



Introduction to Memory Management

Lecture 28

Lawrence Angrave/Klara Nahrstedt

CS241 Administrative

- Read Stallings Chapter 7 & 7A about Memory Management
- **Regular Quiz 9 this week on Memory Management**
- Discussion Sessions this week (on Memory Management – needed for LMP2)
- **LMP2 (Long Machine Problem) starts today**
 - LMP 2 is part of three major assignments (LMP1, LMP2, LMP3)
 - LMP2 will be split into two parts, PART I and overall LMP2 delivery.
 - You will need to deliver PART I by Monday, April 9 and the final overall LMP2 will be due by April 16. For each delivery you will receive points.
 - LMP2 quiz will be on April 16.
 - The graders will provide feedback on the Part I that should assist you in the overall LMP2 delivery.

Contents

Addressing Requirements for a Process

Names and Binding, Loading, Compiling, Run-Time

Storage hierarchy

Simple Programming

Overlays

Resident monitor

Multi-Programming

Fixed Partitioning

Relocation Register

Swapping

Names and Binding

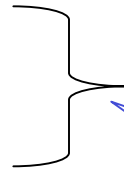
Naming = very deep theme that comes up everywhere

Indirection

Symbolic names

Logical names

Physical names



Most used by application programmers

Converted by system programmers of compilers, OS

Symbolic Names

Symbolic names: known in a context or path

File names

Program names

Printer/device names

User names

Variables

Convenient to use/reference

Logical Names

Logical names: label a specific entity

... but independent of specific physical entity

Inode number

Job number

Major, minor device numbers

uid, pid, gid

Absolute address in program

Physical Names

Physical names: address of entity

Physical Inode address on disk or memory (track, cylinder, ...)

Entry point or variable address

Memory-mapped register addresses

Process control block address

Difficult to use by programmers

Therefore, system software (compilers, loaders, OS) are usually the ones that convert symbolic names/logic names to physical names

Binding Time of Names

Binding: map names to locations (values)

Late binding ... Early binding

Different times

Source program (.c file)

Compile time

Object module (.o file)

Link time

Load module

Load time

Run-time

Binding at Compile & Link time

Create object code

```
gcc -c part1.c
```

```
gcc -c part2.c
```

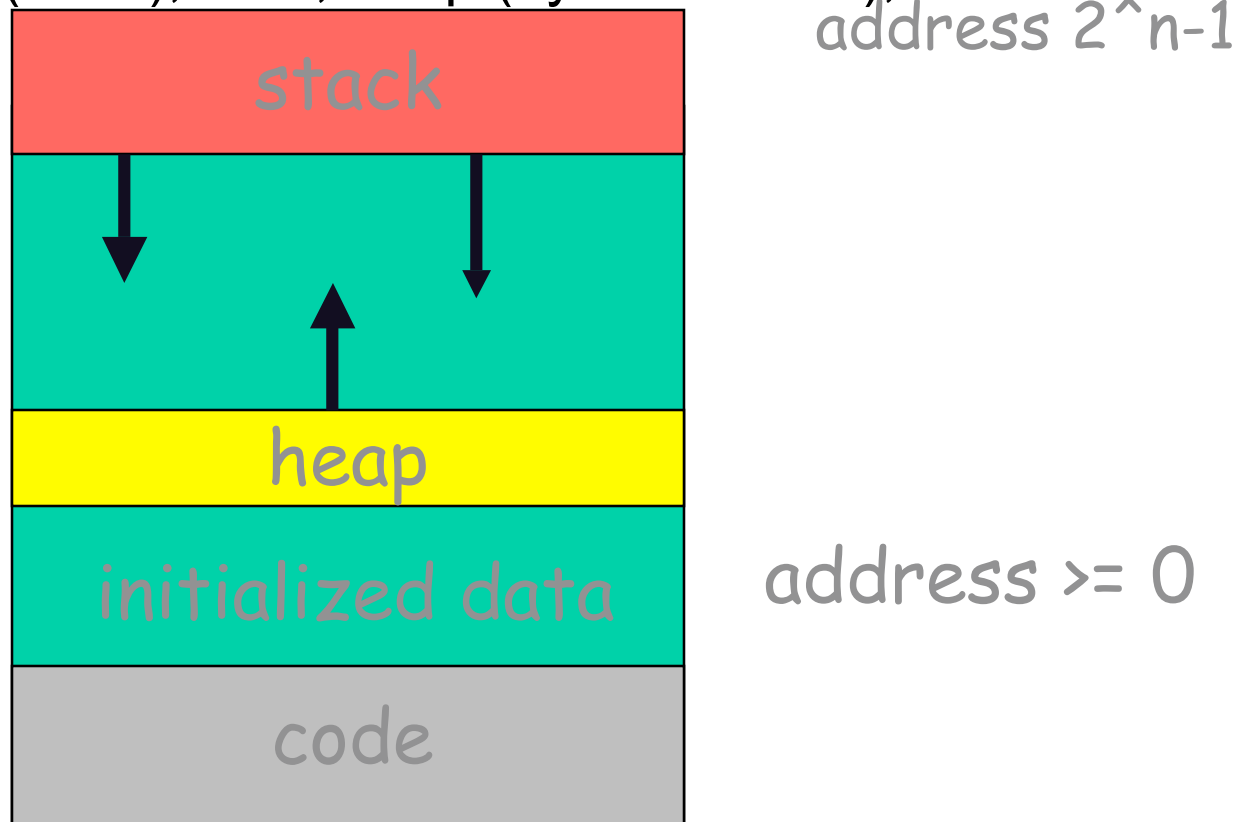
```
gcc -o wowie part1.o part2.o
```

gcc uses linker program 'ld' to link object code together.

What does a process look like? (Unix)

Process address space divided into “segments”

text (code), data, heap (dynamic data), and stack



Who Binds What?

Heap: constructed and layout by allocator (malloc)

compiler, linker not involved other than saying where it can start
namespace constructed dynamically and managed by
programmer (names stored in pointers, and organized using
data structures)

Stack: alloc dynamic (proc call), layout by compiler

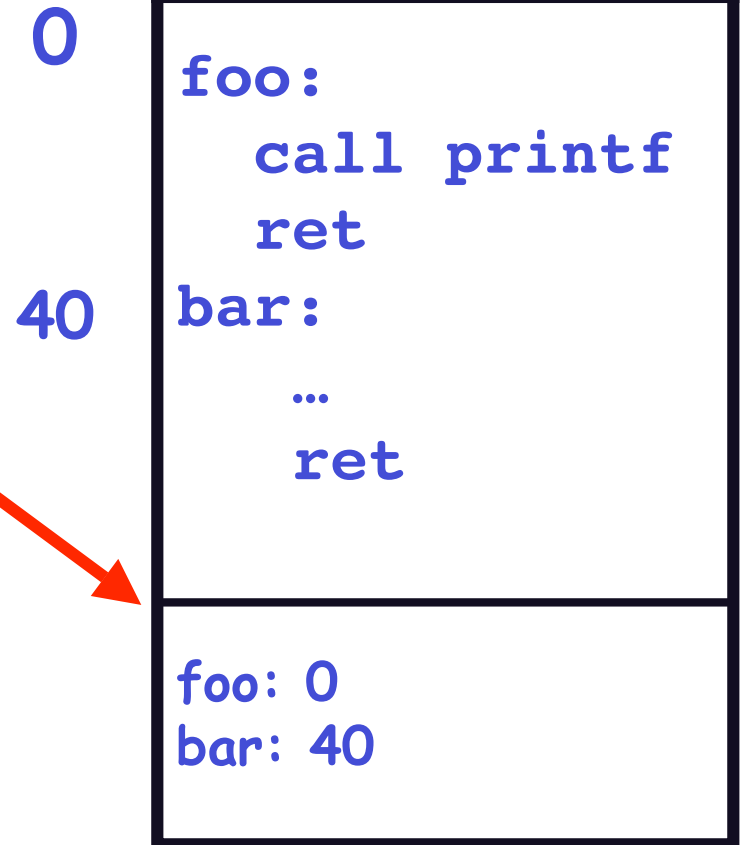
names are relative off of stack pointer
managed by compiler (alloc on proc entry, dealloc on exit)
linker not involved because name space entirely local: compiler
has enough information to build it.

Global data & code: allocation static (compiler),
layout (linker)

compiler emits them and can form symbolic references between
them (“jalr _printf”)
linker lays them out, and translates references

Compiling

- Compiler:
 - doesn't know where data/code should be placed in the process's address space
 - assumes everything starts at zero
 - emits symbol table that holds the name and offset of each created object
 - routine/variables exported by the file are recorded **global definition**
- Simpler perspective:
 - code is in a big char array
 - data is in another big char array
 - compiler creates (object name, index) tuple for each interesting thing
 - linker then merges all of these arrays



Linkers (Linkage editors)

Unix: ld

usually hidden behind compiler

Three functions:

collect together all pieces of a program

coalesce like segments

fix addresses of code and data so the program can run

Result: runnable program stored in new object file

Why can't compiler do this?

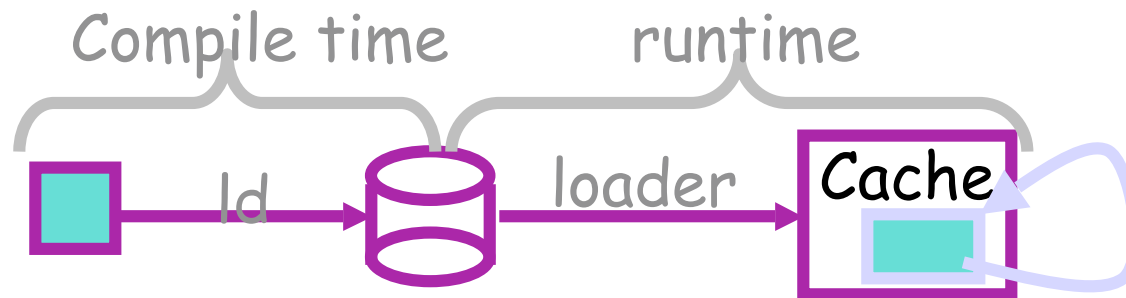
Limited world view: one file, rather than all files

Note *usually*: linkers only shuffle segments, but do not rearrange their internals.

E.g., instructions not reordered; routines that are never called are not removed from a.out

Loading before Running

On Unix systems, read by “loader”



reads all code/data segs into buffer cache; maps code (read only) and initialized data (r/w) into addr space

Optimization opportunities:

Zero-initialized data does not need to be read in.

Demand load: wait until code used before get from disk

Copies of same program running? Share code

Multiple programs use same routines: share code (harder)

Run-time

Maps virtual addresses to physical addresses

VM, overlaying, recursion, relocation

Binding Time Trade-offs(1)

Early binding

Compiler

- + Produces efficient code
- + Allows checking to be done early
- + Allows estimates of running time and space
- inflexible
- require fixed hardware
- difficult to support multiprogramming

Binding Time Trade-offs(2)

Delayed binding

Linker, loader

- + Produces efficient code
- + Allows separate compilation
- + Provides libraries, system interfaces
- + Portability and sharing of object code
- + Checking of consistency
- inflexible
- medium difficult to support multiprogramming

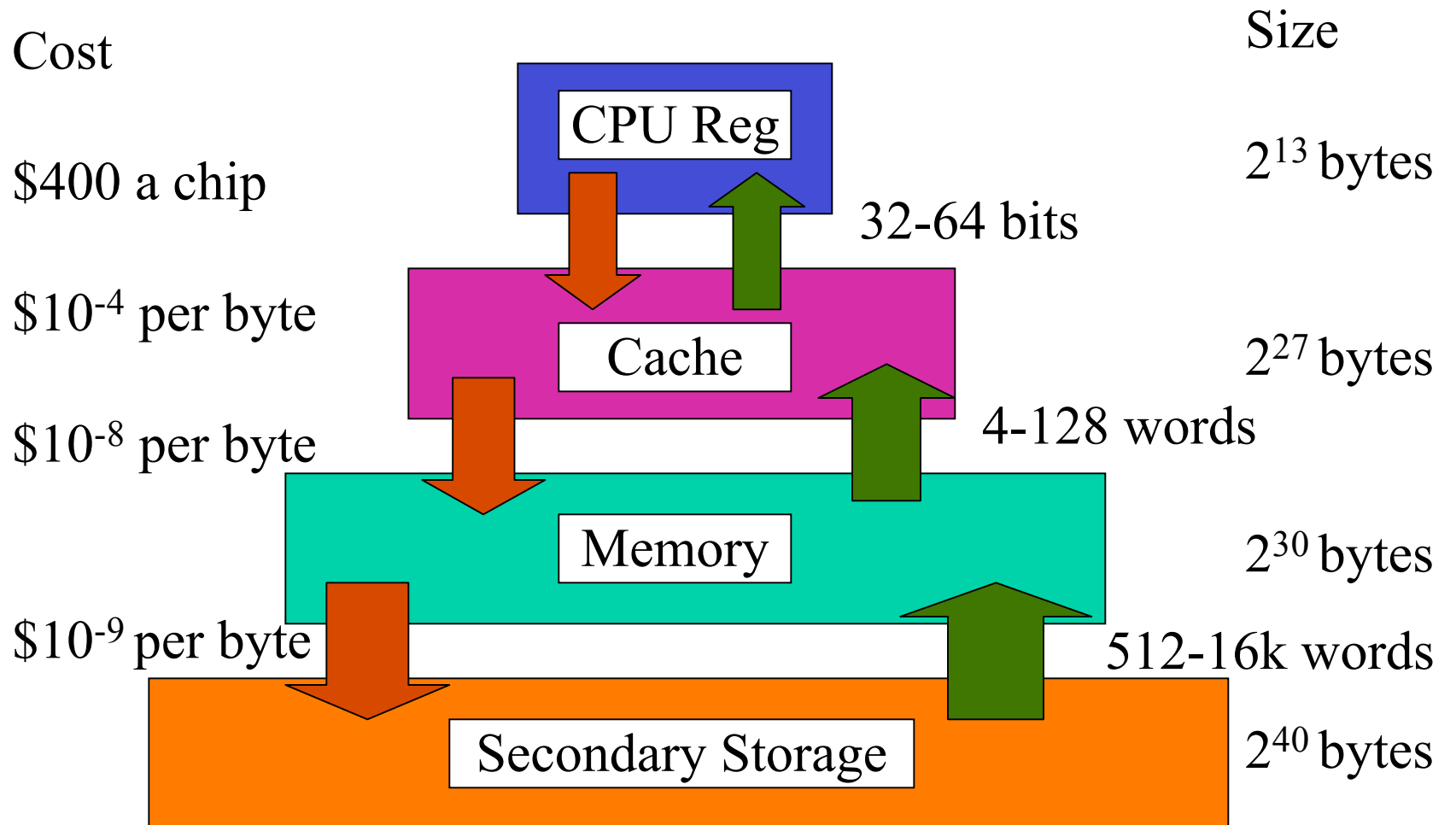
Binding Time Trade-offs (3)

Late binding

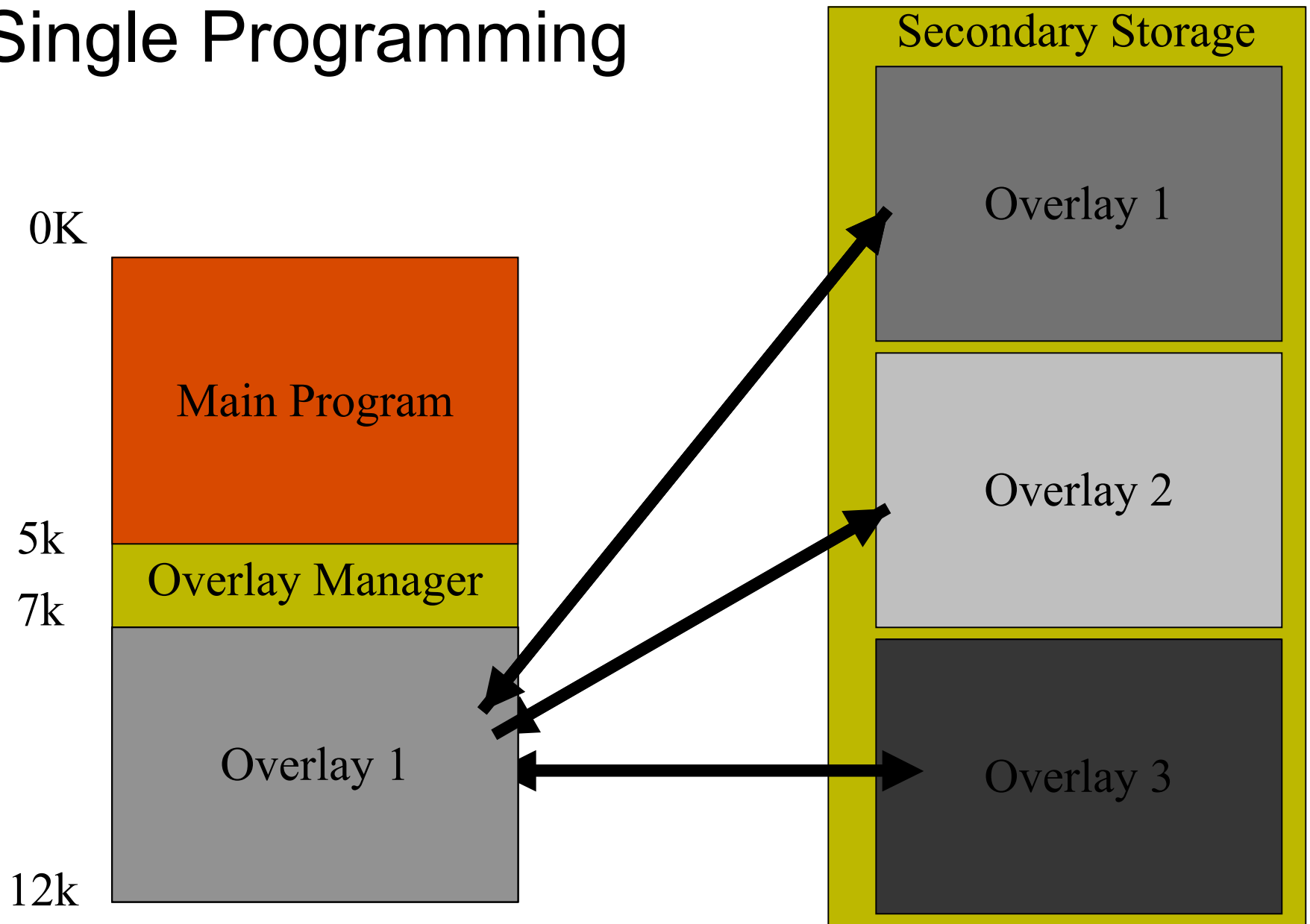
VM, dynamic linking and loading, overlaying, interpreting

- Code less efficient
- Checks must be done at run time
- + Flexible
- + Provides abstraction of hardware
- + Allows dynamic reconfiguration

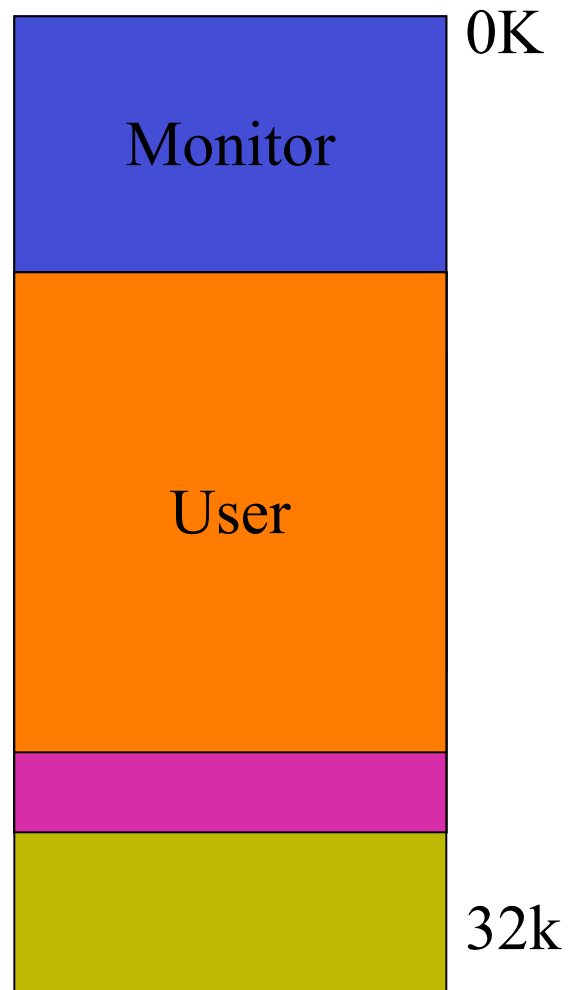
Storage Hierarchy



Single Programming



Resident Monitor



Multiprogramming

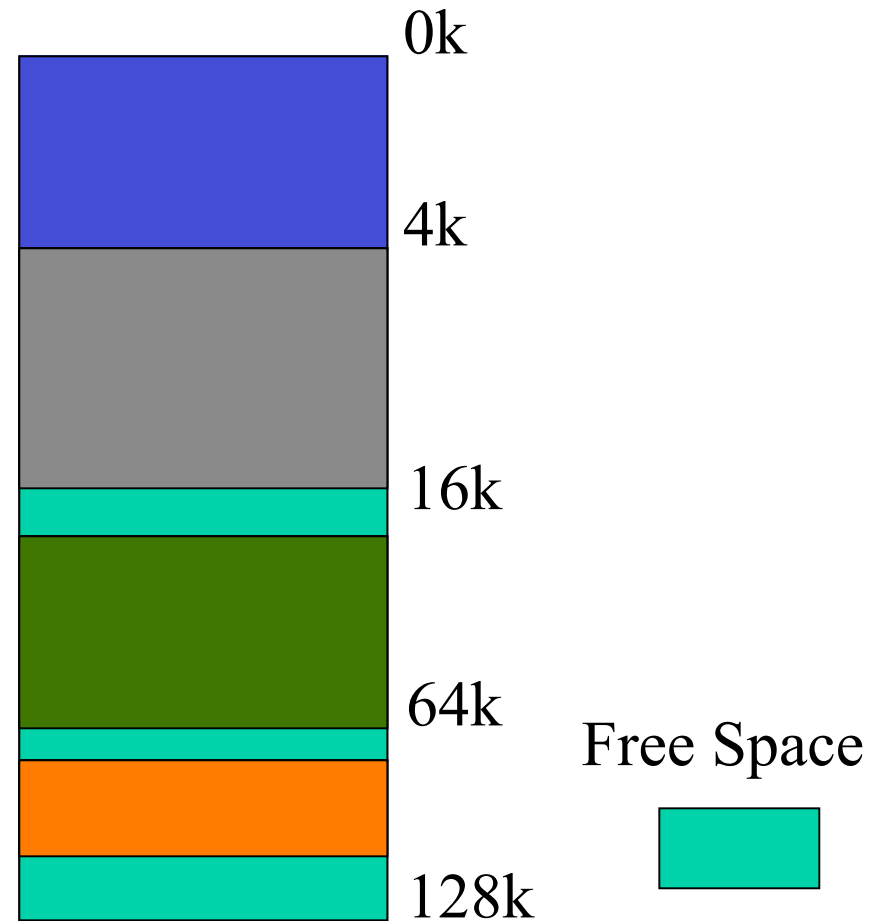
How to keep multiple programs in memory?

Multiprogramming with Fixed Partitions

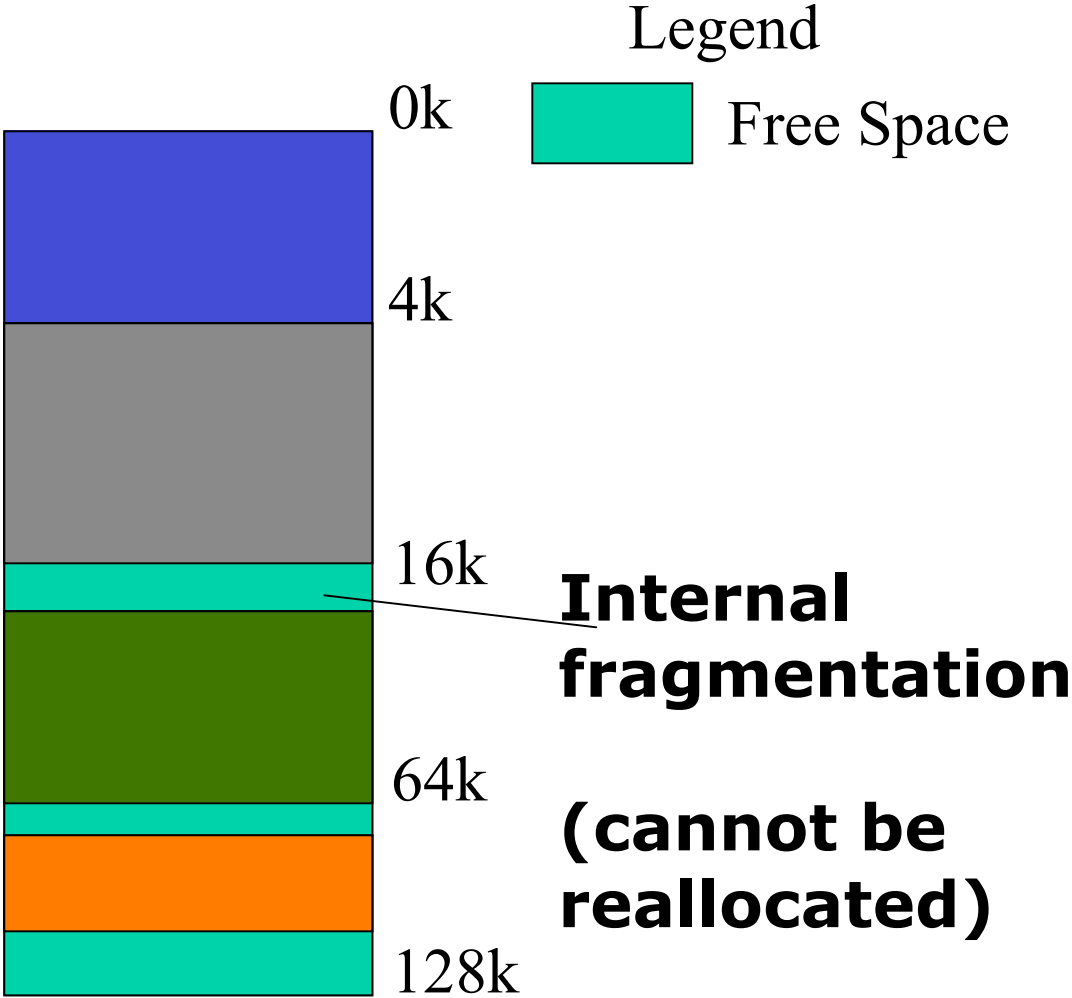
Divide memory into n
(possible unequal)
partitions.

Problem:

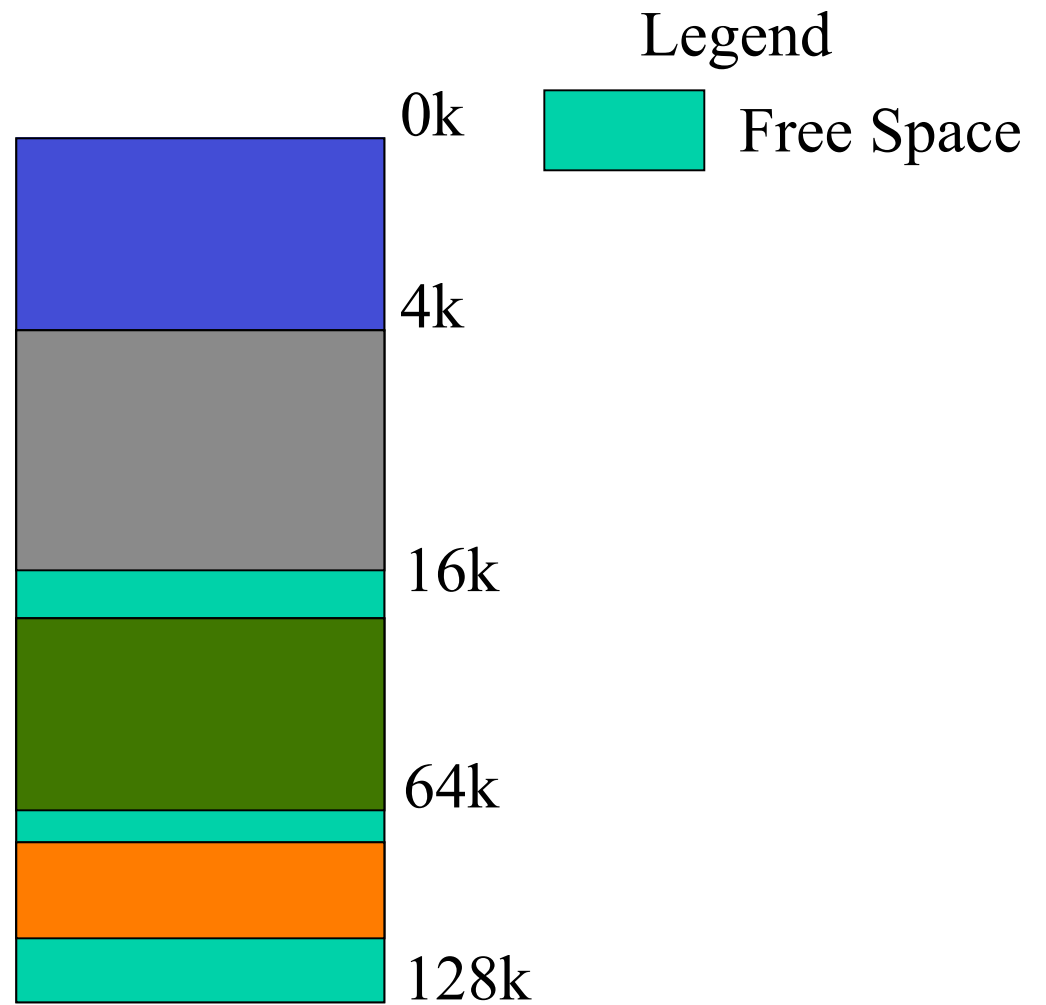
Fragmentation



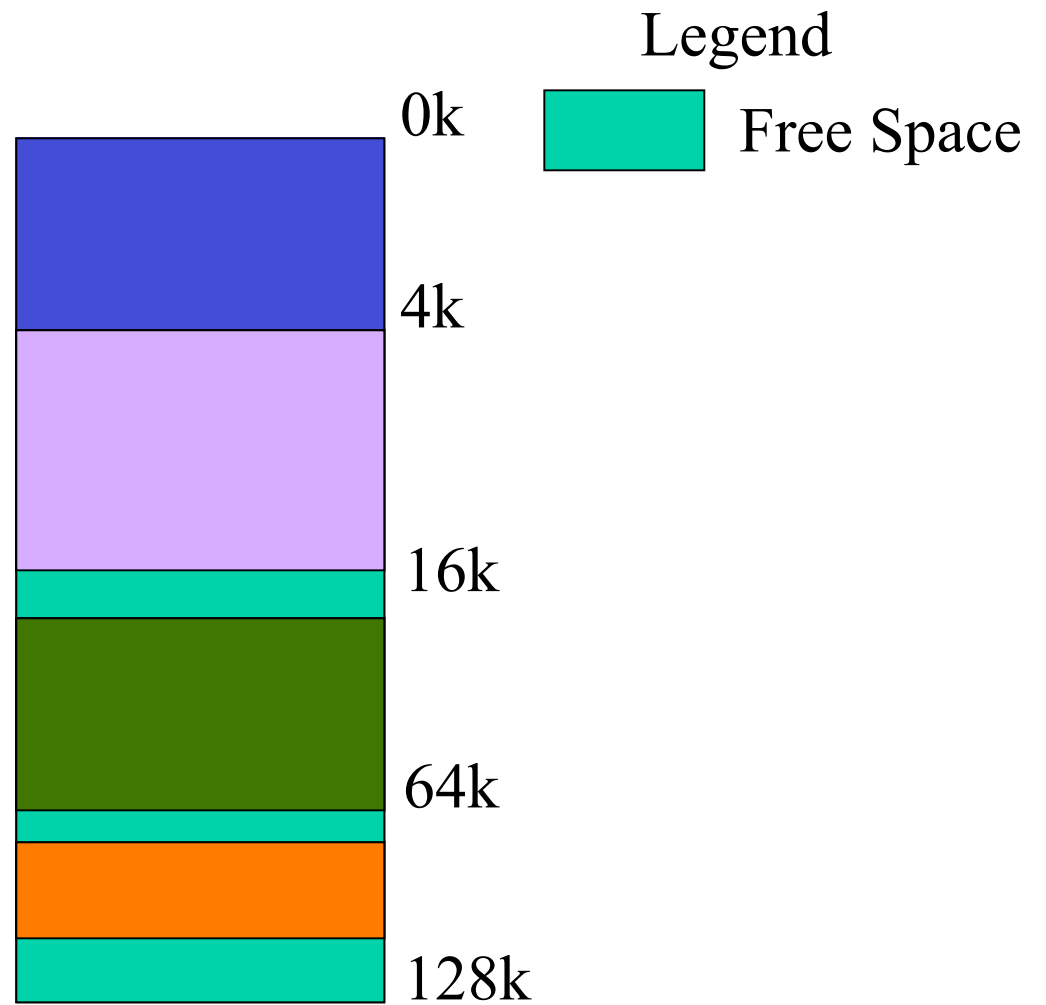
Fixed Partitions



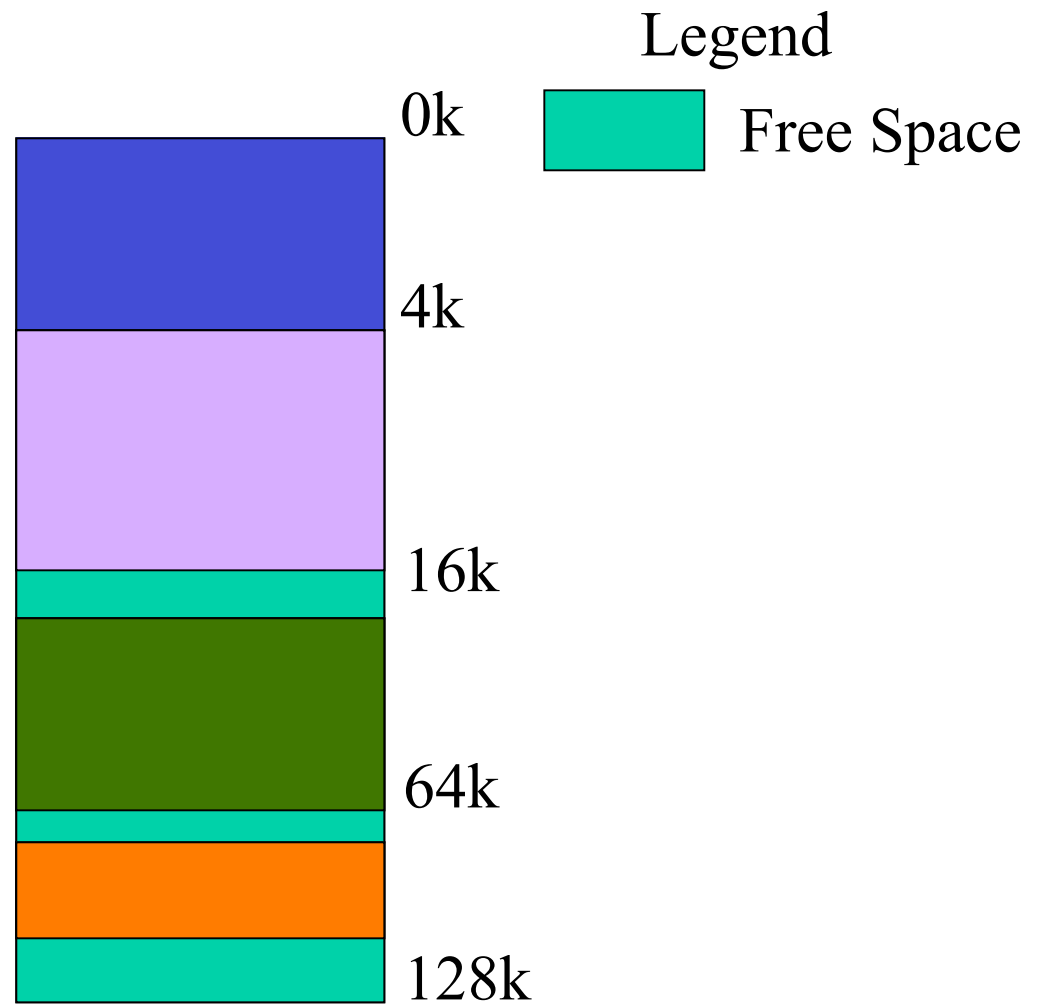
Fixed Partitions



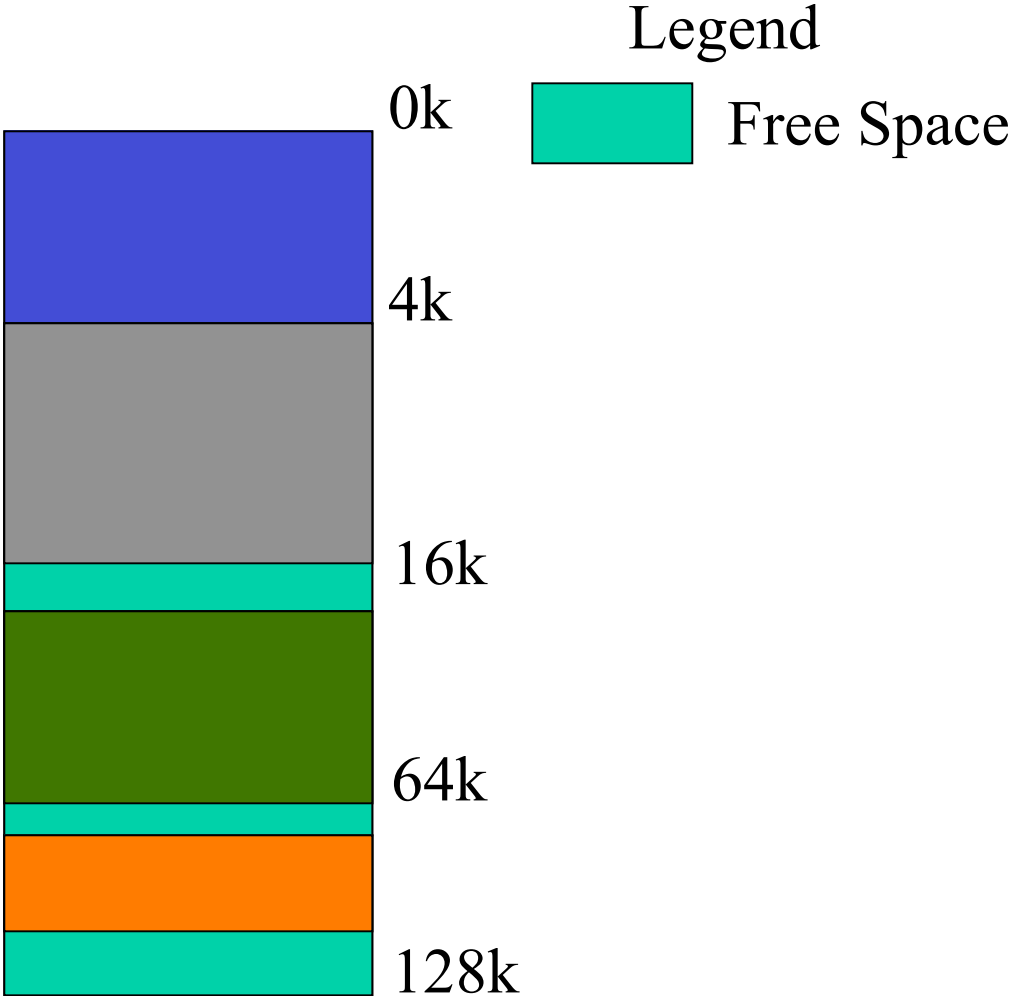
Fixed Partitions



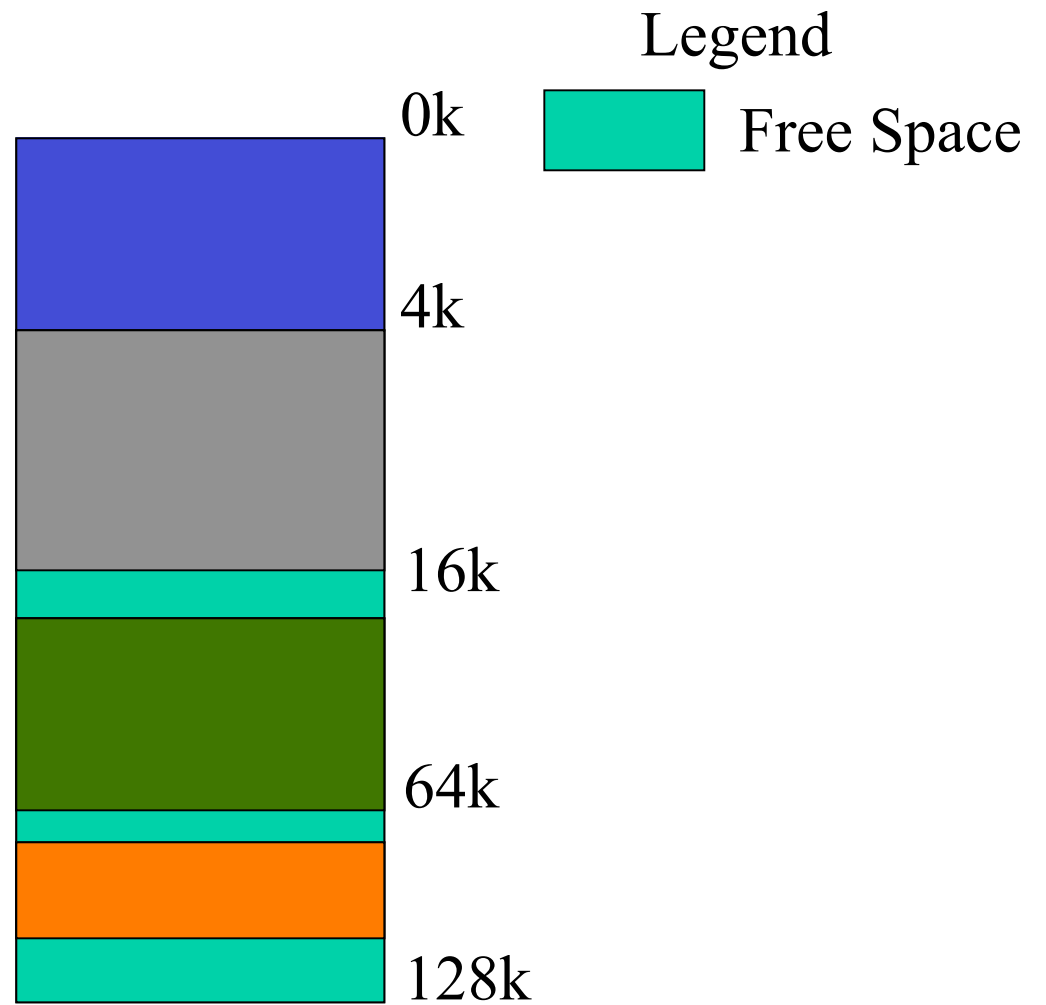
Fixed Partitions



Fixed Partitions



Fixed Partitions



Fixed Partition Allocation Implementation Issues

Separate input queue for each partition

Requires sorting the incoming jobs and putting them into separate queues

Inefficient utilization of memory

when the queue for a large partition is empty but the queue for a small partition is full. Small jobs have to wait to get into memory even though plenty of memory is free.

One single input queue for all partitions.

Allocate a partition where the job fits in.

Best Fit

Available Fit

Relocation

Correct starting address when a program should start in the memory

Different jobs will run at different addresses

When a program is linked, the linker must know at what address the program will begin in memory.

Logical addresses, Virtual addresses

Logical address space , range (0 to max)

Physical addresses, Physical address space

range (R+0 to R+max) for base value R.

User program **never sees** the real physical addresses

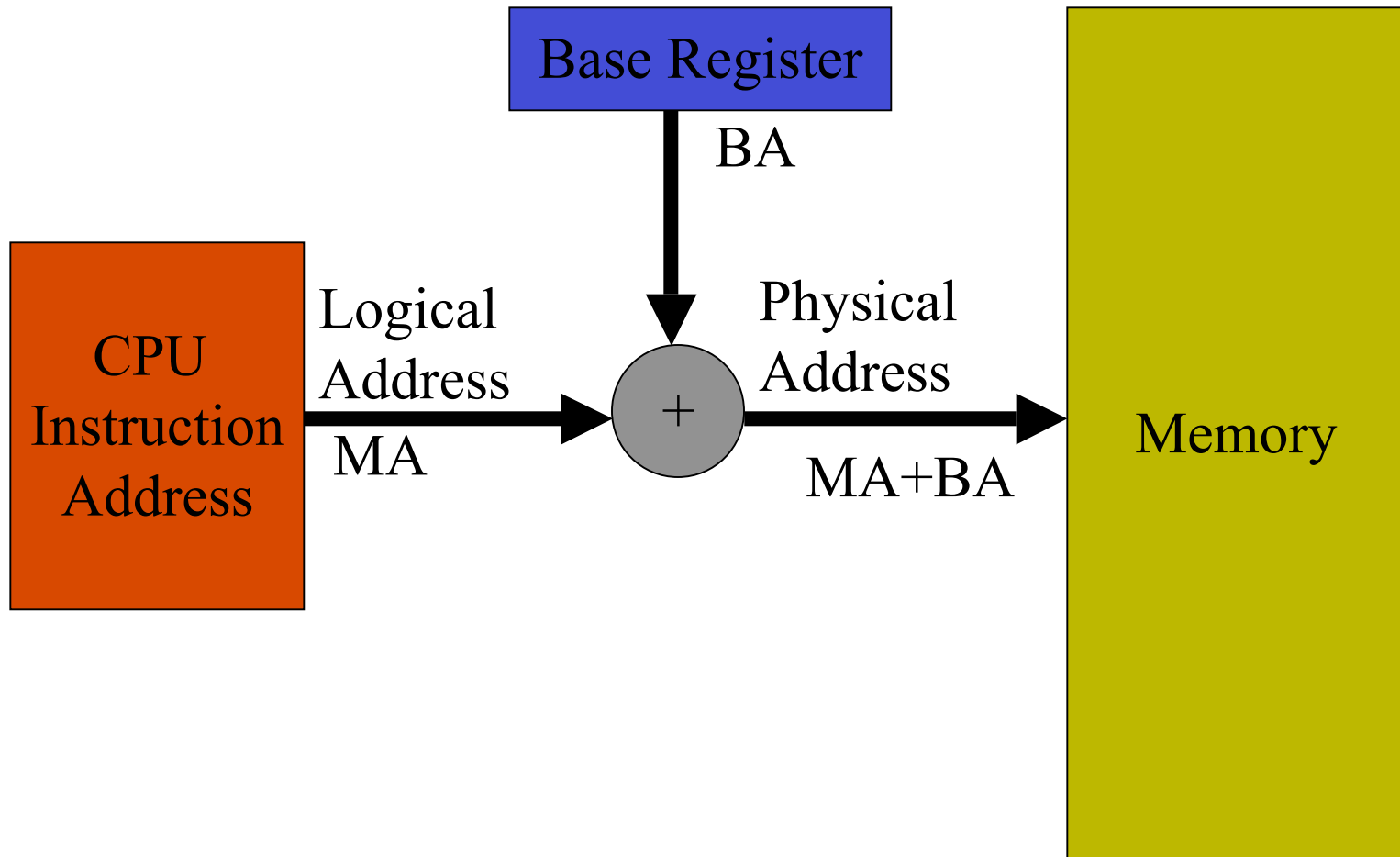
Memory-management unit (MMU)

map virtual to physical addresses.

Relocation register

Mapping requires hardware (MMU) with the base register

Relocation Register



Question 1 - Protection

Problem:

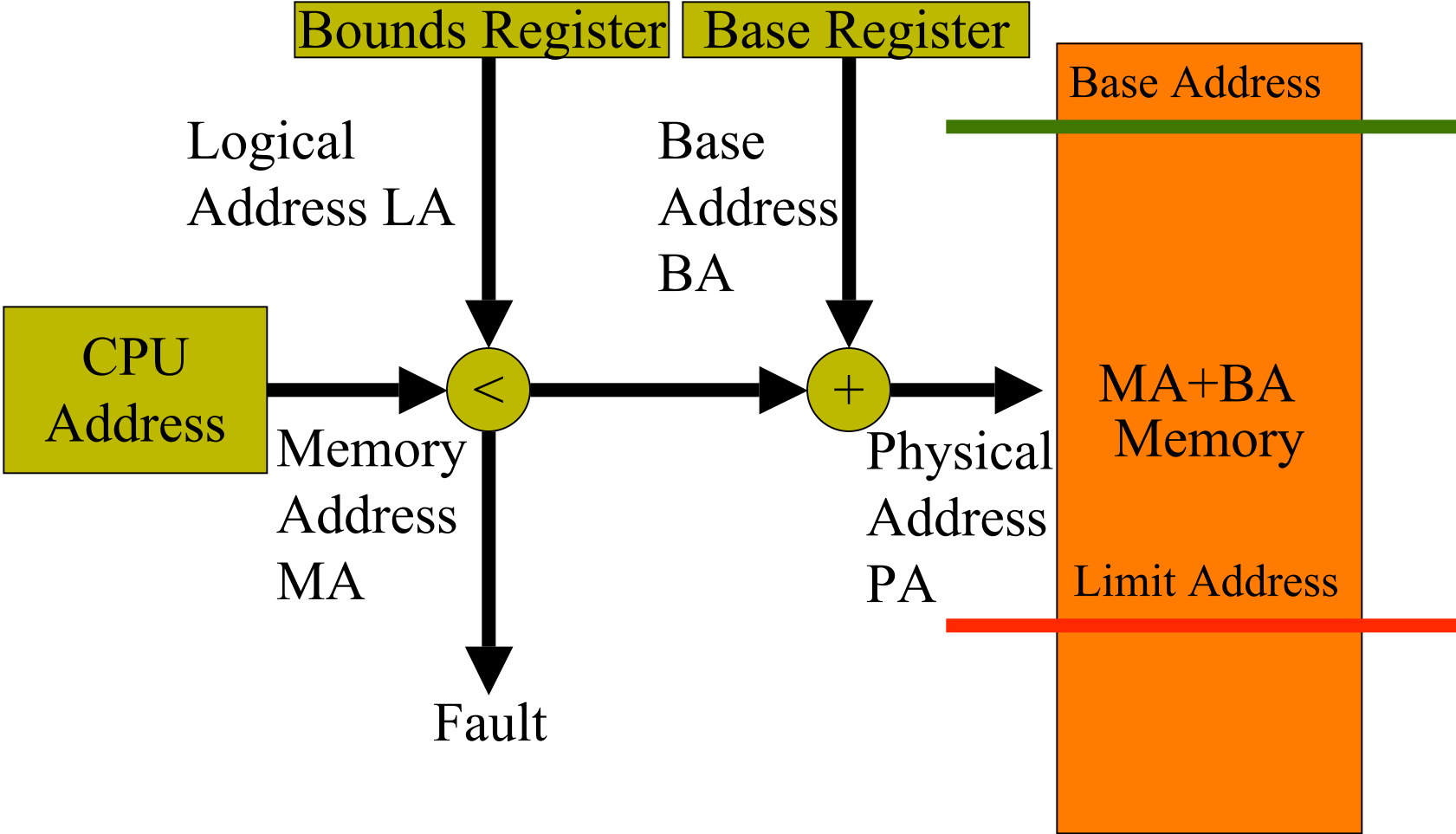
How to prevent a malicious process to write or jump into other user's or OS partitions

Solution:

Base bounds registers



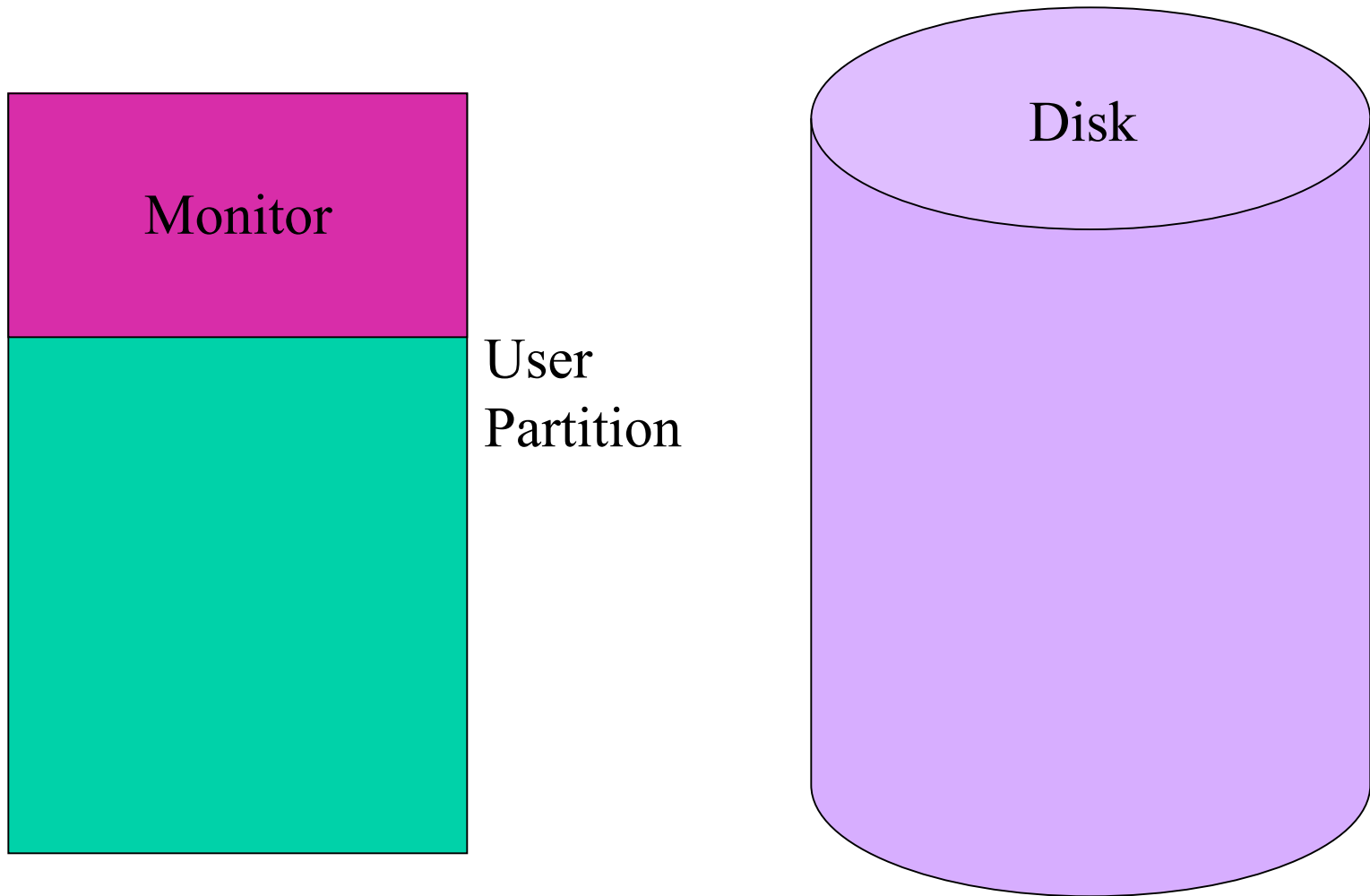
Base Bounds Registers



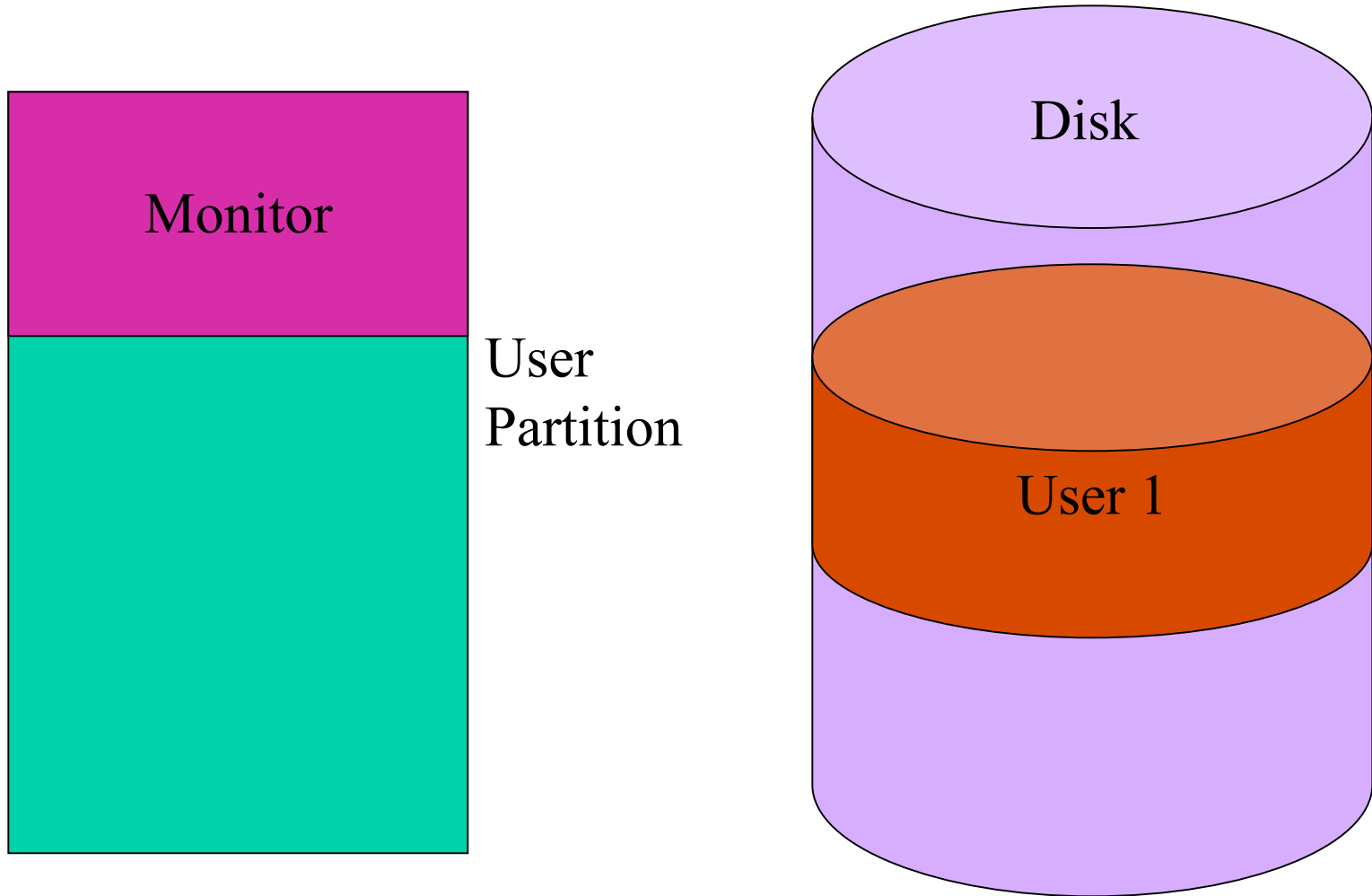
Question 2

What if there are more processes than what could fit into the memory?

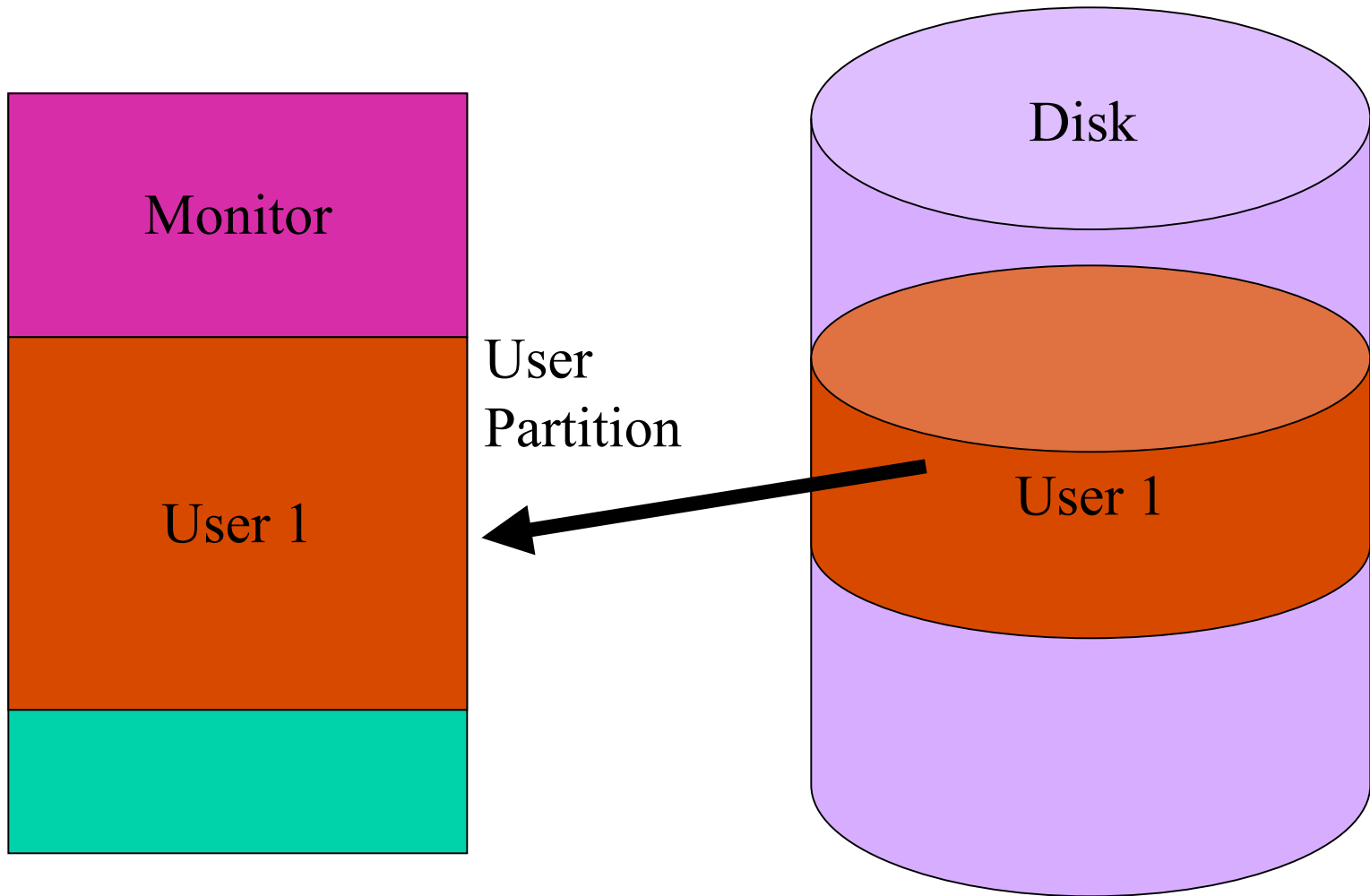
Swapping



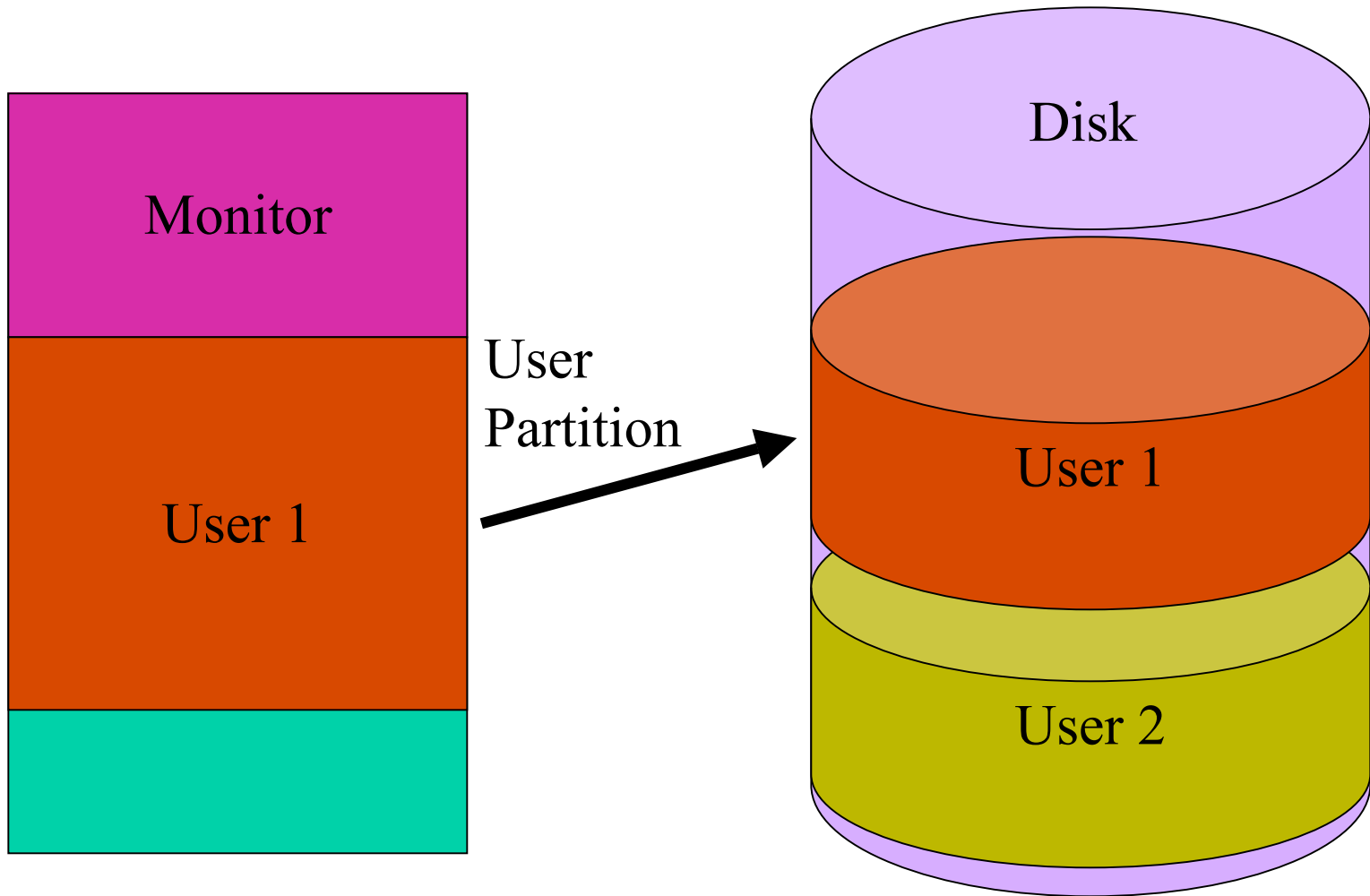
Swapping



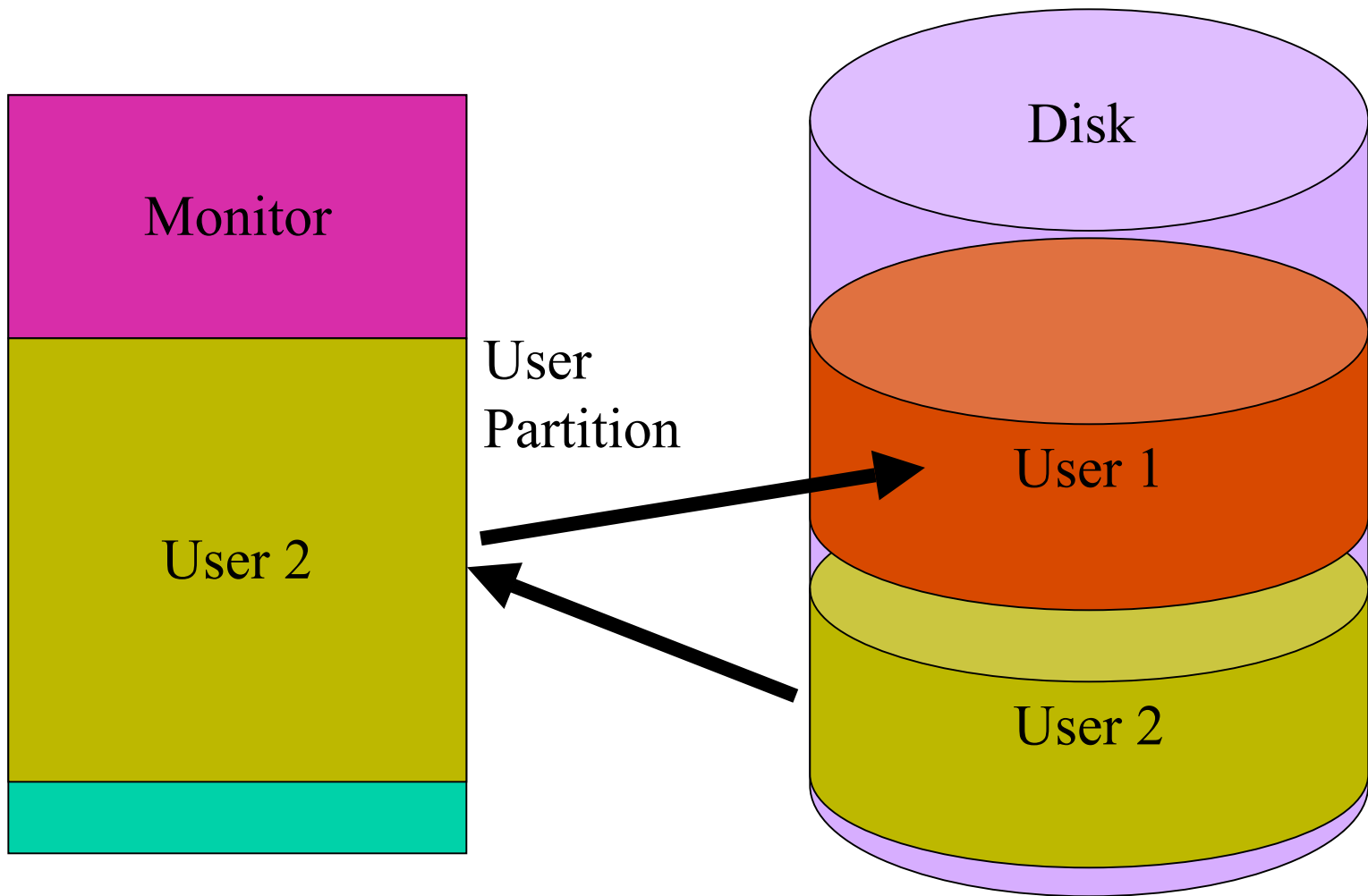
Swapping



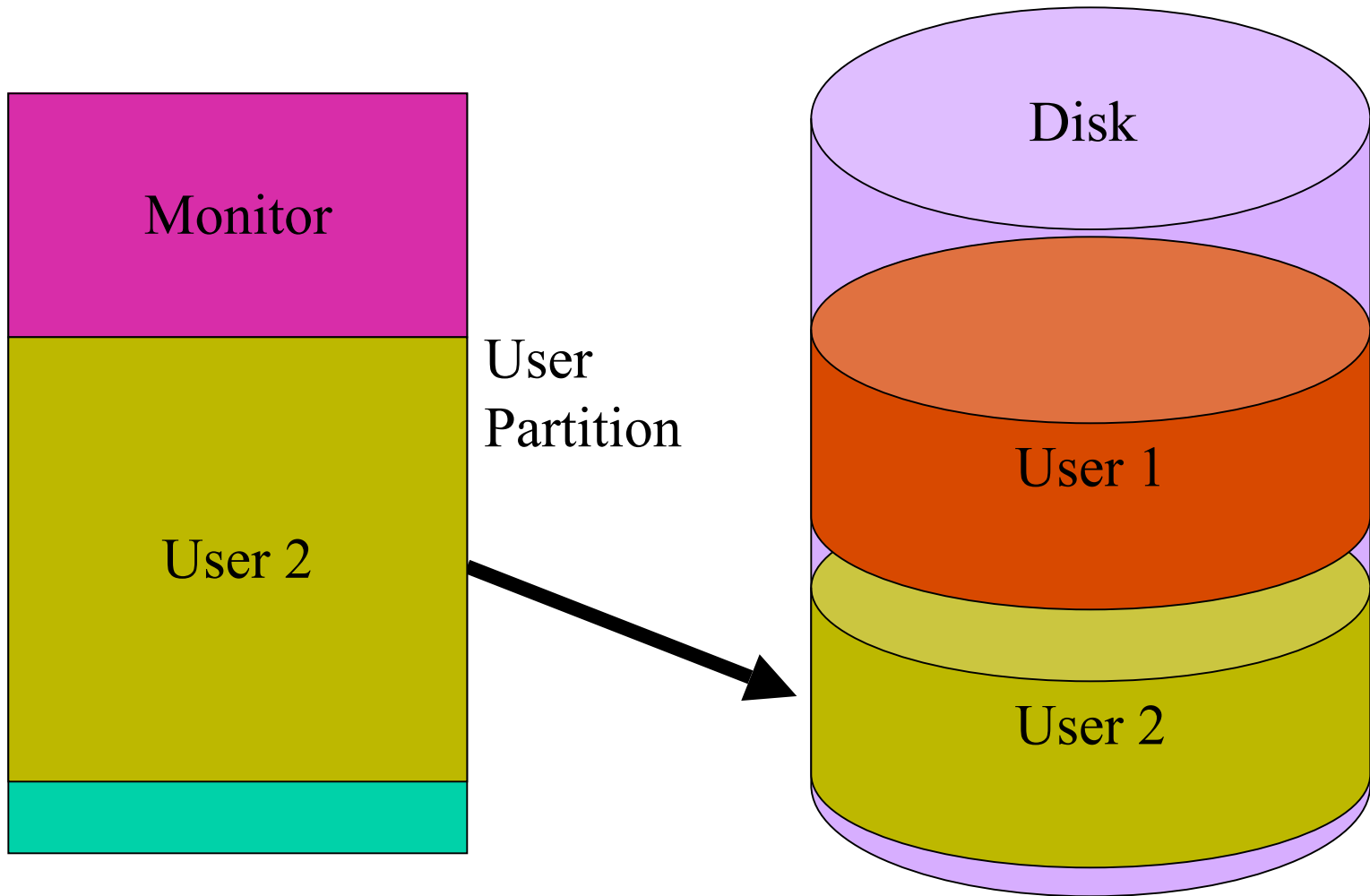
Swapping



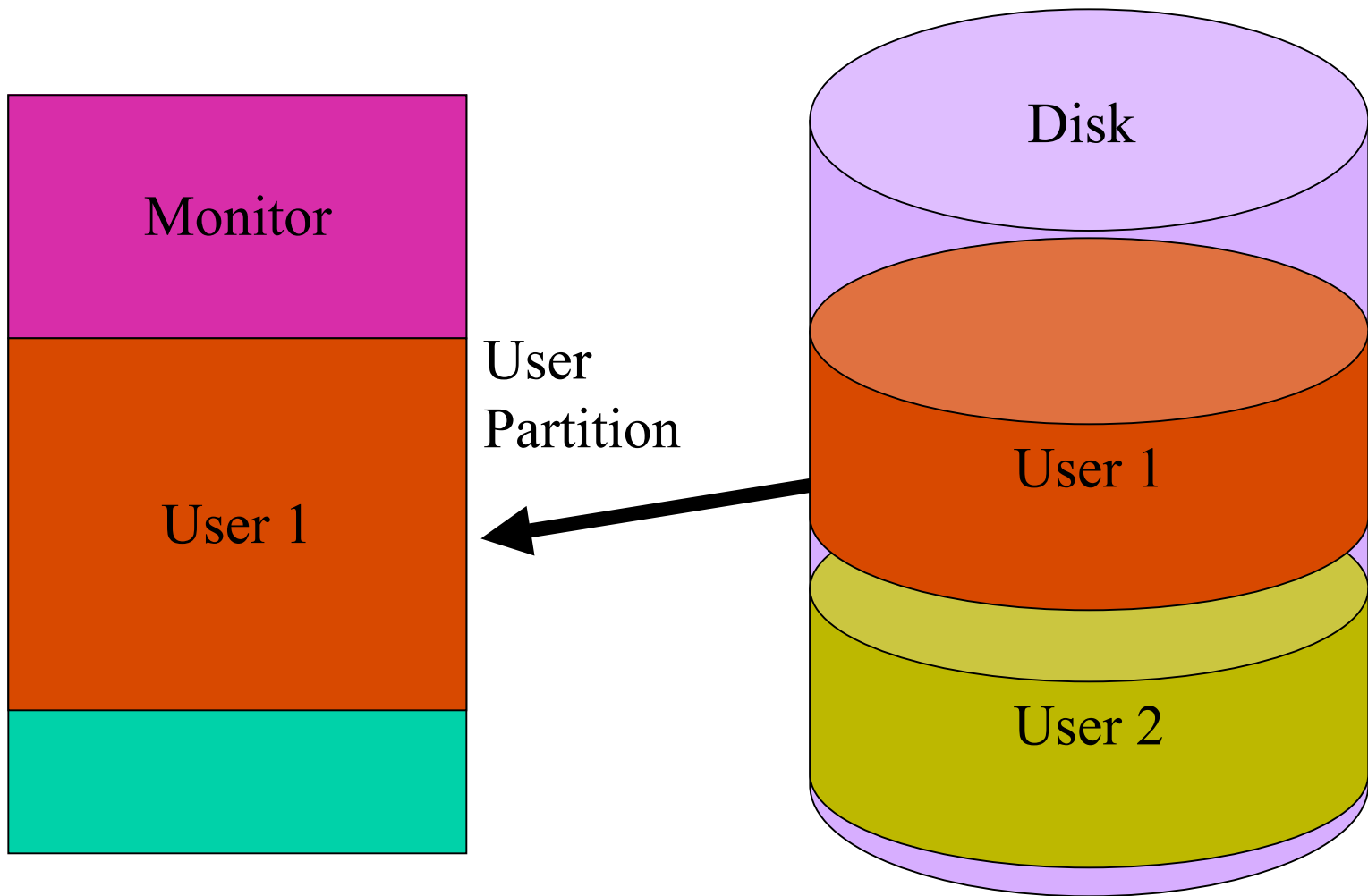
Swapping



Swapping



Swapping



Summary

Concept of Names/Addresses during

Compile

Link

Load

Running

Simple Programming

Overlays

Relocation Register

Multi-Programming

Fixed Partitions

Swapping