System Architecture

17 Address Spaces

Address Space Management
Linking & Loading
Swapping

January 14 2009
Winter Term 2008/09
Gerd Liefländer

Recommended Reading

- Bacon, J.: Operating Systems (5)
- Bovet, D.: Understanding the Linux Kernel (7)
- Nehmer, J.: Grundlagen modener BS, (4)
- Silberschatz, A.: Operating System Concepts (7)
- Stallings, W.: Operating Systems (7)
- Tanenbaum, A.: Modern Operating Systems (4)

Agenda

- Review on MM
- Motivation
  - Protection & Sharing
- Basic Notions
  - Address Scope
  - Address Space
  - Address Region
- Mapping of LAS → RAM
- Address Space Management
  - Single-Programming
  - Multi-Programming
    - Fixed-Partition
    - Variable-Sized Partition
- **Linking & Loading**

Address Space (AS) Concepts

- Physical AS (2^N bytes, N = address width of system/memory bus)
  - non-linearly addressable set of I/O-interfaces and RAM/ROM/… parts
  - can contain holes
- Logical AS (2^M bytes, M = address width of CPU)
  - Linearly addressable
- Virtual AS (2^K bytes)
  - $K > N$ with storage banking, overlay technique etc.
  - $K \leq M$

Basic Notions

- Physical address: reference of a specific RAM/ROM cell
- Logical address: program address used at run time to denote a specific data/instruction cell within the LAS of the executing program
- Relative address: logical address related to some fix point within the LAS of the executing program, e.g.
  - instruction pointer
  - start address of program
  - stack frame pointer
- Virtual address: mapped logical address into virtual AS (in many cases this mapping is 1:1)*

*For simplification in our course logical address = virtual address

Why Address Spaces?

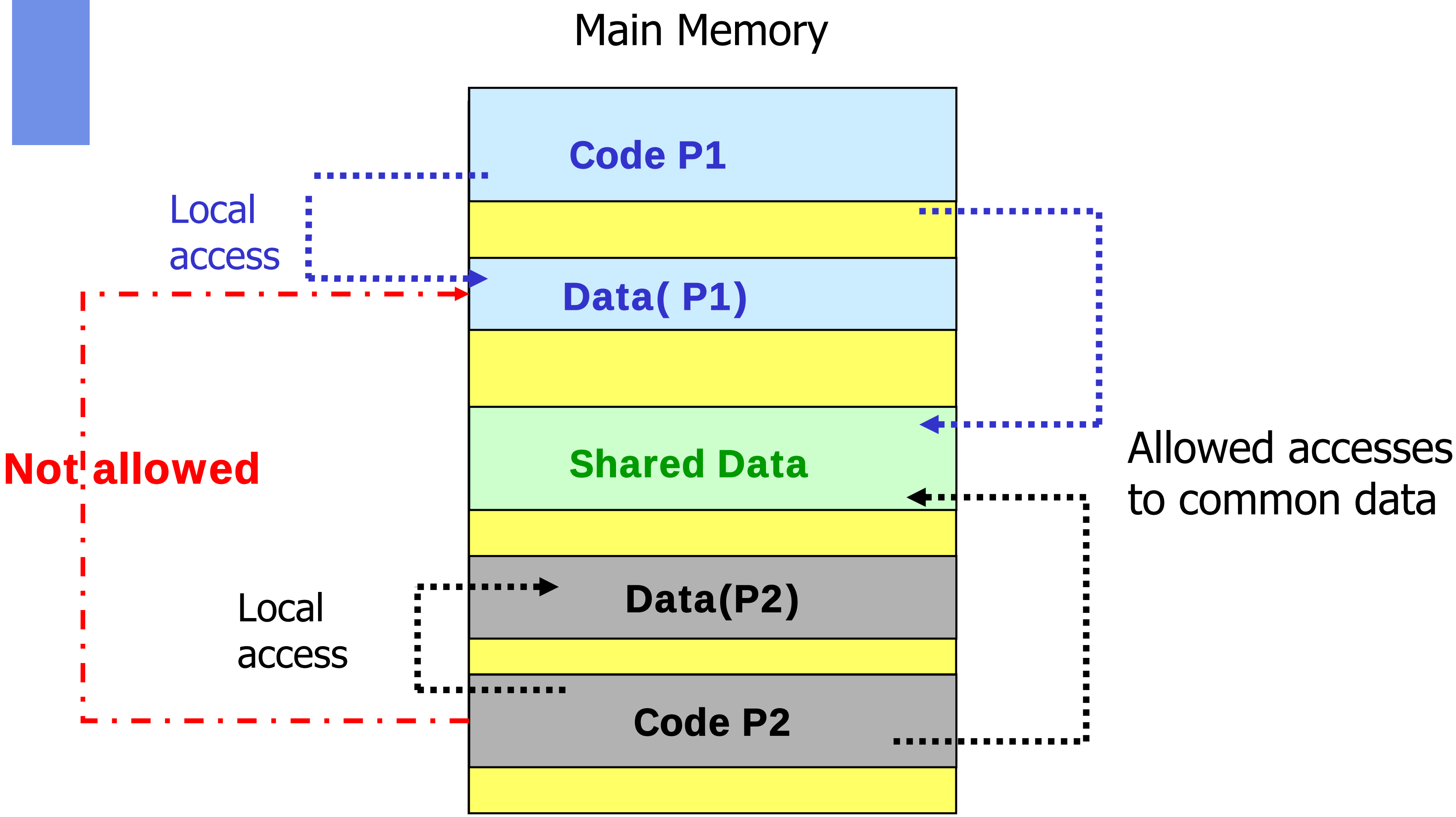
- In order to achieve the intended results, each application runs in its own address LAS ⇒
  - No unwanted interference with another application will occur, i.e. each LAS executes within a "protected area"
- Each shared object & communication path (channel, mailbox etc.) with another LAS has an impact on
  - robustness (e.g. due to race conditions)
  - security (cooperation with untrusted software)
- Only, for efficiency reasons we offer explicit LAS sharing, e.g. Linux or UNIX "shared memory", i.e. parts of n>2 LAS are identical

Why Sharing?

- Sharing when
  - n>2 tasks/processes want to cooperate
  - n>2 tasks want to use common code/data in order to reduce load overhead

- Typical examples for shared objects:
  - Libraries
  - Code (e.g. C compiler)
  - Common data (e.g. buffers)

Sharing


Main Memory
Code P1
Local
access
Data(P1)
Not allowed
Shared Data
Allowed accesses
to common data
Data(P2)
Local
access
Code P2

Protection and Sharing?

- Define logical entities with
  - guarded borders and
  - common address regions

$\Rightarrow$

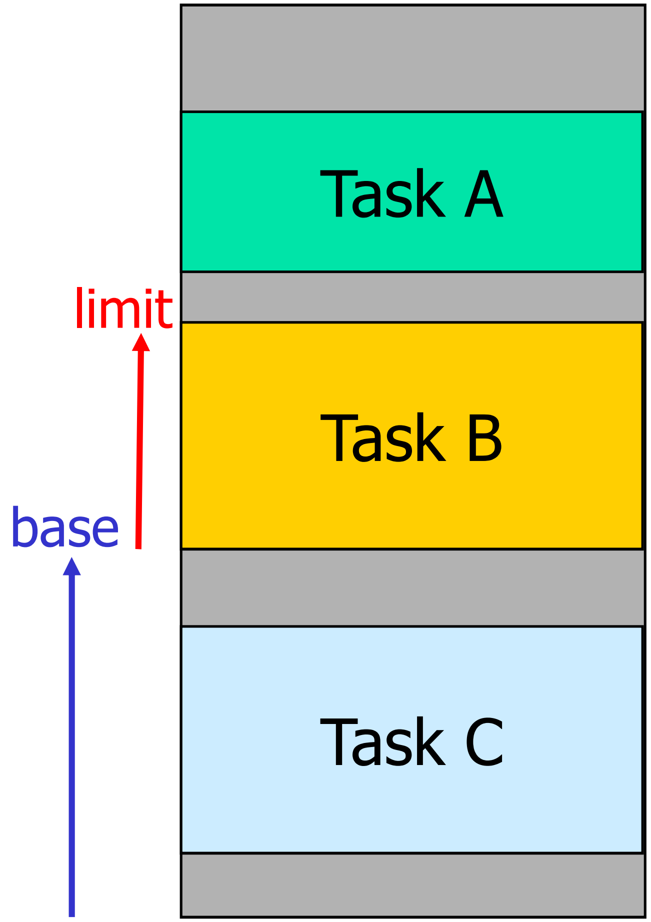
2. Basic Abstraction of System Architecture:

Address Space

(Address) Region

HW Support for Runtime Protection

- Need two registers to run task B
  - Base register
  - Limit register
- Need to add an appropriate offset to a logical address
  - Achieves relocation
  - Protects memory locations lower than base
  - Protects memory location higher than base + limit



Base and Limit Register

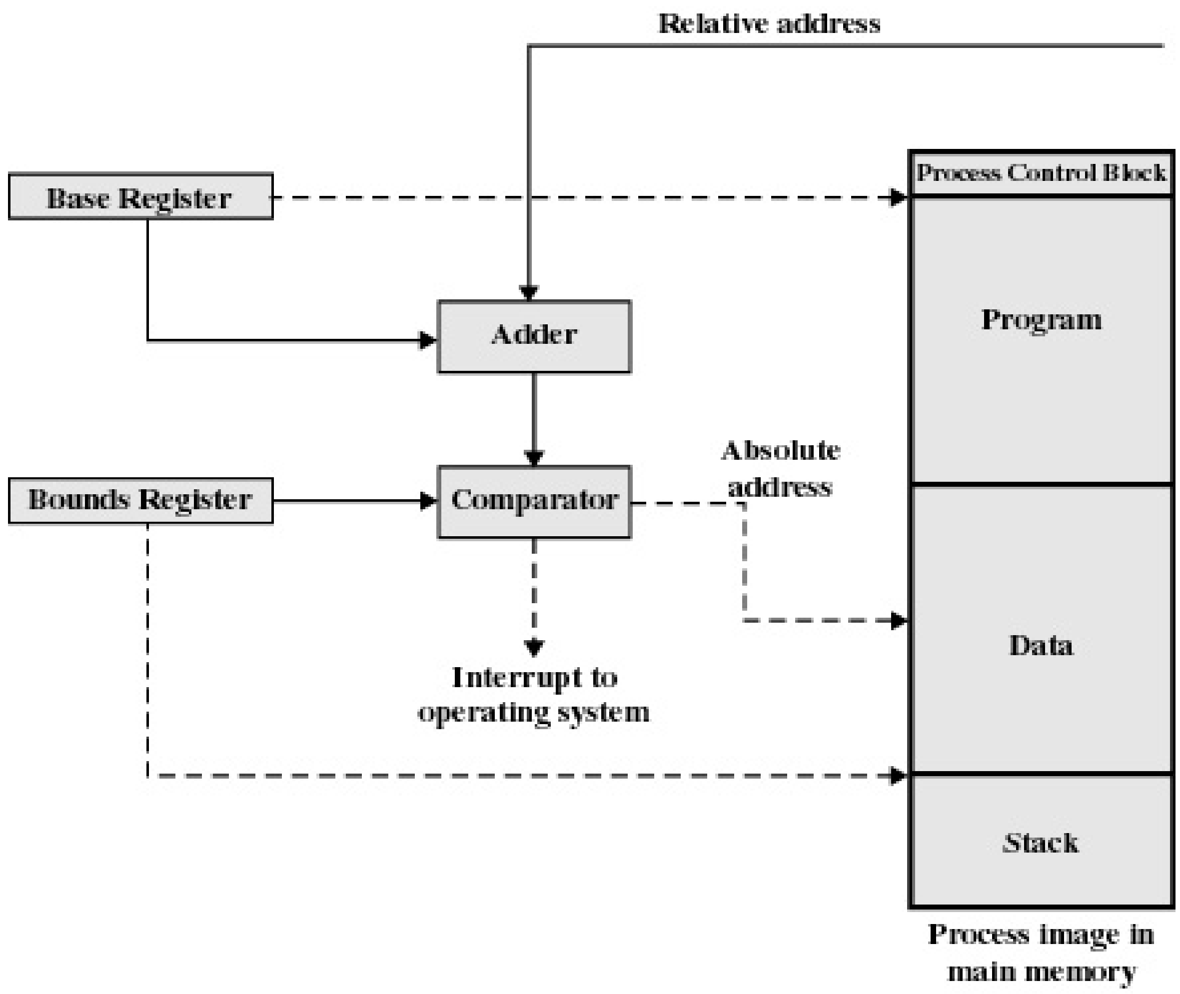
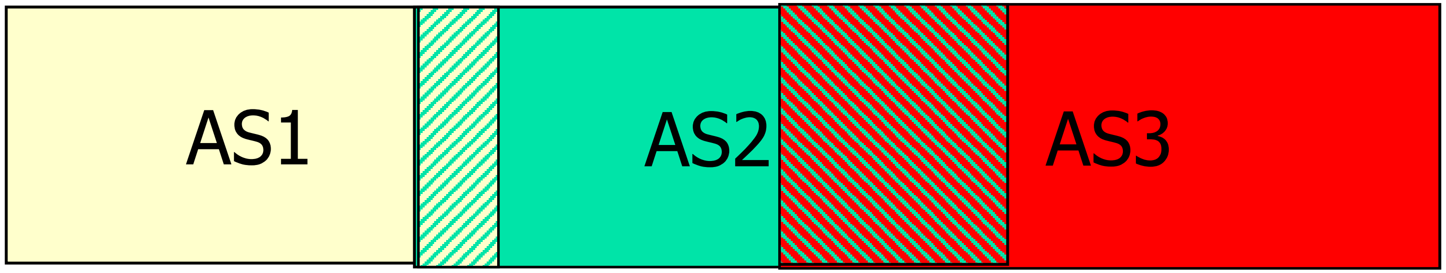


Figure 7.8 Hardware Support for Relocation

Summary: Base & Limit Register

- Disadvantages
  - Allocated memory must be contiguous, i.e. it can be hard to find a fitting free memory partition
  - Complete task/process must be in memory, i.e. if AS contains holes, i.e. the corresponding mapped memory parts are not used
  - No scalable support for partially sharing of ASes

Sharing Problem



Consequence:

⇒ Shared AS regions should be mapped independently of their ASes

⇒ Each AS region can be mapped individually

Implementing Sharing efficiently?

- Whenever we are able to map parts of an AS separately sharing is no longer a problem
- Solution is scalable (provide usage counter)

Logical Organization

Programmers view towards software:

Sampling of

- code entities (thread, procedure etc.) and
- data entities (struct, array, module, object etc.)

SW entities have different access characteristics, e.g.:

- Execute only (e.g. code)
- Read only (e.g. catalogue)
- Read-Write

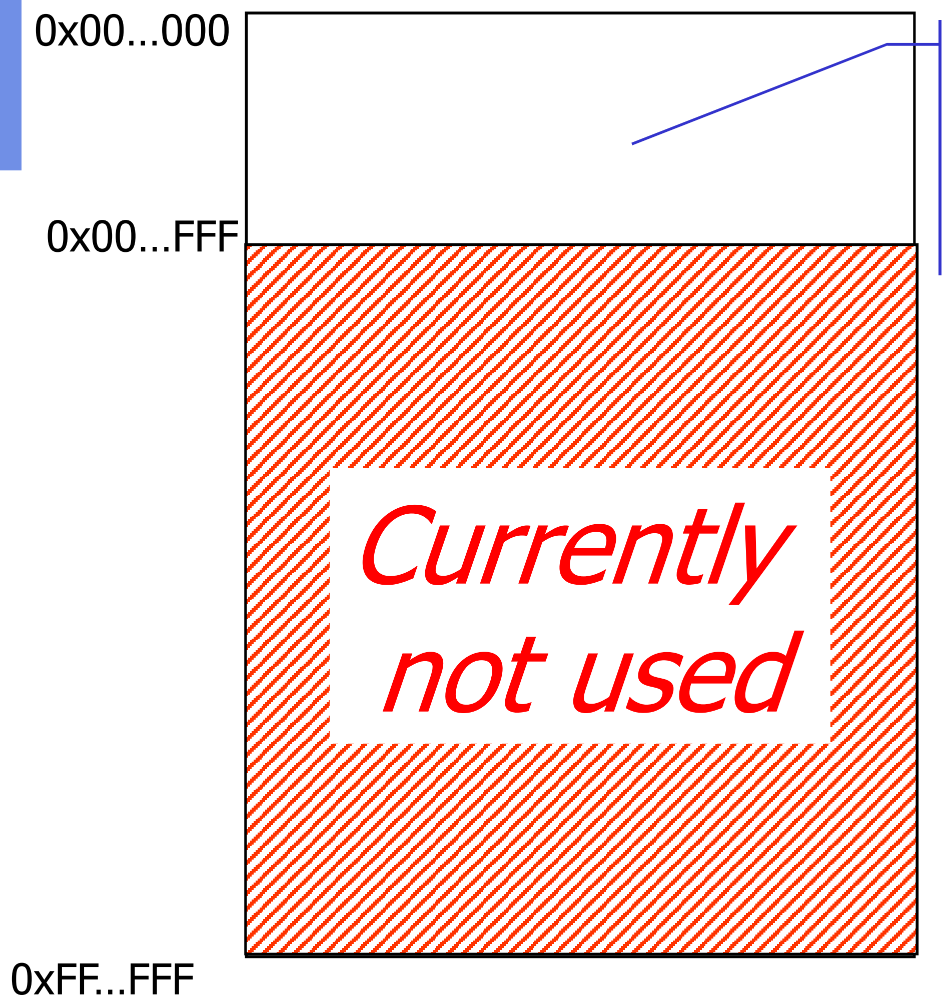
- Standard HW supports this idea, however, some commodity OSes don't use this HW feature

Logical Organization

<u>Definition:</u> The <u>Address Scope</u>* limits the range of addresses a compiler, linker, and loader can give to an executable

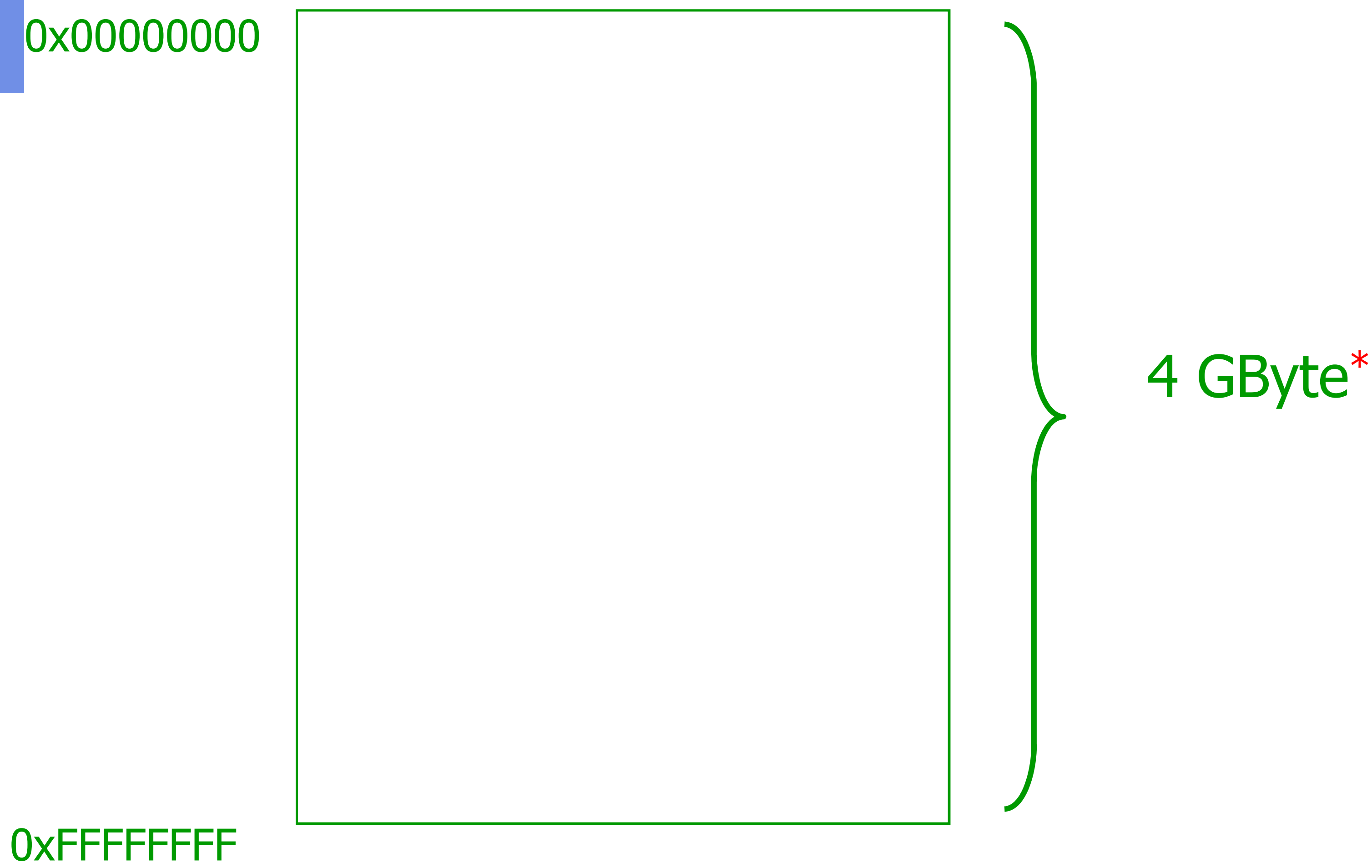
[1]KA specific

Logical Address Scope

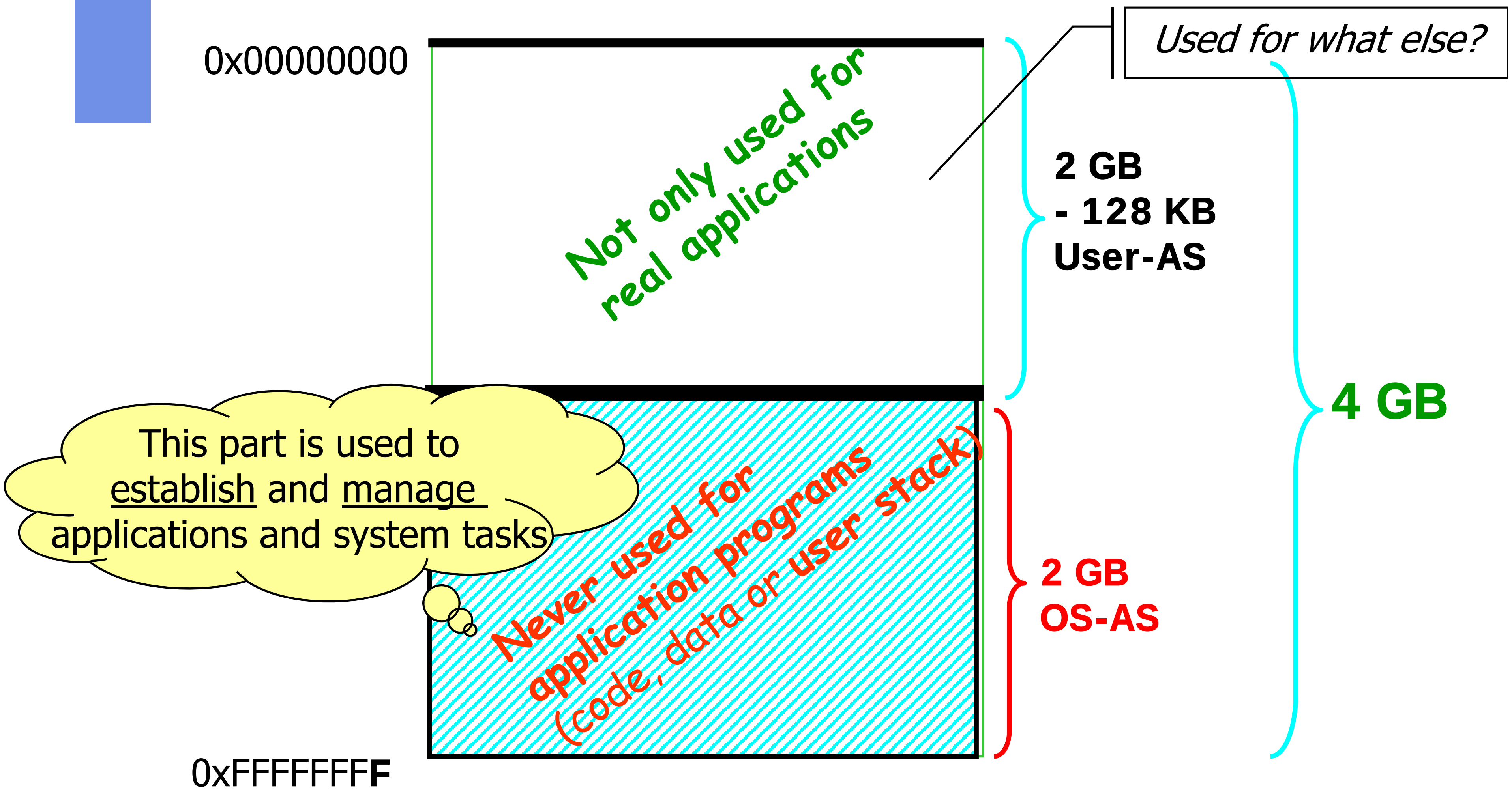


Some systems use only a small part of the whole address scope, e.g. current **64-bit machines** only use 40…48 address bits

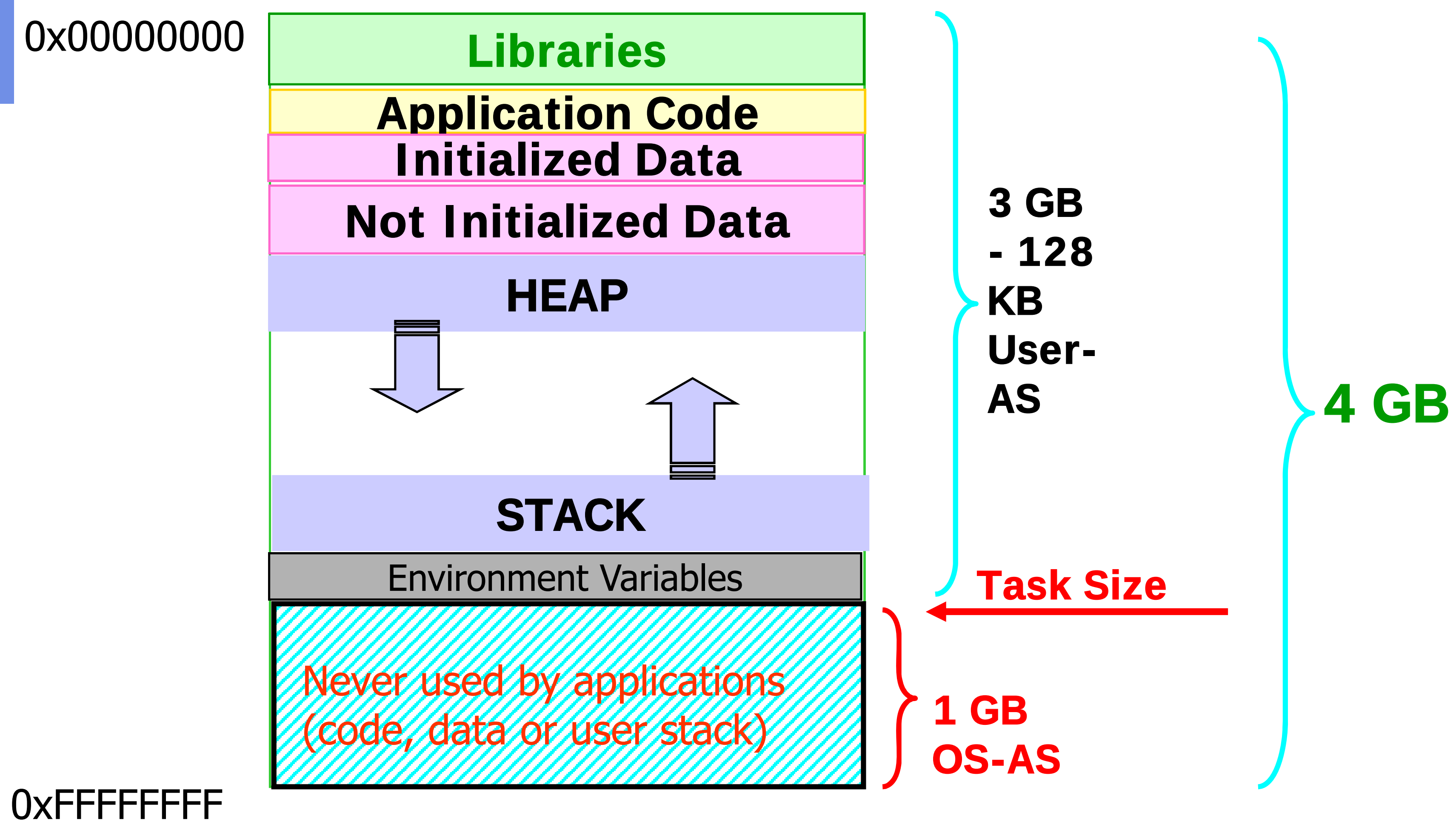
Intel’s x86 Address Scope



Splitting Address Scope (Win. NT)



Linux Address Space Layout



Logical Address Space (1)

Logically associated parts -mapped to available addresses of the address scope- form another logical unit:

Definition: A ("logical") address space LAS is the range of addresses within the address scope accessible for an "executable task", i.e. either for a process (= single-threaded task) or for a multi-threaded task

Task or process can be an application or a system server

Logical Address Space (2)

Question:

What will happen if a thread of a task tries to reference a logical address not belonging to its LAS?

⇒ Exception is raised: "address violation"

⇒ Remember: Main purpose of a LAS is:

!!! PROTECTION !!!

Address Space* (3)

2 implementation for AS:

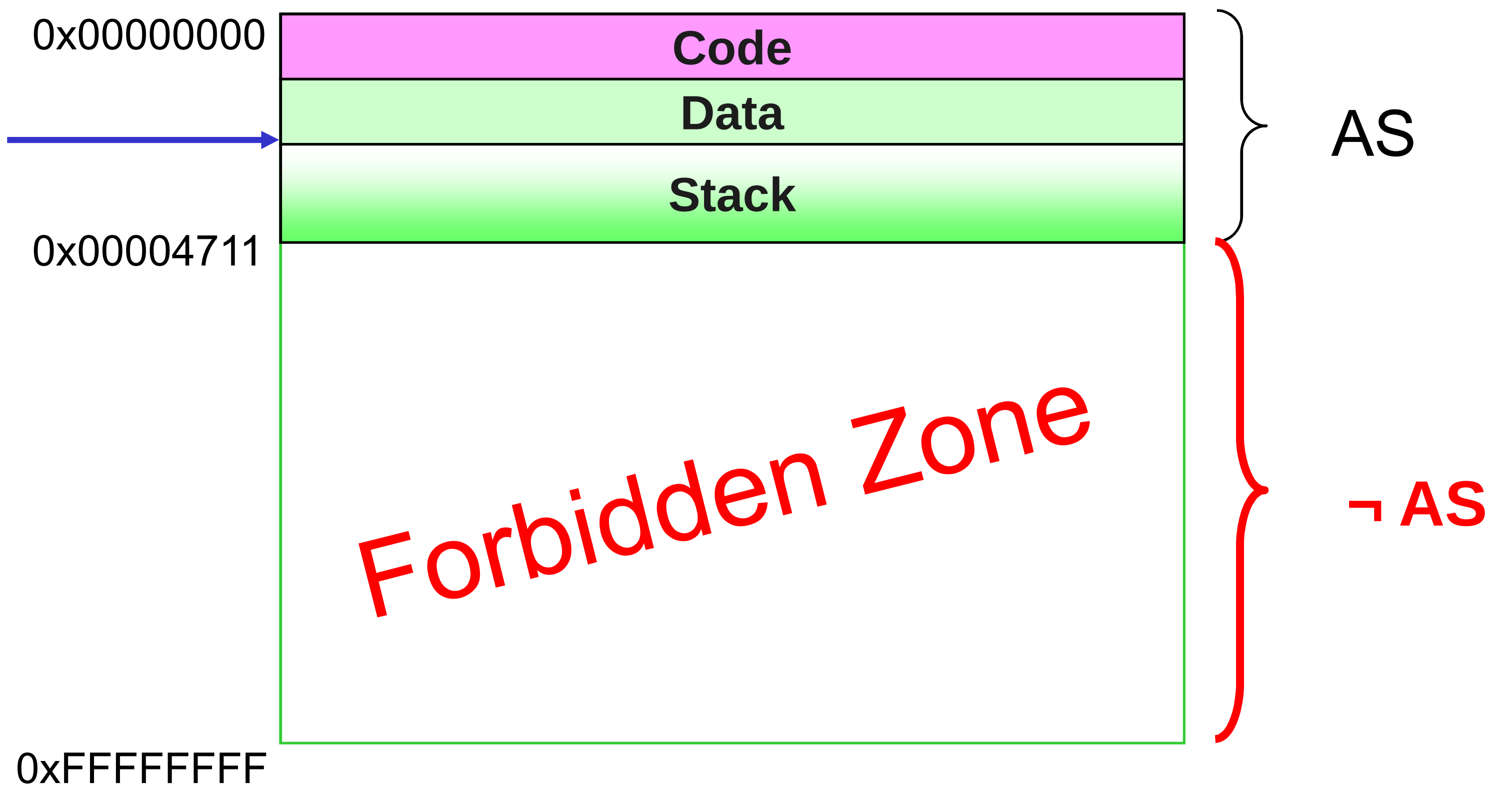
- Contiguous AS
- Dispersed AS

2 characteristics of AS:

- Fixed
  - No changes of the AS size at run time
- Dynamic
  - Growing and shrinking parts of AS a run time

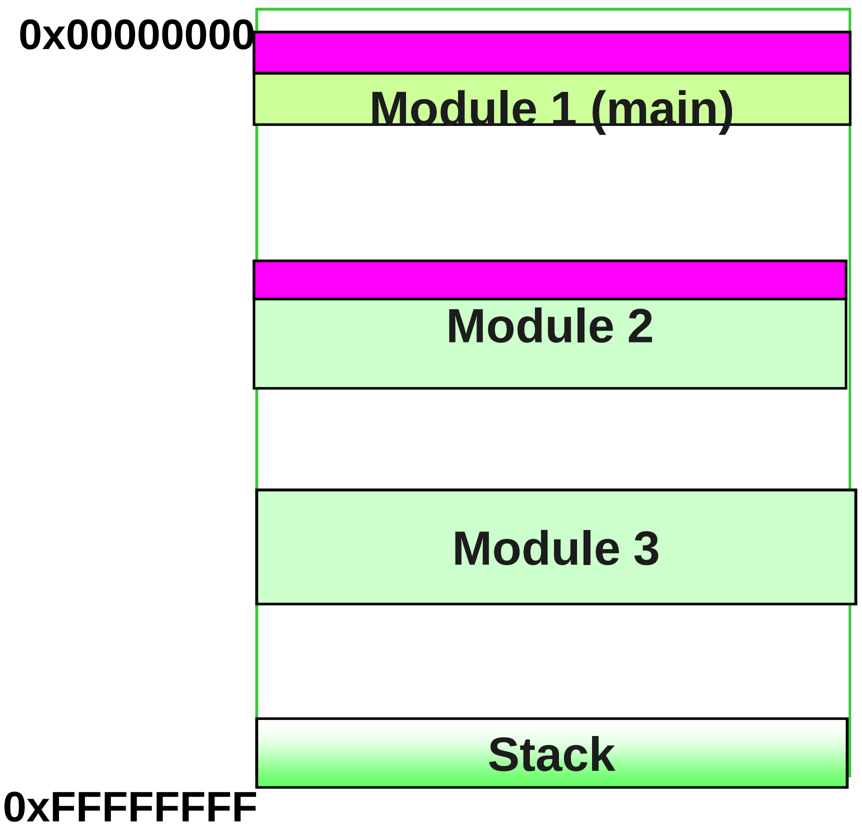
*In the following slides AS = Logical Address Space

Contiguous Address Space



Discuss pros and cons of this concept

Dispersed Address Space



Pros and cons of this concept?

Address Regions

Address spaces may overlap each other,
sharing common portions of their ASs

$\Rightarrow$

How to name private or shared contiguous portions of an AS?

Definition: A *contiguous* AS block is a *region* (e.g. a segment)

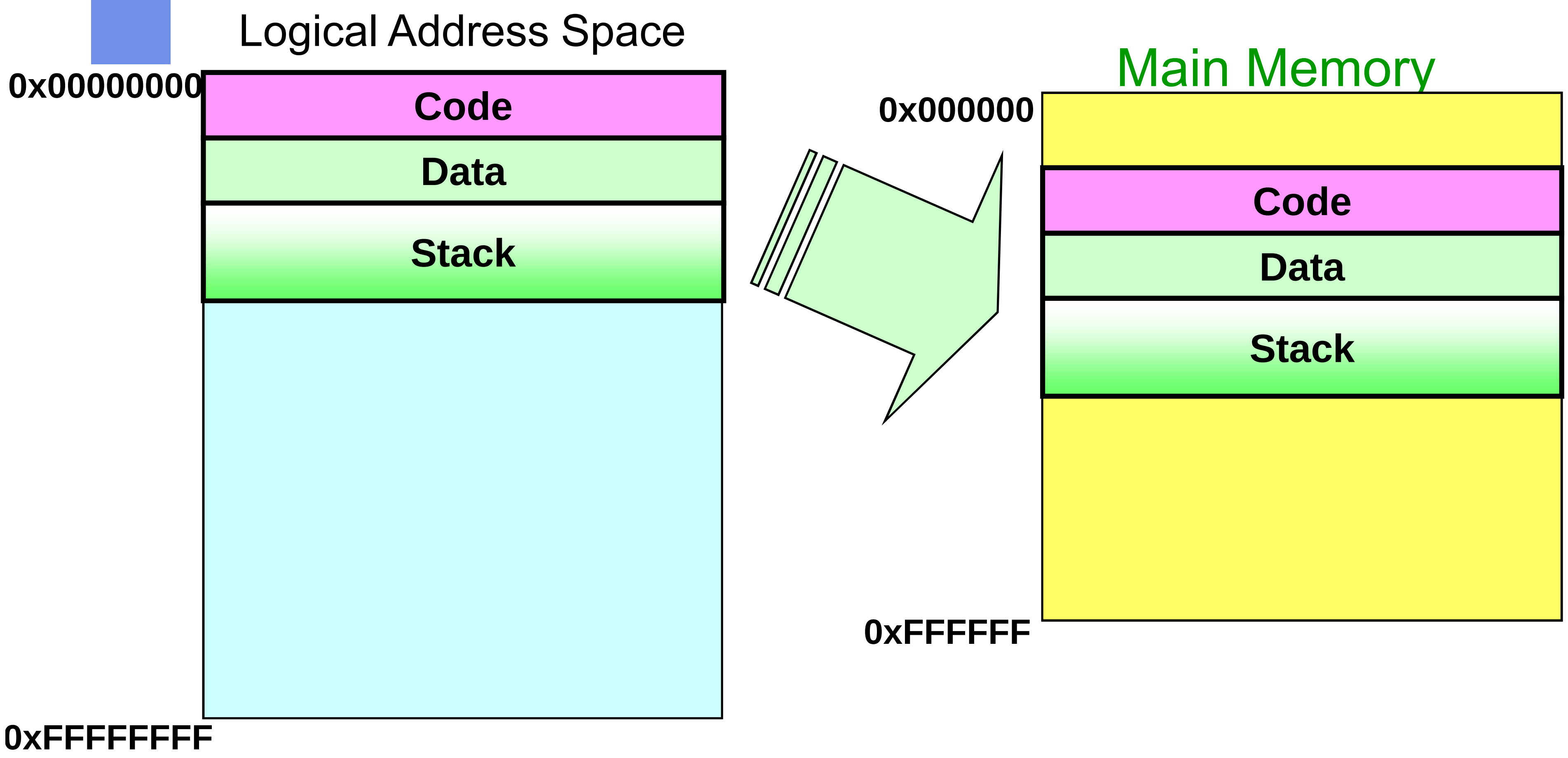
Typical examples in Unix: code(text), data and stack

Mapping AS to RAM

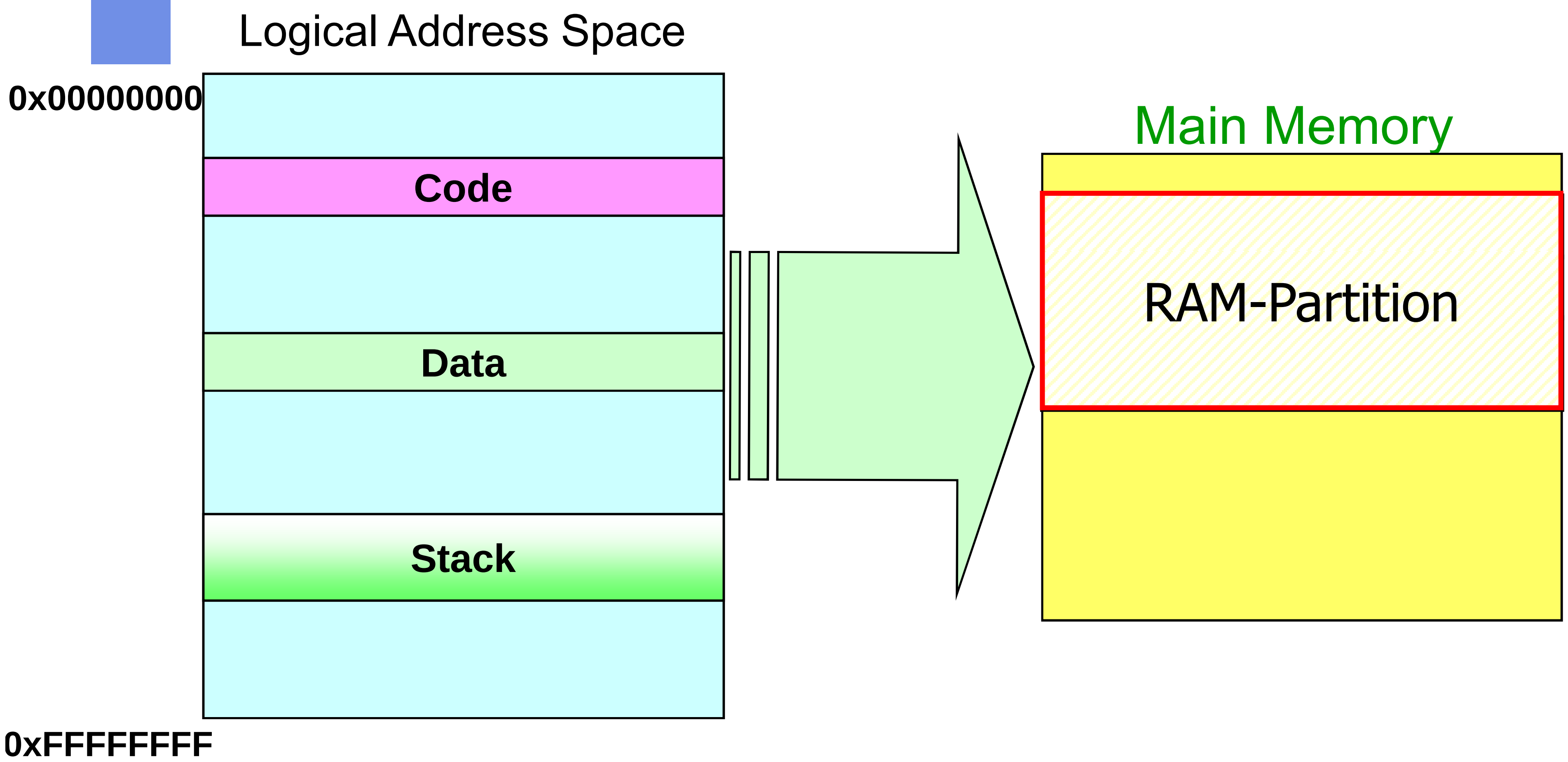
Mapping can be done orthogonal to the layout of a logical and of the physical address space:

- Complete AS (AS is either mapped or not at all)
- Portions of the AS
  - Fixed sized logical portions (pages) or
  - Variable sized logical portions (segments)
- Contiguous memory partition (MP) or
- Non contiguous memory partitions
  - Fixed sized memory portions or
  - Variable sized memory portions

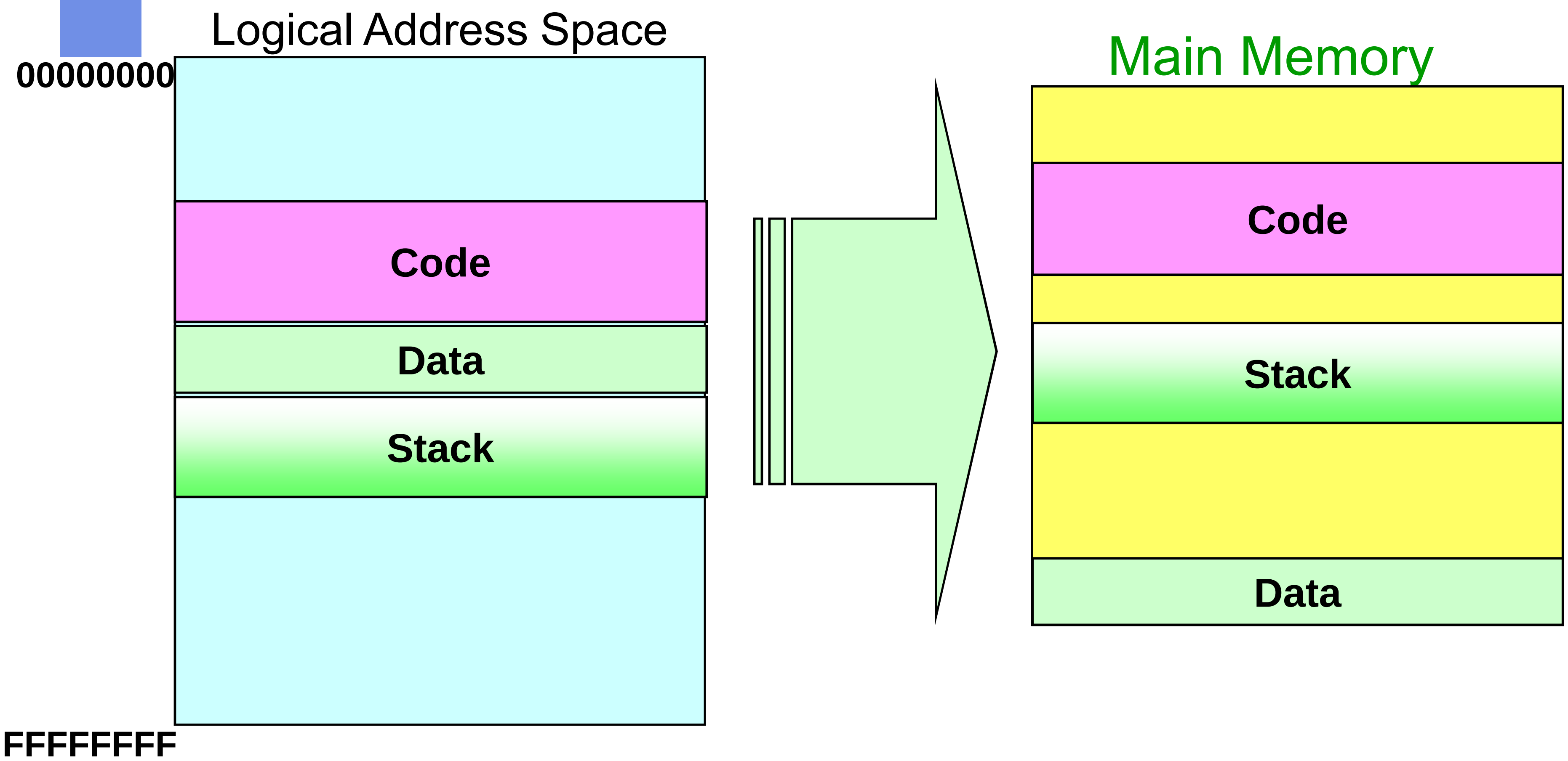
Contiguous AS → Contiguous MP



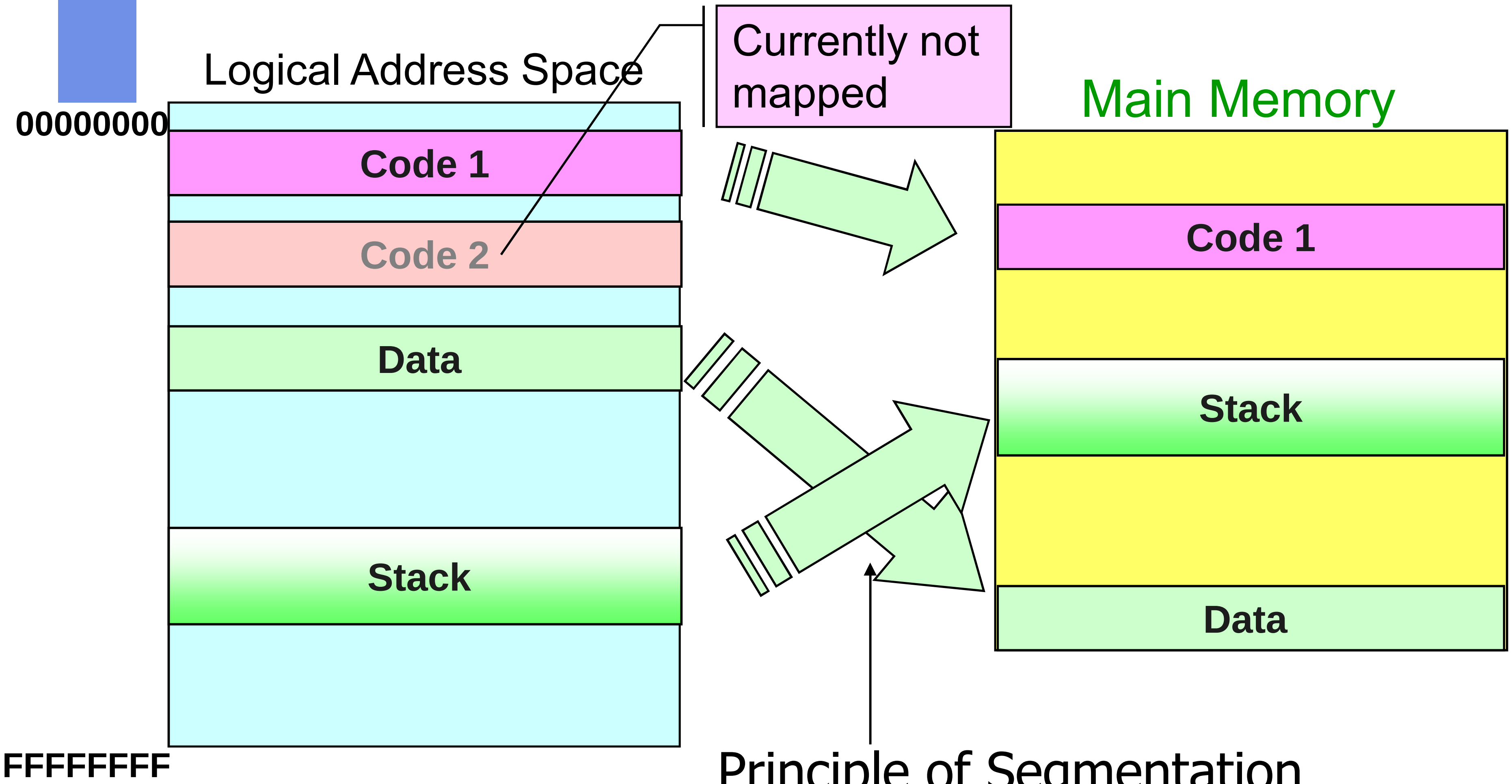
Non Contiguous AS → Contig. MP



Contig. AS → Non Contiguous MP

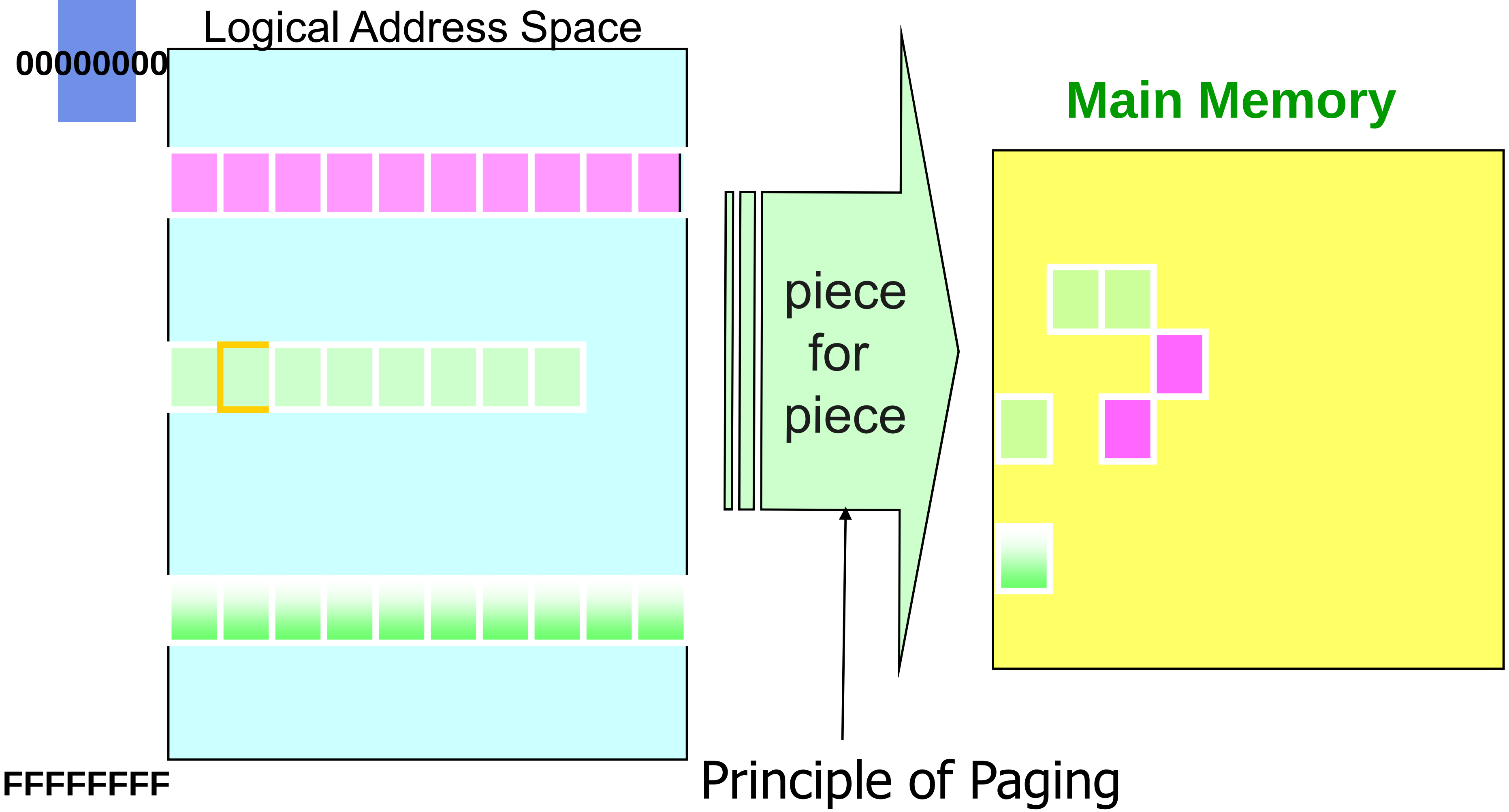


Partially Non Cont. AS → Non Cont. MP



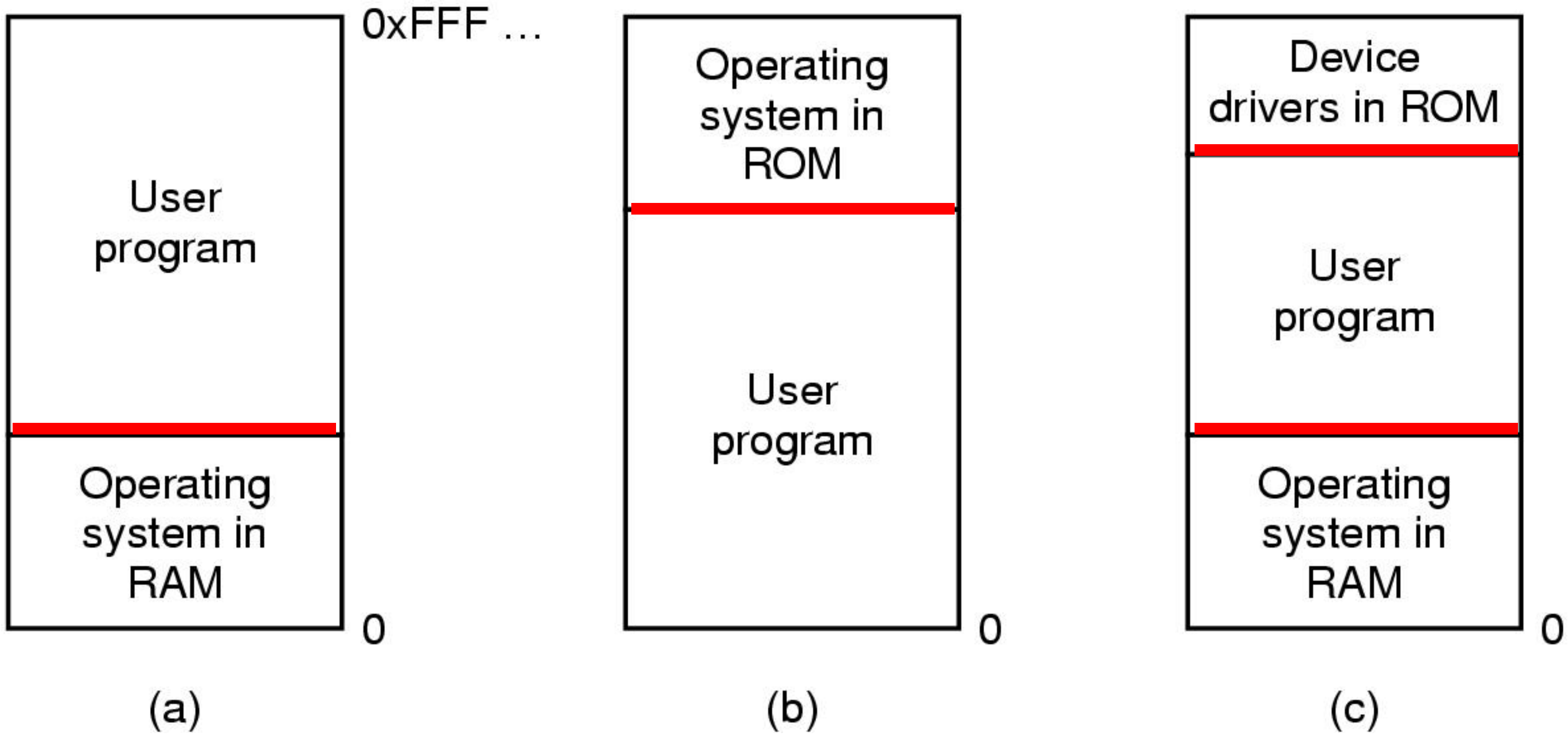
Principle of Segmentation

Fixed Parts of Non Cont. AS → Cont. MP



Single- & Multi-Programming

Elementary AS Management



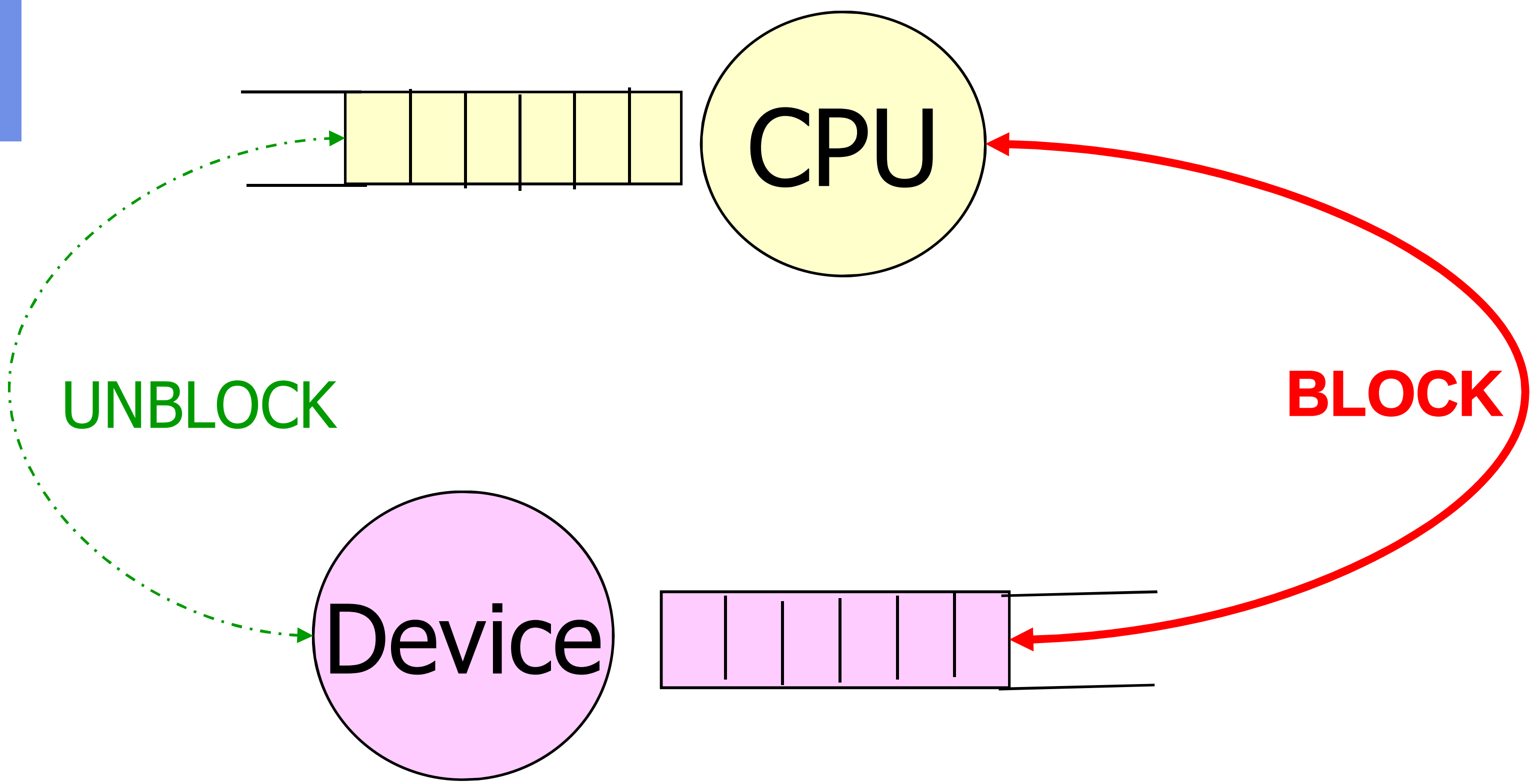
Three ways of organizing memory

- OS with 1 application, i.e. single-programming

Analysis of Single-Programming

- OK if
  - Only one task
  - Memory available ~ required memory
- Otherwise
  - Poor CPU utilization during blocking I/O
  - Poor memory utilization with varying jobs
- Better idea:
  - Subdivide memory in partitions and run more than one task or process

Fast CPU & Slow I/O-Device



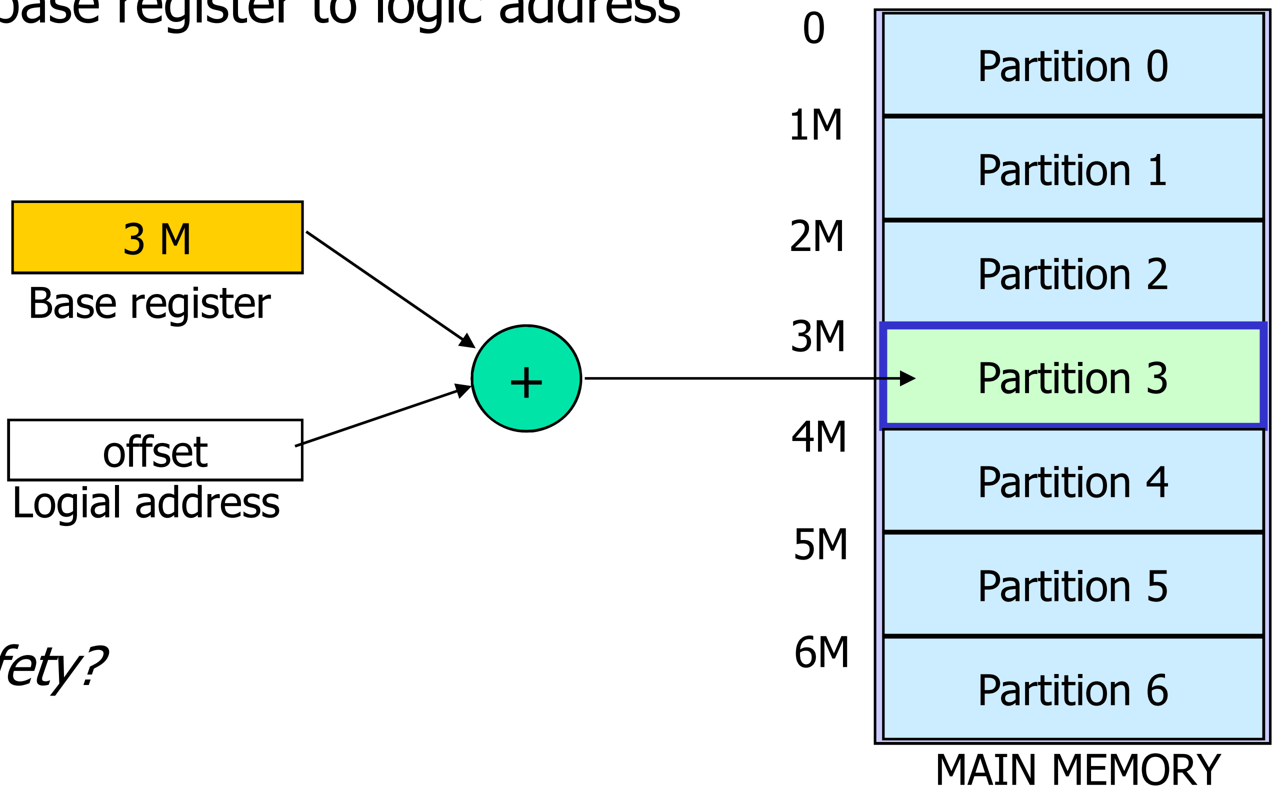
These: The faster the CPU, the more it runs idle

How to divide Main Memory?

- Fixed Partition
  - A process ≤ partition size can be loaded
  - Fast Context Switch, only need to update base register
  - Simple Find empty partition when loading a new task
  - Internal fragmentation
- Variable Partitions
  - More complex, but still fast context switch possible, only need to update base register and limit register
  - Instead of internal we have external fragmentation

Fixed Partition

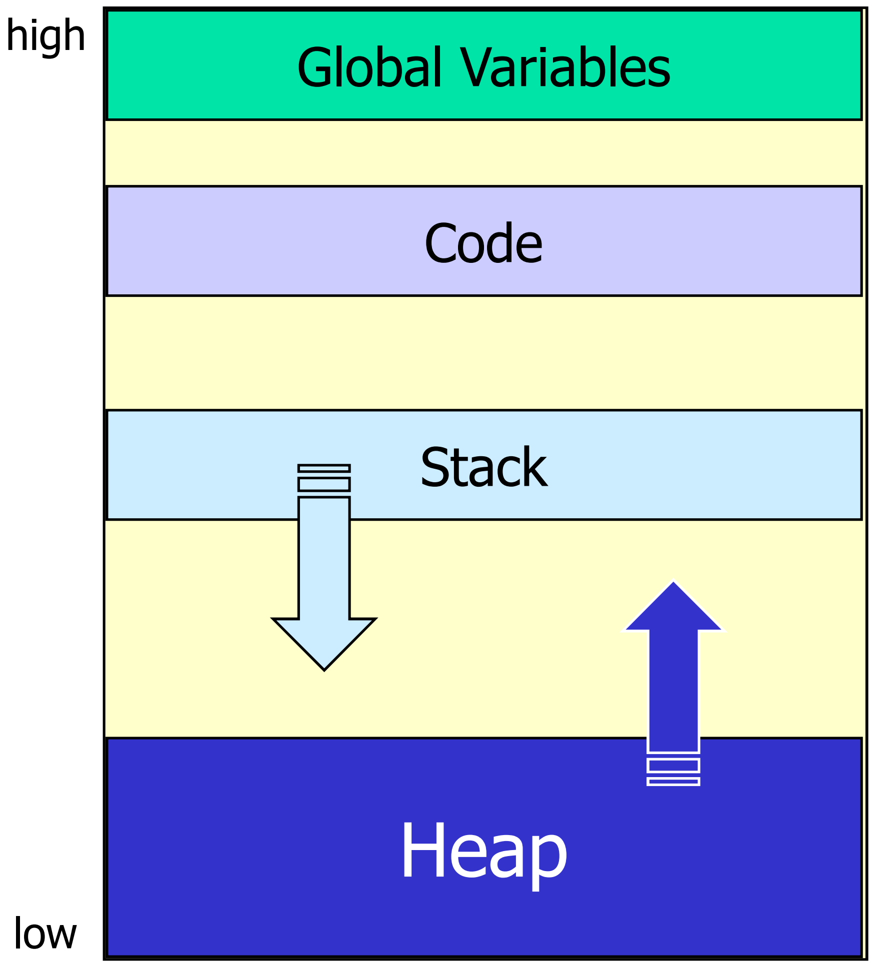
- Break main memory into fixed-size partitions
  - Hardware requirement: **base register**
  - Translation from logical address to physical address: simply add base register to logic address



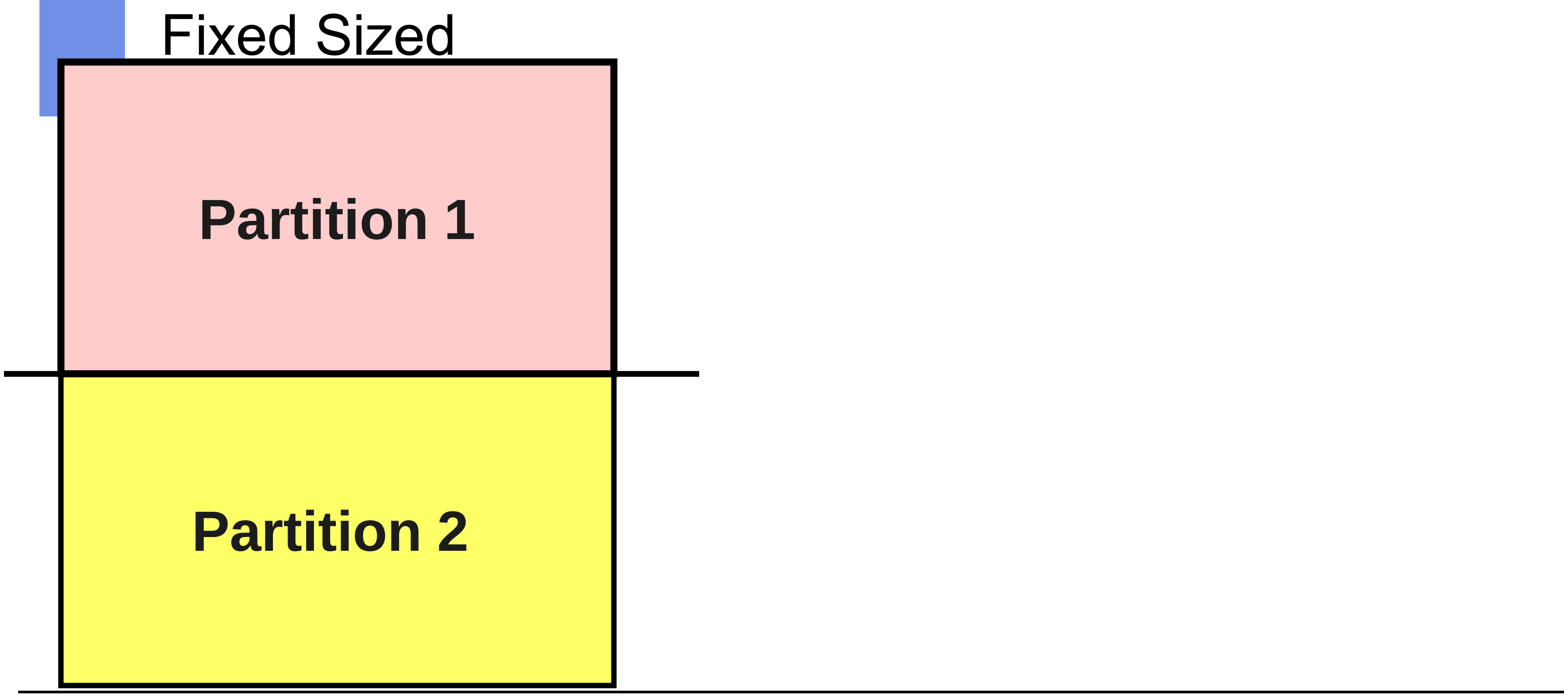
Problem: safety?

Potential Structure of a Partition

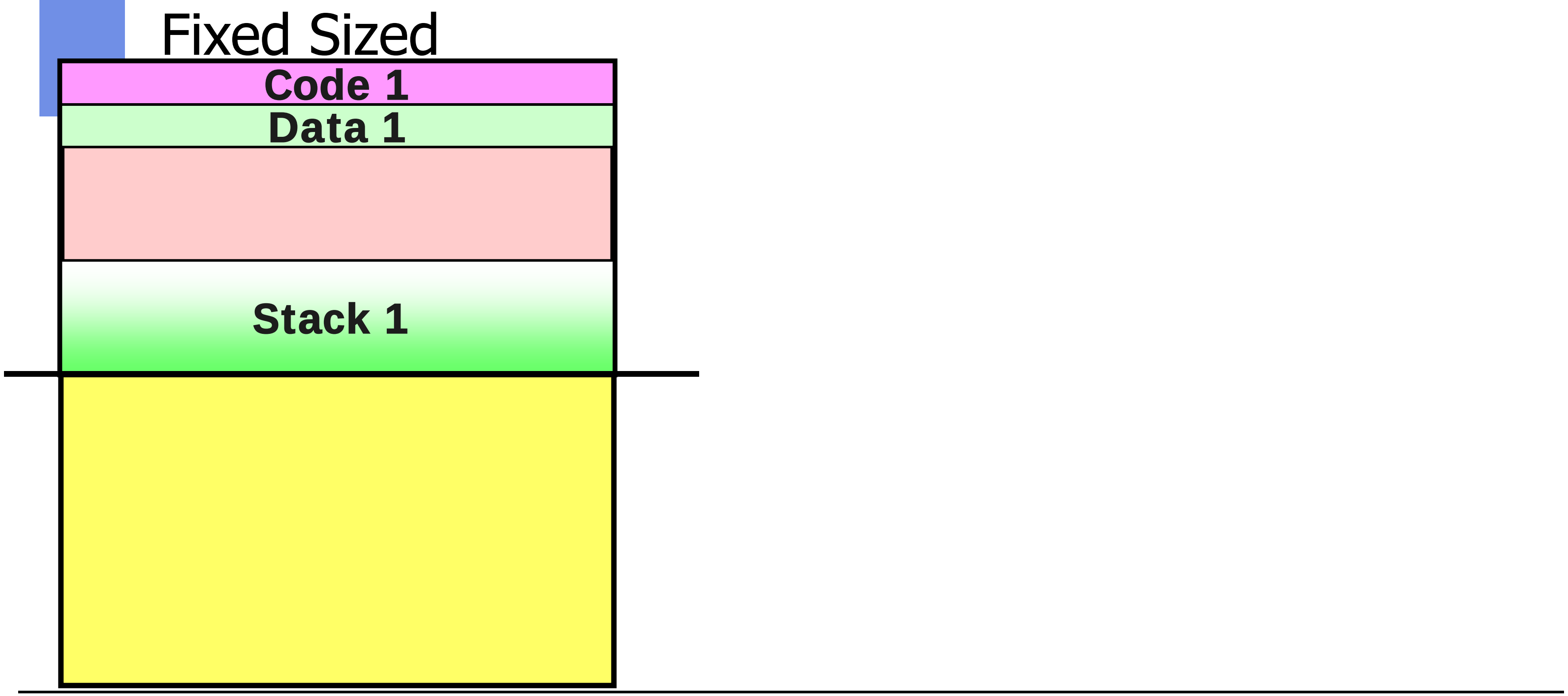
- Heap
  - Allocating at run-time
  - For dynamic objects and data structures
  - Resources (code, buffer,…)
- Stack
  - Parameter
  - Local variables
  - Return addresses, nesting
- Global variables
- Code section



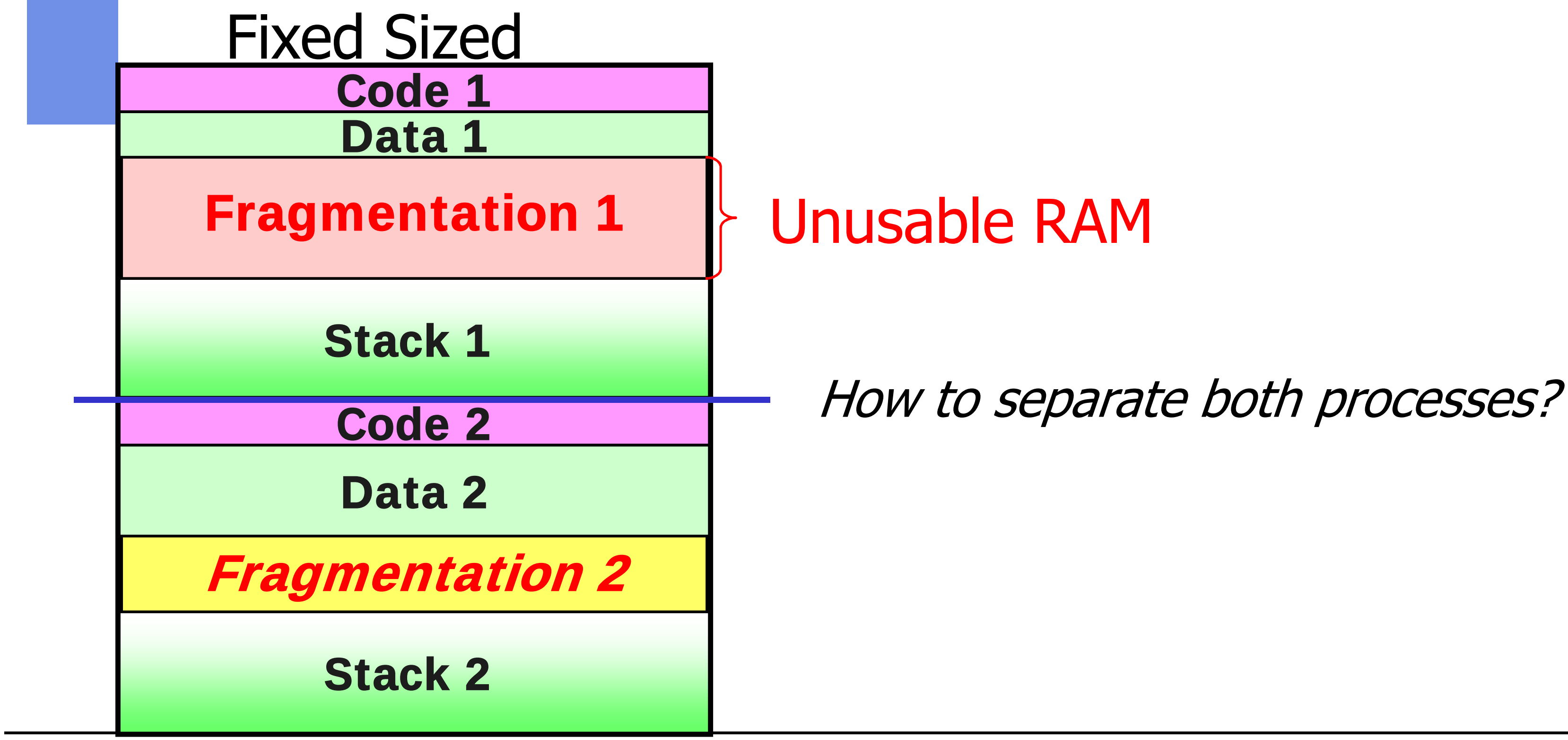
Fixed Sized Memory Partitions



Fixed Sized Partitions



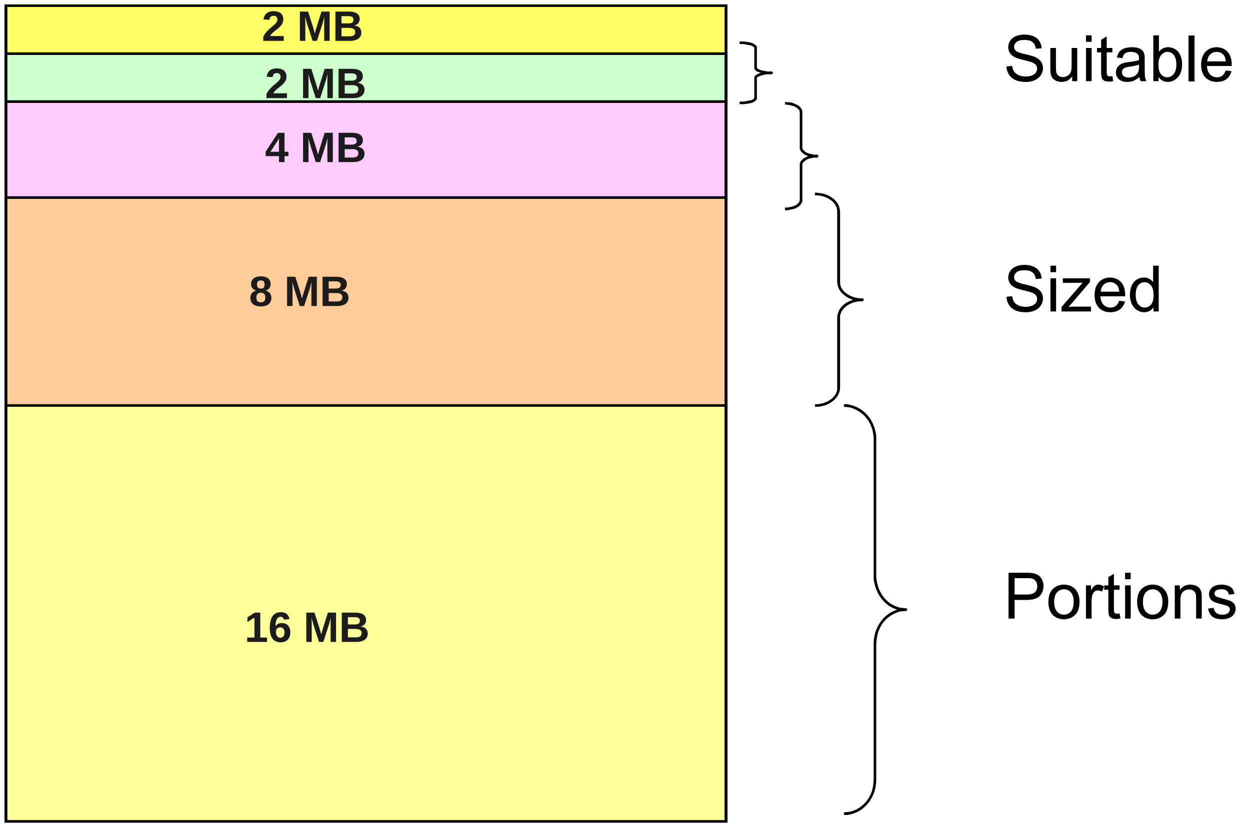
Fixed Sized Partitions



Pro: Easy to implement

Con: Internal fragmentation &number of tasks is limited

Flexible Fixed Partitions



Pro: For some dedicated systems less internal fragmentation

Con: More system overhead

Comments on Fixed Partitioning

Poor usage of memory, because each task, no matter how small, needs an entire partition

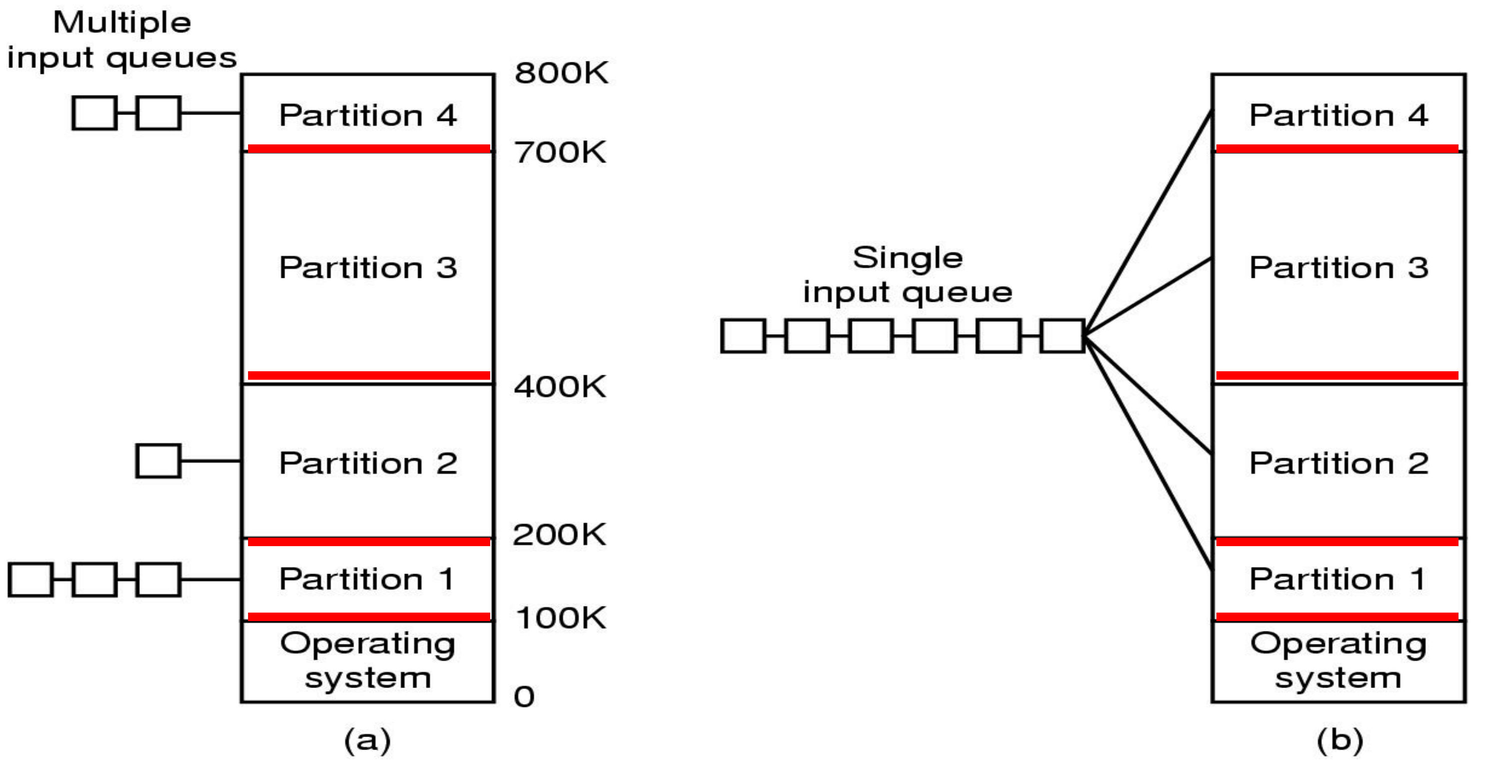
⇒ internal fragmentation

Suitable-sized partitions lessen this problem, but internal fragmentation still holds

Equal-sized partitions used in early IBM's **OS / MFT** (**M**ultiprogramming with a **F**ixed number of **T**asks ⇒ the maximal multi programming degree is fixed)

Basic Design Flaw?

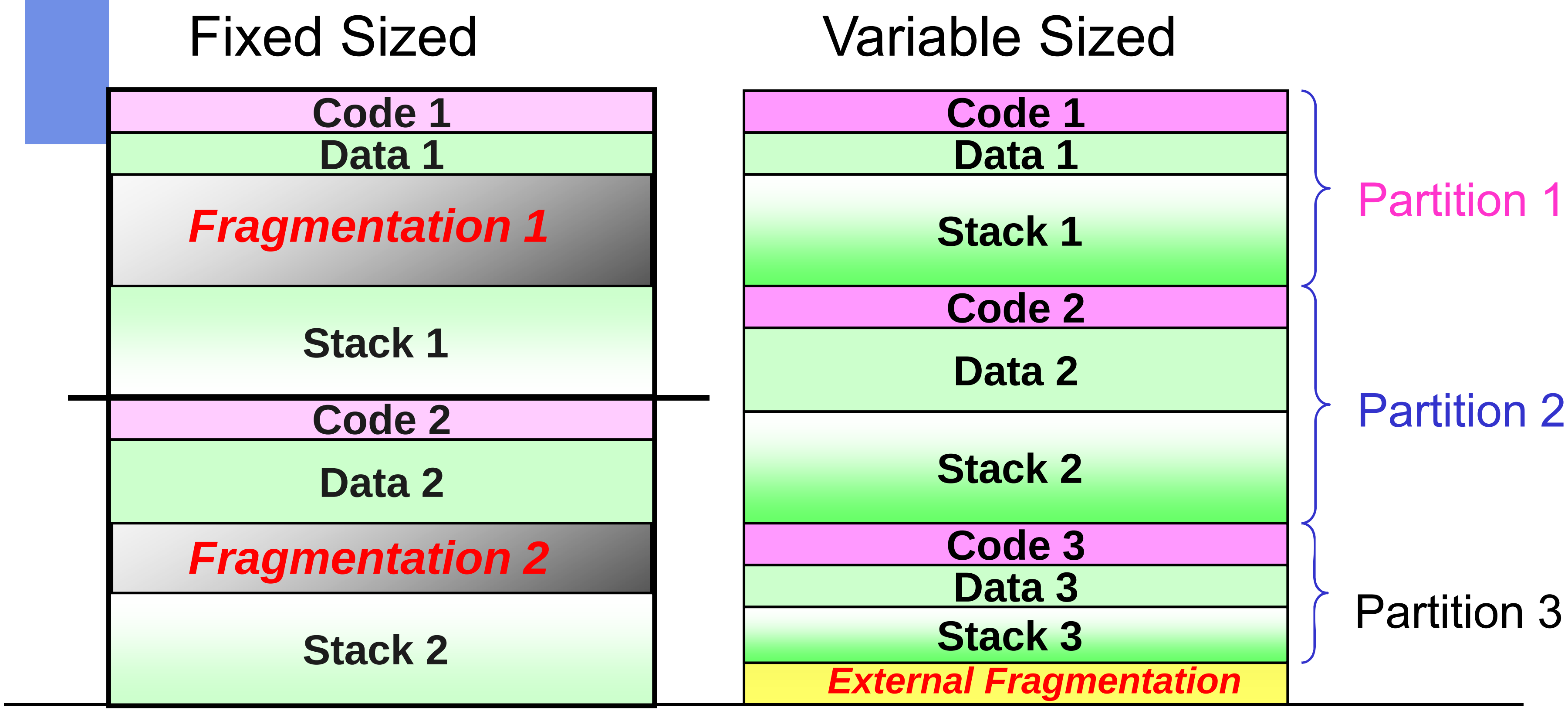
Implementing Fixed Partitions



Fixed memory partitions

- separate input queues for each partition
- single input queue for all partitions

Fixed & Variable Sized Partitions



Pro: No internal fragmentation, better multiprogramming

Con: External fragmentation, more complicated

Variable Partitions

Partitions are of variable length and number:

Each task gets exactly as much memory as it requires

After a task terminates, "memory holes" may appear

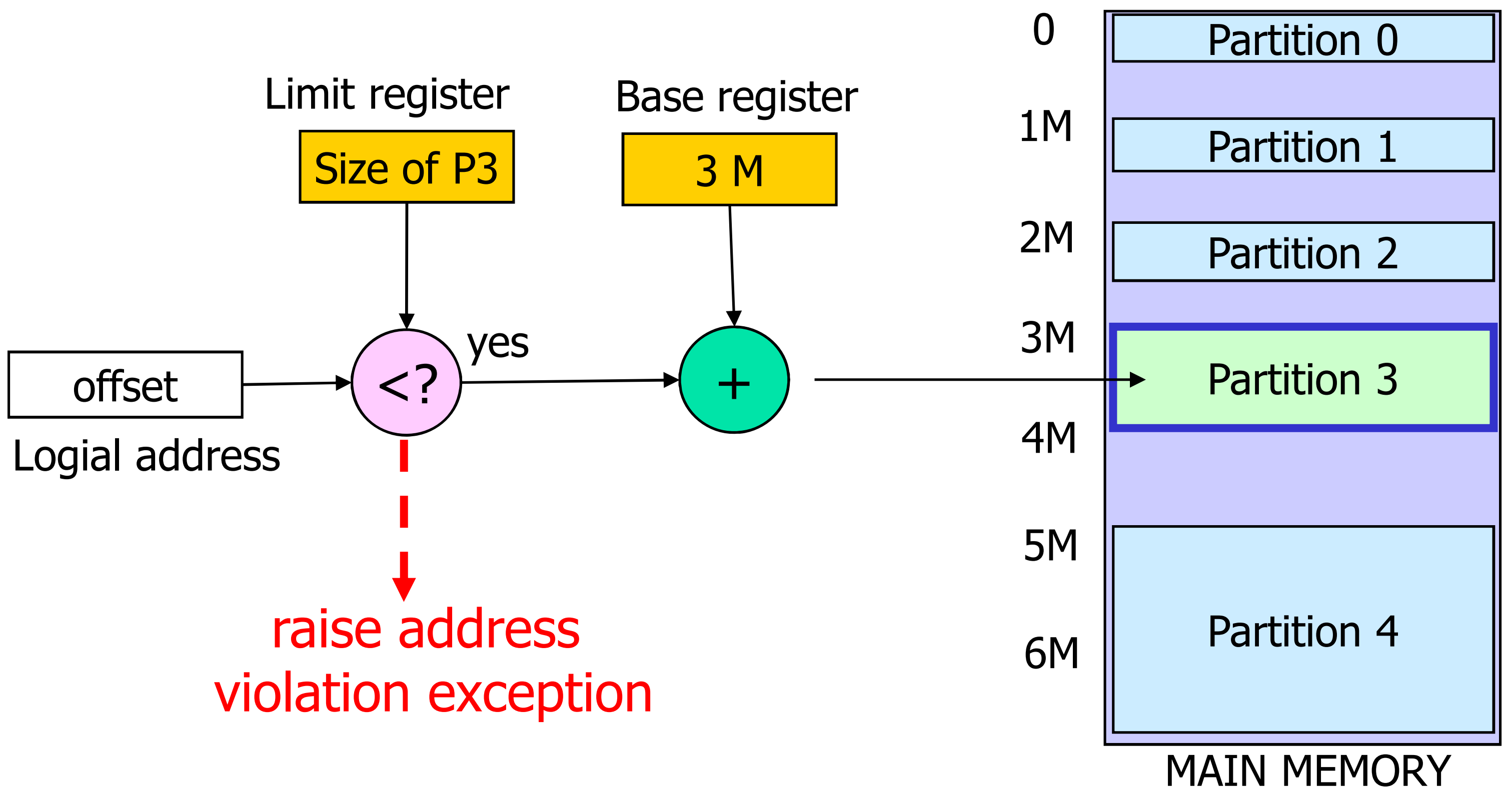
⇒ external fragmentation

Must use compaction to shift tasks,
to get a larger block of free memory

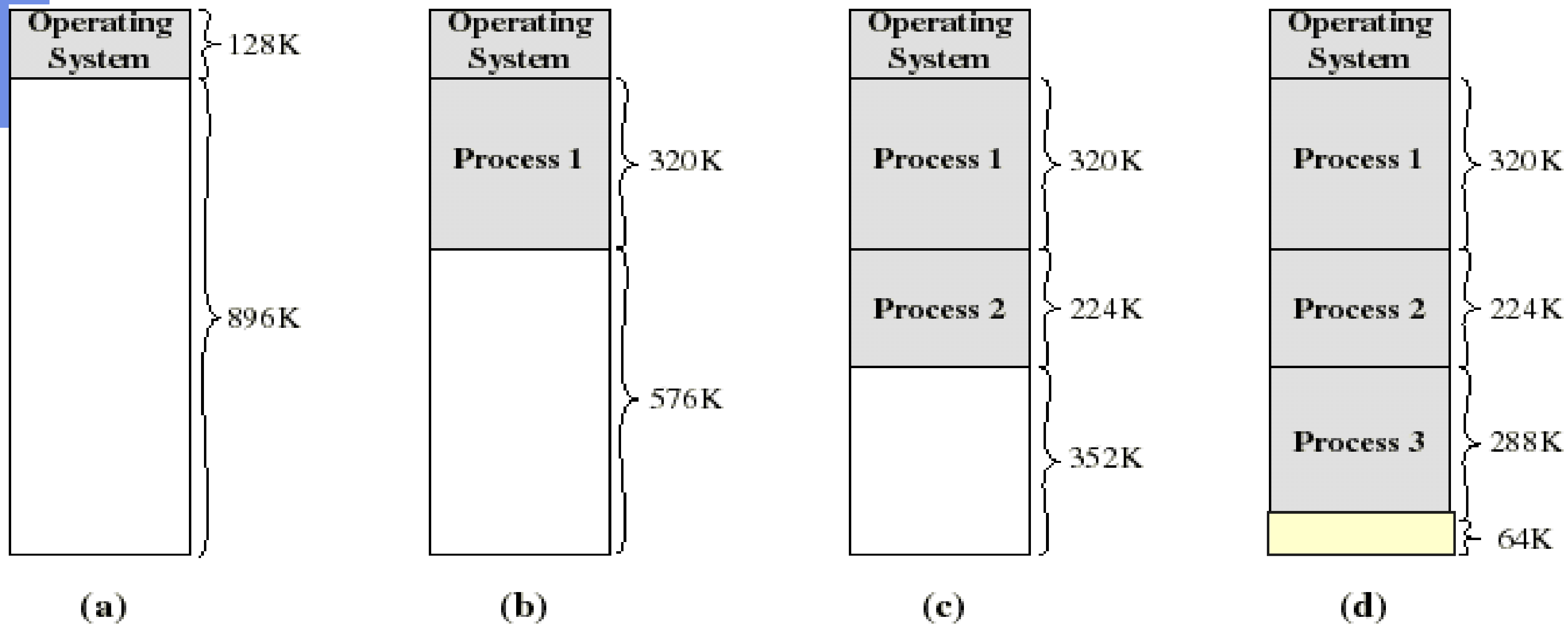
Used in IBM's OS/**MVT** (**M**ultiprogramming with a **V**ariable number of **T**asks)

Requirements of Variable Partitions

- Break memory in variable-sized partitions
  - Hardware requirements: base register and limit register



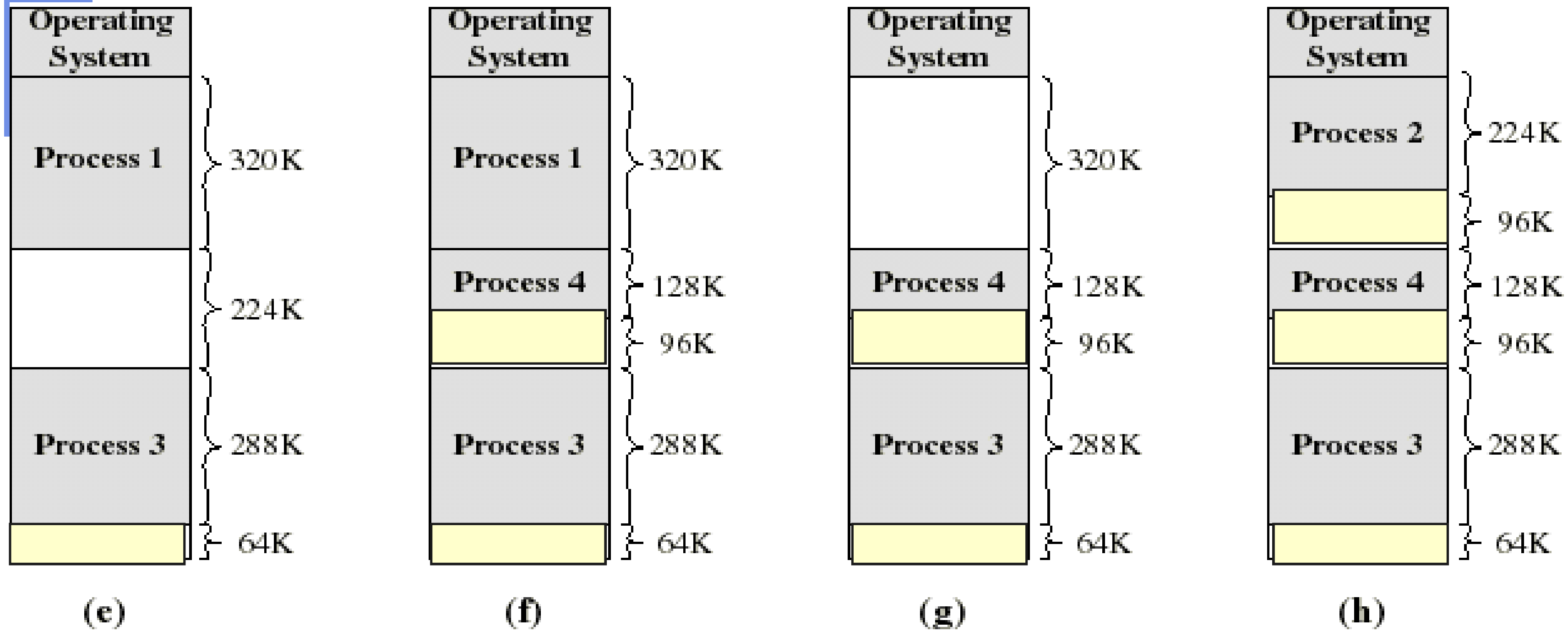
Variable Partitions: Example (1a)



A hole of 64K is left after loading 3 tasks: not enough room for another task

If each task is blocked, OS swaps out task2 in order to swap in task4

Variable Partitions: Example (1b)



Another hole of 96K is created, if task4 is also blocked ⇒

OS swaps out task1, swaps in task 2 ⇒ another hole of 96K ⇒

Danger of splitting up memory (compare to Swiss cheese pattern)

Analysis of Variable Partitions

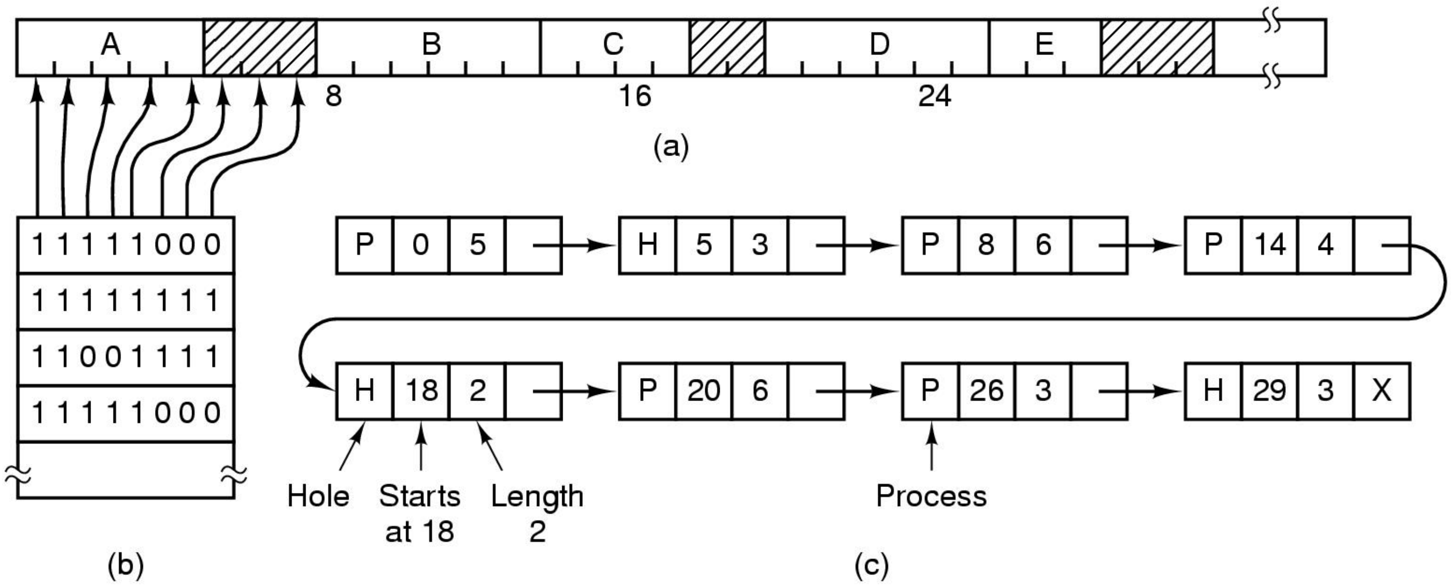
- In previous slide
  - We have 256 KB free in total, but if a new task requires 100 KB, we cannot satisfy its request
  - External fragmentation
- We end up with lots of unusable memory holes
- We could use compaction
  - Shuffle allocated memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at run time

Managing Variable Partitions

- Basic Requirements
  - Find a fitting free partition as fast as possible
  - Minimize external fragmentation
  - Support eager reunification of neighbored free partitions

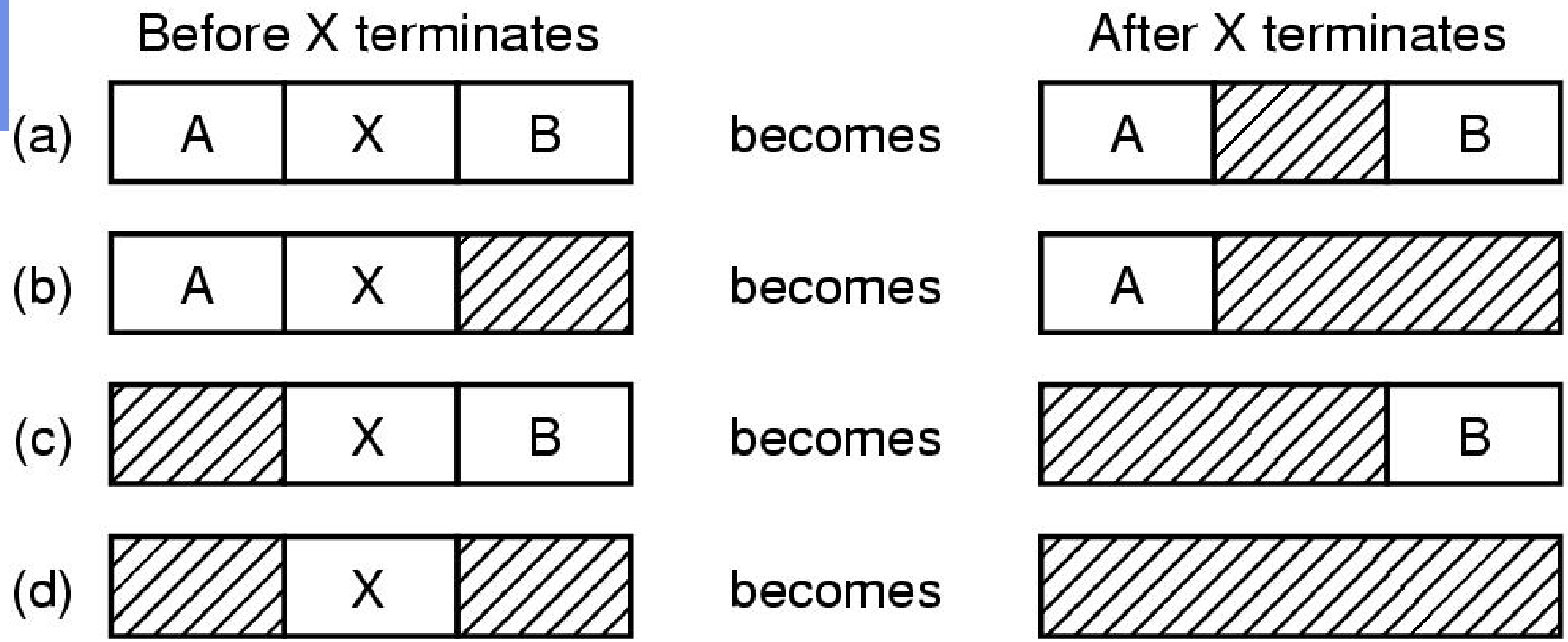
Question: What memory manager would you use?

Bit Map/List for Tracing Partitions



- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- Corresponding bit map
- Same information as a list

Linked Lists for Tracing Partitions



Four combinations for the terminating process X if eager reunification is used

Overview on Allocation Policies

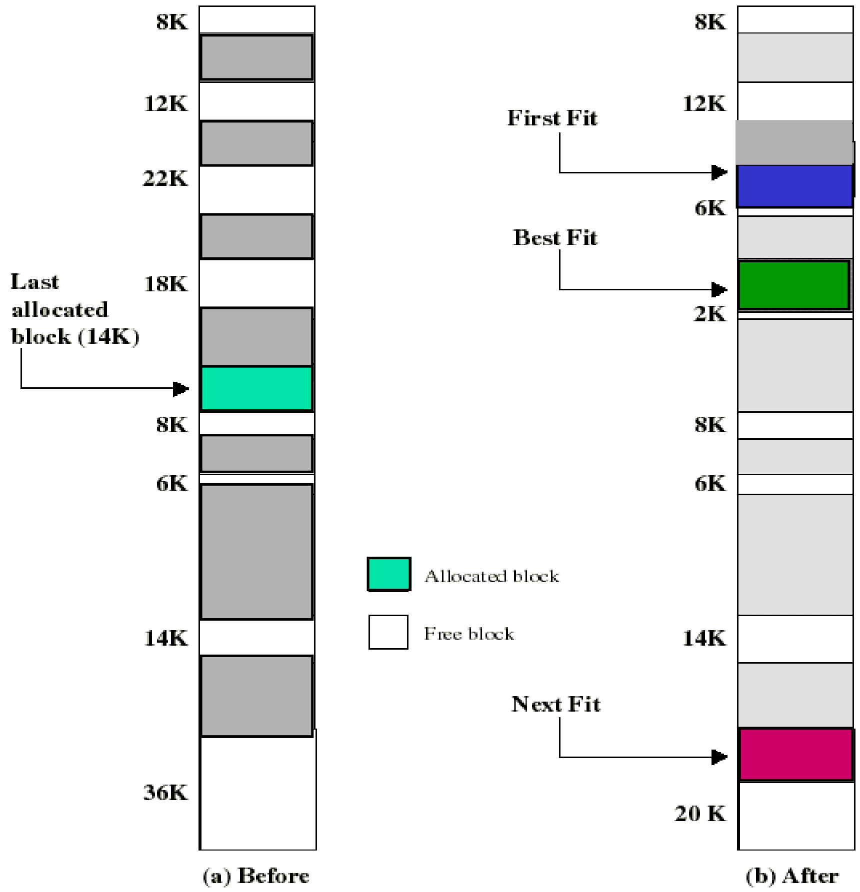
Used to decide which free block to allocate to a requesting task

Goal:
Reduce usage of compaction (being quite time consuming)

Possible algorithms:

- First-fit: choose always very first hole from beginning
- Best-fit: choose smallest hole
- Next-fit: choose first hole from last placement
- Nearest-fit: choose nearest hole from last placement

Using information of the last allocated block



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

Mapping Variable Partitions

- First-fit
  - Scan the list or bit map for the first entry that fits
    - If larger in size, break it into an allocated and a free part, iff free part is large enough to be used
  - Many processes loaded into the front end of memory that must be searched over and over when trying to find a free block (~ inefficient)
  - Can have some unusable holes at the beginning
    - External fragmentation

Mapping Variable Partitions (2)

- Next fit
  - Like first-fit, except it begins its search from that point in the list or bit map where the previous request had succeeded
    - More often allocates a block of memory at the end of memory where the largest block is found
    - Largest block is broken up into smaller blocks
    - Compaction is required to obtain a large block at the end of memory
    - *Simulation show next-fit slightly slower than first-fit*

Mapping Variable Partitions (3)

- Best-fit
  - Choose that block that is closest in size to the request
  - Poor performance
    - Often has to search the complete list or bit map
    - Since smallest fitting block is chosen for a request, the smallest amount of fragmentation is left in the memory $\Rightarrow$ compaction must be done more often

Mapping Variable Partitions (4)

- Worst-fit
  - Choose the block that is largest in size
    - Idea is to leave a usable new free fragment over
  - Poor performance
    - Often has to search complete list or bit map
    - Simulations show only limited effects

Linking & Loading

Study for yourselves

Use slides from previous Proseminars