

# CSI62 Operating Systems and Systems Programming Lecture 11

## Scheduling (finished), Deadlock, Address Translation

February 26<sup>th</sup>, 2018

Profs. Anthony D. Joseph and Jonathan Ragan-Kelley  
<http://cs162.eecs.Berkeley.edu>

## Recall: 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!!!

2/26/18

CSI62 ©UCB Fall 2018

Lec 11.2

## Recall: Real-Time Scheduling (RTS)

- Hard Real-Time
  - *Attempt to meet all deadlines*
  - EDF (Earliest Deadline First), LLF (Least Laxity First), RM (Rate-Monotonic), DM (Deadline Monotonic)
- 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)

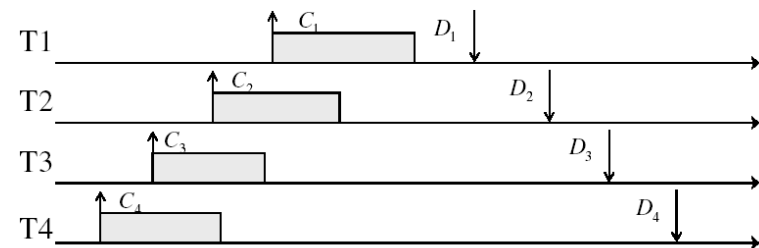
2/26/18

CSI62 ©UCB Fall 2018

Lec 11.3

## Example: Workload Characteristics

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

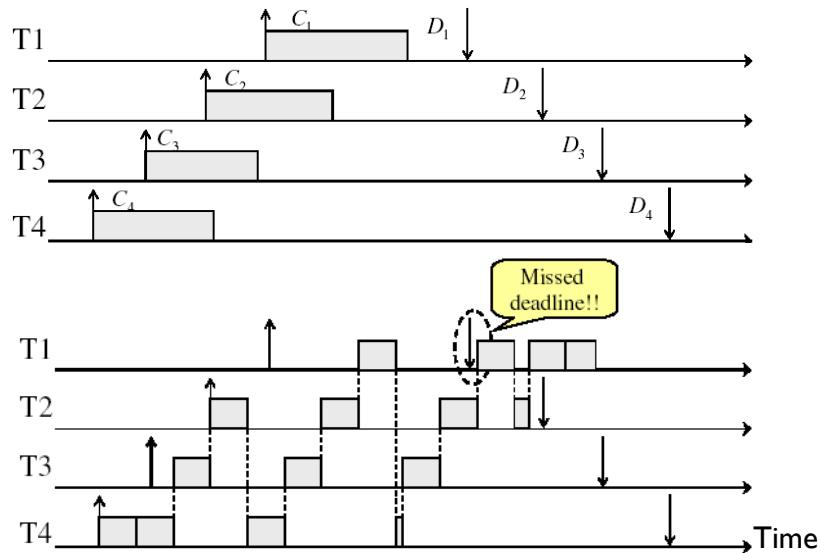


2/26/18

CSI62 ©UCB Fall 2018

Lec 11.4

## Example: Round-Robin Scheduling Doesn't Work



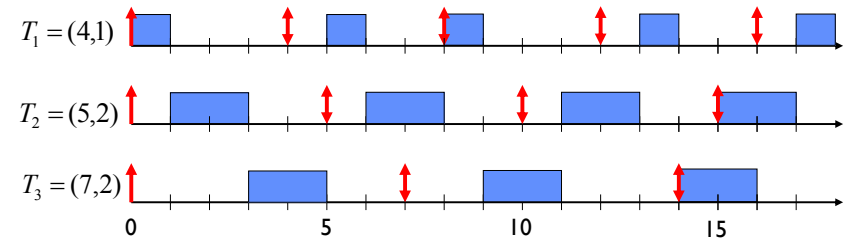
2/26/18

CSI62 ©UCB Fall 2018

Lec 11.5

## Earliest Deadline First (EDF)

- Tasks periodic with period  $P$  and computation  $C$  in each period:  $(P, C)$
- Preemptive priority-based dynamic scheduling
- Each task is assigned a (current) priority based on how close the absolute deadline is
- The scheduler always schedules the active task with the closest absolute deadline



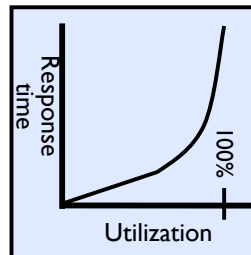
2/26/18

CSI62 ©UCB Fall 2018

Lec 11.6

## A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or ...)
  - One approach: Buy it when it will pay for itself in improved response time
    - » Assuming you're paying for worse response time in reduced productivity, customer angst, etc...
    - » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization  $\rightarrow$  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



2/26/18

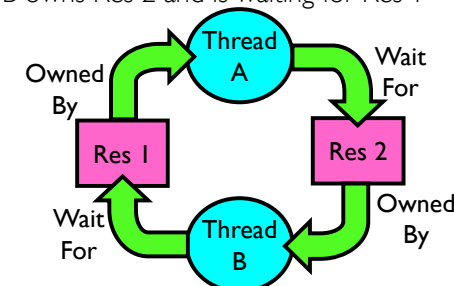
CSI62 ©UCB Fall 2018

Lec 11.7

## 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

2/26/18

CSI62 ©UCB Fall 2018

Lec 11.8

## Conditions for Deadlock

- Deadlock not always deterministic – Example 2 mutexes:

### Thread A

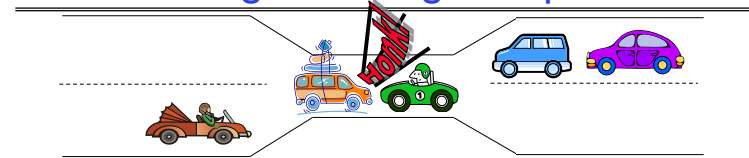
```
x.P();
y.P();
y.V();
x.V();
```

### Thread B

```
y.P();
x.P();
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 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

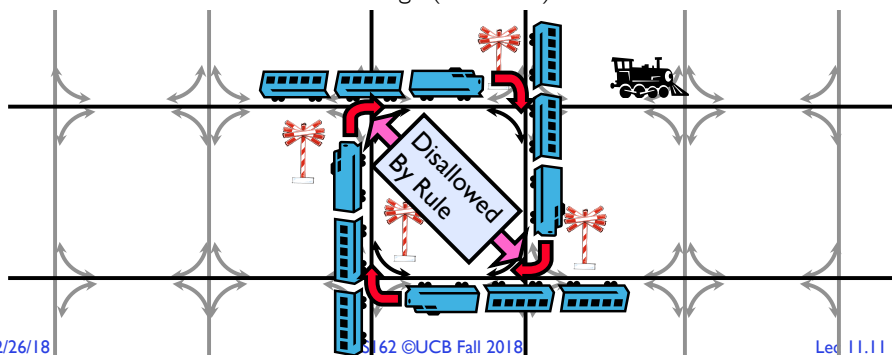
## Bridge Crossing Example



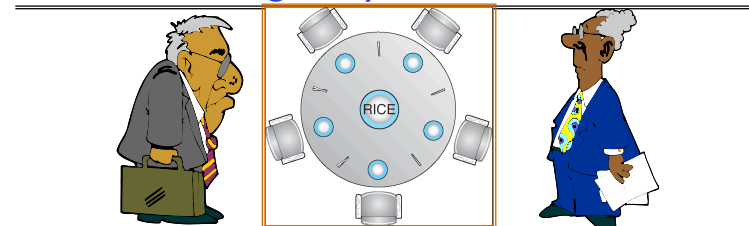
- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires 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  $\Rightarrow$  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  $\{T_1, \dots, T_n\}$  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  $T_1$

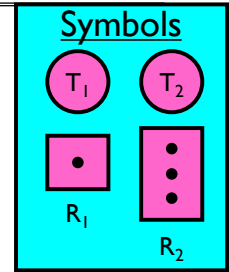
2/26/18

CSI62 ©UCB Fall 2018

Lec 11.13

## Resource-Allocation Graph

- System Model
  - A set of Threads  $T_1, T_2, \dots, T_n$
  - Resource types  $R_1, R_2, \dots, R_m$ 
    - 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:
    - »  $T = \{T_1, T_2, \dots, T_n\}$ , the set threads in the system.
    - »  $R = \{R_1, R_2, \dots, R_m\}$ , the set of resource types in system
  - request edge – directed edge  $T_i \rightarrow R_j$
  - assignment edge – directed edge  $R_j \rightarrow T_i$



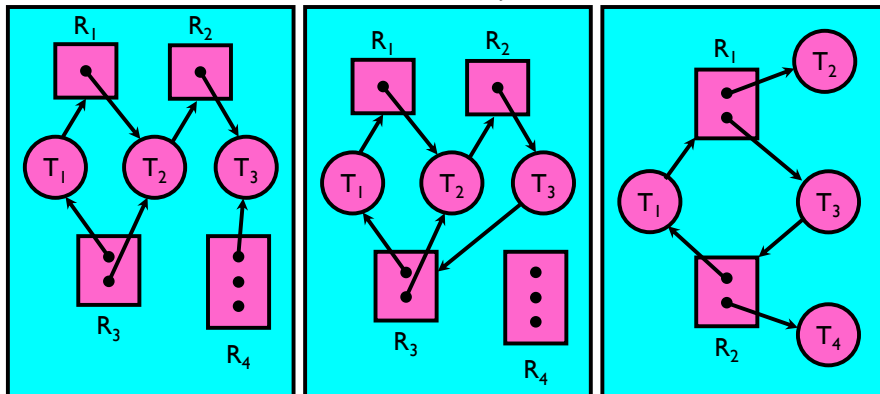
2/26/18

CSI62 ©UCB Fall 2018

Lec 11.14

## 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

2/26/18

CSI62 ©UCB Fall 2018

Lec 11.15

## Methods for Handling Deadlocks



- 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

2/26/18

CSI62 ©UCB Fall 2018

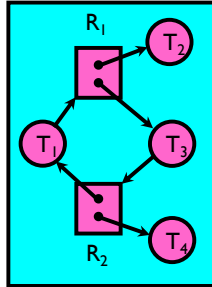
Lec 11.16

## Deadlock Detection Algorithm

- Only one of each type of resource  $\Rightarrow$  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
    - $[Request_x]$ : 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 nodes to UNFINISHED
do {
  done = true
  Foreach node in UNFINISHED {
    if ([Requestnode] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Allocnode]
      done = false
    }
  }
} until(done)
```

- Nodes left in **UNFINISHED**  $\Rightarrow$  deadlocked



2/26/18

CSI62 ©UCB Fall 2018

Lec 11.17

## 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

2/26/18

CSI62 ©UCB Fall 2018

Lec 11.18

## Administrivia

- Midterm I: 6:30-8:30pm, this Wednesday
  - Li Ka Shing 245: IDs ending in 0,1,2,3
  - Hearst Field Annex A1: ...4,5,6
  - VLSB 2060: ...7,8
  - Barrows 20: ...9
  - CS 189: Wurster 102 (no electronics)
  - DSP: Soda 465 & 606
- Covers everything through lecture 10 + project 1
  - Everyone in group should understand all parts of the project!
- Project I: code due Friday, 3/2

2/26/18

CSI62 ©UCB Fall 2018

Lec 11.19

# BREAK

2/26/18

CSI62 ©UCB Fall 2018

Lec 11.20

## 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!

2/26/18

CSI62 ©UCB Fall 2018

Lec 11.21

## 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
  - Example (x.P, y.P, z.P,...)
    - » Make tasks request disk, then memory, then...
    - » Keep from deadlock on freeways around SF by requiring everyone to go clockwise

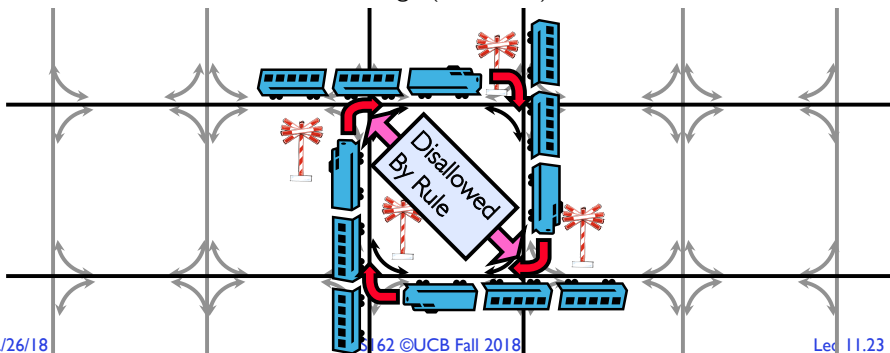
2/26/18

CSI62 ©UCB Fall 2018

Lec 11.22

## 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)



2/26/18

CSI62 ©UCB Fall 2018

Lec 11.23

## Banker's Algorithm for Preventing Deadlock

- Toward right idea:
  - State maximum (max) 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
 
$$([Max_{node}] - [Alloc_{node}] \leq [Avail]) \text{ for } ([Request_{node}] \leq [Avail])$$
 Grant request if result is deadlock free (conservative!)



2/26/18

CSI62 ©UCB Fall 2018

Lec 11.24

## Banker's Algorithm for Preventing Deadlock

- ```

[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
    done = true
    Foreach node in UNFINISHED {
        if ([Requestnode] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Allocnode]
            done = false
        }
    }
} until(done)

```



» 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

- ```

[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
    done = true
    Foreach node in UNFINISHED {
        if ( $[Max_{node}] - [Alloc_{node}] <= [Avail]$ ) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Allocnode]
            done = false
        }
    }
} until(done)

```



» 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!)

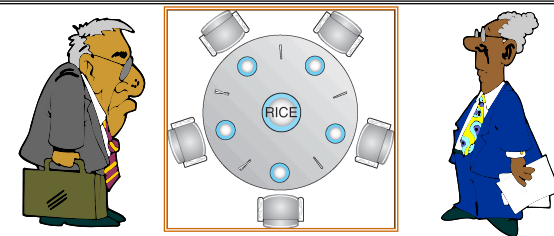
## Banker's Algorithm for Preventing Deadlock

- 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  $([Max_{node}] - [Alloc_{node}] <= [Avail])$  for  $([Request_{node}] <= [Avail])$   
Grant request if result is deadlock free (conservative!)
    - » Keeps system in a "SAFE" state, i.e. there exists a sequence  $\{T_1, T_2, \dots, T_n\}$  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

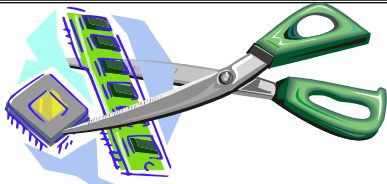


## Banker's Algorithm Example

- 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 2<sup>nd</sup> to last, and no one would have k-1
    - » It's 3<sup>rd</sup> to last, and no one would have k-2
    - » ...



## Virtualizing Resources



- Physical Reality:
 Different Processes/Threads share the same hardware
  - Need to multiplex CPU (Just finished: scheduling)
  - Need to multiplex use of Memory (starting today)
  - Need to multiplex disk and devices (later in term)
- 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)

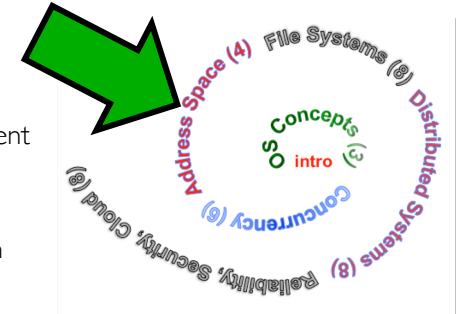
2/26/18

CSI62 ©UCB Fall 2018

Lec 11.29

## Next Objective

- Dive deeper into the concepts and mechanisms of memory sharing and address translation
- Enabler of many key aspects of operating systems
  - Protection
  - Multi-programming
  - Isolation
  - Memory resource management
  - I/O efficiency
  - Sharing
  - Inter-process communication
  - Debugging
  - Demand paging
- Today: Translation

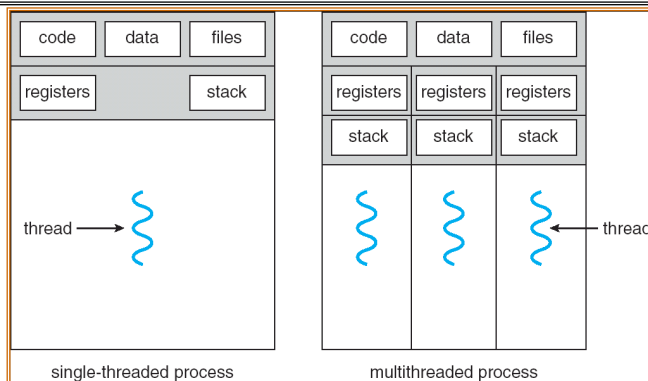


2/26/18

CSI62 ©UCB Fall 2018

Lec 11.30

## Recall: Single and Multithreaded Processes



- Threads encapsulate concurrency
  - “Active” component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - “Passive” component of a process

2/26/18

CSI62 ©UCB Fall 2018

Lec 11.31

## Important Aspects of Memory Multiplexing

- 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
- Controlled overlap:**
  - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  - Conversely, would like the ability to overlap when desired (for communication)
- Translation:**
  - 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

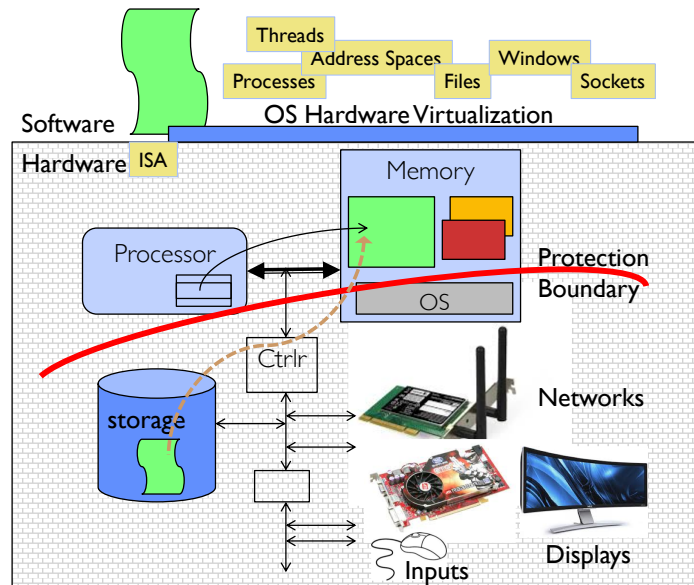
2/26/18

CSI62 ©UCB Fall 2018

Lec 11.32



## Recall: Loading

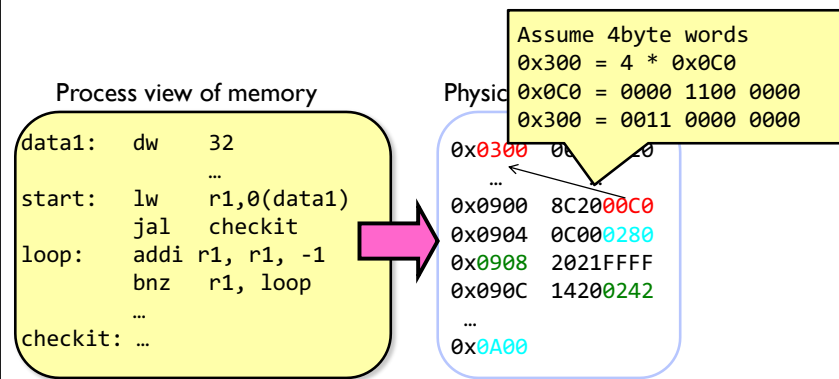


2/26/18

CSI62 ©UCB Fall 2018

Lec 11.33

## Binding of Instructions and Data to Memory

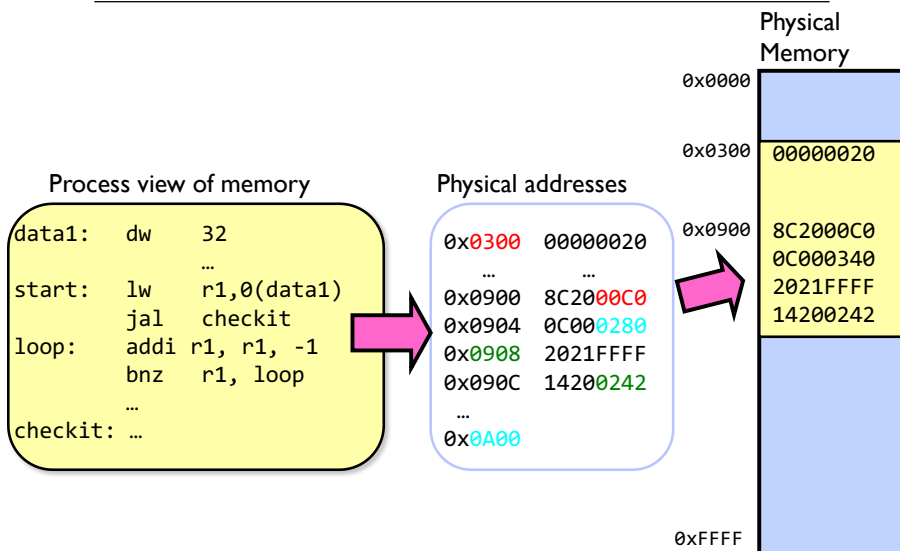


2/26/18

CSI62 ©UCB Fall 2018

Lec 11.34

## Binding of Instructions and Data to Memory

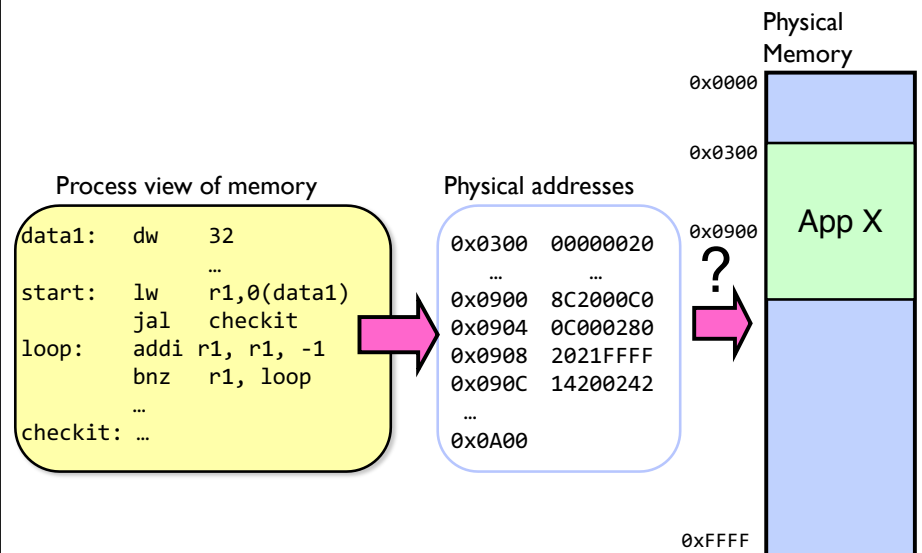


2/26/18

CSI62 ©UCB Fall 2018

Lec 11.35

## Second copy of program from previous example



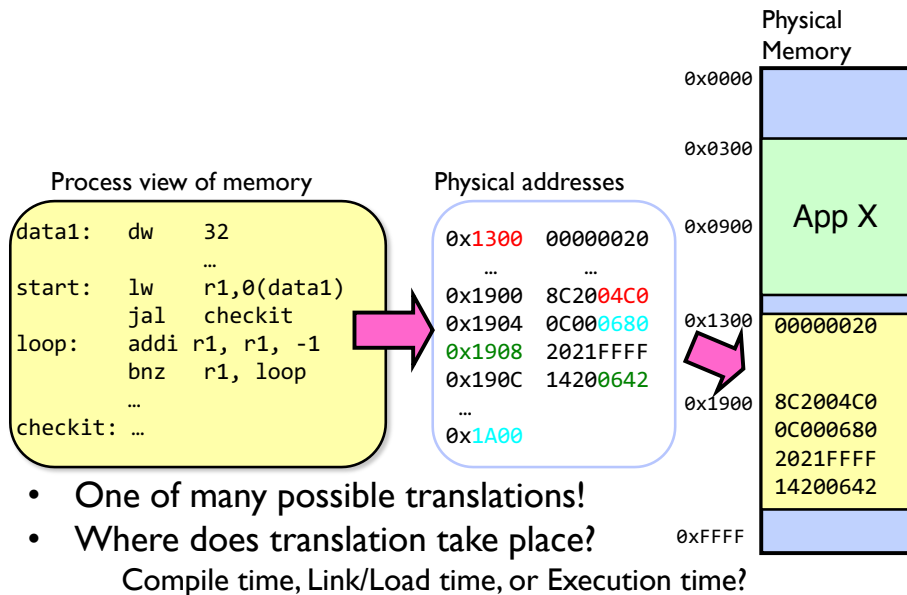
Need address translation!

2/26/18

CSI62 ©UCB Fall 2018

Lec 11.36

## Second copy of program from previous example



## Summary

- Real-time scheduling
  - Need to meet a deadline, predictability essential
  - Earliest Deadline First (EDF) and Rate Monotonic (RM) scheduling
- Four requirements for deadlock:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Simple Protection through segmentation
  - Base + Limit registers restrict memory accessible to user
  - Can be used to translate as well