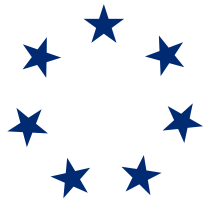
Memory Management

Instructor: Hengming Zou, Ph.D.

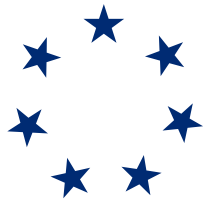
In Pursuit of Absolute Simplicity 求于至简，归于永恒

Lecture Arrangement



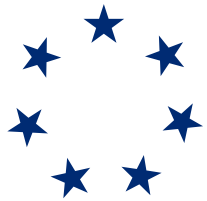
- Four lectures:
- Lecture 1: (3)
  - Basic memory management
- Lecture 2: (3)
  - Paging systems
- Lecture 3: (2)
  - Segmentation systems

Lecture 1: Basic memory management



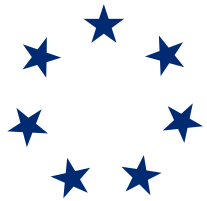
- Memory management environment
- Fixed location
- Base and bound
- Address translation
- Swapping
- Virtual memory

Lecture 2: Paging systems



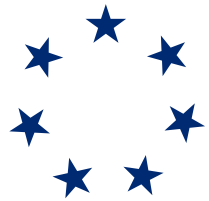
- Paging
- Page table
- Page replacement
- Design issues for paging systems

Lecture 3: Segmentation Systems



- Drawback of paging system
- Segmentation systems
- Segmentation with paging
- Case study: Multics

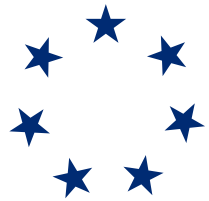
Lecture 4: Advanced Topics



- Translation data
- Kernel vs. User Mode
- Passing Arguments to System Call
- Separation of Policy and Mechanism
- Case study: Linux Memory Management

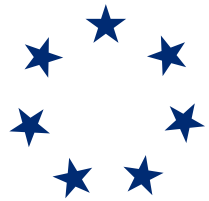
Basic Memory Management

Content

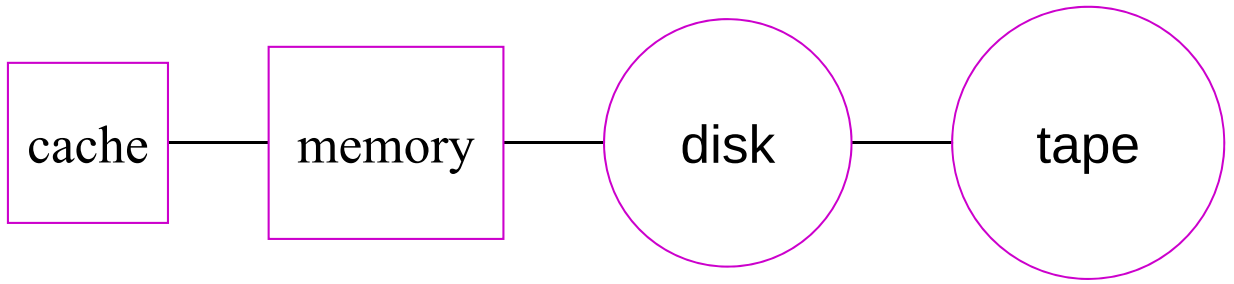


- Memory management environment
- Memory hierarchy and virtual memory
- (Pure) segmentation
- Fixed location
- Fixed Partition
  - Address translation
  - Base and bound
- Swapping

Memory Management Environment

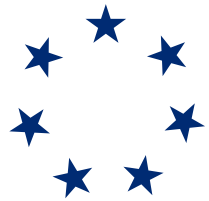


- Ideally programmers want memory that is
  - Large, fast, and non volatile
- The Reality: the memory hierarchy:
  - Cache: Small amount of fast, expensive memory –
  - Main memory: Some medium-speed, medium price memory
  - Disk storage: Gigabytes of slow, cheap memory



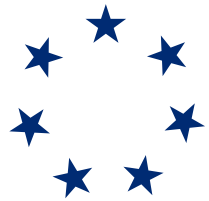
Memory hierarchy

Memory Management Environment



- Memory manager handles the memory hierarchy
  - through virtual memory
- What is virtual memory:
  - An illusion provided to the applications
  - Built on top of the memory hierarchy
- Provides two abstractions:
  - an address space that can be larger than the amount of physical memory
  - a storage space that is fast than the physical memory
- In the future will provide a storage space that is durable

Memory Management Objective

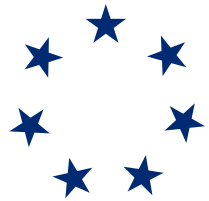


- Meet two address objectives:

- **Address independence**
  - **The address given by program is independent of physical address**

- **Address protection**
  - **One process does not access another's address space**

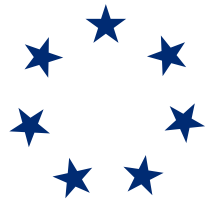
Basic Memory Management



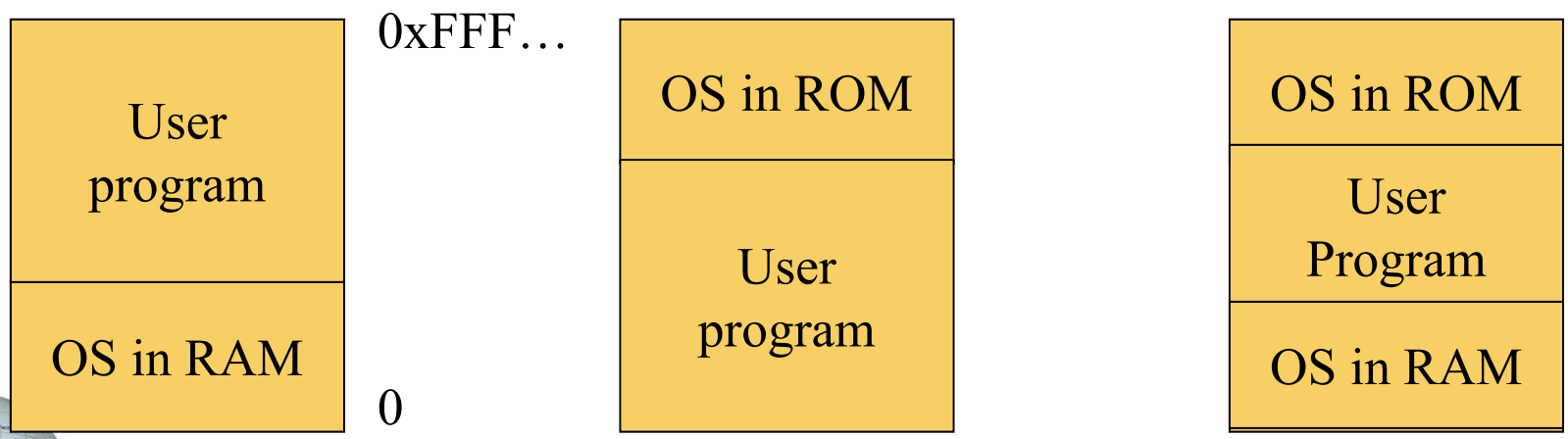
- Pure Segmentation
  - Treat each program as a single logical segment

- Can be further divided into:
- Fixed location
  - Applies to uni-programming
- Fixed partitions
  - Applies to multi-programming
- Swapping
  - Applies to multi-programming

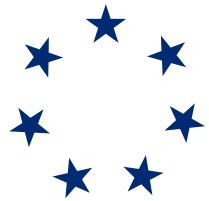
Fixed Location for Uni-programming



- One process runs at a time
  - One process occupies memory at a time
- Always load process into the same memory spot
- And reserve some space for the OS
- 3 ways of organizing memory for an OS with one user process



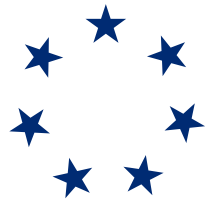
Fixed Location for Uni-programming



- Achieves address independence by
  - Loading process into same physical memory location
  - Therefore physical addresses can be computed before hand
- Achieve address protection by:
  - Doing nothing!

- Problems with uni-programming?
  - Load processes in its entirety (no enough space?)
  - Waste resource (both CPU and Memory)

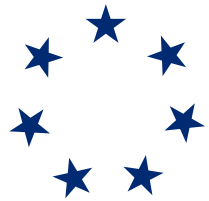
Multi-programming



- More than one process is in memory at a time

- More must be done by memory management
  - Need to support address translation
    - →Address from instruction may not be the final address
    - →Physical addresses cannot be computed before hand
  - Need to support protection
    - →Each process cannot access other processes' space

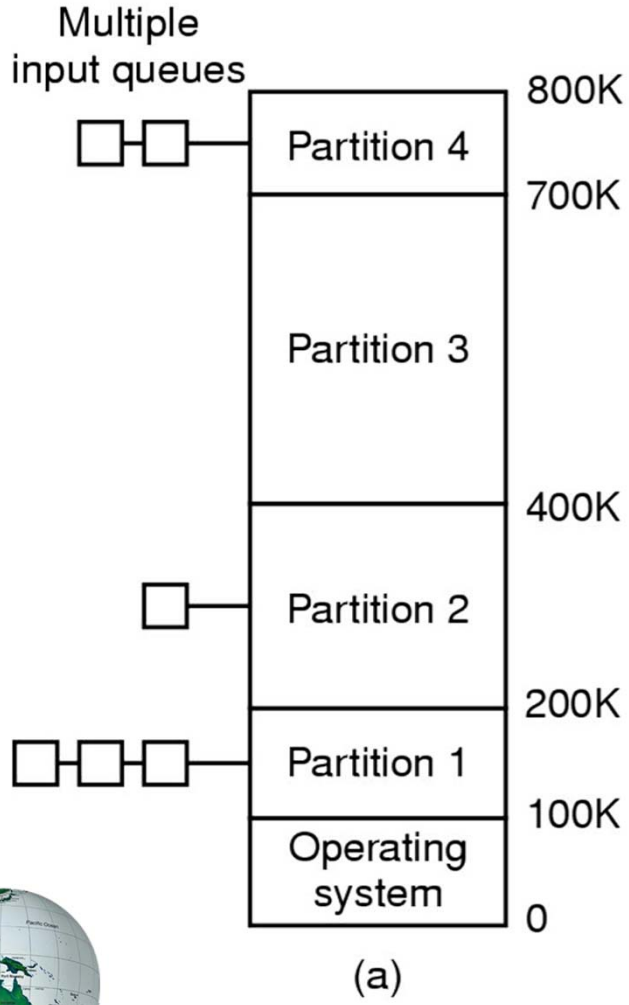
Multiprogramming with Fixed Partitions

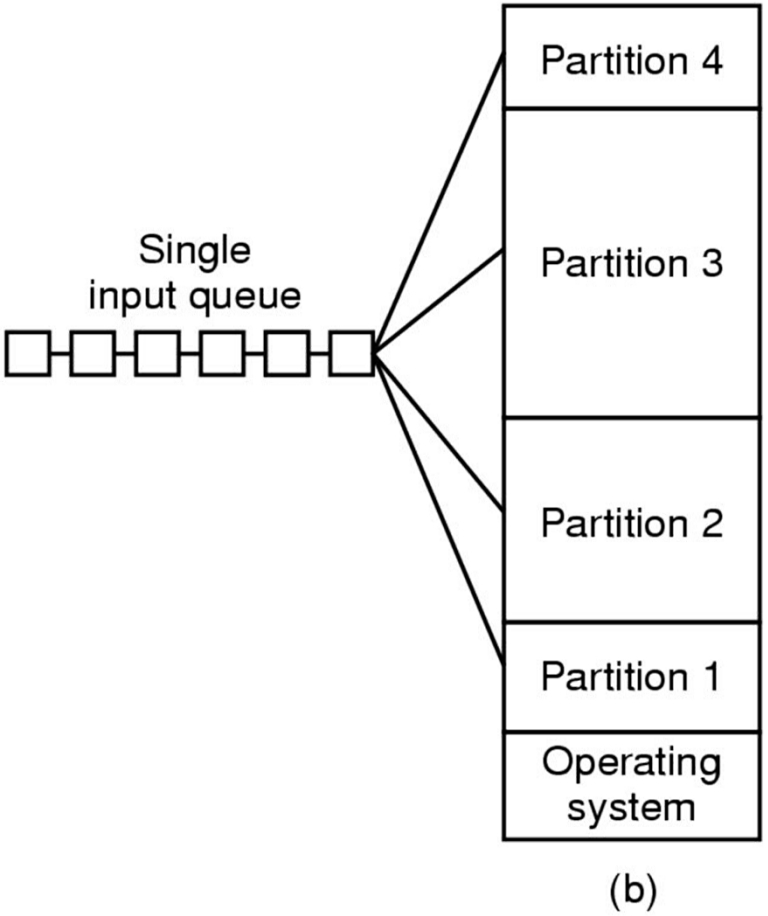


- The simplest form of memory mgmt for multiprogramming
  - Memory are partitioned into fixed-size partitions
  - Programs are loaded into fixed partitions

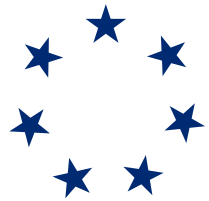
- Two options exist for loading programs into fixed memory partitions
  - Separate input queues for each partition
    - →Incoming processes are allocated into fixed partition
  - Single input queue
    - →Incoming processes can go to any partition
    - →Assume partition is big enough to hold the processes

Multiprogramming with Fixed Partitions



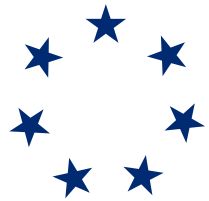


Multiprogramming Issues



- Cannot be sure where program will be loaded
  - Many partitions exist that can be used to hold the program
  - Address locations of variables can not be absolute
  - Code routines cannot be absolute
- Solution:
  - Use base and limit values
  - Address added to base value to map to physical address
  - Address locations larger than limit value is an error

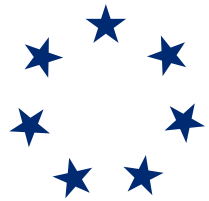
Multiprogramming Issues



- Protection processes from each other
  - Must keep a program out of other processes' partitions
- Solution:
  - Address translation
  - Must translate addresses issued by a process

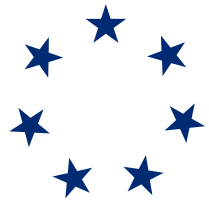
    →so they don't conflict with addresses issued by other processes

Address Translation



- Static address translation
  - Translate addresses before execution
  - Translation remains constant during execution
- Dynamic address translation
  - Translate addresses during execution
  - Translation may change during execution

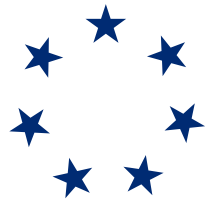
Address Translation



- Is it possible to:
  - Run two processes at the same time (both in memory), and
  - provide address independence with only static address translation?
- Does this achieve the other address space objective?
  - No (i.e. does not offer address protection)

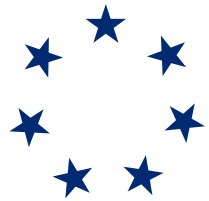
- Achieving all address space objectives (protection/independence)
  - requires doing some work on every memory reference
- Solution:
  - Dynamic address translation

Dynamic Address Translation



- Translate every memory reference from virtual to physical address
- Virtual address:
  - An address viewed by the user process
  - The abstraction provided by the OS
- Physical address
  - An address viewed by the physical memory

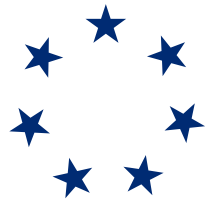
Benefit of Dynamic Translation



- Enforces protection
  - One process can't refer to another process's address space

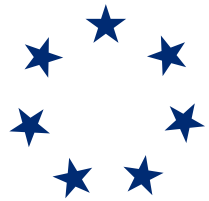
- **Enables virtual memory**
  - A virtual address only needs to be in physical memory
    - →when it's being accessed
  - Change translations on the fly
    - →as different virtual addresses occupy physical memory
- Does dynamic address translation require hardware support?
  - It's better to have but not absolutely necessary

Implement Translator



- Lots of ways to implement the translator
- Tradeoffs among:
  - Flexibility (e.g. sharing, growth, virtual memory)
  - Size of translation data
  - Speed of translation

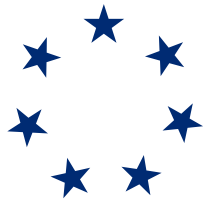
Base and Bounds



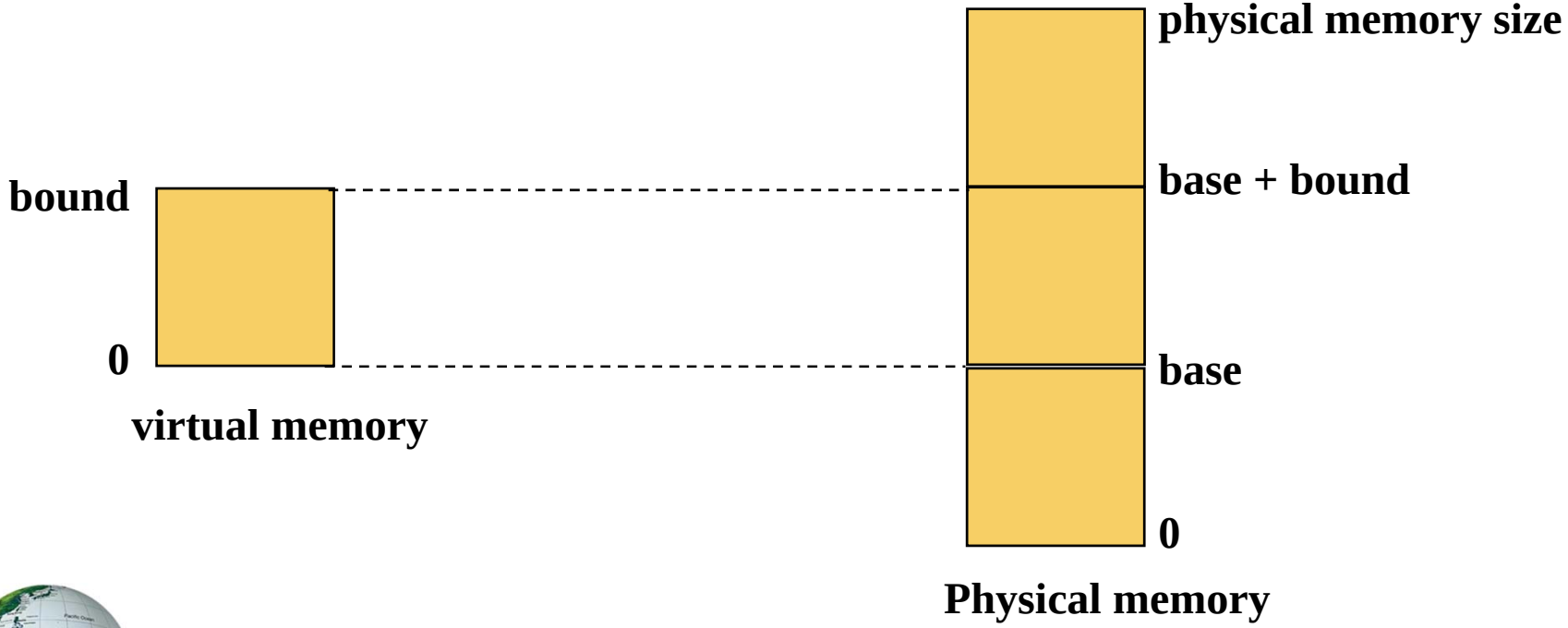
- The simplest solution
- Load each process into contiguous regions of physical memory
- Prevent each process from accessing data outside its region

- if (virtual address > bound) {
- trap to kernel; kill process (core dump)
- } else {
- physical address = virtual address + base
- }

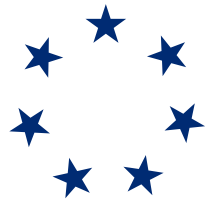
Base and Bounds



- Process has illusion of running on its own dedicated machine
  - with memory [0, bound)



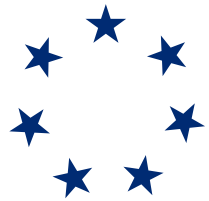
Base and Bounds



- This is similar to linker-loader
  - But also protect processes from each other
- Only kernel can change base and bounds
- During context switch, must change all translation data
  - (base and bounds registers)

- What to do when address space grows?

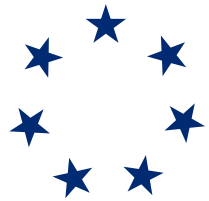
Pros and Cons of Base and Bounds



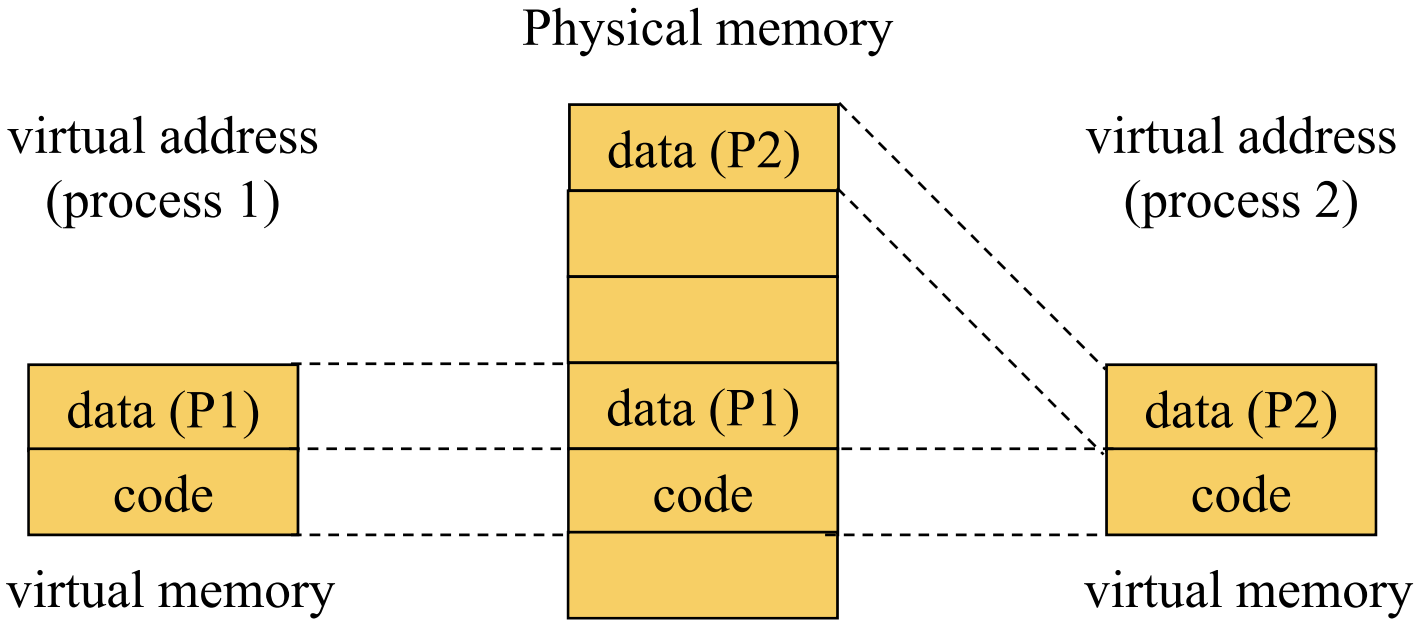
- Pros:
  - Low hardware cost
    - → 2 registers, adder, comparator
  - Low overhead
    - →Add and compare on each memory reference

- Cons:
  - Hard for a single address space to be larger than physical memory
  - But sum of all address spaces can be larger than physical memory
    - →Swap an entire address space out to disk
    - →Swap address space for new process in

Cons of Base and Bounds

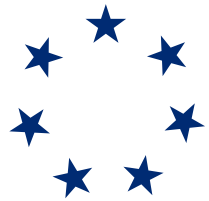


- Can't share part of an address space between processes

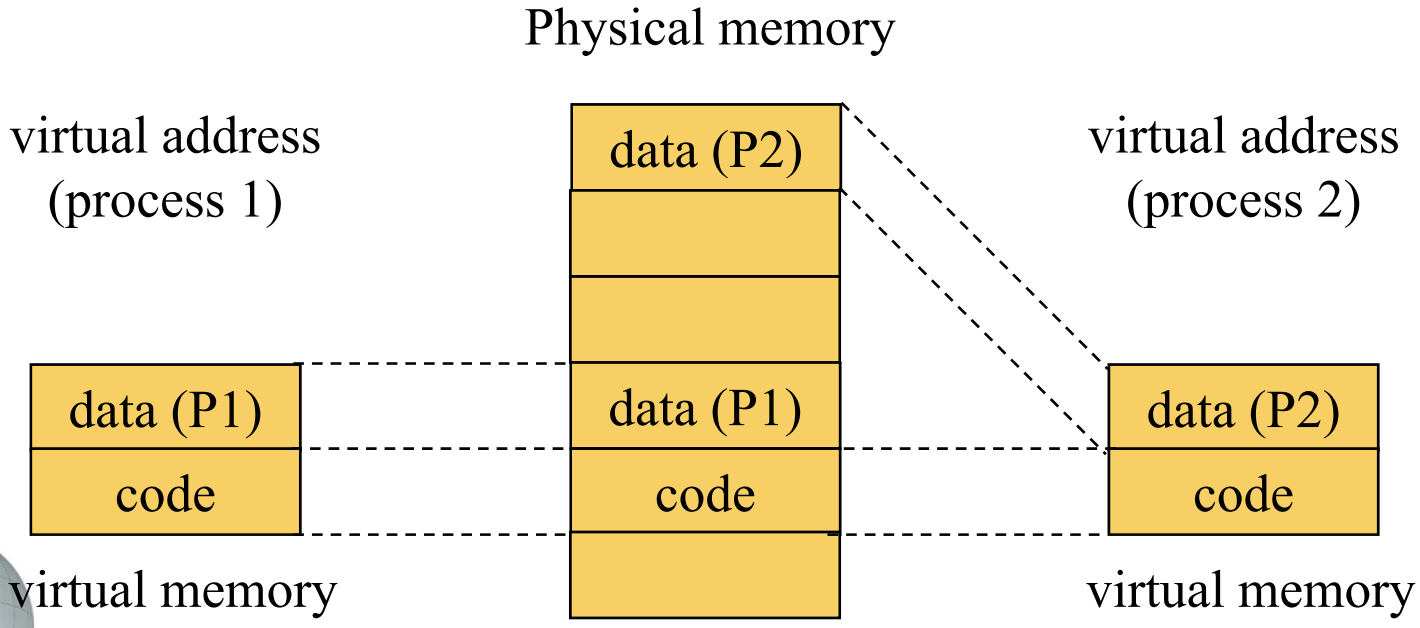


Does this work under base and bound?

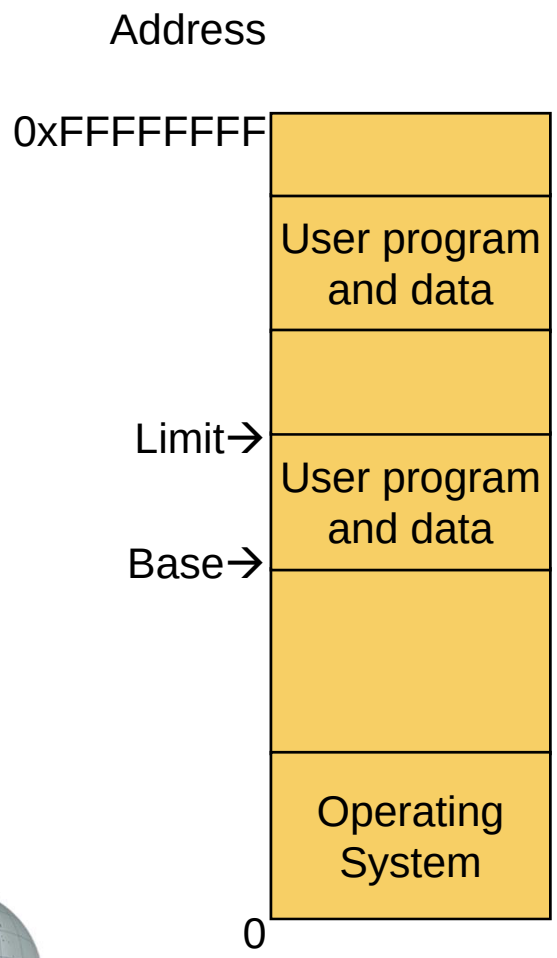
Cons of Base and Bounds

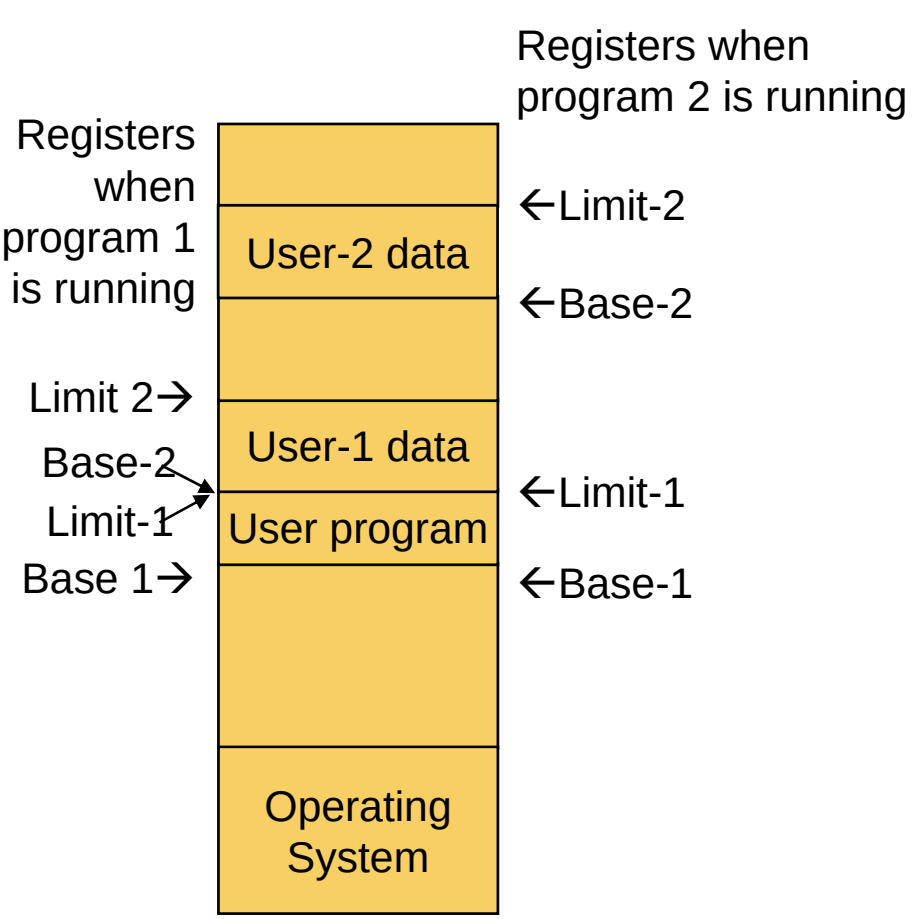


- Solution:
- Multiple sets of base and bounds
  - Specifically, use two sets of base and bounds for two processes sharing
    →one for code section, one for data section

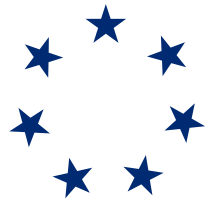


1 Base-Limit & 2 Base-Limit Pairs





Swapping

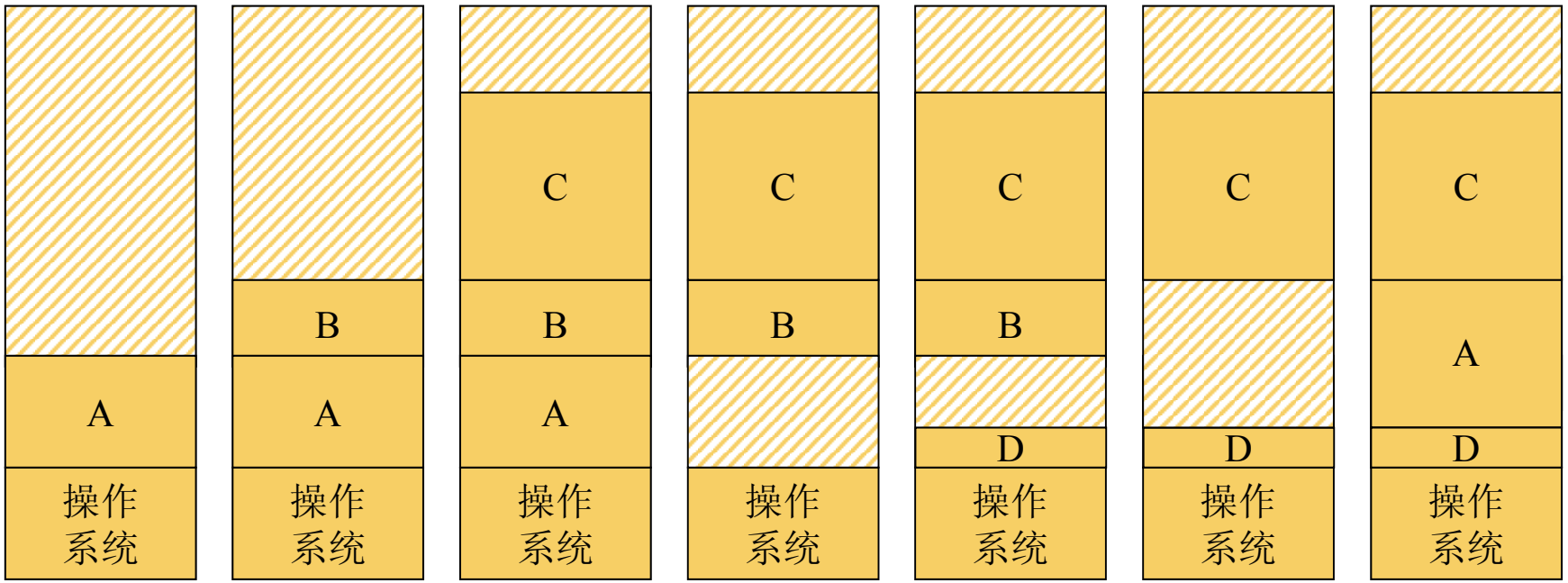


- Fixed partitions is inflexible in utilizing the physical memory space
- A flexible management scheme allows varying partitions

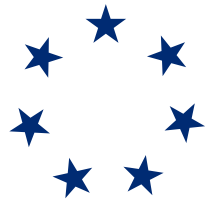
- Swapping is the solution:
  - **Take processes out from memory and bring process into memory**

- Memory allocation changes as:
  - Processes come into memory
  - Processes leave memory

Swapping

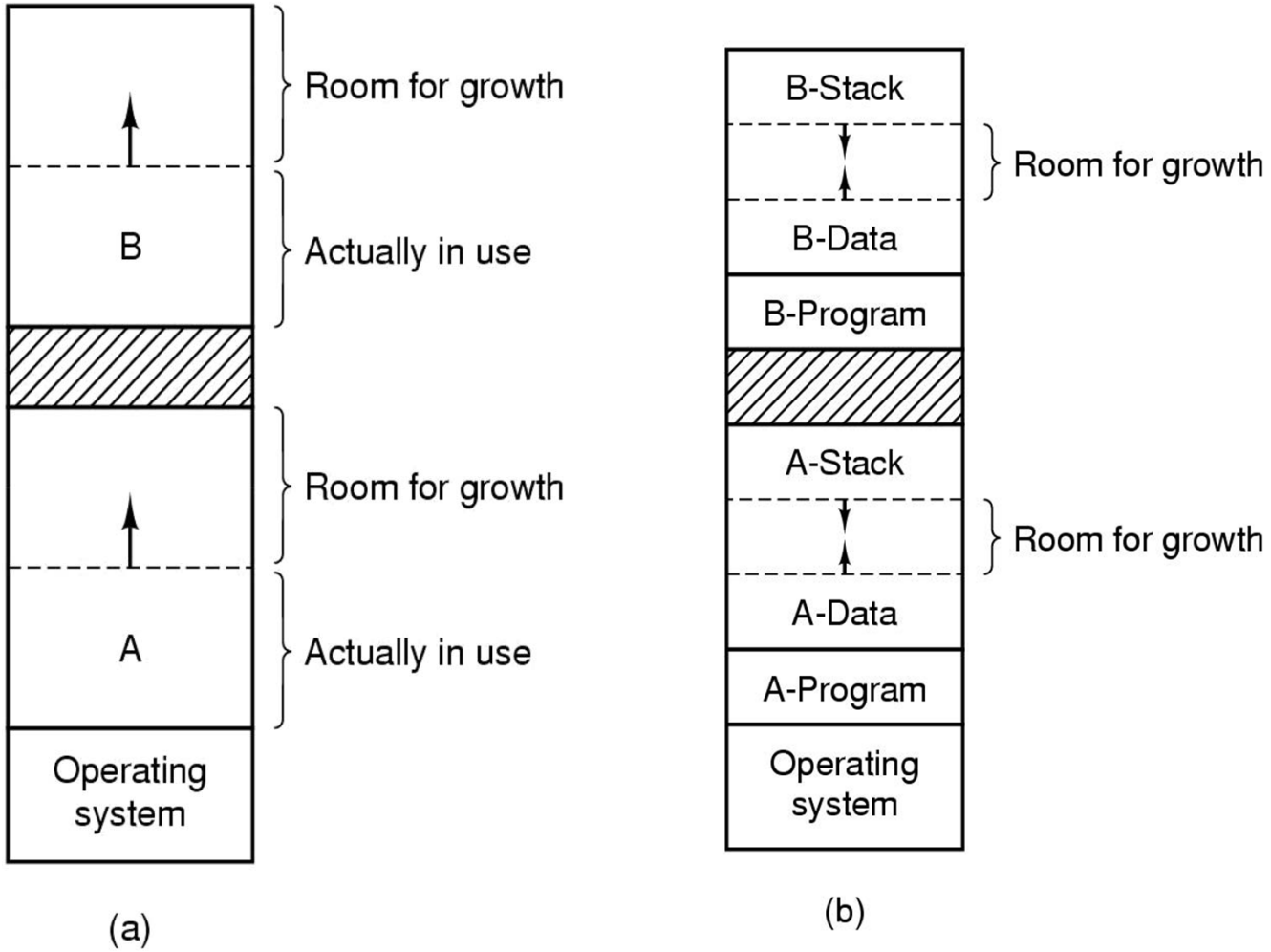


Swapping

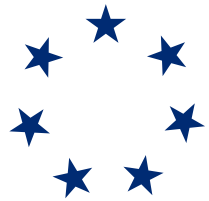


- Problem with previous situation?
- Difficult to grow process space
  - i.e. stack, data, etc.
- Solution:
  - Allocating space for growing data segment
  - Allocating space for growing stack & data segment

Swapping

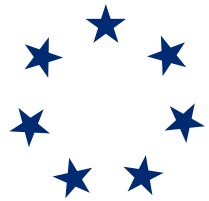

Room for growth
B
Actually in use
Room for growth
A
Actually in use
Operating system
(a)
B-Stack
Room for growth
B-Data
B-Program
A-Stack
Room for growth
A-Data
A-Program
Operating system
(b)

External Fragmentation with Swap



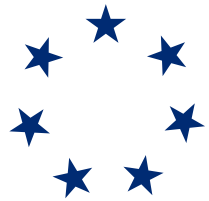
- In a swap system, processes come and go
- Can leave a mishmash of available memory regions
- Some regions may be too small to be of any use
- Hard to grow address space
  - Might have to move to different region of physical memory (which is slow)

External Fragmentation



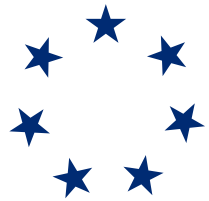
- P1 start:100 KB (phys. mem. 0-99 KB)
- P2 start:200 KB (phys. mem. 100-299 KB)
- P3 start:300 KB (phys. mem. 300-599 KB)
- P4 start:400 KB (phys. mem. 600-999 KB)
- P3 exits (frees phys. mem. 300-599 KB)
- P5 start:100 KB (phys. mem. 300-399 KB)
- P1 exits (frees phys. mem. 0-99 KB)
- P6 start:300 KB

External Fragmentation



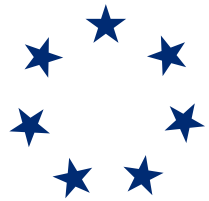
- 300 KB are free (400-599 KB; 0-99 KB)
  - but not contiguous
- This is called “external fragmentation”
  - wasted memory between allocated regions
- Can waste lots of memory

Strategies to Minimize Fragmentation



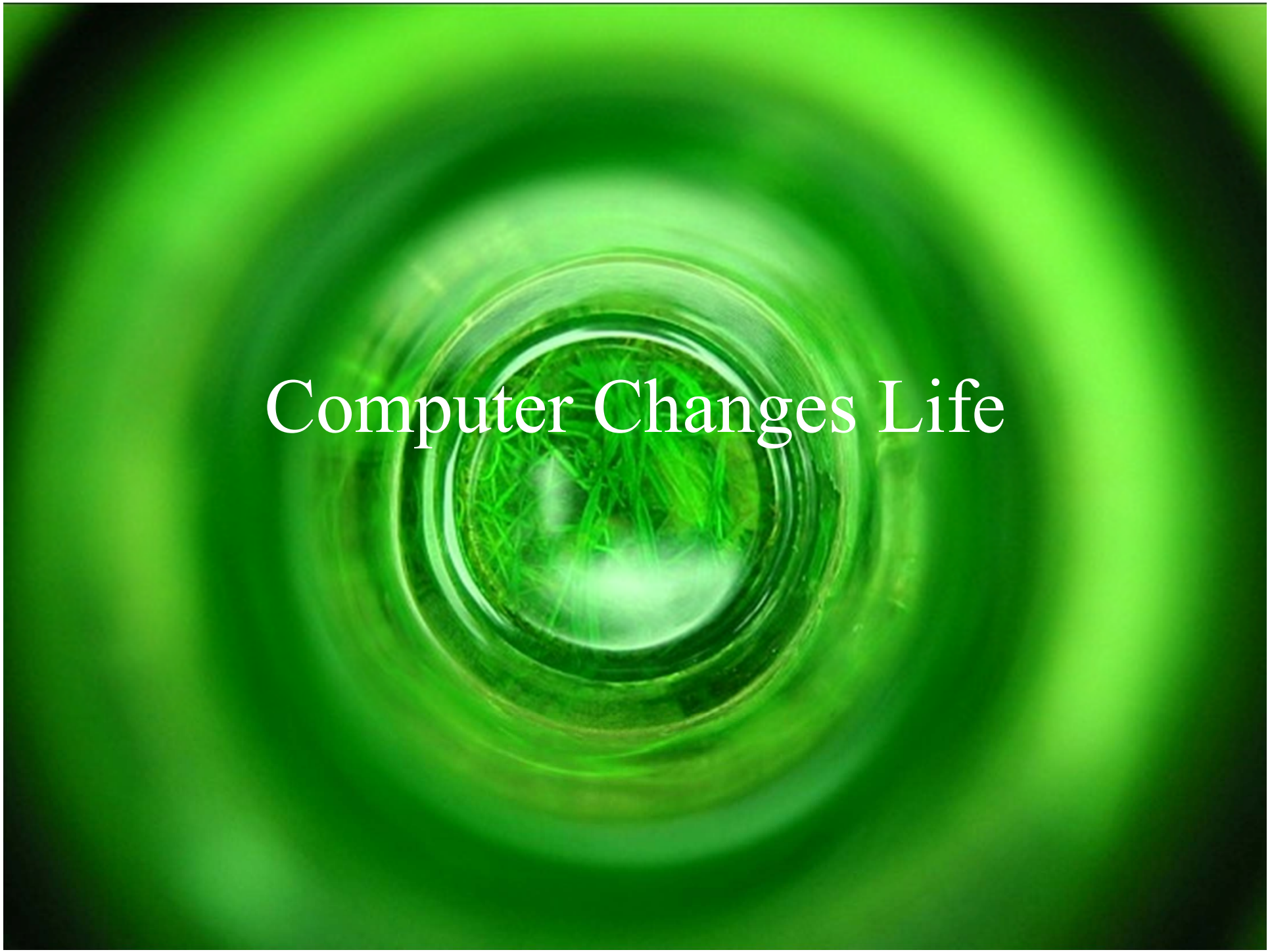
- Best fit:
  - Allocate the smallest memory region that can satisfy the request
    - →(least amount of wasted space)
- First fit:
  - Allocate the memory region that you find first that can satisfy the request
- In worst case, must re-allocate existing memory regions
  - by copying them to another area

Problem with Pure Segmentation



- Hard to grow address space (for a data structure for example)
- Unable to run program that is larger than physical memory
- External fragmentation

- How to extend more than one contiguous data structure in VM?
- How to run a program larger than physical memory?
  - PAGING


Computer Changes Life