infinite disk space (not realistic yet?)

No Sharing of resources

- Totally independent threads
- Not very realistic

Techniques for Preventing Deadlock

Don't allow waiting

- How the phone company avoids deadlock
  - Call to your Mom in Haifa, works its way through the phone lines, but if blocked get busy signal.
- Technique used in Ethernet and some multiprocessor nets
  - Everyone speaks at once. On collision, back off and retry
- Inefficient, since need to retry
  - Consider: Driving to Tel Aviv; when hit traffic jam, suddenly you're transported back home and told to retry!

Image source: http://images.myparkingsign.com/img/lg/K/Fire-Zone-No-Parking-Sign-K-4442.gif

Request Ahead

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 Ayalon Fwy at a time

Image source: http://ralphies.com/assets/images/order-before.jpg

Techniques for Preventing Deadlock

Ordering

Force all threads to request resources in a particular order preventing any cyclic use of resources

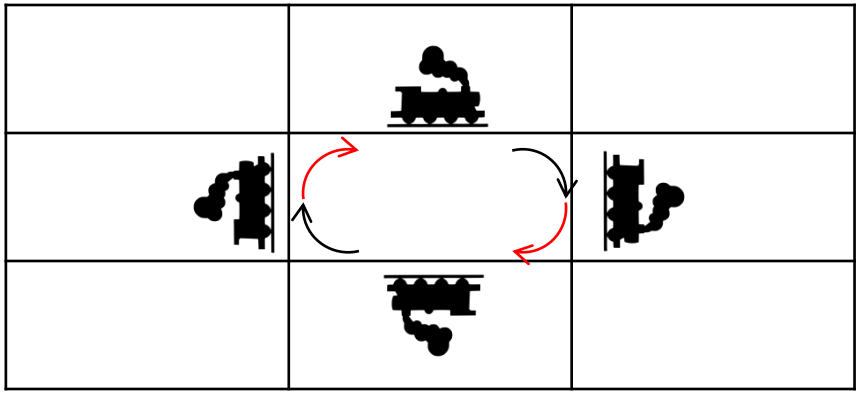
- Thus, preventing deadlock

Example (`x.P, y.P, z.P,…`)

- Make tasks request disk, then memory, then…
- Keep from deadlock on freeways around Tel Aviv by requiring everyone to go clockwise

Image source: http://www.drivingtesttips.biz/wp-content/uploads/2014/05/mini-roundabout-sign.jpg

Recall: Dimension Ordering



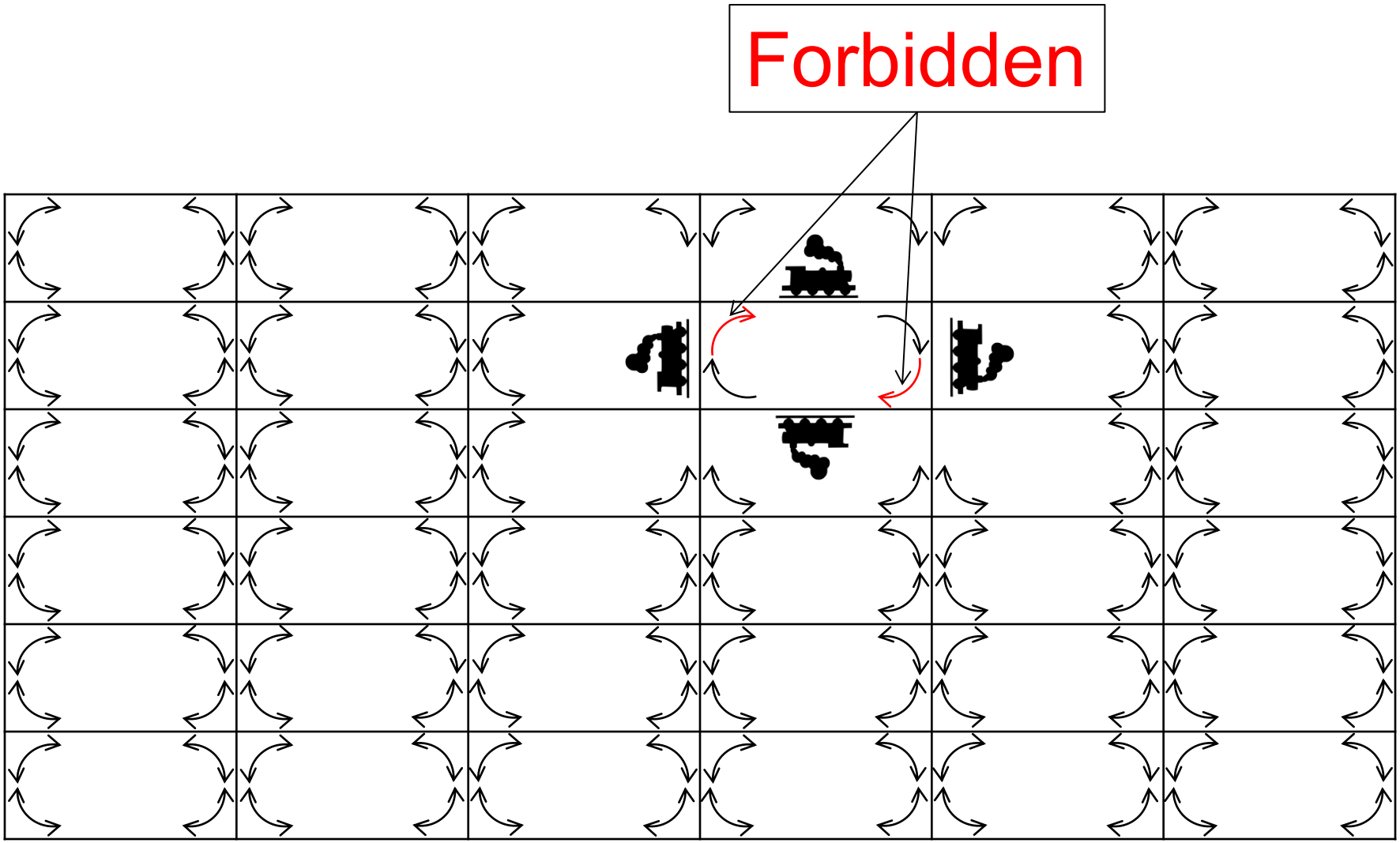
- Force ordering of channels (tracks)
- Protocol: Always go east-west first, then north-south
  - X then Y

Trains (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Similar problem to multiprocessor networks

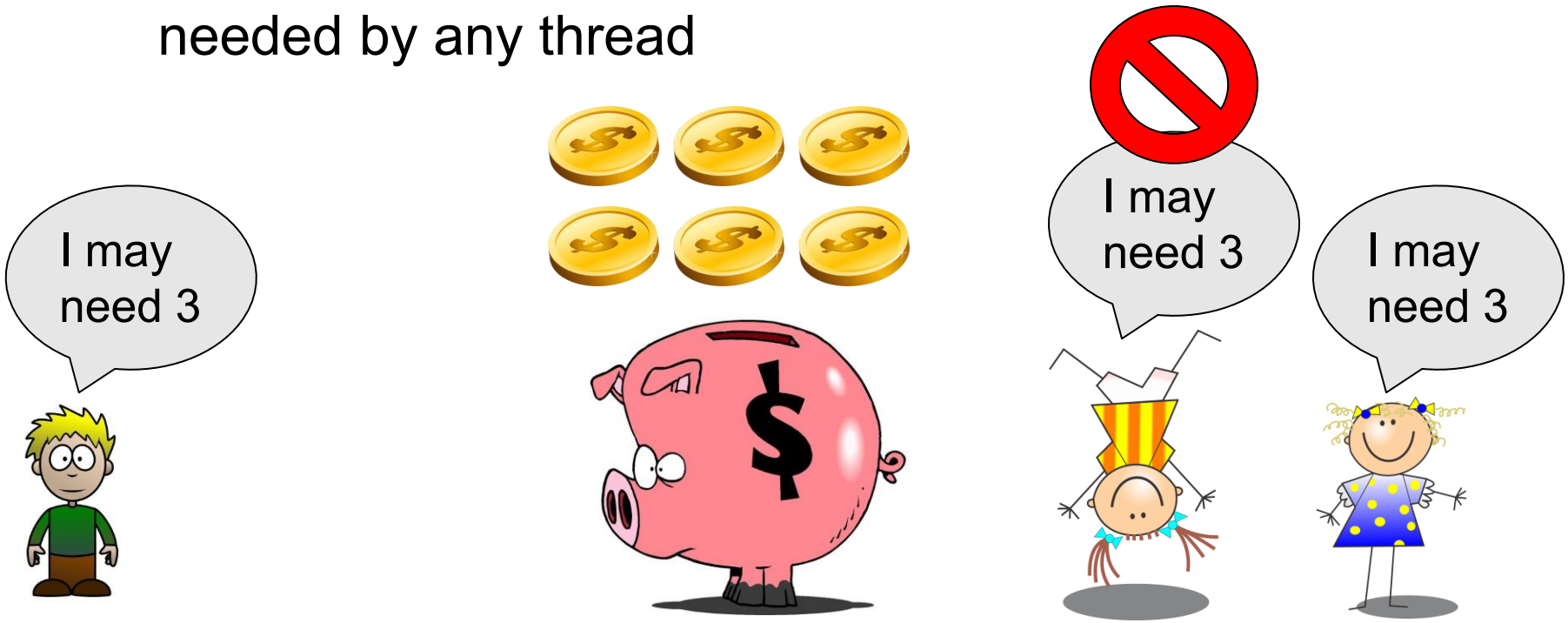
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - Protocol: Always go east-west first, then north-south
  - "dimension ordering" (X then Y)

Dimension Ordering



Another Idea: Accounting

- First Idea:
  - Thread t states maximum resource needs in advance
  - System allows thread t to proceed if: $([\text{Avail}] - [Request_t] \geq Max)$ remaining that might be needed by any thread

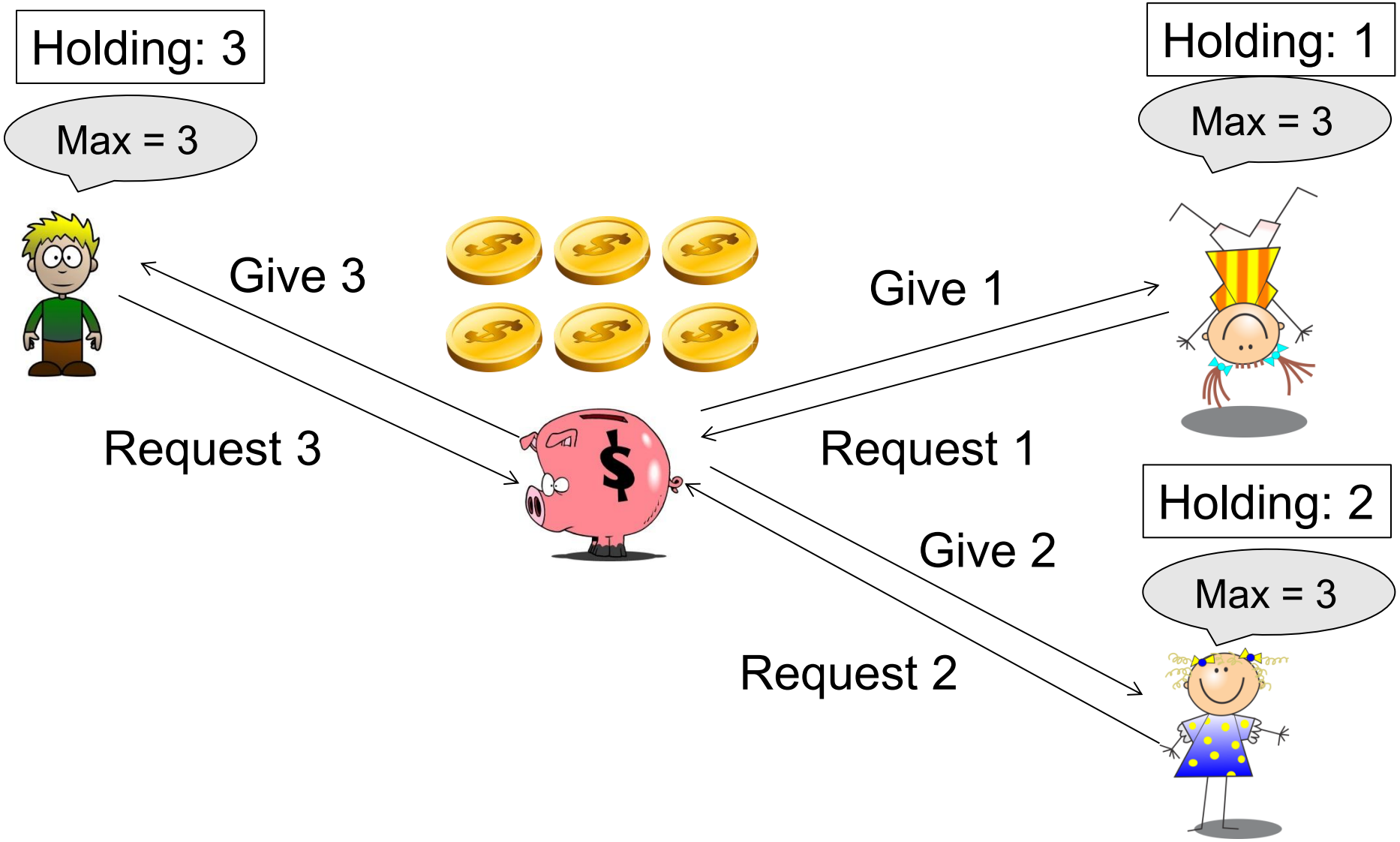


Banker's algorithm (less conservative):

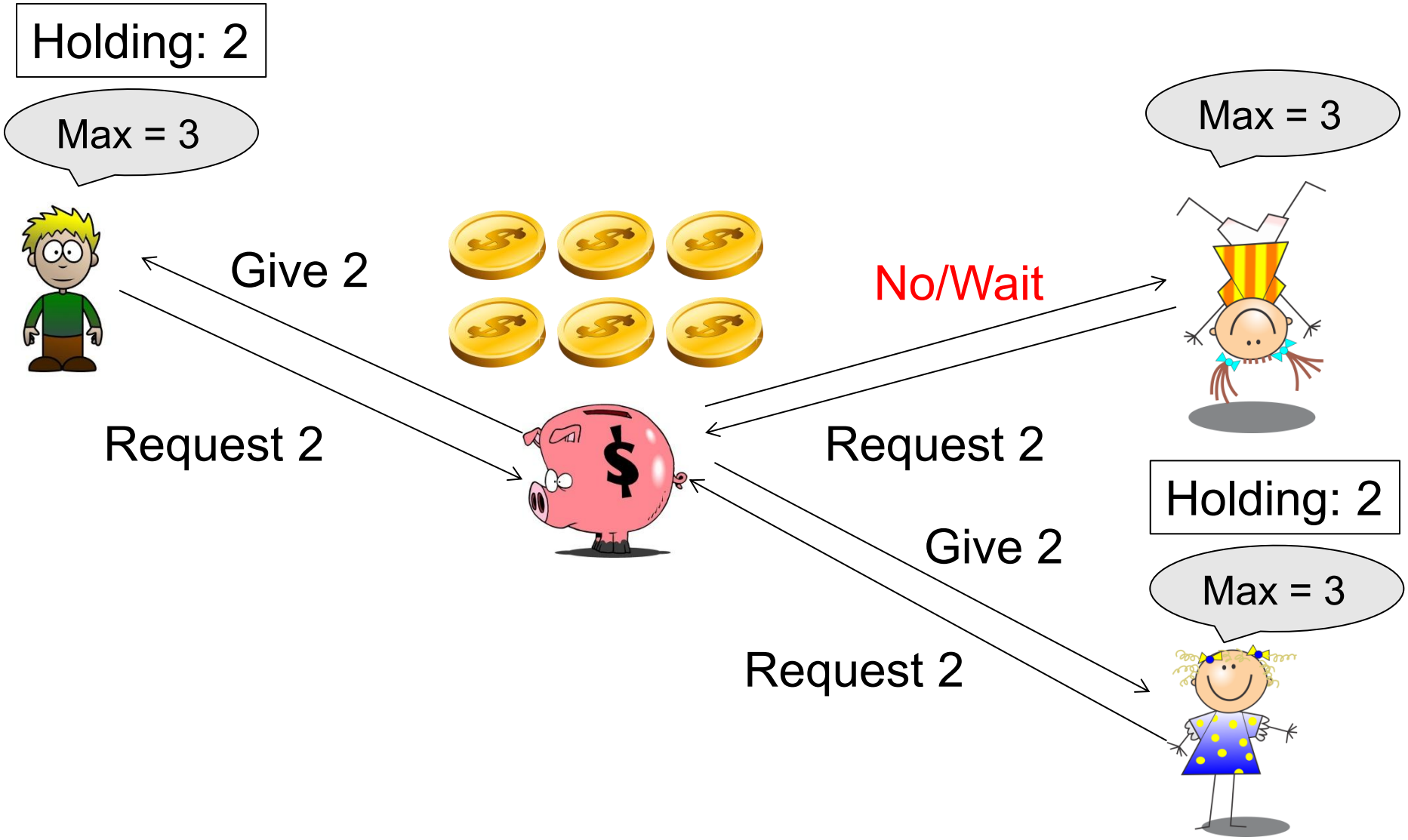
- Allocate resources dynamically
  - Evaluate each request and grant if some ordering of threads is still deadlock free afterward
  - Technique: Pretend each request is granted, then run deadlock detection algorithm, substituting $([Max_{node}] - [Alloc_{node}] \leq [Avail])$ for $([Request_{node}] \leq [Avail])$
    Grant request if result is deadlock free (conservative!)
  - Keeps system in a safe state, i.e. there exists a sequence $\{T_1, T_2, \ldots, T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..

- Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Banking Example 1

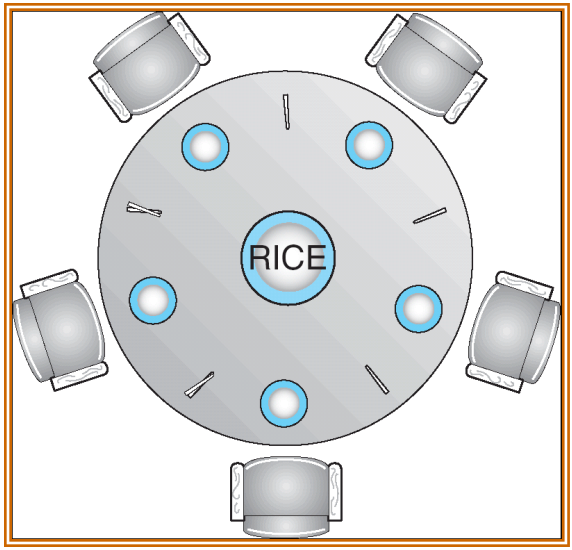


Banking Example 2



Banker's Algorithm Example

Banker's algorithm with dining students



- Safe (won't cause deadlock) if when you try to grab chopstick either:
  1. Not the last chopstick
  2. Is the last chopstick but someone will have two afterwards

- What if k-handed students? 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 $k-2$
  - etc…

Image Source: http://gmythology.blogspot.com/2008/12/hundred-handed-ones-hecatoncheires.html

So Far

- Starvation and Deadlock
- Address Spaces and Segmentation
  - Loading and Translating
  - Segments
  - Fragmentation

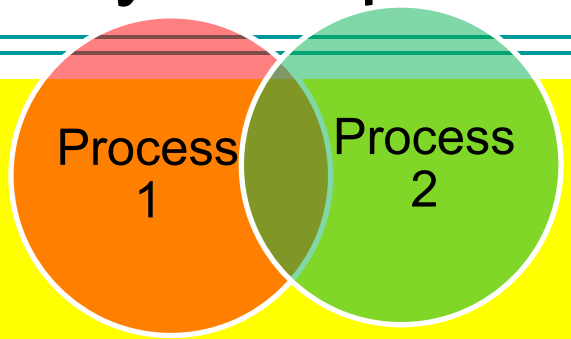
Virtualizing Resources

- Physical Reality: Different Processes/Threads share the same hardware
  - Need to multiplex CPU (Just finished: scheduling)
  - Need to multiplex use of Memory (Today)
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different threads of control use the same memory
    - Physics: Two different pieces of data cannot occupy the same locations in memory
  - Probably don't want different threads to even have access to each other's memory (protection)

Important Aspects of Memory Multiplexing

Translation

$a \rightarrow b$



Controlled Overlap

Protection

Translation: $a \rightarrow b$

Ability to translate accesses from one address space (virtual) to a different one (physical)

When translation exists, processor uses virtual addresses physical memory uses physical addresses

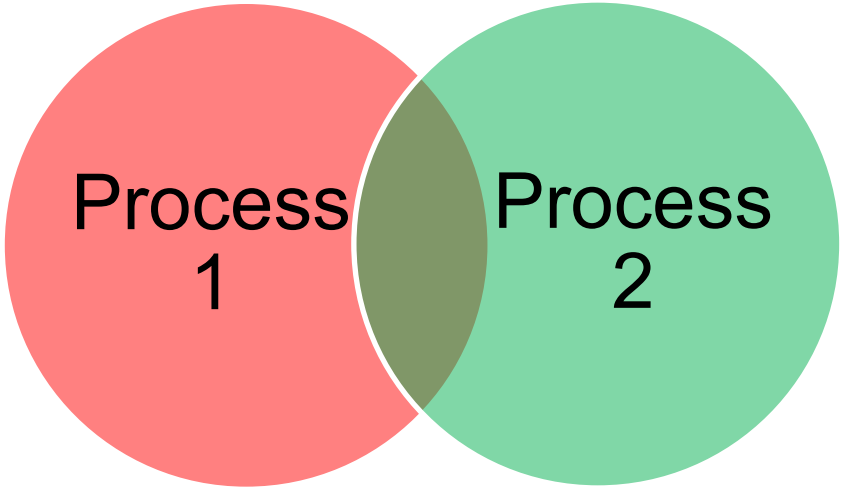
Side effects:

- Can be used to avoid overlap
- Can be used to give uniform view of memory to programs

Controlled Overlap

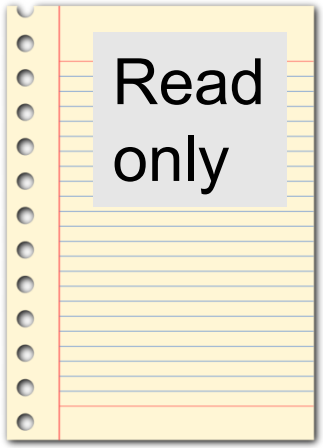
- Separate state of threads should not collide in physical memory.
  - Unexpected overlap causes chaos!

- Want to be able to overlap when desired for communication
  - Also debugging…



Protection

- Prevent access to private memory of other processes
- Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).

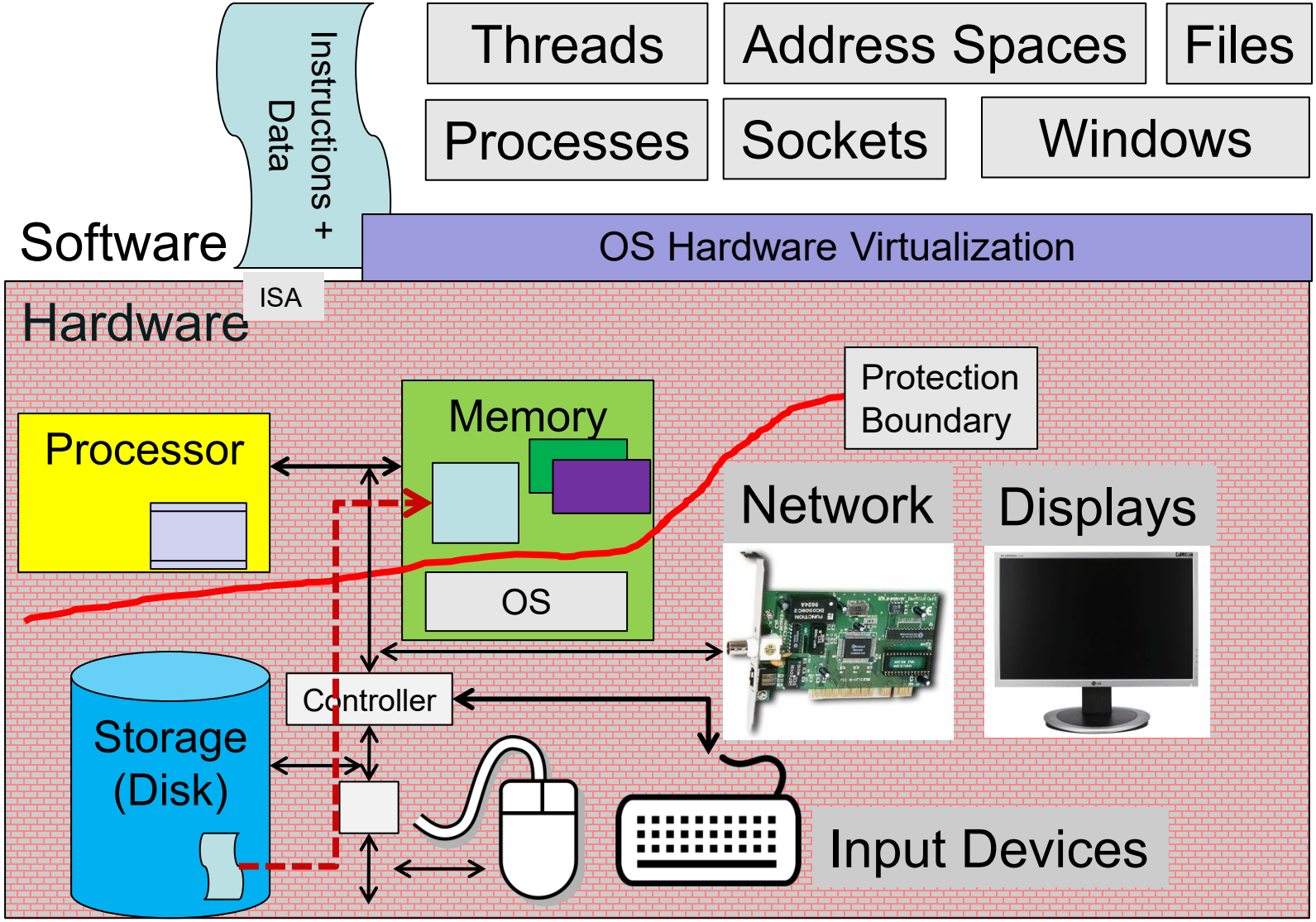


- Kernel data protected from User programs
- Programs protected from themselves

So Far

- Starvation and Deadlock
- Address Spaces and Segmentation
  - Loading and Translating
  - Segments
  - Fragmentation

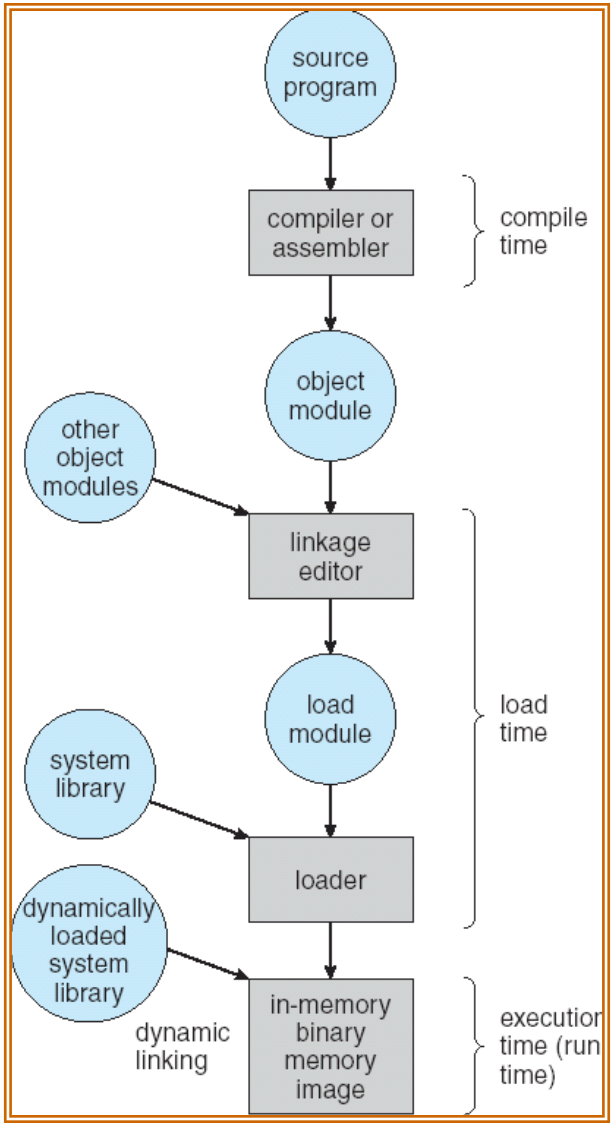
Recall: Loading



Recall a bit of the old days → Relocating loaders

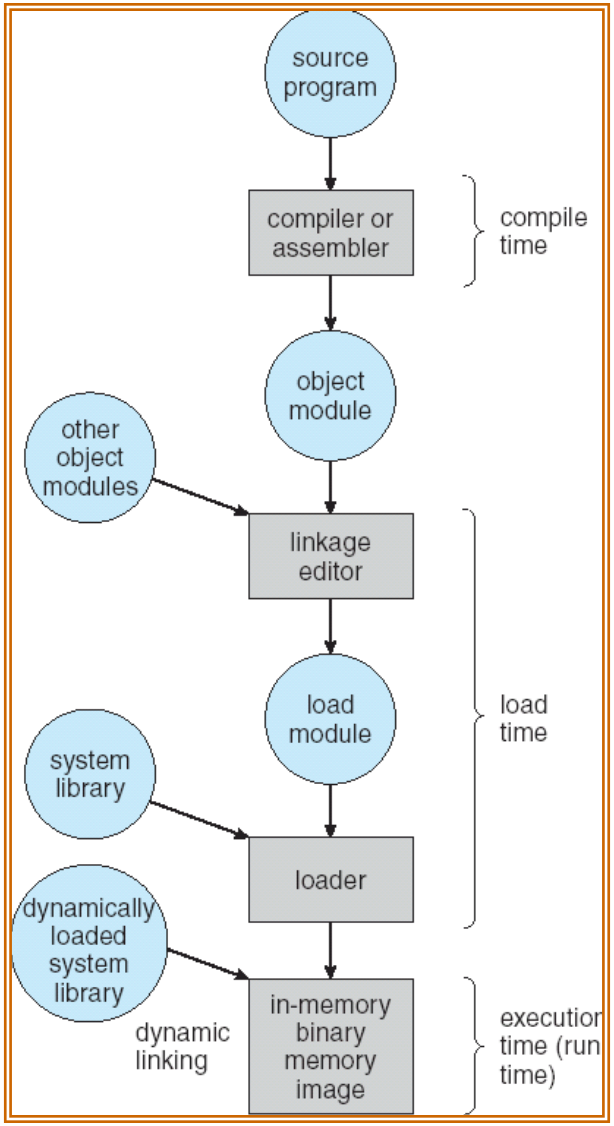
Multi-step Processing of a Program for Execution

- Preparation of a program for execution involves components at:
  - Compile time (ex. `gcc`)
  - Link/Load time (UNIX `ld` does link)
  - Execution time (ex. dynamic libs)
- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system



Multi-step Processing of a Program for Execution

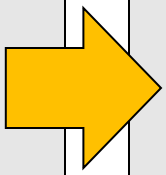
- Dynamic Libraries
  - Linking postponed until execution
  - Small piece of code, *stub*, used to locate appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine



Binding of Instructions and Data to Memory

Process View of Memory

```
data1:  dw      32
                …
start:  lw      r1,0(data1)
        jal     checkit
loop:   addi    r1, r1, -1
        bnz     r1, loop
                …
checkit: …
```

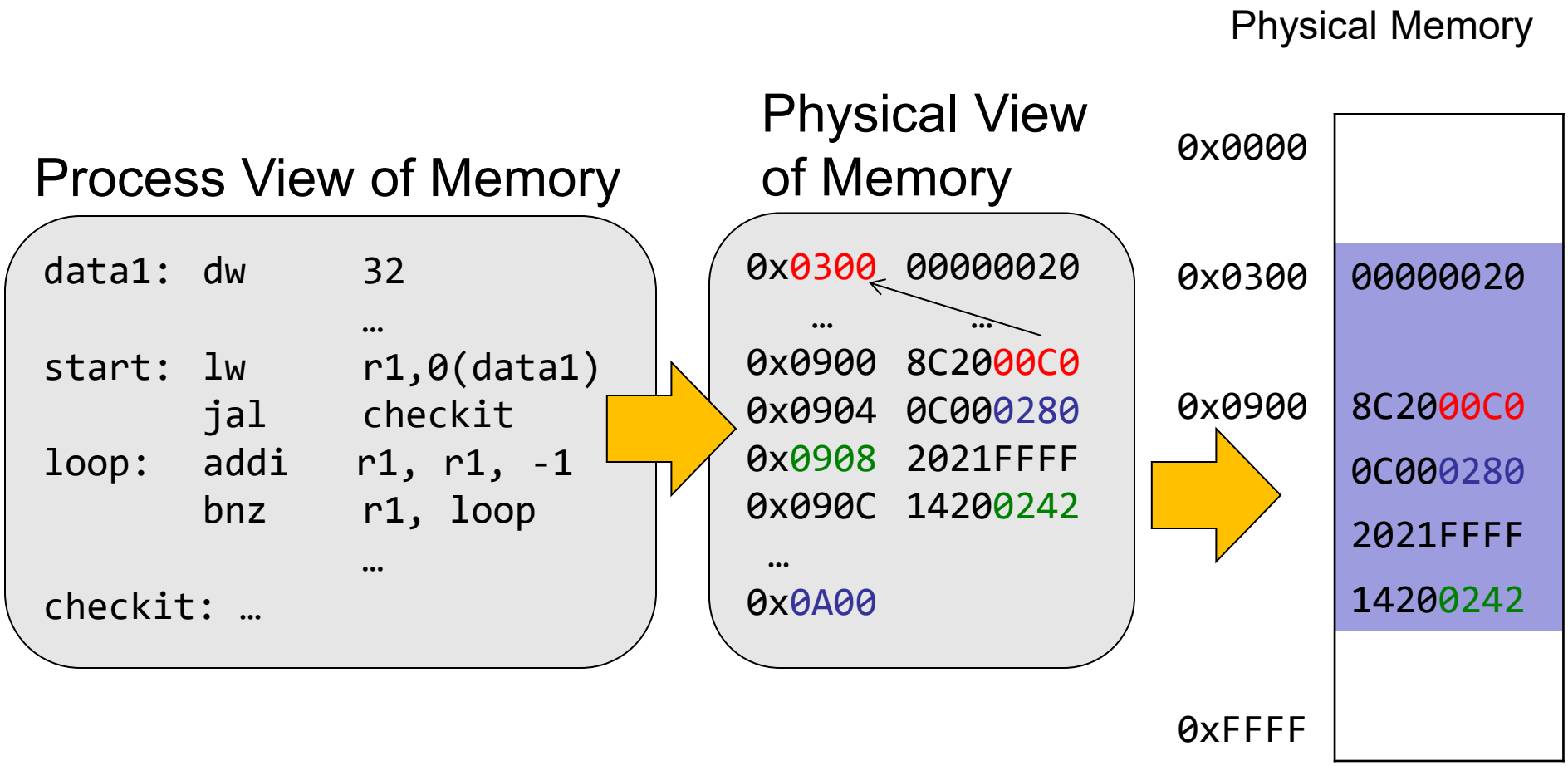


Physical View of Memory

```
0x0300  00000020
   …        …
0x0900  8C2000C0
0x0904  0C000280
0x0908  2021FFFF
0x090C  14200242
 …
0x0A00
```

Assume 4 Byte words
0x300 = 4 x 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000

Binding of Instructions and Data to Memory



Second copy of the program

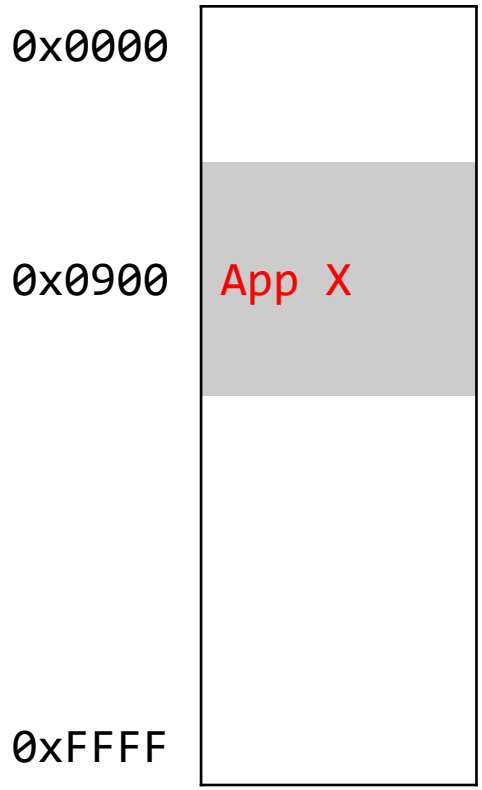
Physical Memory

Process View of Memory

```
data1:  dw      32
                …
start:  lw      r1,0(data1)
        jal     checkit
loop:   addi    r1, r1, -1
        bnz     r1, loop
                …
checkit: …
```

Physical View of Memory

```
0x0300  00000020
   …        …
0x0900  8C2000C0
0x0904  0C000280
0x0908  2021FFFF
0x090C  14200242
 …
0x0A00
```



What do we do with the addresses? Need translation!

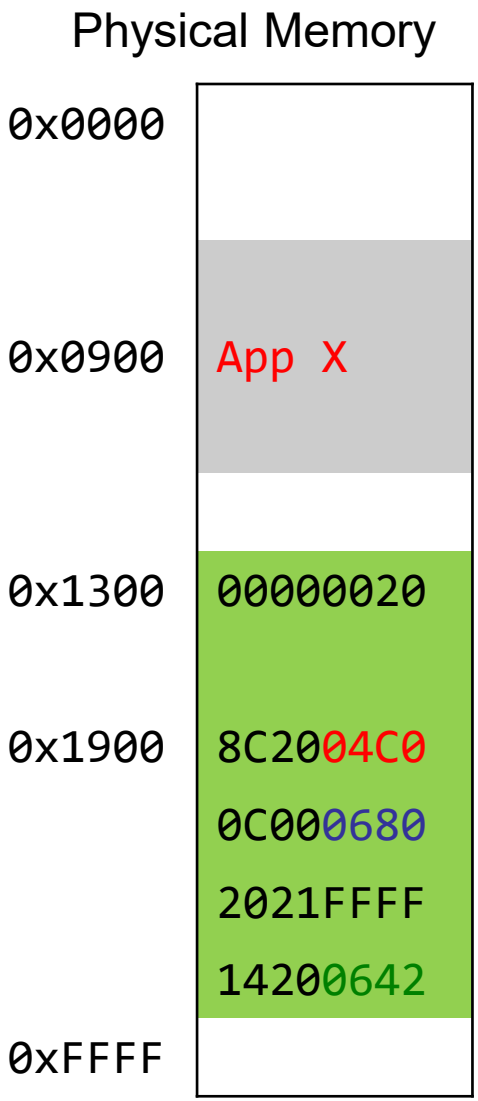
Second copy of the program

Process View of Memory

```
data1:  dw      32
        …
start:  lw      r1,0(data1)
        jal     checkit
loop:   addi    r1, r1, -1
        bnz     r1, loop
        …
checkit: …
```

Physical View of Memory

```
0x1300  00000020
   …       …
0x1900  8C2004C0
0x1904  0C000680
0x1908  2021FFFF
0x190C  14200642
 …
0x1A00
```



- One of many possible translations!
- When does translation take place?
  - Compile time, Link/Load time, or Execution time?

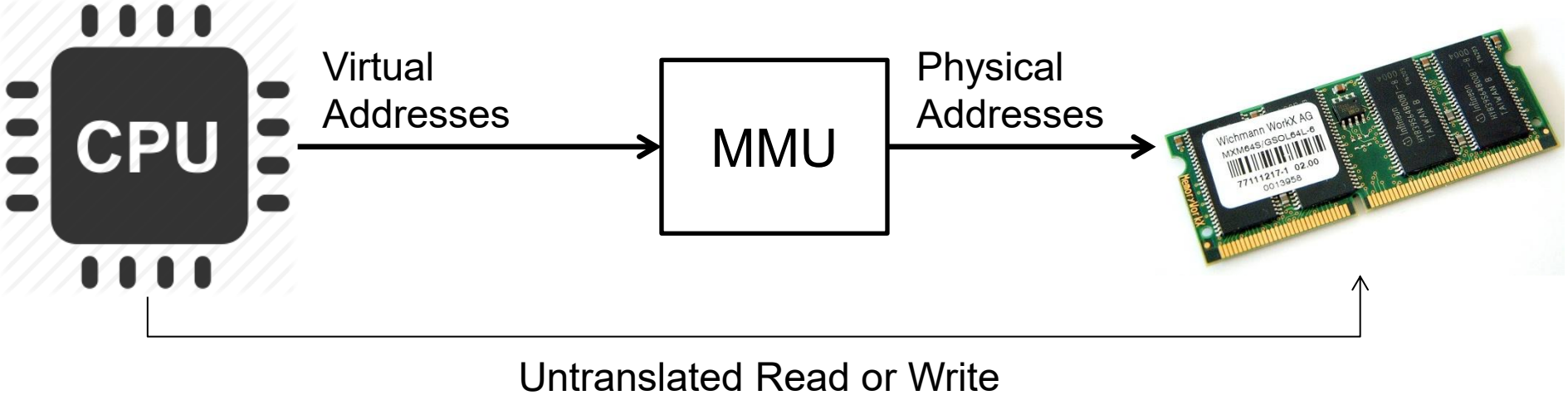
So Far

- Starvation and Deadlock
- Address Spaces and Segmentation
  - Loading and Translating
  - Segments
  - Fragmentation

General Address translation (Review)

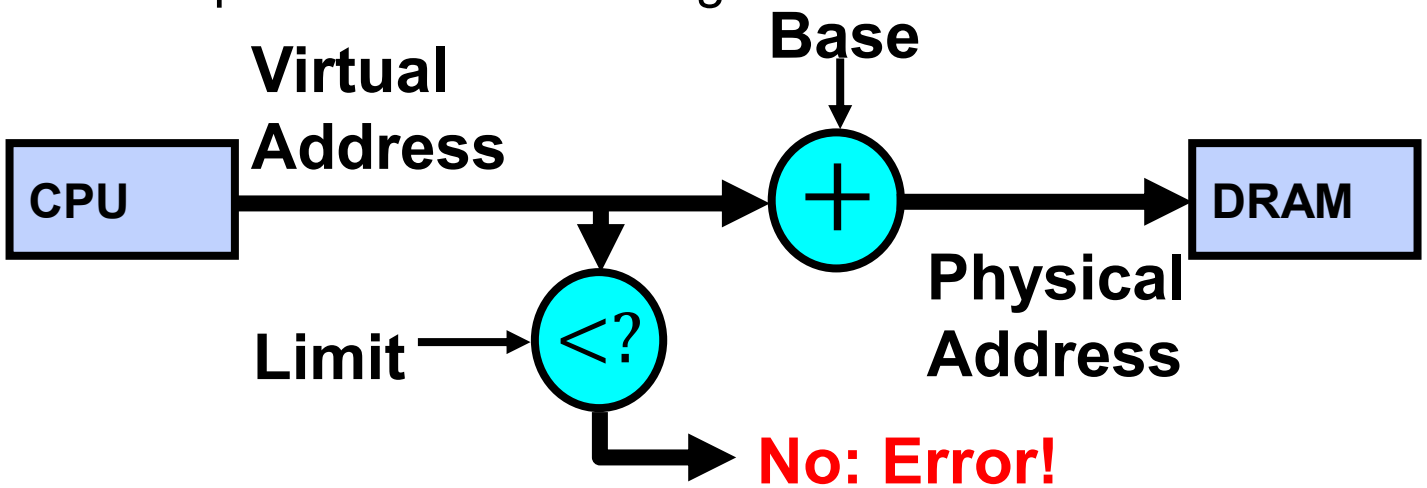
- Address Space: All the addresses and state a process can touch
  - Each process and kernel have different address spaces
- Therefore → Two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (MMU) converts between the two views
- Translation makes it much easier to implement protection
  - If Job A cannot even gain access to Job B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into the same region of user address space

General Address translation (Review)

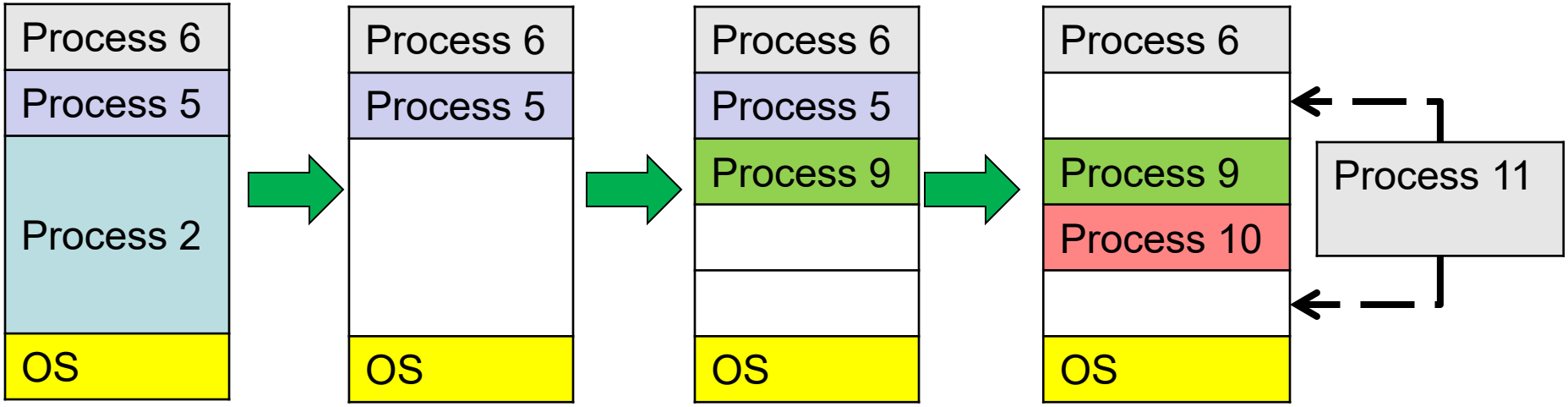


Simple Example: Base and Bounds (CRAY-1)

- Base + limit for dynamic address translation (runtime)
  - Alter address of every load/store by adding “base”
  - Error if address > limit
- Gives program illusion it is running alone, with memory starting at `0`
  - Program gets continuous region of memory
  - Addresses in program do not need relocation when program placed in different region of DRAM



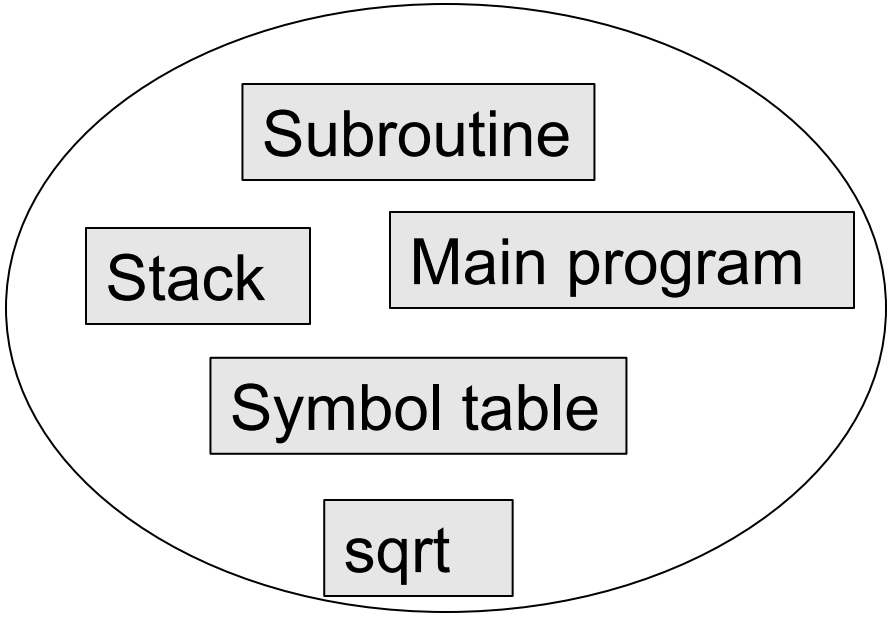
Issues with Simple B&B Method



- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- Missing support for sparse address space
  - Would like to have multiple chunks per program
  - Ex: Code, Data, Stack
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process

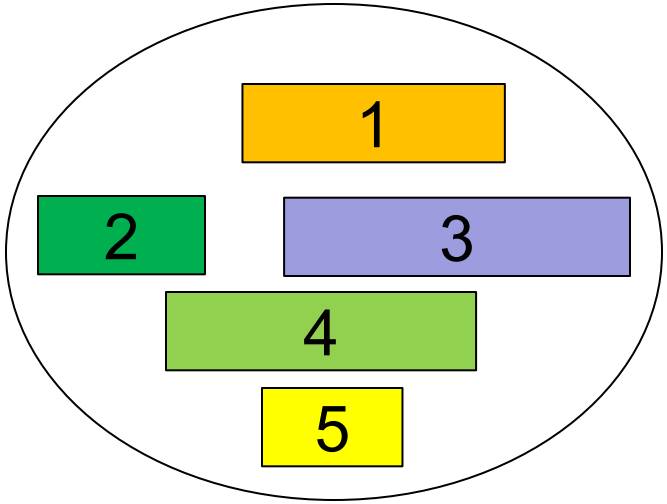
More Flexible Segmentation

- Logical View: Multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

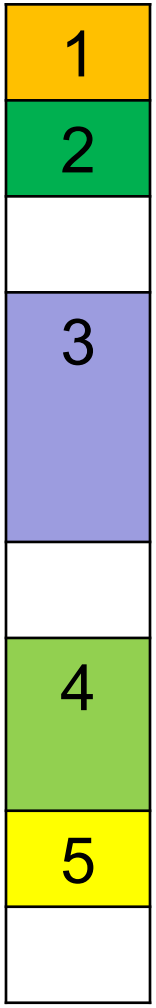


Logical Addresses

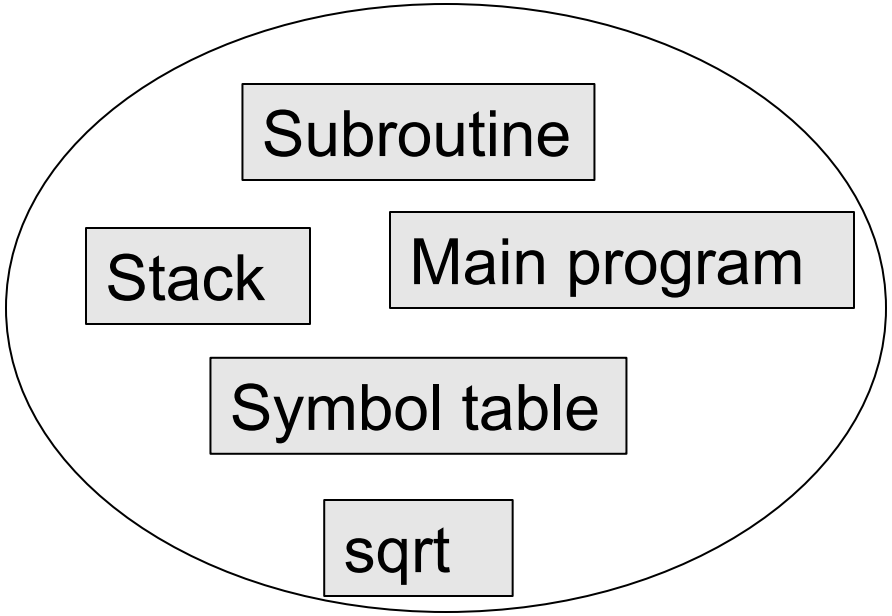
More Flexible Segmentation



User view
of memory space



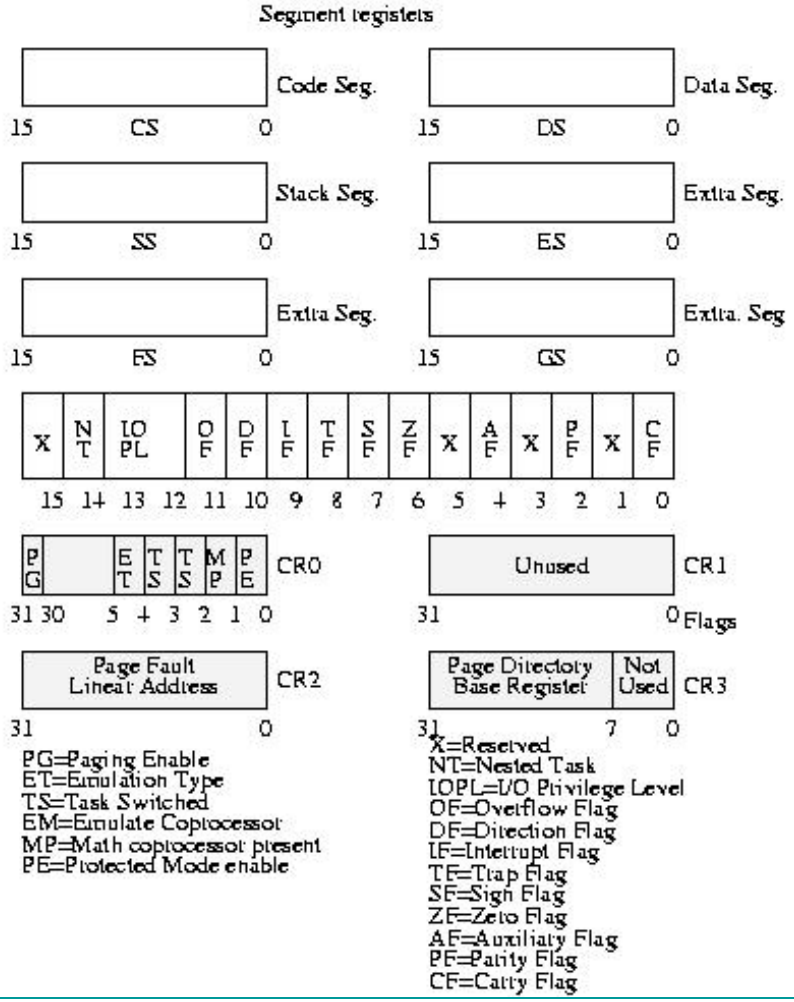
Physical memory space



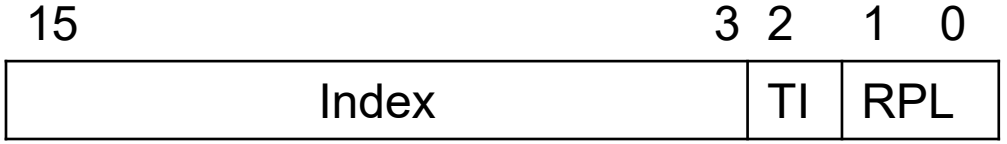
Logical Addresses

Intel x86 Special Registers

80386 Special Registers



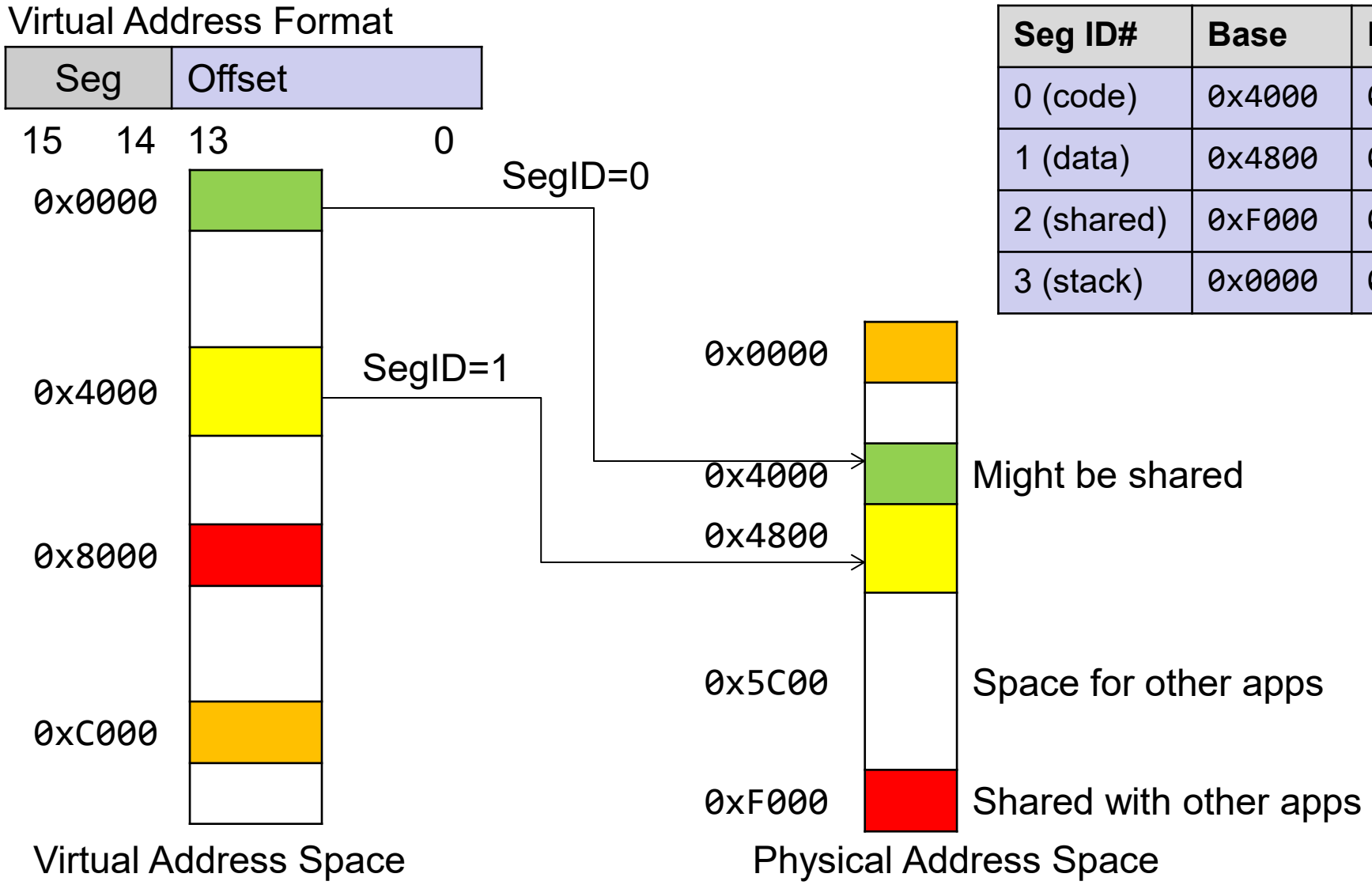
Typical Segment Register Current Priority is RPL of Code Segment (CS)



- RPL: Requestor Privilege Level
- TI: Table Indicator
  - 0 = GDT, 1=LDT
- Index: Index into table

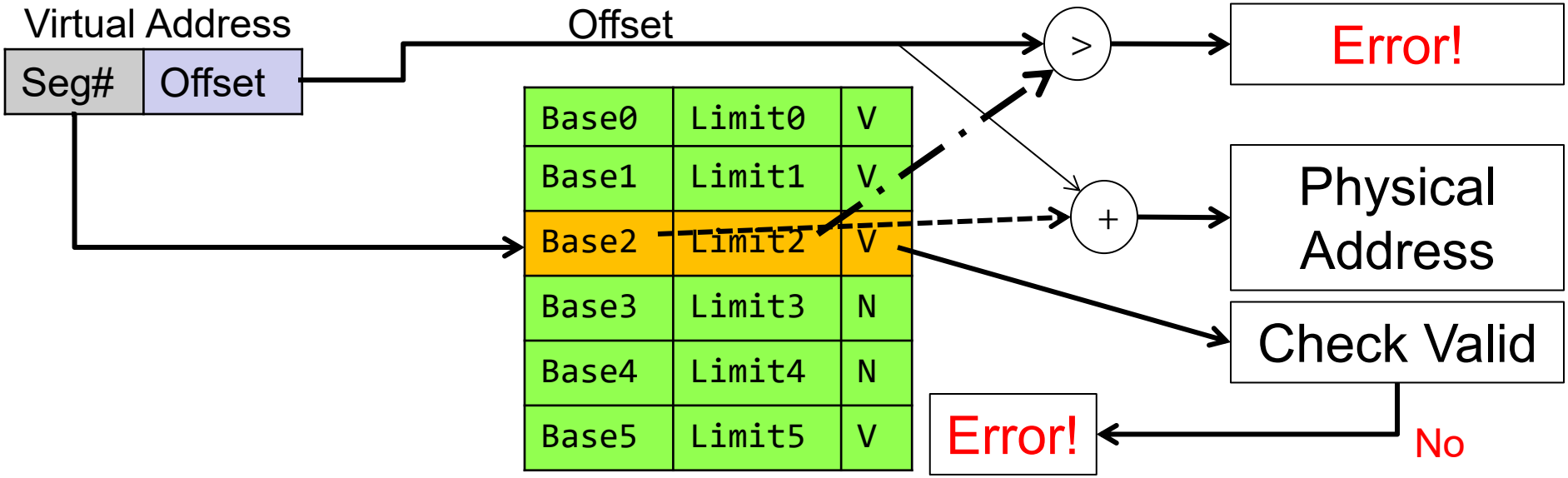
- Protected memory segment selector

Ex: 4 Segments (16b addresses)



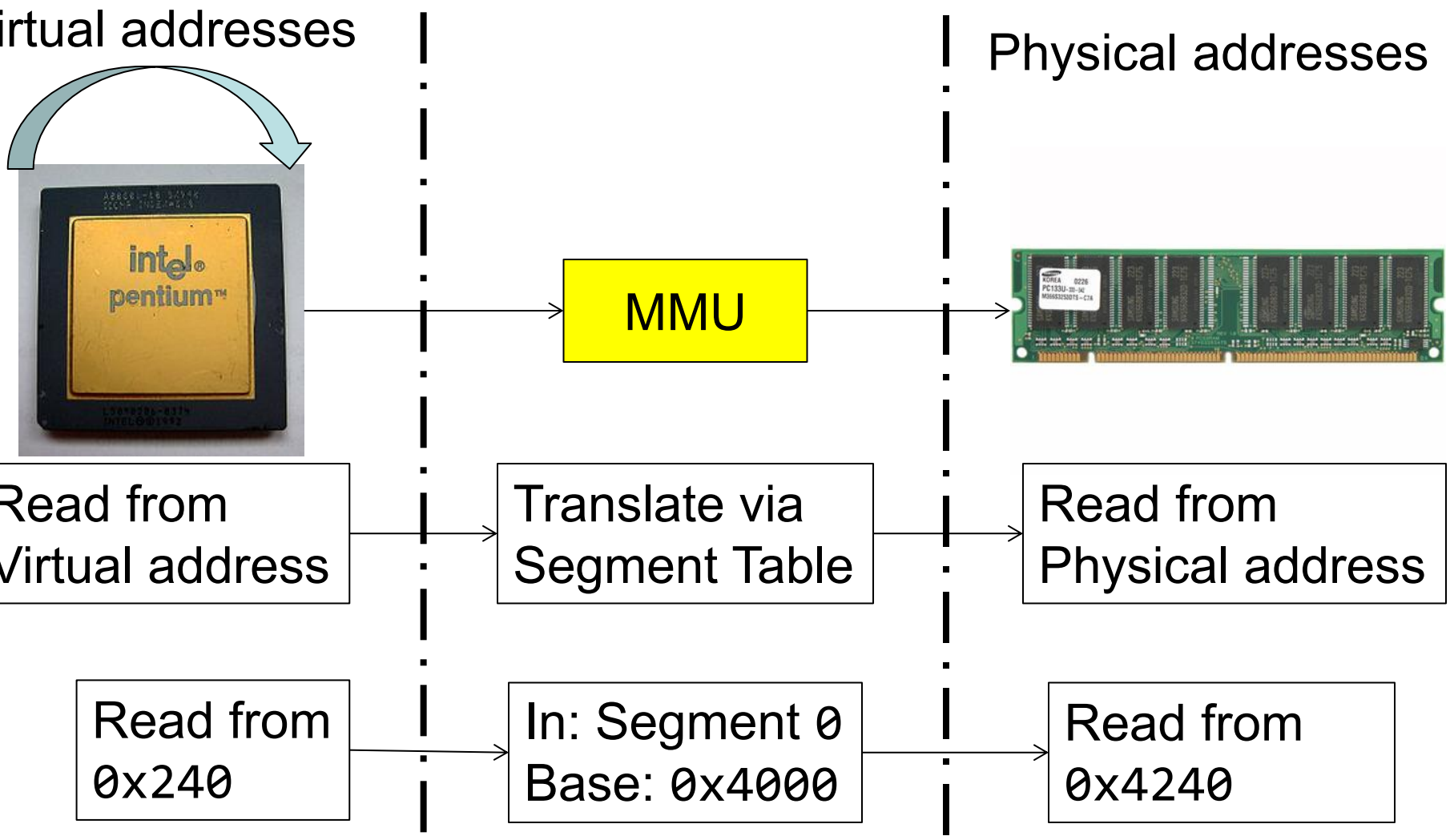
Seg ID#	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Implementation of Multi-Segment Model



- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - x86 Example: `mov [es:bx],ax.`
- What is `V/N` (valid / not valid)?
  - Can mark segments as invalid; requires check as well

Virtual versus Physical

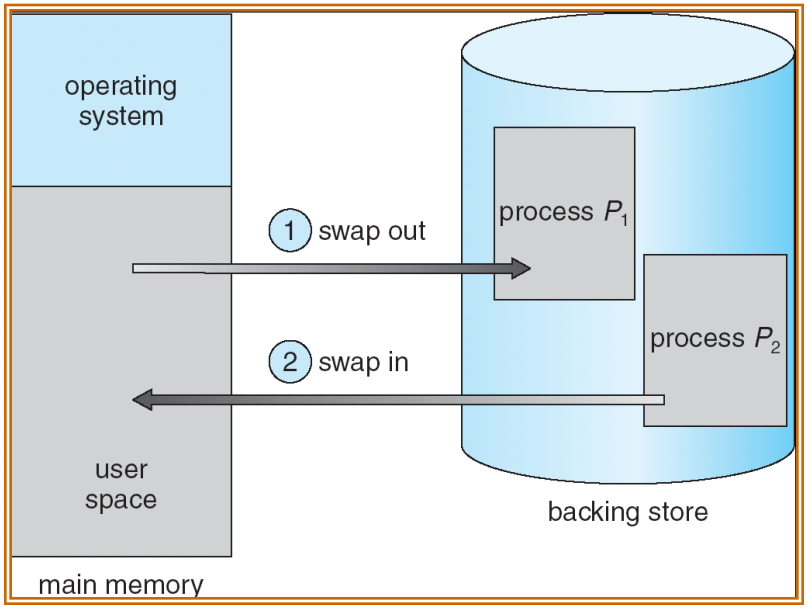


Running more programs than fit in memory: Swapping

Q: What if not all processes fit in memory?

A: Swapping: Extreme form of Context Switch

- To make room for next process, some or all the previous process is moved to disk
- Greatly increases the cost of context switching

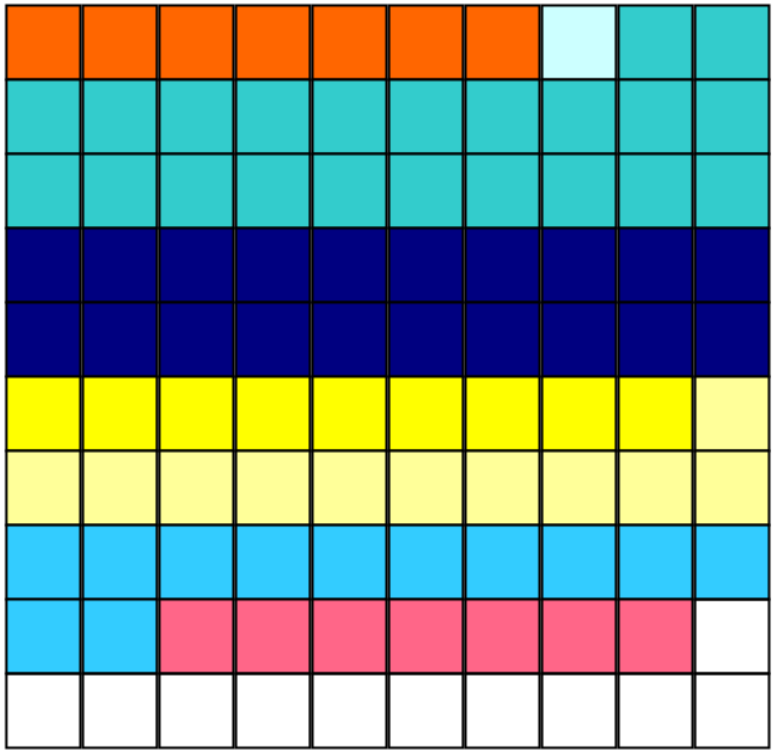


What else could we do?

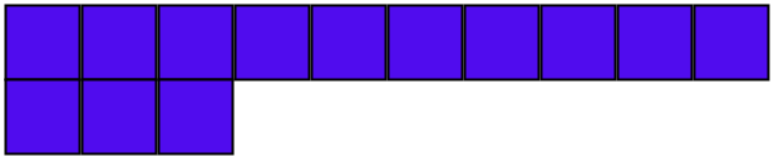
- Keep only active portions of a process in memory
- Need finer granularity control over physical memory

Swapping imagined

In Memory



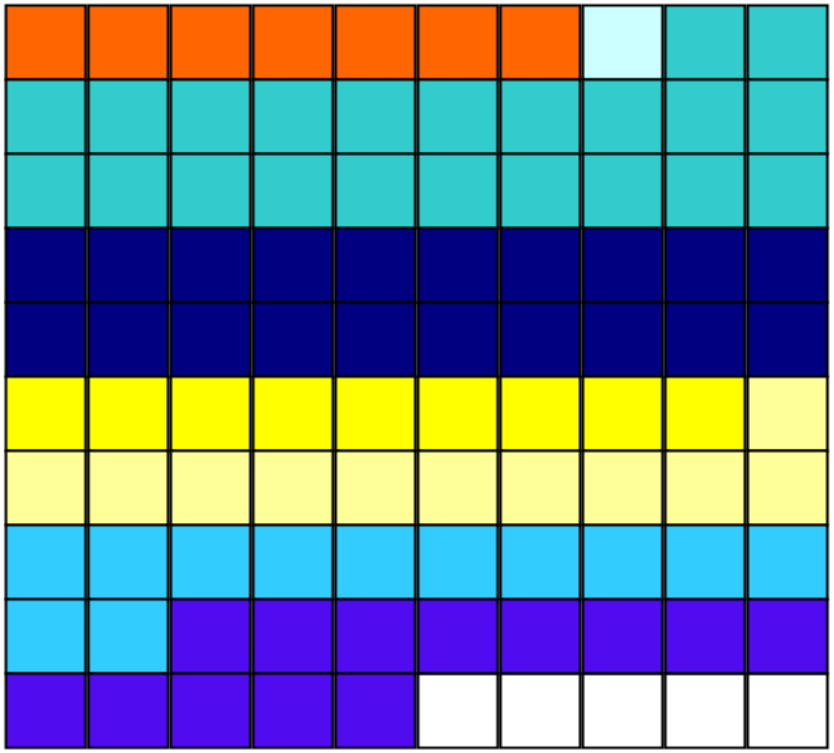
Segment to load



Free = white blocks

Swapping imagined

In Memory



Free = white blocks

Stored on disk

Memory Layout for Linux 32-bit

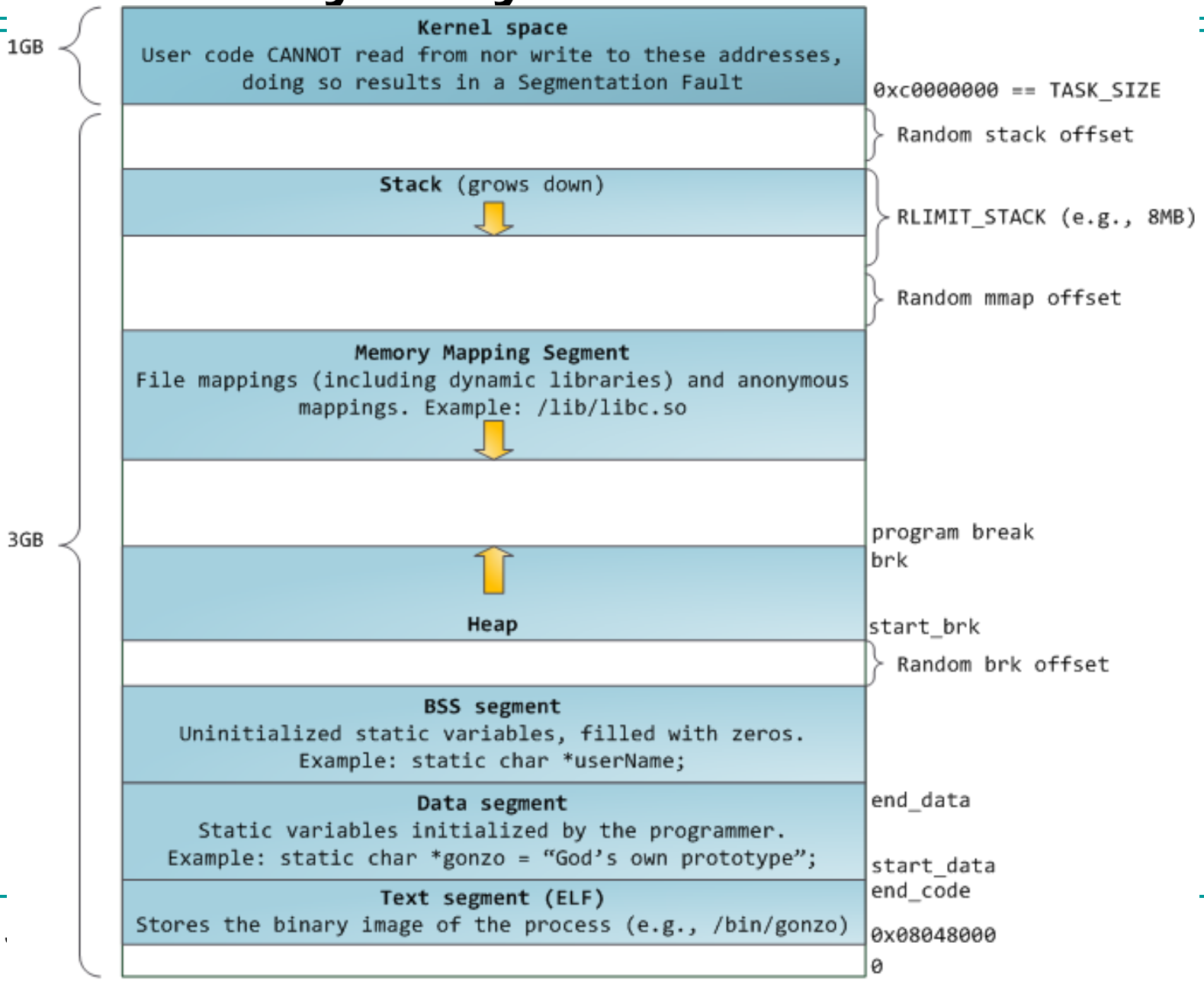


Image source: http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png

Problems with Segmentation

Must fit variable-sized chunks into physical memory

May move processes multiple times to fit everything

Limited options for swapping to disk

Fragmentation: wasted space

- External: free gaps between allocated chunks
- Internal: don't need all memory within allocated chunks

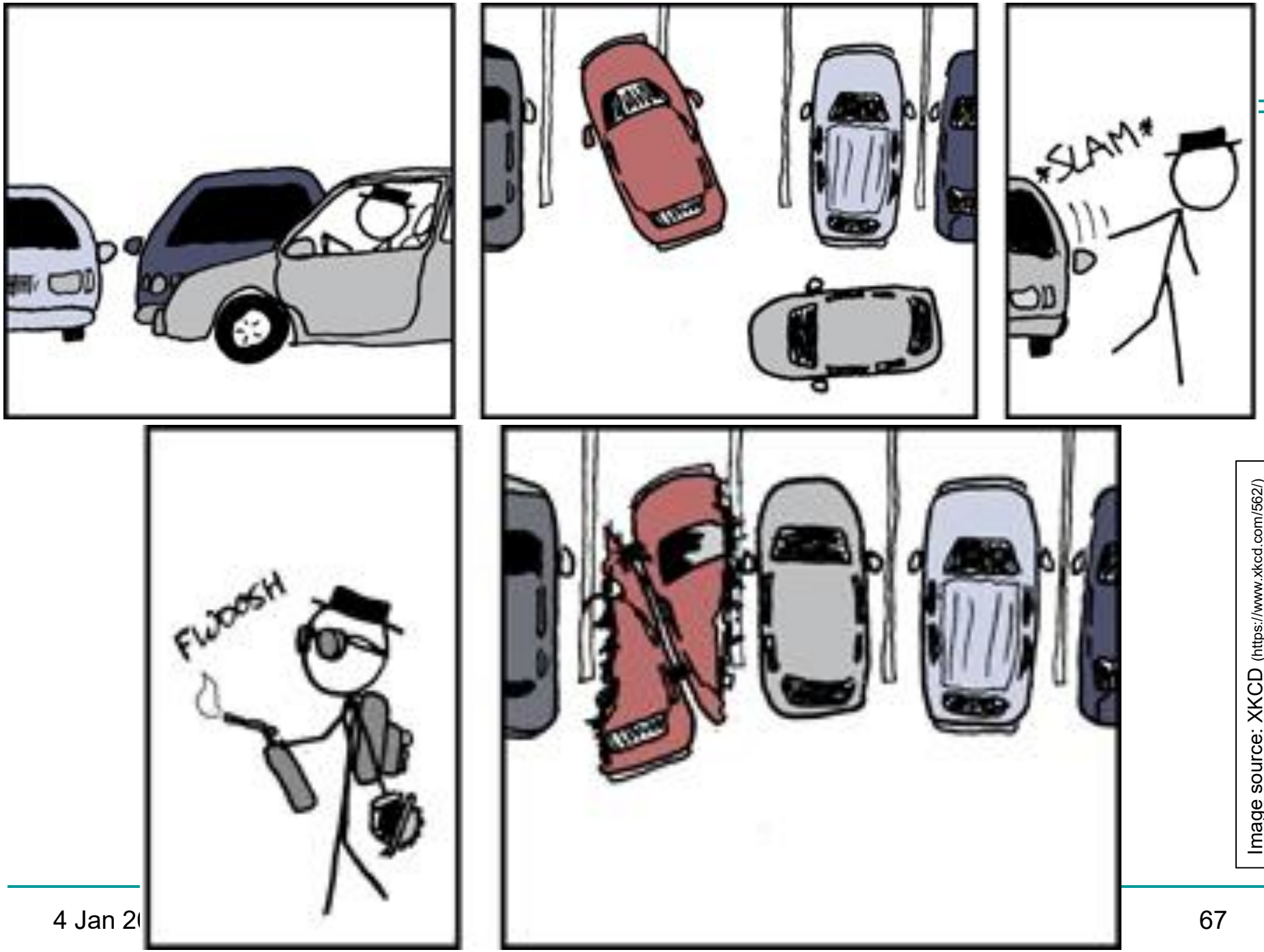


Image source: XKCD (https://www.xkcd.com/562/)

Image source: http://imgur.com/OFEy7

Conclusion

- Starvation and Deadlock
- Address Spaces and Segmentation
  - Loading and Translating
  - Segments
  - Fragmentation