CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Spring 2021 L15

Deadlocks

UTAs: Undergraduate Teaching Assistants

Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

FAQ

- What is the meaning of life? One answer video
- What are resources? Drives, memory blocks, locks for critical sections, etc.
- Why modern OSs do not actively prevent deadlocks?
  - If a process does not progress for a few seconds, the system may generate a message. ".. not responding"
  - May be used by embedded/rear-time OSs,
  - data-bases, locked files may be checked for deadlocks
  - Some version of Windows and Linux may have capability to check for deadlocks involving locks for critical sections.
  - Use of mechanism by which locks are always acquired in a defined order

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
    - ensuring that at least one of the 4 conditions cannot hold
  - Deadlock avoidance
    - Dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Allow the system to enter a deadlock state
  - Detect and then recover. Hope is that it happens rarely.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock.

Mutual exclusion: only one process at a time can use a resource
Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
Circular wait: there exists a set $\{P_0, P_1, …, P_n\}$ of waiting processes that are circularly waiting.

Colorado State University

Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- each process declares the ***maximum number*** of resources of each type that it may need
- *Resource-allocation state* is defined by the number of available and allocated resources, and the maximum demands of the processes
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
  - Ensures all allocations result in a safe state

Safe Sequence

System must decide if immediate allocation leaves the system in a safe state

System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes such that

- for each P_i, the resources that P_i can still request can be satisfied by
  - currently available resources +
  - resources held by all the P_j, with $j < i$
  - That is
    - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished and released resources
    - When P_i terminates, P_{i+1} can obtain its needed resources, and so on
- If no such sequence exists: system state is **unsafe**

Example A: Assume **12 Units** in the system

	Max need (initially declared)	Current holding at time T0
P0	10	5
P1	4	2
P2	9	2

At T0:
9 units allocated
12-9 = **3 units available**

A unit could be a drive, a block of memory etc.

- Is the system in a safe state at time **T0**?
  - Try sequence <P1, P0 , P2>
  - P1 can be given 2 units
  - When P1 releases its resources; there are 5 units available
  - P0 uses 5 and subsequently releases them (10 available now)
  - P2 can then proceed.
- Thus <P1, P0 , P2> is a safe sequence, and at T0 system was in a safe state

More detailed look

Colorado State University

Avoidance Algorithms

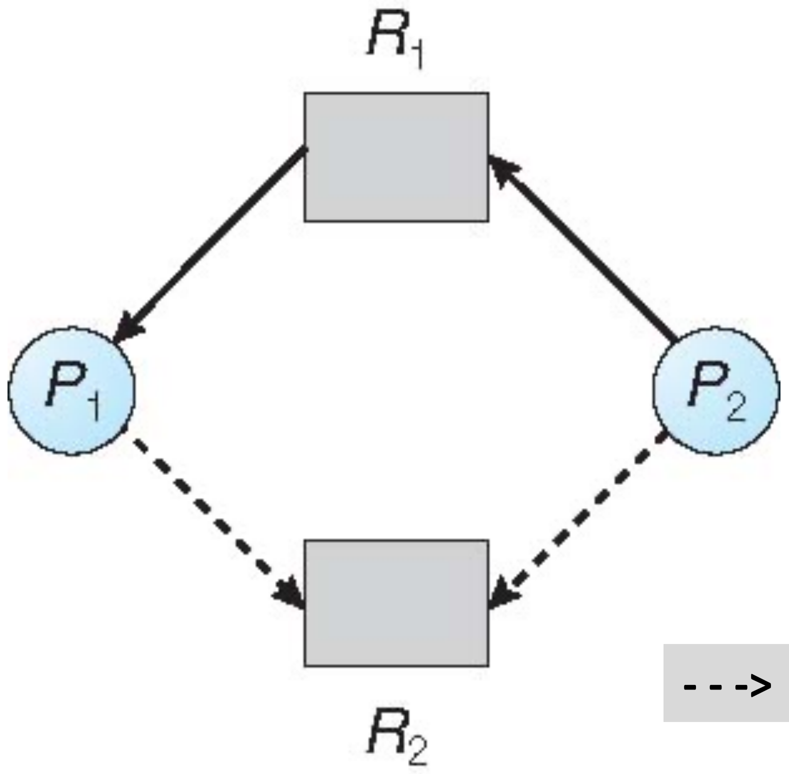
- Single instance of a resource type
  - Use a resource-allocation graph scheme

- Multiple instances of a resource type
  - Use the banker's algorithm (Dijkstra)

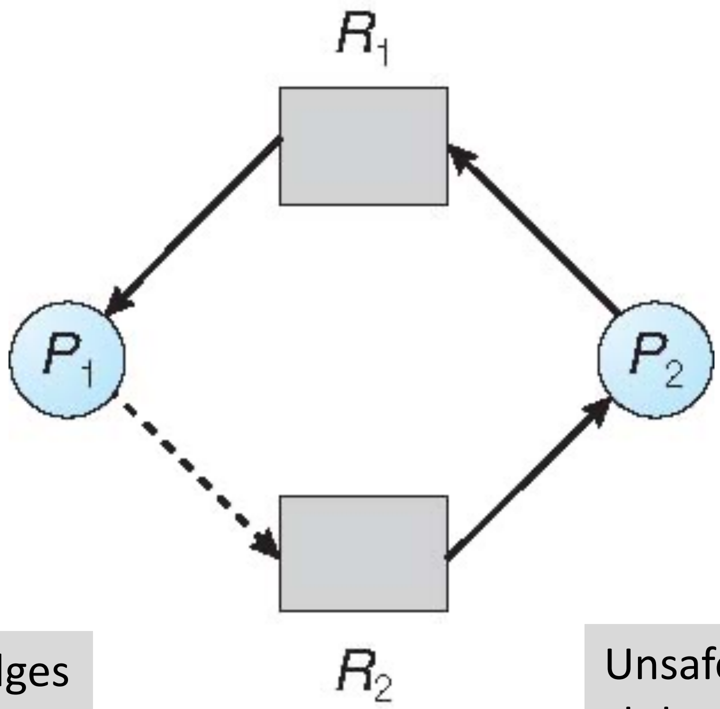
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j; represented by a dashed line. This is new.
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Requirement: Resources must be claimed *a priori* in the system

Resource-Allocation Graph



- - -> Claim edges



Unsafe state

Suppose *P2* requests *R2.* Can R2 be allocated to P2?
Although *R2* is currently free, we cannot allocate it to *P2,* since this action will create a cycle getting system in an unsafe state. If *P1* requests *R2,* and *P2* requests *R1,* then a deadlock will occur.

Banker’s Algorithm: examining a request

- Multiple instances of resources.
- Each process must a priori claim maximum use
- When a process requests a resource,
  - it may have to wait until the resource becomes available (resource request algorithm)
  - Request should not be granted if the resulting system state is unsafe (safety algorithm)
- When a process gets all its resources it must return them in a finite amount of time
- Modeled after a banker in a small-town making loans.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available

Processes vs resources:

- **Max**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

Safety Algorithm: Is this a safe state?

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work* = Initially *Available resources

   ***Finish* [*i*] = *initially false* for *i* = 0, 1, …, *n*- 1** (processes done)

2. Find a process ***i*** such that both:

   (a) ***Finish* [*i*] = *false***

   (b) ***Need$_i$*** $\leq$ ***Work***

   If no such ***i*** exists, go to step 4

3. ***Work* = *Work* + *Allocation$_i$***

   Finish*[*i*] = *true

   go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

n = number of processes,
m = number of resources types
Need$_i$: additional res needed
Work: res currently free
Finish$_i$: processes finished
Allocation$_i$: allocated to i

Colorado State University

Resource-Request Algorithm for Process P_i

Notation: ***Request***$_i$ = request vector for process $\boldsymbol{P_i}$.

If ***Request***$_i$ **[*j*] = *k*** then process $\boldsymbol{P_i}$ wants ***k*** instances of resource type $\boldsymbol{R_j}$

Algorithm: Should the allocation request be granted?

1. If ***Request***$_i \leq$ ***Need***$_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request***$_i \leq$ ***Available***, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. **Is allocation safe?:** Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

   Available = Available – Request$_i$***;***

   Allocation$_i$ ***= Allocation***$_i$ ***+ Request***$_i$***;***

   Need$_i$ ***= Need***$_i$ ***– Request***$_i$***;***

   - If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$

     Use safety algorithm here
   - If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is preserved.

Colorado State University

Example 1A: Banker's Algorithm

- 5 processes P_0 through P_4;
- 3 resource types: *A* (10 instances), *B* (5 instances), and *C* (7 instances)
- Is it a safe state?

The Need matrix is redundant

Process	Max			Allocation			Need		
type	A	B	C	A	B	C	A	B	C
Currently available				3	3	2			
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	2	0	0	1	2	2
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	0	0	2	4	3	1

Example 1A: Banker's Algorithm

- Is it a safe state?
- Yes, since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria

How did we get to this state?

Process	Max			Allocation			Need		
type	A	B	C	A	B	C	A	B	C
available				3	3	2			
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	2	0	0	1	2	2
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	0	0	2	4	3	1

"Work"

Why did we choose P1?

P1 run to completion. Available becomes [3 3 2]+[2 0 0] = [5 3 2]

P3 run to completion. Available becomes [5 3 2]+[2 1 1] = [7 4 3]

P4 run to completion. Available becomes [7 4 3]+[0 0 2] = [7 4 5]

P2 run to completion. Available becomes [7 4 5]+[3 0 2] = [10 4 7]

P0 run to completion. Available becomes [10 4 7]+[0 1 0] = [10 5 7]

Hence state above is safe.

Colorado State University

Ex 1B: Assume now P_1 Requests (1,0,2)

- Check that ***$Request_i \le Need_i$*** ***and*** $Request_i \le$ Available. (1,0,2) ≤ (3,3,2) → true.
- Check for safety after pretend allocation. P1 allocation would be (2 0 0) + (1 0 2)= 302

Process	Max			Pretend Allocation			Need		
type	A	B	C	A	B	C	A	B	C
Available				**2**	**3**	**0**			
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	**3**	**0**	**2**	**0**	**2**	**0**
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	0	0	2	4	3	1

Sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.

Hence state above is safe, thus the allocation would be safe.

Ex 1C,1D: Additional Requests ..

- Given State is (same as previous slide)

Process	Max			Allocation			Need		
type	A	B	C	A	B	C	A	B	C
available				**2**	**3**	**0**			
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	**3**	**0**	**2**	**0**	**2**	**0**
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	0	0	2	4	3	1

P4 request for (3,3,0): cannot be granted - resources are not available.

P0 request for (0,2,0): cannot be granted since the resulting state is unsafe.

Bankers Algorithm: Practical Issues

- Processes may not know in advance about their maximum resource needs
- Number of processes is not fixed
  - Varies dynamically
- Resources thought to be available can disappear
- Few systems use this algorithm

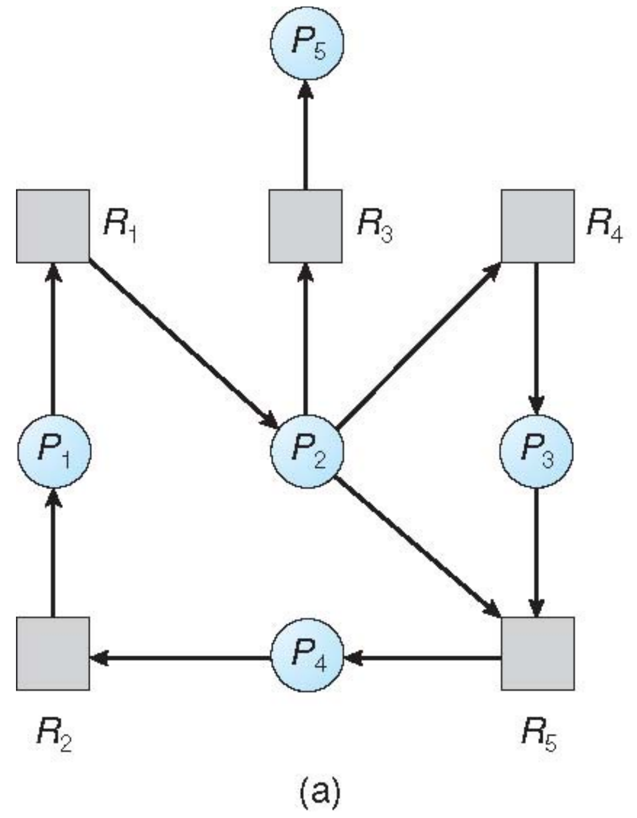
Deadlock Detection

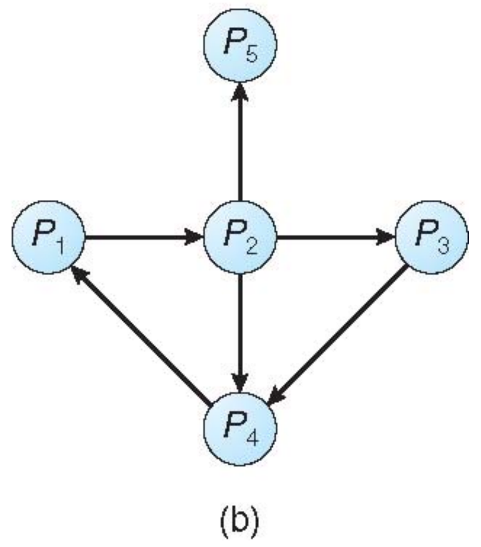
- Allow system to enter deadlock state. If that happens, detect the deadlock and do something about it.
- Detection algorithm
  - Single instance of each resource:
    - wait-for graph
  - Multiple instances:
    - detection algorithm (based on Banker's algorithm)
- Recovery scheme

Single Instance of Each Resource Type

- Maintain **wait-for** graph (based on resource allocation graph)
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
  - ***Deadlock if cycles***
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph

Corresponding wait-for graph

Has cycles. Deadlock.

Colorado State University

Several Instances of a Resource Type

Banker's algorithm: Can requests by all process be satisfied?

- **Available***:* A vector of length ***m*** indicates the number of available (currently free) resources of each type
- **Allocation***:* An ***n* x *m*** matrix defines the number of resources of each type currently allocated to each process
- **Request***:* An ***n* x *m*** matrix indicates the current request of each process. If ***Request* [*i*][*j*] = *k***, then process $\boldsymbol{P_i}$ is requesting ***k*** more instances of resource type $\boldsymbol{R_j}$.

Colorado State University

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively. Initialize:
   (a) ***Work* = initially *available***
   (b) For ***i* = 1,2, …, *n***, if ***Allocation***$_i \neq$ **0**, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***
2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) ***Request***$_i \leq$ ***Work***

   If no such ***i*** exists, go to step 4
3. ***Work = Work + Allocation***$_i$
   Finish*[*i*] = *true

   go to step 2 (find next process)
4. If ***Finish[i] == false***, for some ***i***, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then ***P***$_i$ is deadlocked

n = number of processes,
m = number of resources types
Work: res currently free
Finish$_i$: processes finished
Allocation$_i$: allocated to i

Algorithm requires an order of O(*m* x *n*^{2}) operations to detect whether the system is in deadlocked state

Colorado State University

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all i. ***No deadlock***

Process	Allocation			Request		
type	A	B	C	A	B	C
available	**0**	**0**	**0**			
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	0
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

After	work		
ini	0	0	0
P0	0	1	0
P2	3	1	3
P3	5	2	4
P1	7	2	4
P4	7	2	6

Example of Detection Algorithm (cont)

- ***P_2*** requests an additional instance of type ***C***

Process	Allocation			Request		
type	A	B	C	A	B	C
available	0	0	0			
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	1
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

Sequence

After	work		
ini	0	0	0
P0	0	1	0
P2	-	-	-

- State of system?
  - Can reclaim resources held by process ***P_0***, but insufficient resources to fulfill other processes' requests
  - Deadlock exists, consisting of processes ***P_1, P_2, P_3***, and ***P_4***

Colorado State University

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur
  - How many processes will need to be rolled back
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock: Process Termination

Choices

- Abort all deadlocked processes

- Abort one process at a time until the deadlock cycle is eliminated

In which order should we choose to abort?

1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollbacks in cost factor

Deadlock recovery through rollbacks

- **Checkpoint** process periodically
  - Contains memory image and resource state
- Deadlock detection tells us *which* resources are needed
- Process owning a needed resource
  - **Rolled back** to before it acquired needed resource
    - Work done since rolled back checkpoint discarded
  - **Assign** resource to deadlocked process

Livelocks

In a livelock two processes need each other's resource

- Both run and make no progress, but neither process blocks
- Use CPU quantum over and over without making progress

Ex: If fork fails because process table is full

- Wait for some time and try again
- But there could be a collection of processes each trying to do the same thing
- Avoided by ensuring that only one process (chosen randomly or by priority) takes action

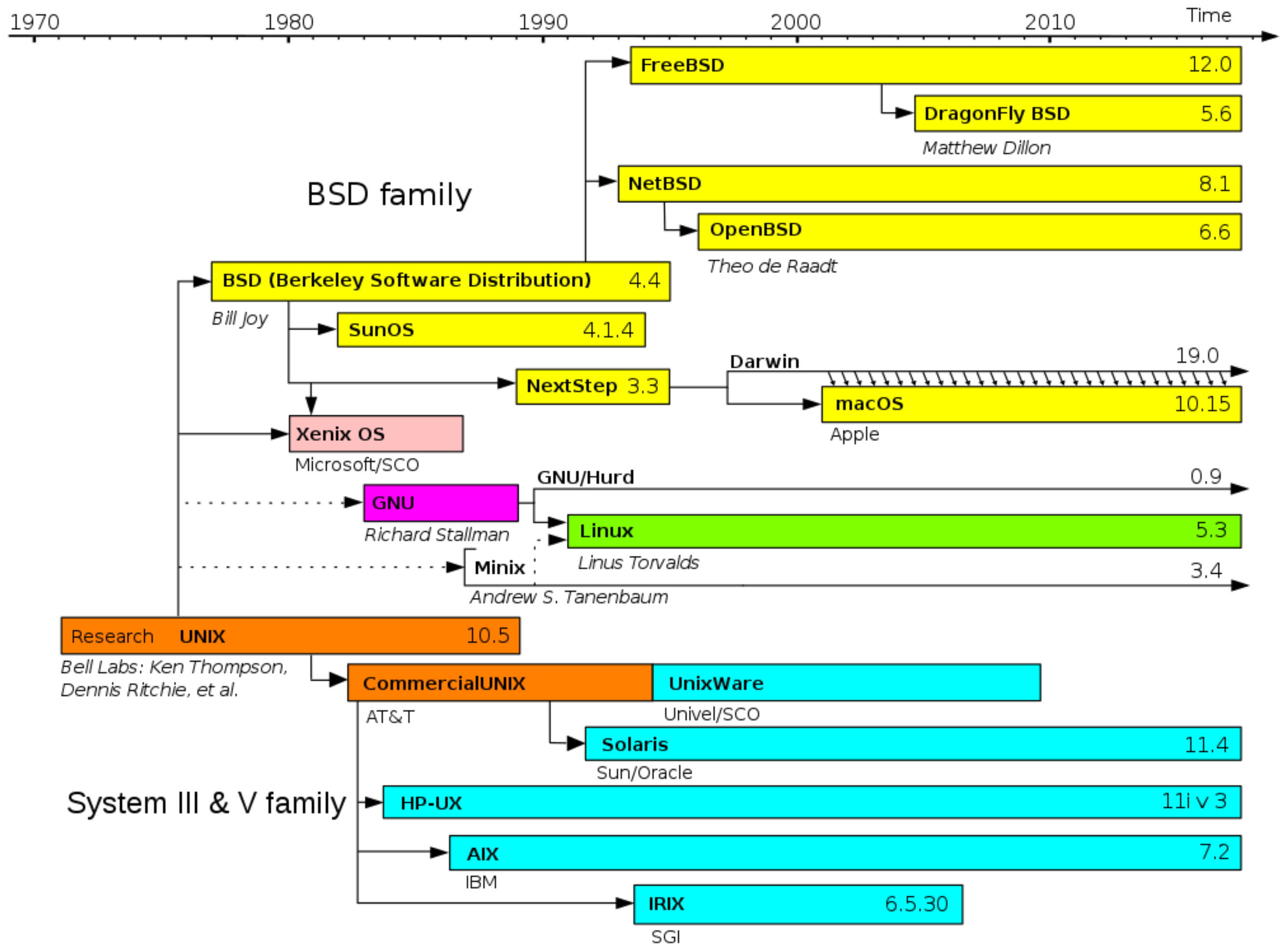
Two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass.
But they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Welcome to CS370 Second Half

- Topics: Memory, Storage, File System, Virtualization
- Class rules: See Syllabus
  - Class, Canvas, Teams
  - participation
  - Final
    - Sec 001, local 801: in class.
    - Sec 801 non-local: on-line.
    - SDC: Sec 001, Sec 801: must be taken at SDC
  - Project, deadlines, Plagiarism

Some OS History Lessons 1: UNIX

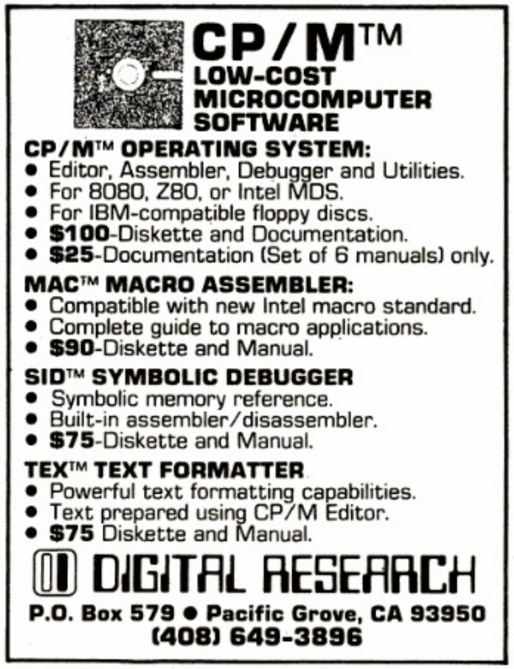
- History in Unix-like OSs



Colorado State University

Some OS History Lessons 2: Windows

- 1974: CP/M Intel 8080, Gary Kildall, Digital Research
  - 8-bit, min 16 kB RAM, floppy
- 1980: 86-DOS, Intel 8086, Tim Paterson, Seattle Computer Products
  - Inspired by CP/M?
- 1981: PC DOS, Bill Gates, Microsoft
  - 86-DOS licensed for $25,000, hired Paterson
- 1985: Windows, Bill Gates, Microsoft
  - GUI inspired by MAC? Xerox PARC Star?



CP/M™
LOW-COST MICROCOMPUTER SOFTWARE

CP/M™ OPERATING SYSTEM:
- Editor, Assembler, Debugger and Utilities.
- For 8080, Z80, or Intel MDS.
- For IBM-compatible floppy discs.
- **$100**-Diskette and Documentation.
- **$25**-Documentation (Set of 6 manuals) only.

MAC™ MACRO ASSEMBLER:
- Compatible with new Intel macro standard.
- Complete guide to macro applications.
- **$90**-Diskette and Manual.

SID™ SYMBOLIC DEBUGGER
- Symbolic memory reference.
- Built-in assembler/disassembler.
- **$75**-Diskette and Manual.

TEX™ TEXT FORMATTER
- Powerful text formatting capabilities.
- Text prepared using CP/M Editor.
- **$75** Diskette and Manual.

DIGITAL RESEARCH
P.O. Box 579 • Pacific Grove, CA 93950
(408) 649-3896

Gary Kildall net worth $1.9 Million at death
Tim Paterson Net Worth: $250,000

Colorado State University

CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Spring 2022

Main Memory

Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

Colorado State University
Yashwant K Malaiya
Spring 2022

Main Memory

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources

Chapter 8: Main Memory

Objectives:

- Organizing memory for multiprogramming environment
  - Partitioned vs separate address spaces
- Memory-management techniques
  - Virtual vs physical addresses
  - Chunks
    - segmentation
    - Paging: page tables, caching ("TLBs")
- Examples: the Intel (old/new) and ARM architectures

What we want

- Memory capacities have been increasing
  - But programs are getting bigger faster
  - Parkinson’s Law*: Programs expand to fill the memory available to hold
- What we would like
  - Memory that is
    - infinitely large, infinitely fast
    - Non-volatile
    - Inexpensive too
- Unfortunately, no such memory exists as of now

*work expands so as to fill the time available for its completion. 1955

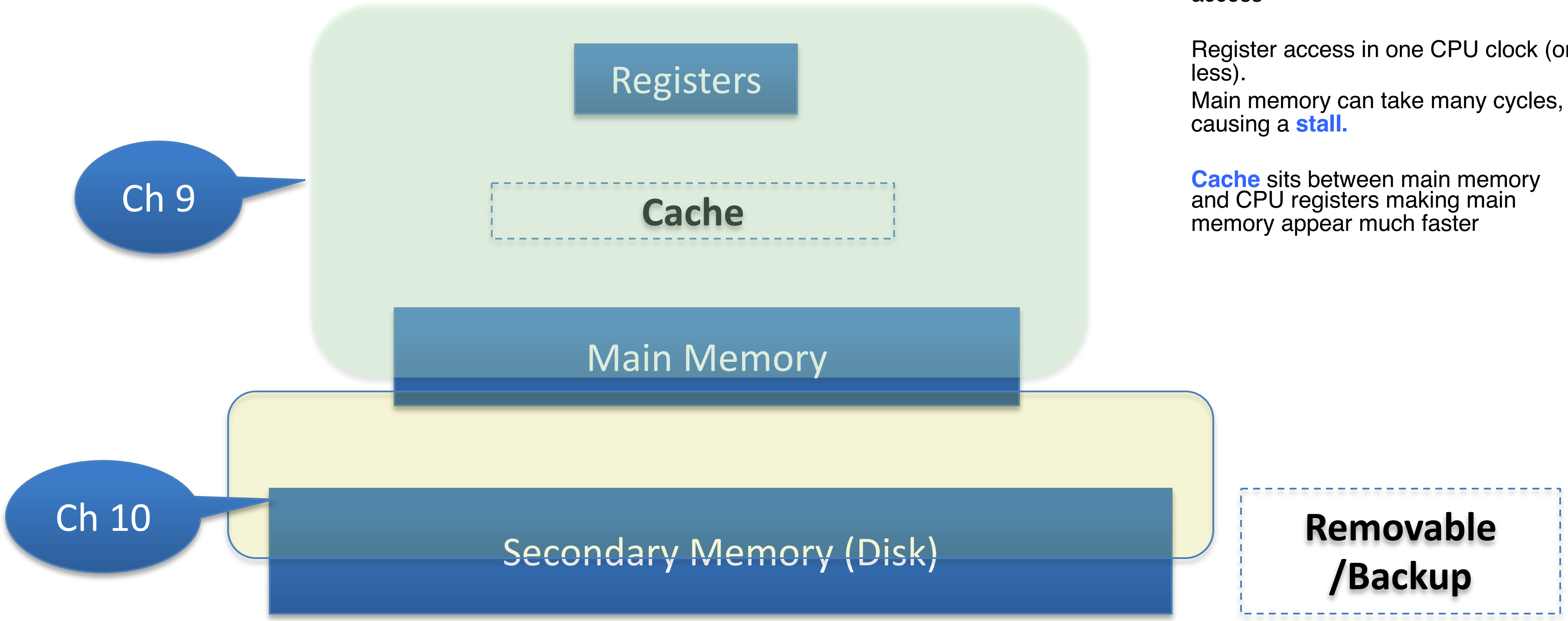
Colorado State University

Background

- Program must be brought (from disk) into memory and run as a process
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of
  - addresses + read requests, or
  - address + data and write requests
- n-bit address: address space of size 2^n bytes.
  - Ex: 32 bits: addresses 0 to (2^{32} -1) bytes
  - Addressable unit is always 1 byte.
- Access times:
  - Register access in one CPU clock (or less)
  - Main memory can take many cycles, causing a **stall**
  - **Cache** sits between main memory and CPU registers making main memory appear much faster
- **Protection** of memory required to ensure correct operation

2^{10}=1,024 ≈ K
2^{20} = 1,048,576 ≈ M
2^{30} ≈ G

Hierarchy



Main memory and registers are only storage CPU can access directly access

Register access in one CPU clock (or less).
Main memory can take many cycles, causing a **stall.**

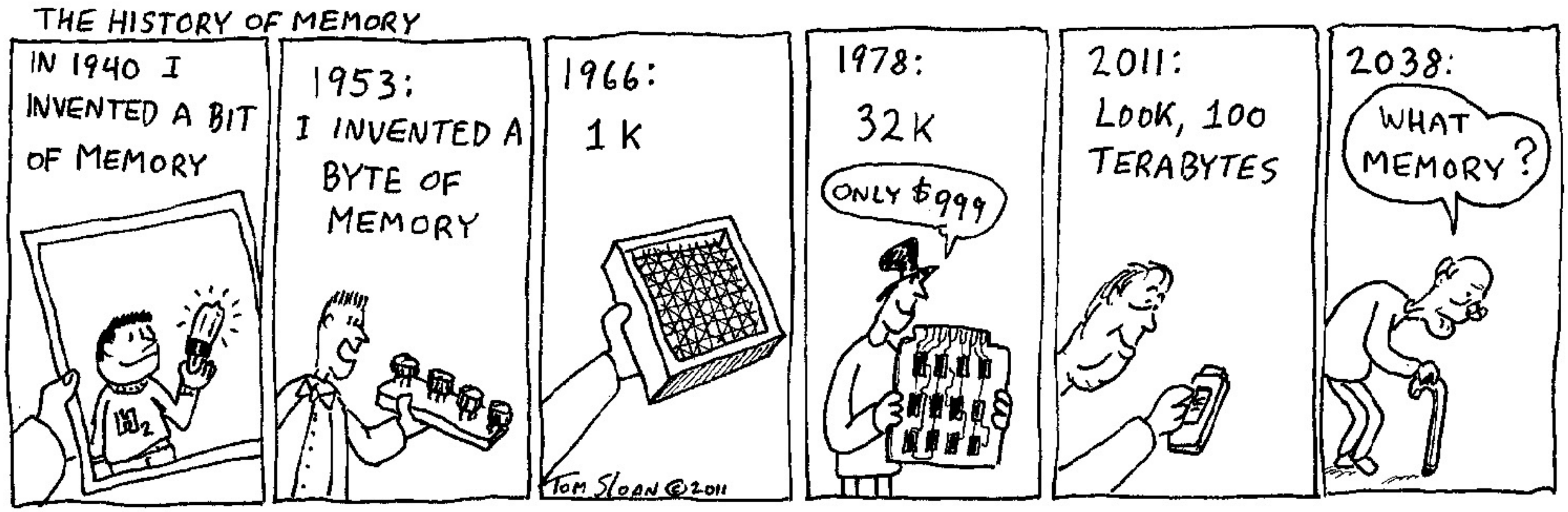
Cache sits between main memory and CPU registers making main memory appear much faster

Ch 11,13,14,16: Disk, file system Cache: CS470

Colorado State University

Memory Technology

somewhat inaccurte


THE HISTORY OF MEMORY
IN 1940 I INVENTED A BIT OF MEMORY
1953: I INVENTED A BYTE OF MEMORY
1966: 1 K
1978: 32 K
ONLY $999
2011: LOOK, 100 TERABYTES
2038:
WHAT MEMORY?
TOM SLOAN ©2011

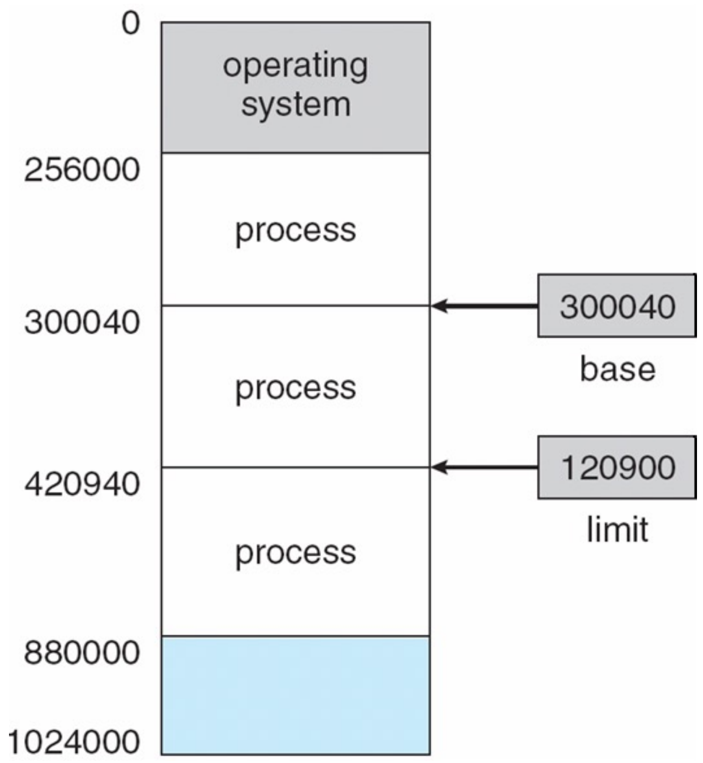
Colorado State University

Protection: Making sure each process has separate memory spaces

- OS must be protected from accesses by user processes
- User processes must be protected from one another
  - Determine range of legal addresses for each process
  - Ensure that process can access only those
- Approaches:
  - Partitioning address space (early system)
  - Separate address spaces (modern practice)

Partitioning: Base and Limit Registers

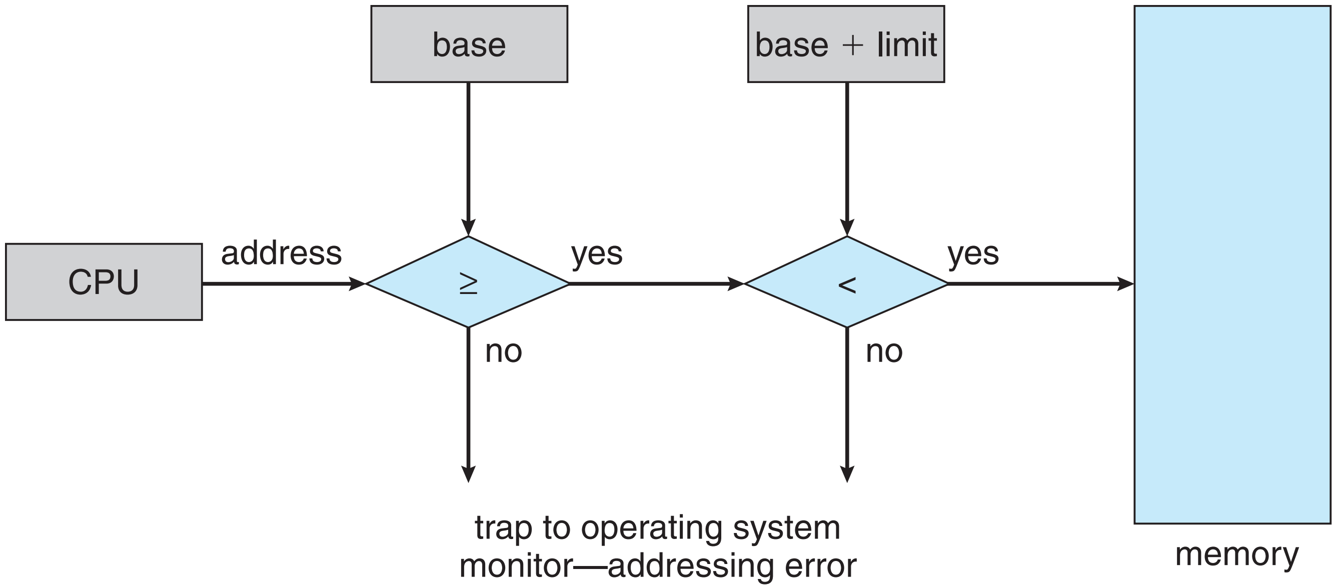
- Base and Limit for a process
  - **Base**: Smallest legal physical address
  - **Limit**: Size of the range of physical address
- A pair of **base** and **limit registers** define the logical address space for a process
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
- Base: **Smallest** legal physical address
- Limit: Size of the **range** of physical address
- Eg: Base = 300040 and limit = 120900
- Legal: 300040 to (300040 + 120900 -1) = 420939



Addresses: decimal, hex/binary

Colorado State University

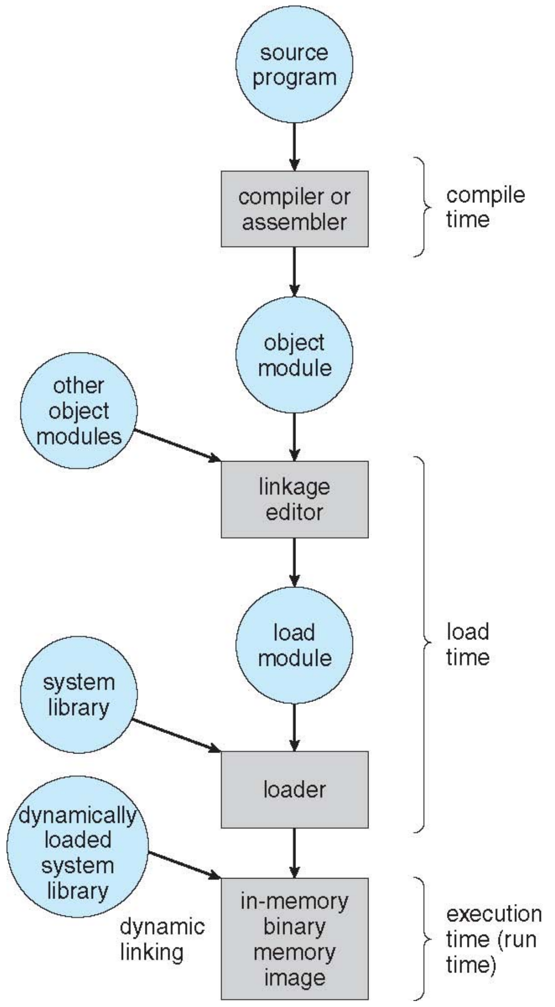
Hardware Address Protection



Legal addresses: **Base address** to **Base address + limit -1**

Colorado State University

Multistep Processing of a User Program



Colorado State University

Address Binding Questions

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program's life
  - **Source code** addresses are symbolic
  - **Compiled code** addresses **bind** to relocatable addresses
    - i.e., "14 bytes from beginning of this module"
  - **Linker or loader** will bind relocatable addresses to absolute addresses
    - i.e., 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Separate Address Spaces Modern

- Each process has its own private address space.
  - **Logical address space** is the set of all logical addresses used by a process.
- However, the physical memory has just one address space.
  - **Physical address space** is the set of all physical addresses
- Need to map one to the other.

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses