Memory Management

DRS. KROPP, YOUSSFI, AND STALLINGS

Objectives

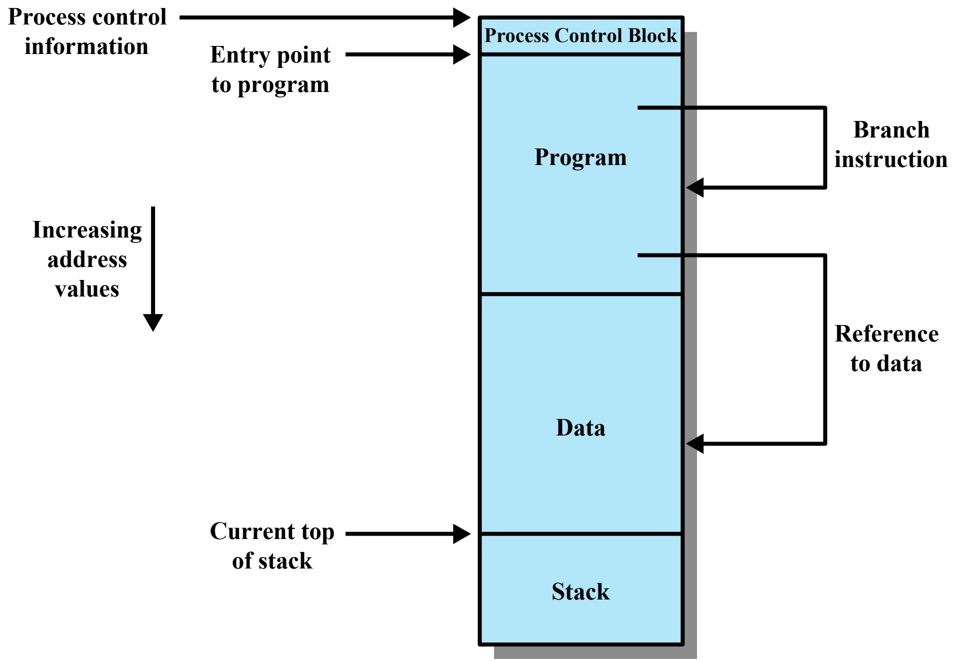
- Main requirements
- Memory partitioning
- Paging
- Segmentation
- Relative advantages/disadvantages of paging and segmentation
- Loading and linking

Requirements for memory management systems

- Relocation
- Protection
- Sharing
- Logical organization
- Physical organization

Relocation Requirement

- Cannot expect programmer to know location of program in memory
- Need to swap out blocked processes and swap in ready processes to use memory efficiently
  - Therefore, cannot expect process to be swapped in at the same memory location
- OS and hardware must be able to translate memory references to physical memory addresses



Memory locations that an operating system must keep track of

Protection Requirement

- Any process should not access memory of other processes without permission
- Location of process is not known at compile time
- Dynamic allocation and generation of addresses at run time
- Memory references generated by a process must be checked at run time
- OS and processor hardware must check for protection
  - OS alone cannot efficiently check for protection
- Relocation requirement increases difficulty of the protection requirement

Sharing Requirement

- Memory management must allow controlled access to shared areas of memory
- Shouldn’t compromise protection
- Cooperating processes can access shared data structures
- Mechanisms for relocation should support sharing capabilities

Logically Organization Requirement

- Memory is organized linearly
- If an operating system can logically organize programs as modules, then programs can:
  - Can be written and compiled independently
  - Might have different protection
  - Sharing on module level

Physical organization requirement

- Main memory versus secondary memory
- Organizing the flow between memory levels
- Main memory maybe too small → programmer can employ overlaying
  - But this is too complex for programmer

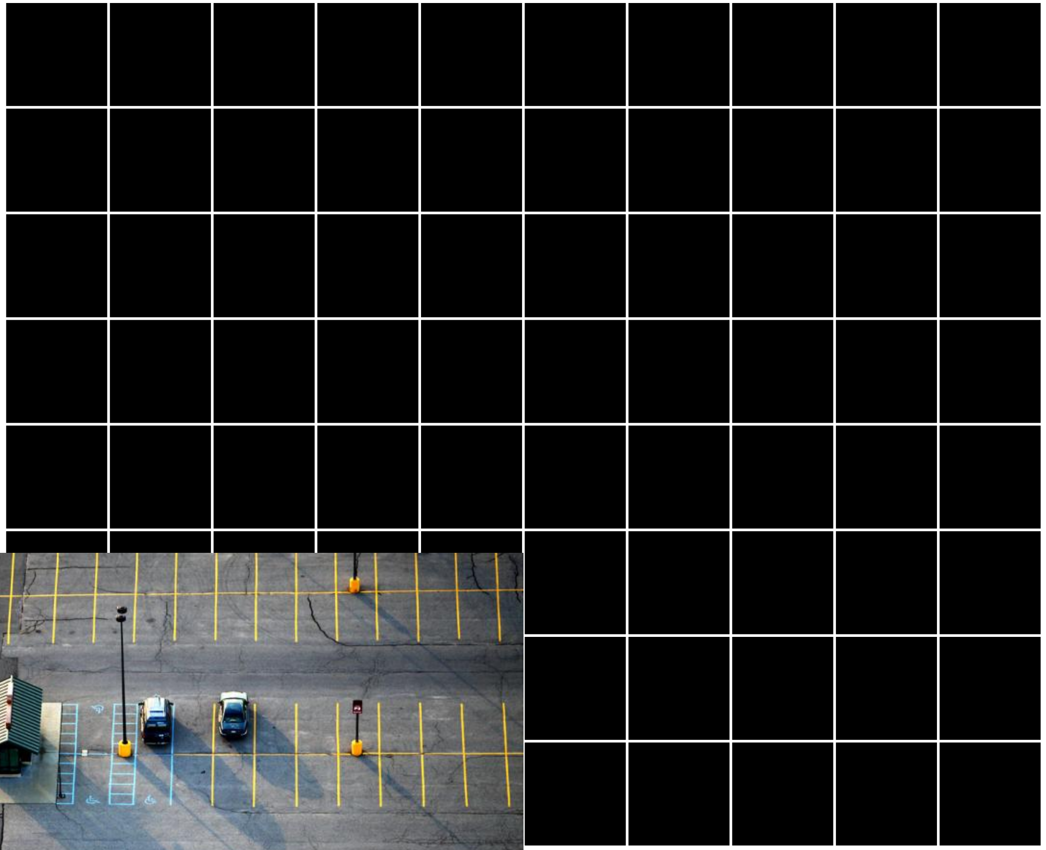
Memory partitioning

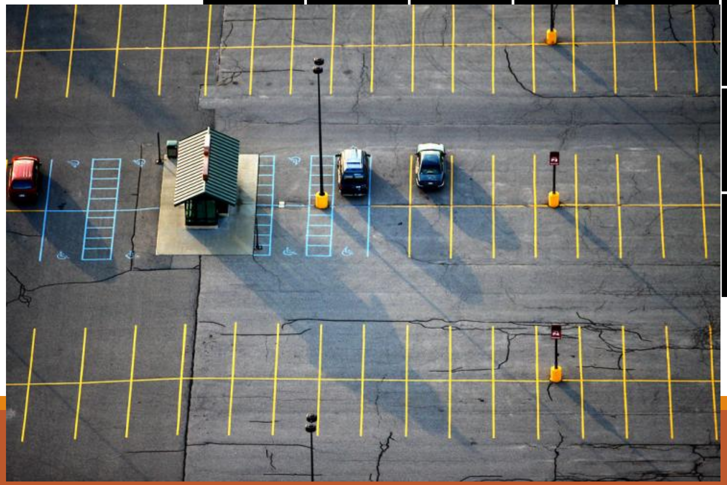
HOW DO YOU DIVIDE MAIN MEMORY IN A WAY THAT OPTIMIZES THE ALLOCATION OF MEMORY TO DIFFERENT PROCESSES

Memory partitioning

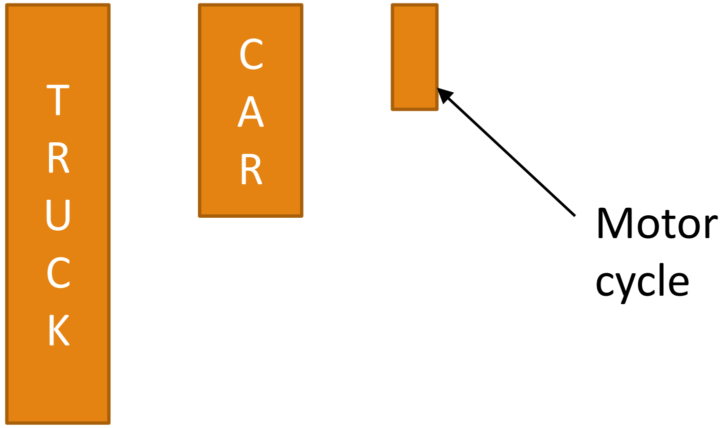
- Most modern operating systems use a system called virtual memory
  - Paging
  - Segmentation
- We’re going to look at an older systems to understand core concepts behind virtual memory
  - Memory partitioning
  - Simple paging
  - Simple segmentation

Painting the parking lot: round 1



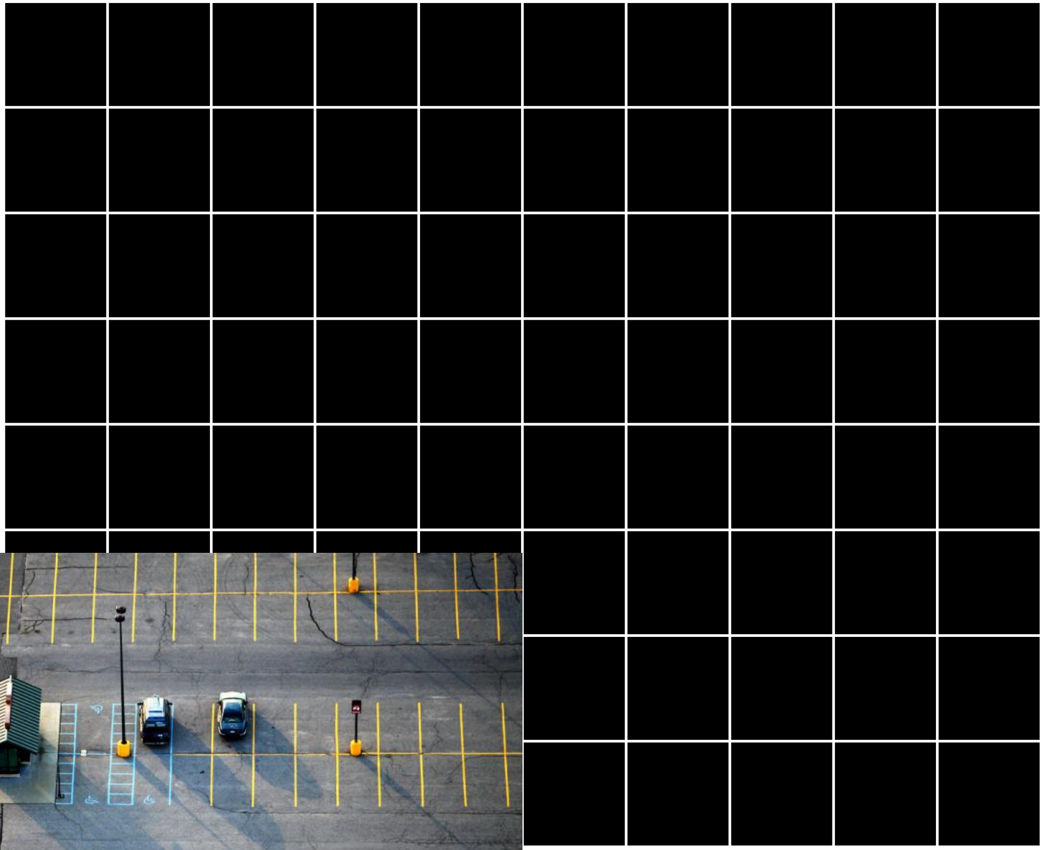


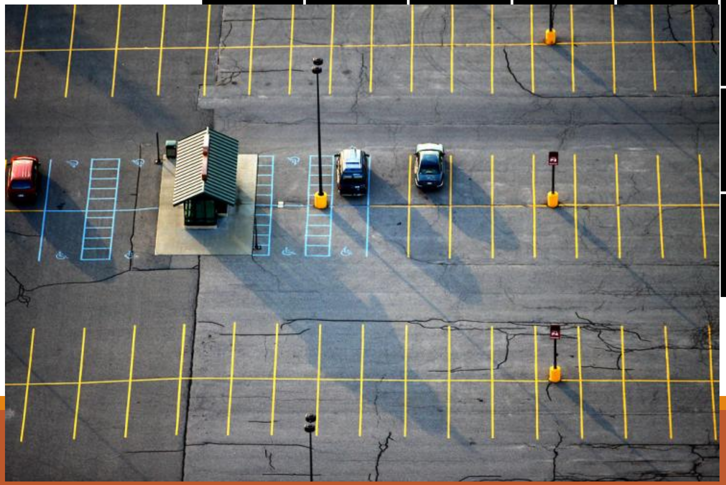
- This grid represents the area of a parking unpainted parking lot
- Paint the lines that divide the vehicles up in the parking lot
- CONSTRAINTS:
  - Expect trucks, cars, and motorcycles to park here
  - There must be at least a $1 \times n$ path for the vehicles to navigate
  - All parking spaces must be equal sized!



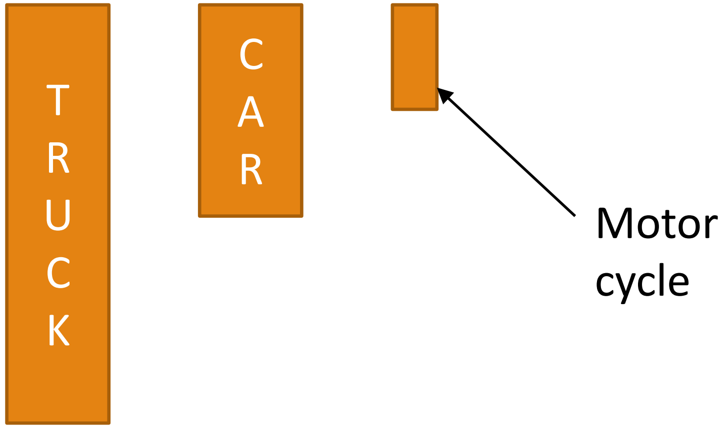
https://c1.staticflickr.com/3/2454/4088871694_5a1b654ed7_b.jpg

Painting the parking lot: round 2



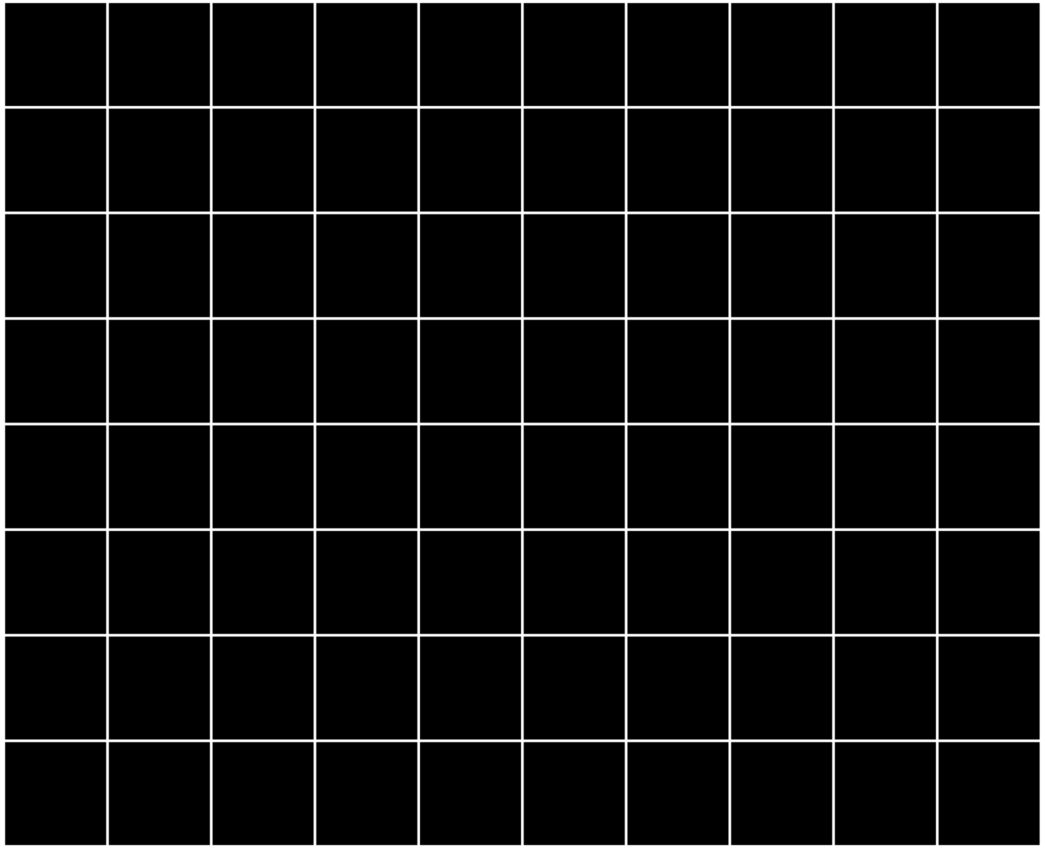


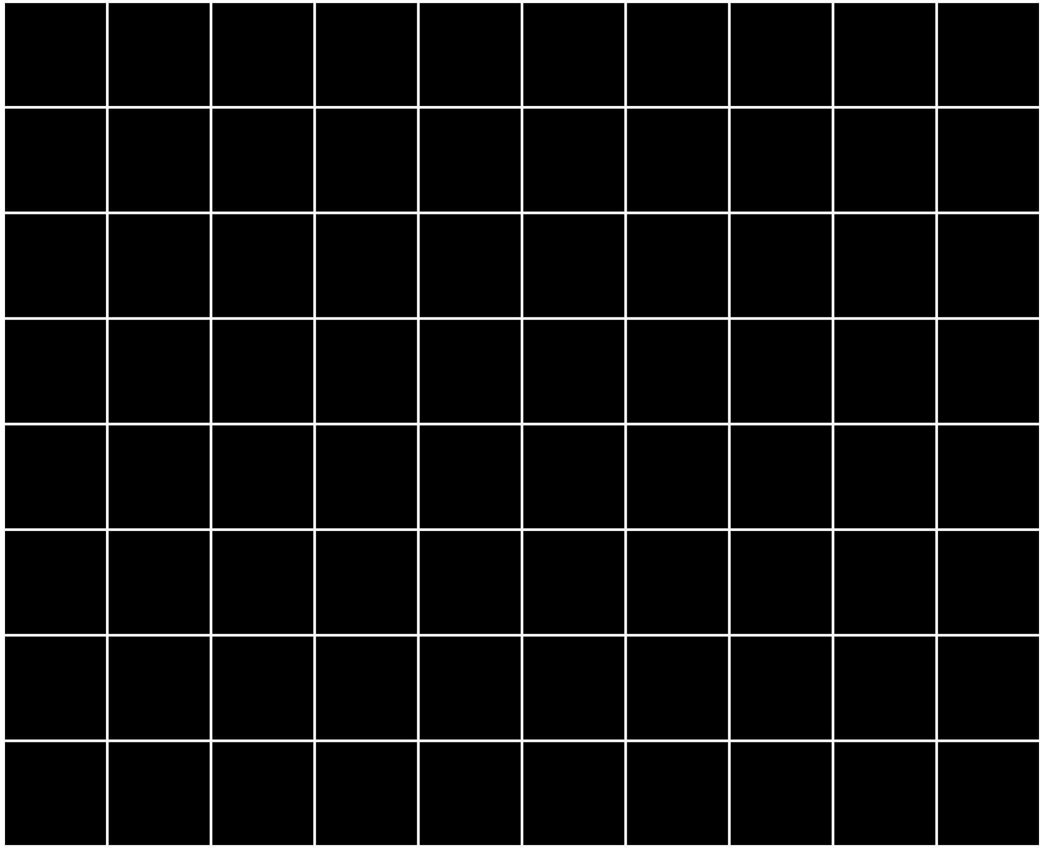
- This grid represents the area of a parking unpainted parking lot
- Paint the lines that divide the vehicles up in the parking lot
- CONSTRAINTS:
  - Expect trucks, cars, and motor cycles to park here
  - There must be at least a $1 \times n$ path for the vehicles to navigate
  - Now, you can paint your parking spots different sizes!



https://c1.staticflickr.com/3/2454/4088871694_5a1b654ed7_b.jpg

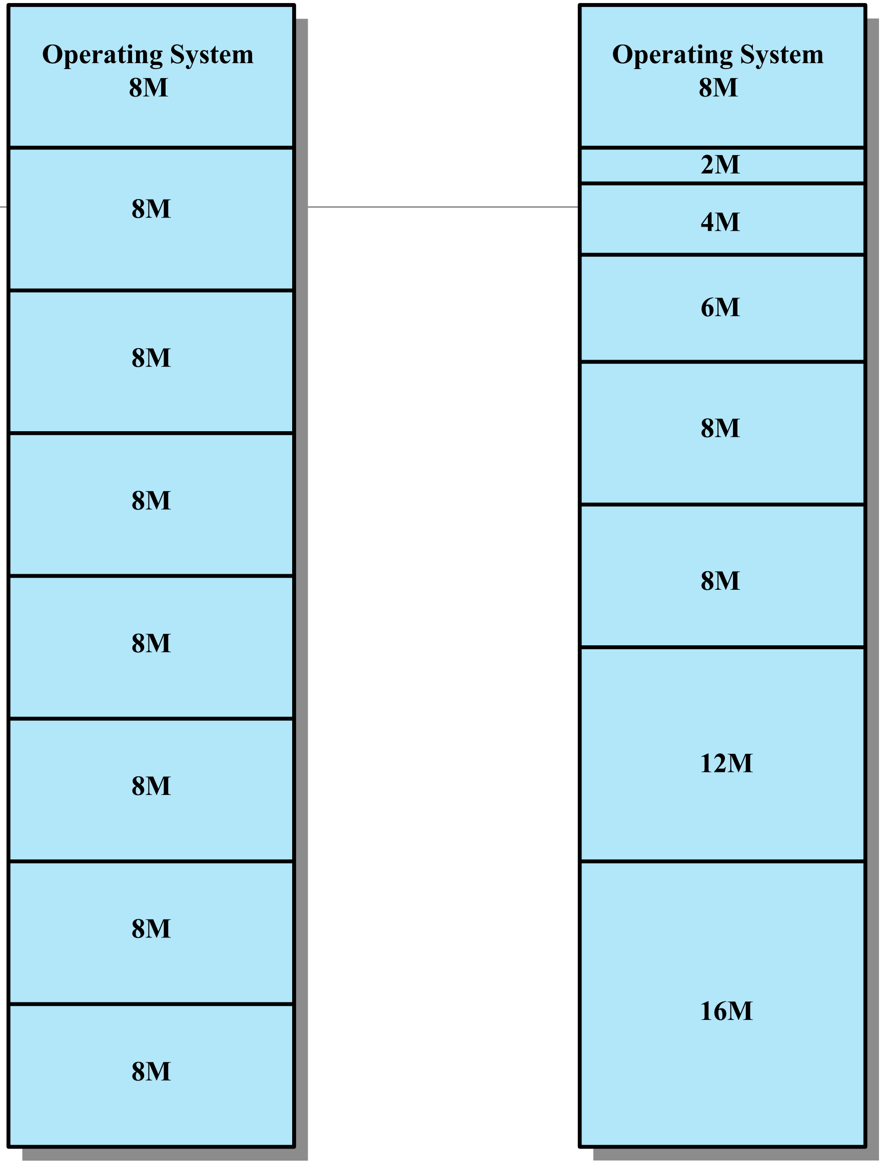
Painting the parking lot: round 2





Memory Partitioning

- Common to have a:
  1. Region of memory with the operating system
  2. The remainder of the memory is broken into partitions with fixed boundaries
- Types of partitions
  - Equal size
  - Unequal size



(a) Equal-size partitions

(b) Unequal-size partitions

Difficulties with equal-sized Partitioning

- Partition size might be too small
  - If use bigger partition → internal fragmentation
  - Fix number of partitions → fixed number of active processes
- Both issues can be lessened by unequal-sized fixed partitions

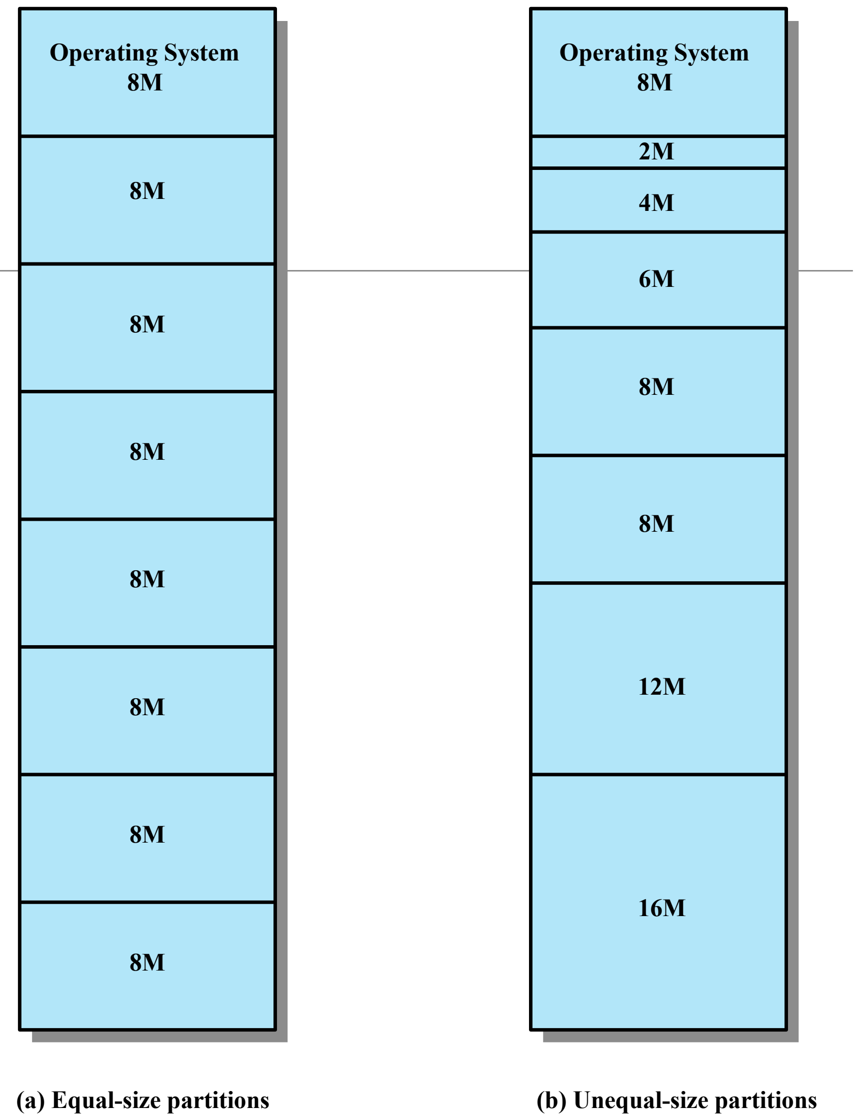
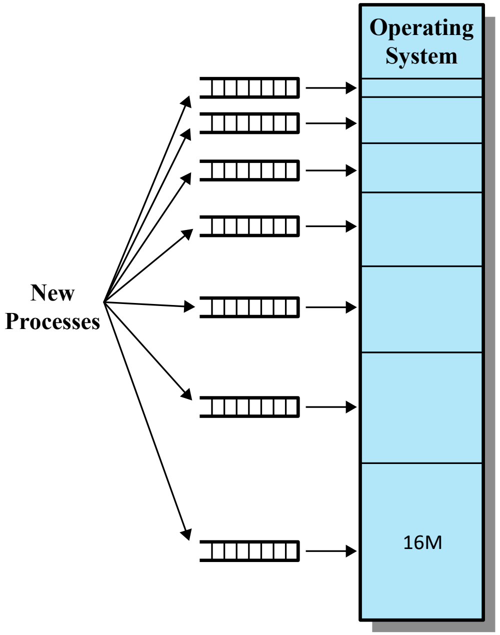


Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

Placement Algorithms Unequal Size

- How do we place processes into these slots?
- Algorithm a:
  - Algorithm puts the process in the smallest possible slot it can fit in.
  - One queue per slot
  - Issue: what if all processes are under 16M of memory?



(a) One process queue per partition

Placement Algorithms Unequal Size

- How do we place processes into these slots?
- Algorithm a:
  - Algorithm puts the process in the smallest possible slot it can fit in.
  - One queue per slot
  - Issue: what if all processes are under 16M of memory?
- Algorithm b:
  - One queue for all slots
  - Smallest available slot is chosen
  - But then we need an algorithm to figure out what processes to swap out when more space is needed

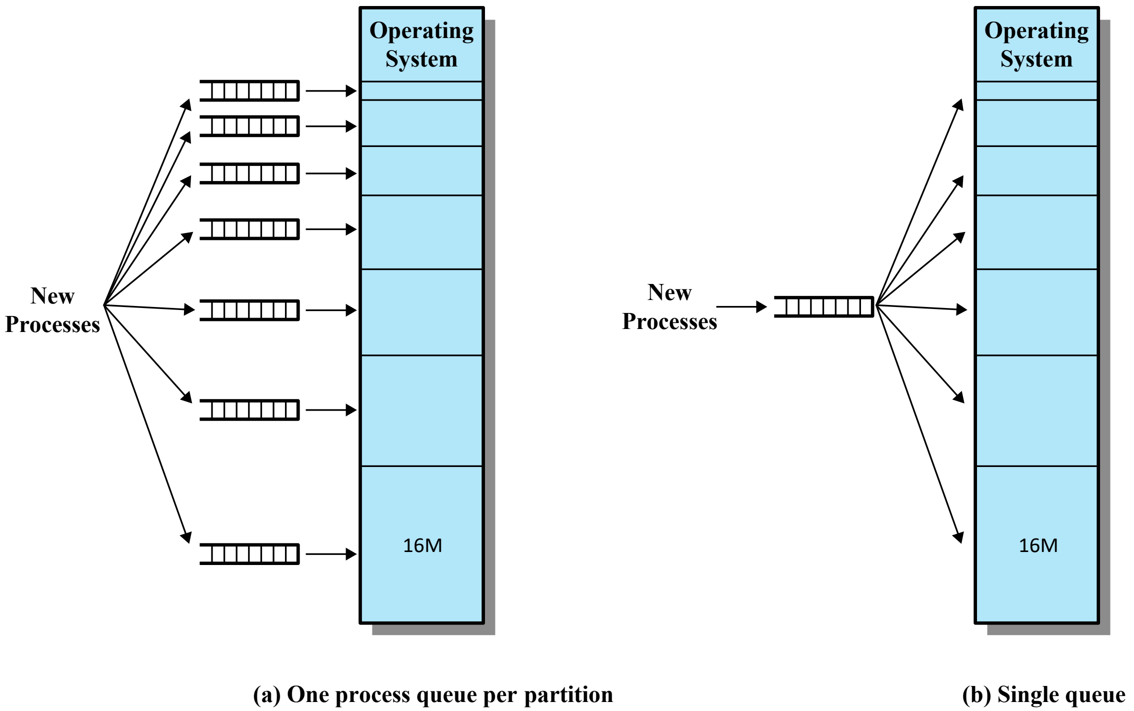
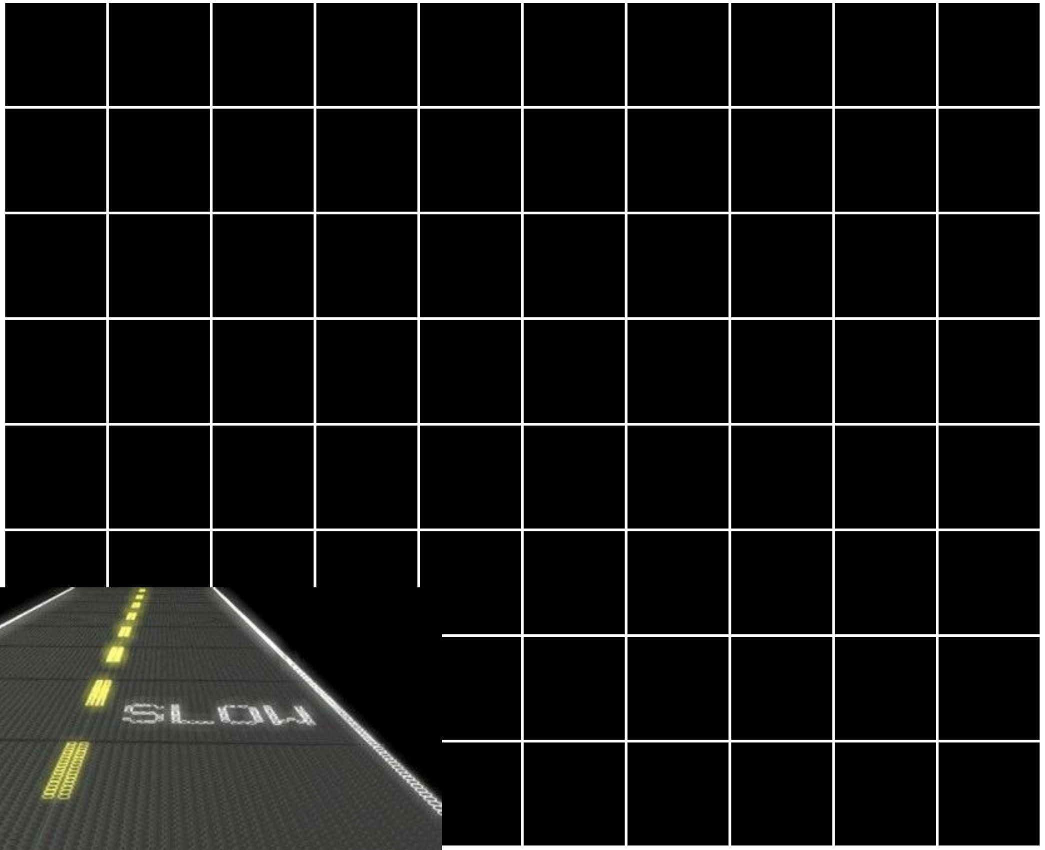
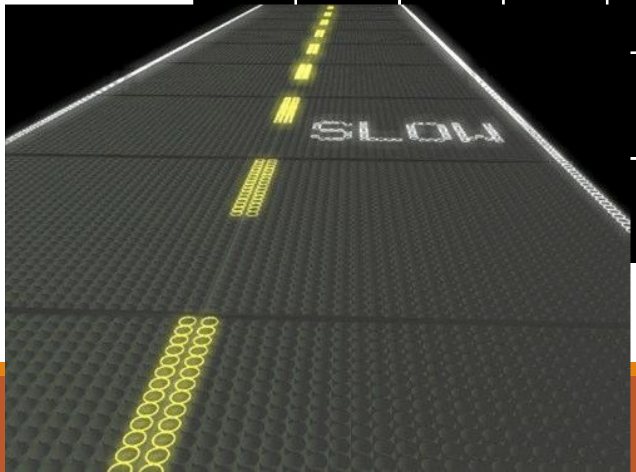


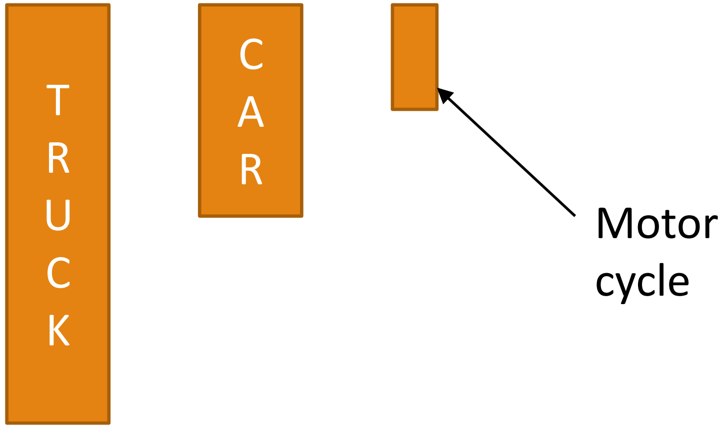
Figure 7.3 Memory Assignment for Fixed Partitioning

Painting the parking lot: round 3





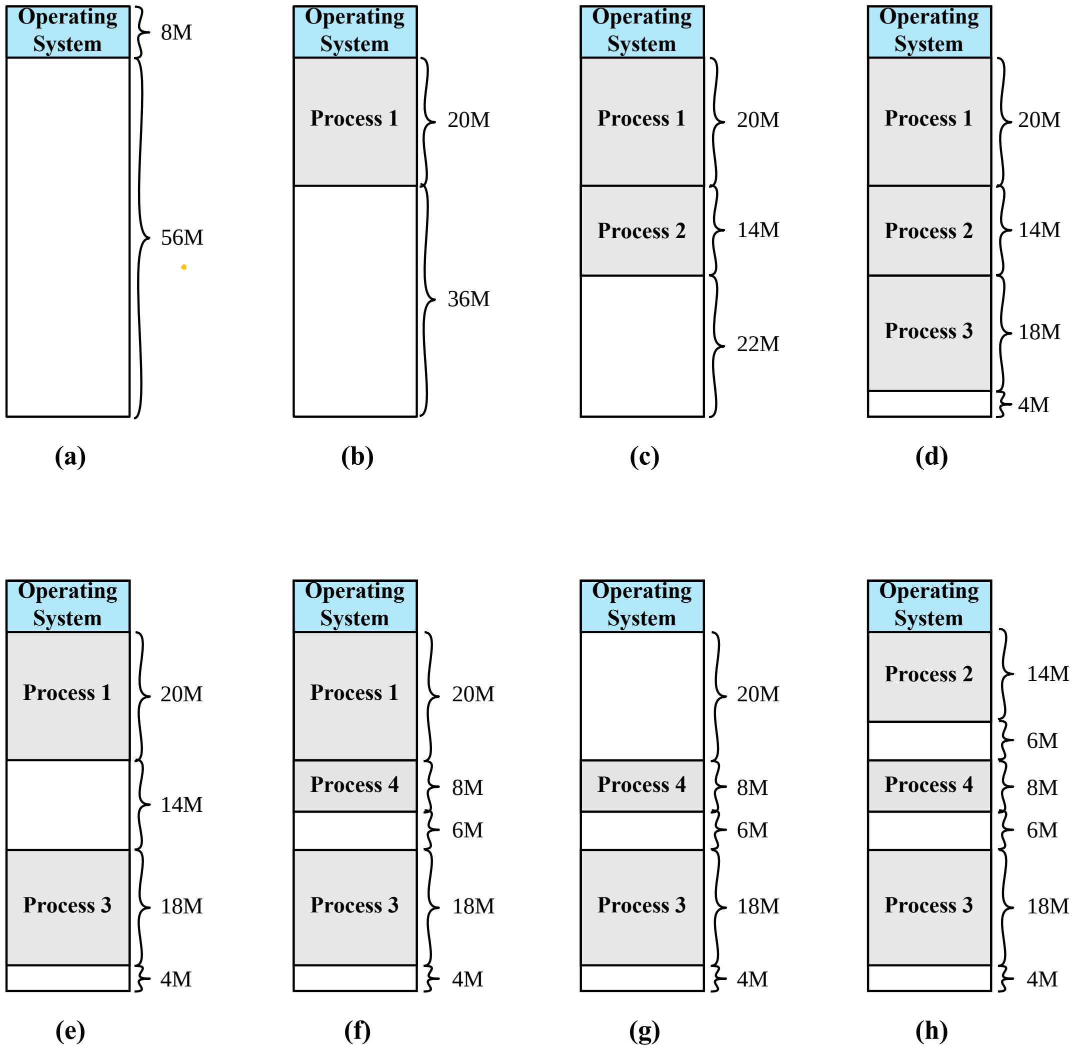
- Now imagine we have a parking lot that can dynamically change the lines as cars come into the parking lot.
- Write an algorithm for drawing these lines as cars come and go



https://www.thecityfix.org/wp-content/uploads/2009/09/SolarRoadways.jpg

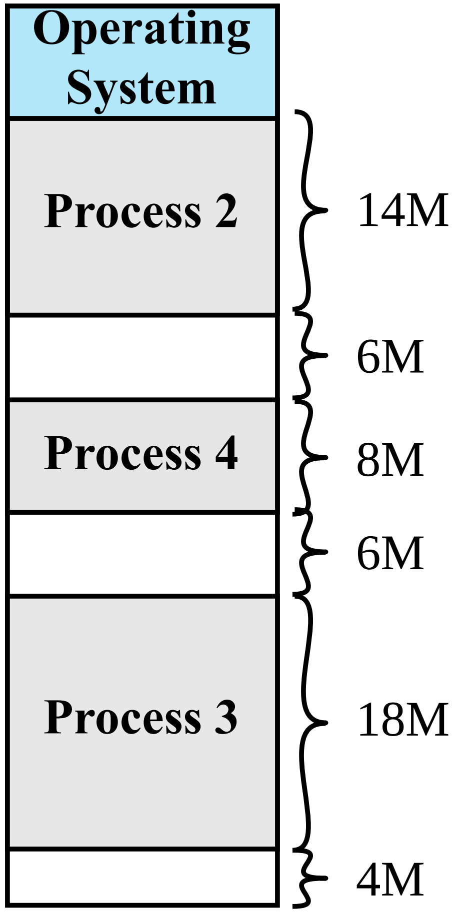
Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as it requires
- Example:
  a) No processes loaded
  b) Process 1 is loaded in
  c) Process 2 is loaded in
  d) Process 3 is loaded in
  e) Process 2 is swapped out for process 4
  f) Process 4 is loaded in
  g) All processes are blocked, so Process 1 is swapped out to load Process 2 back in
  h) Process 2 is loaded back in



Dynamic Partitioning

External fragmentation



Compaction is an option, but time consuming

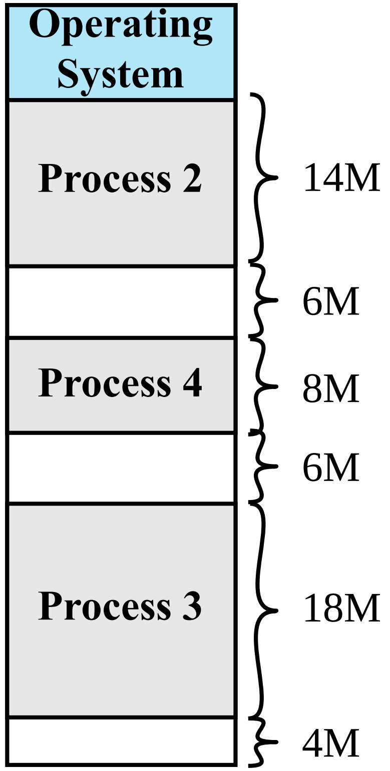
(h)

Dynamic Partitioning Placement

Best-fit	First-fit	Next-fit
• Chooses the block that is closest in size to the request	• Begins to scan memory from the beginning and chooses the first available block that is large enough	• Begins to scan memory from the location of the last placement and chooses the next available block that is large enough

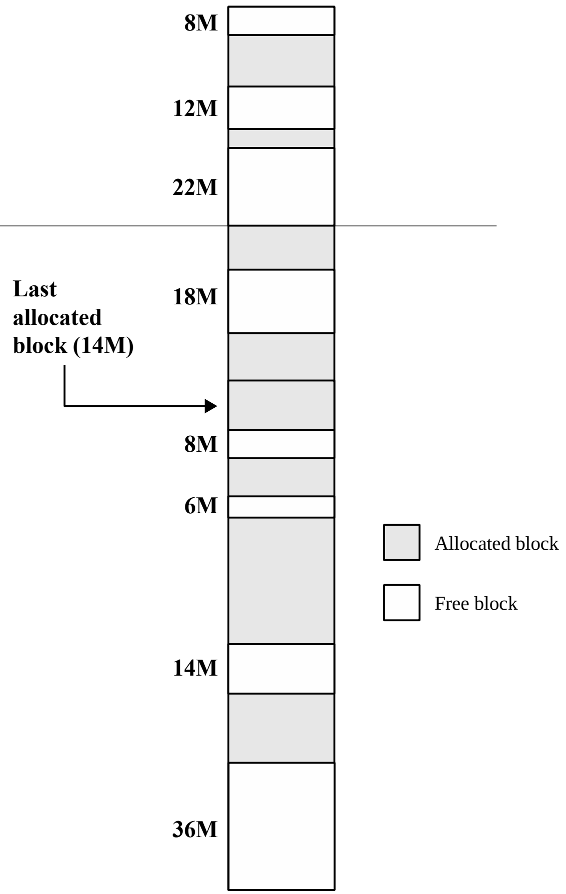
Dynamic Partitioning Placement Example 1

Best-fit	First-fit	Next-fit
• Chooses the block that is closest in size to the request	• Begins to scan memory from the beginning and chooses the first available block that is large enough	• Begins to scan memory from the location of the last placement and chooses the next available block that is large enough



Dynamic partitioning placement example 2

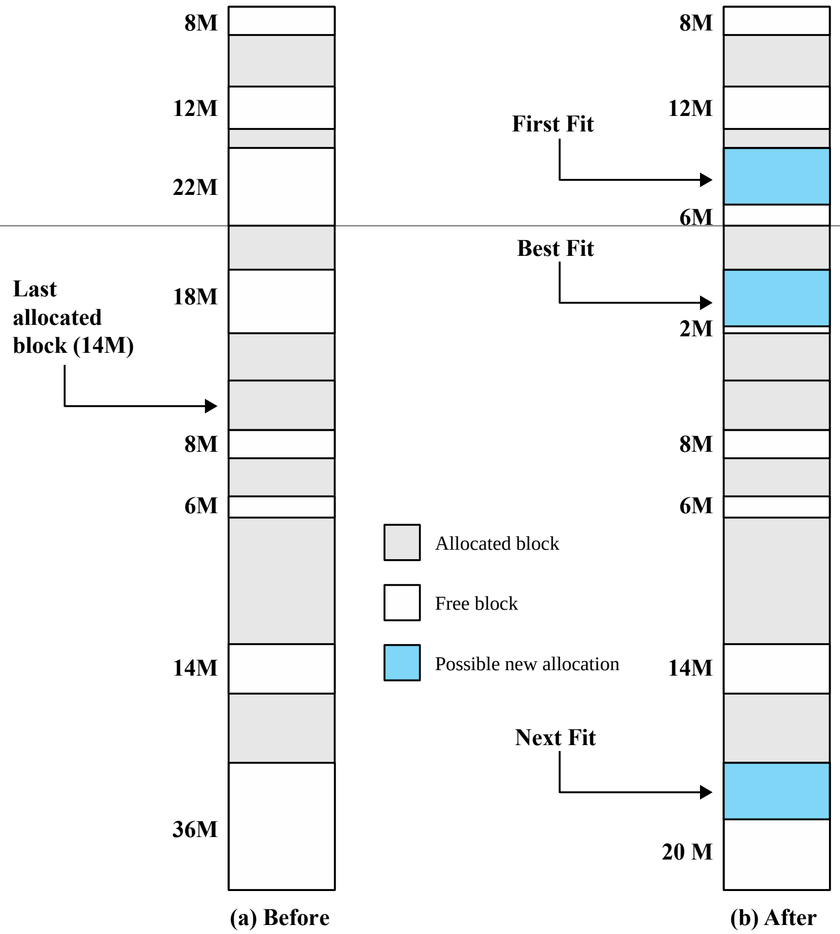
- New request: 16 Mbyte
- Where would each algorithm place the new data?
  - Best-fit
  - First-fit
  - Next-fit



(a) Before

Dynamic partitioning placement example 2

- Which performs better? Best fit, first fit, or next fit?
- Best-fit is the slowest and also leads to external fragmentation
- First-fit and next-fit → similar speed, however, with next-fit, it fragments large segments at the end of memory
- First-fit overall performs best



Addresses for relocation

Logical

- Reference to a memory location independent of the current assignment of data to memory

Relative

- A particular example of logical address, in which the address is expressed as a location relative to some known point

Physical or Absolute

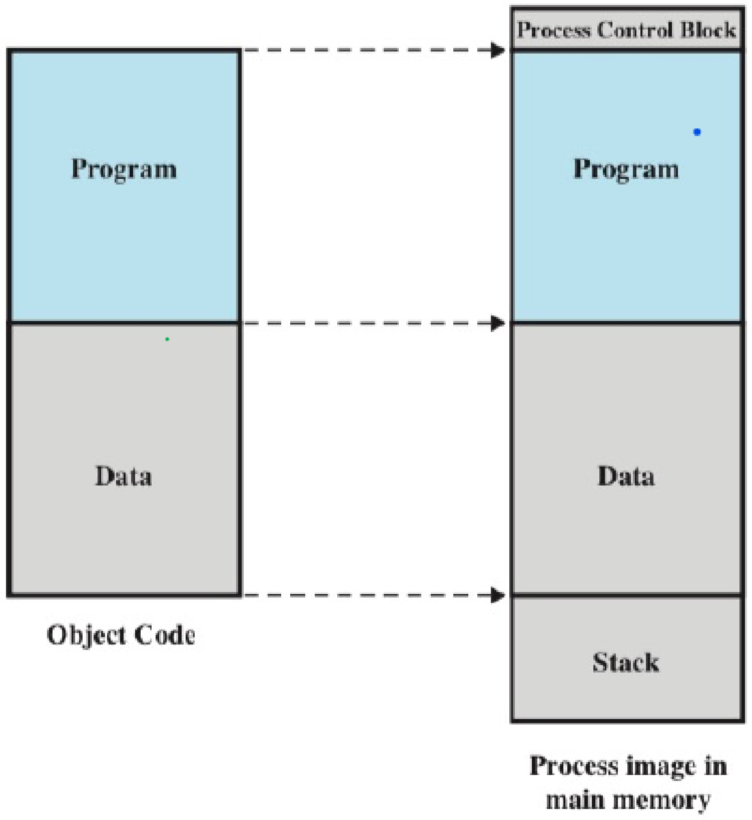
- Actual location in main memory

Loading

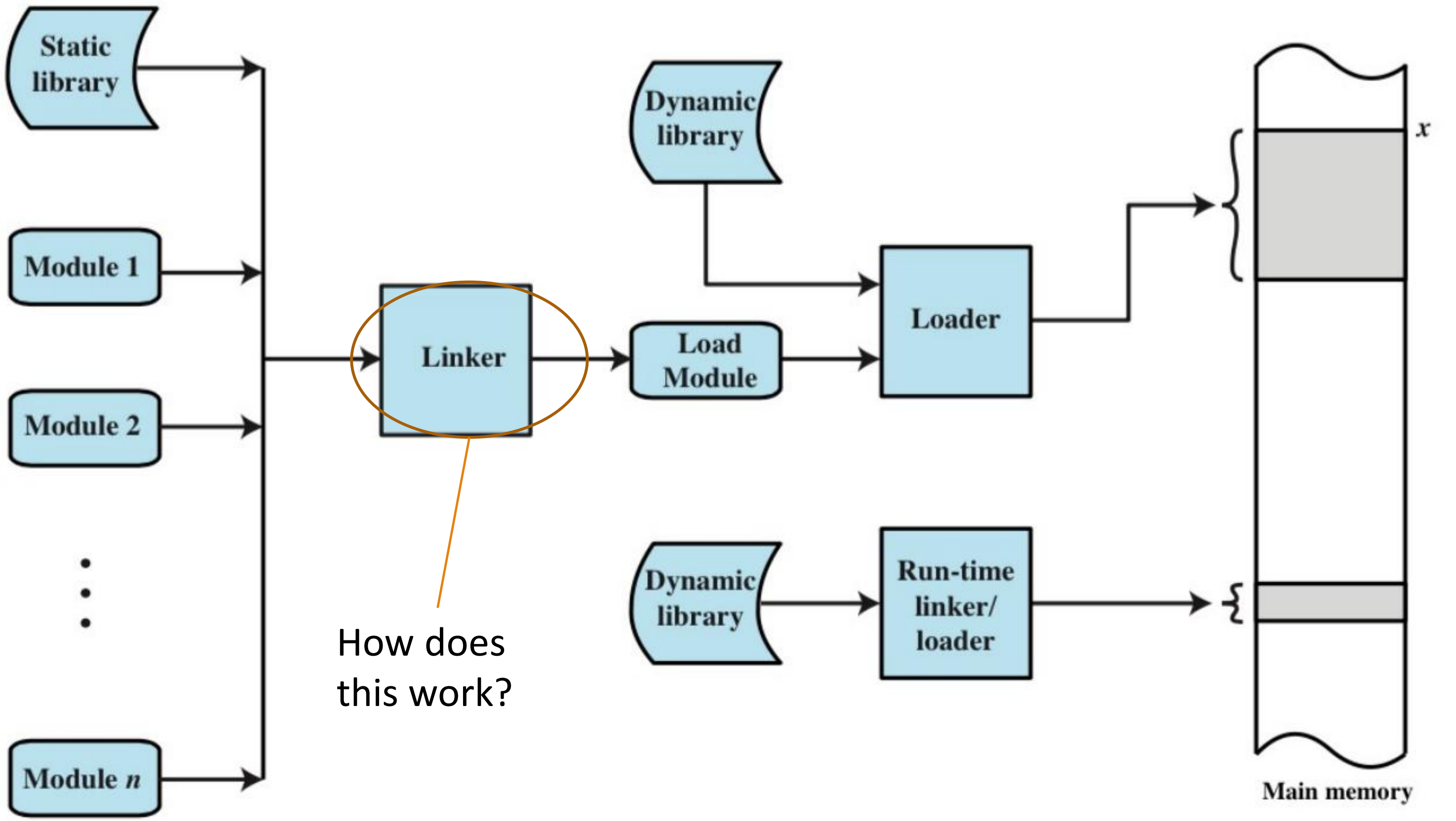
Let's dive deeper into the process of loading a process into memory

First step in creating an active process:

- create a process image
- load a program into main memory

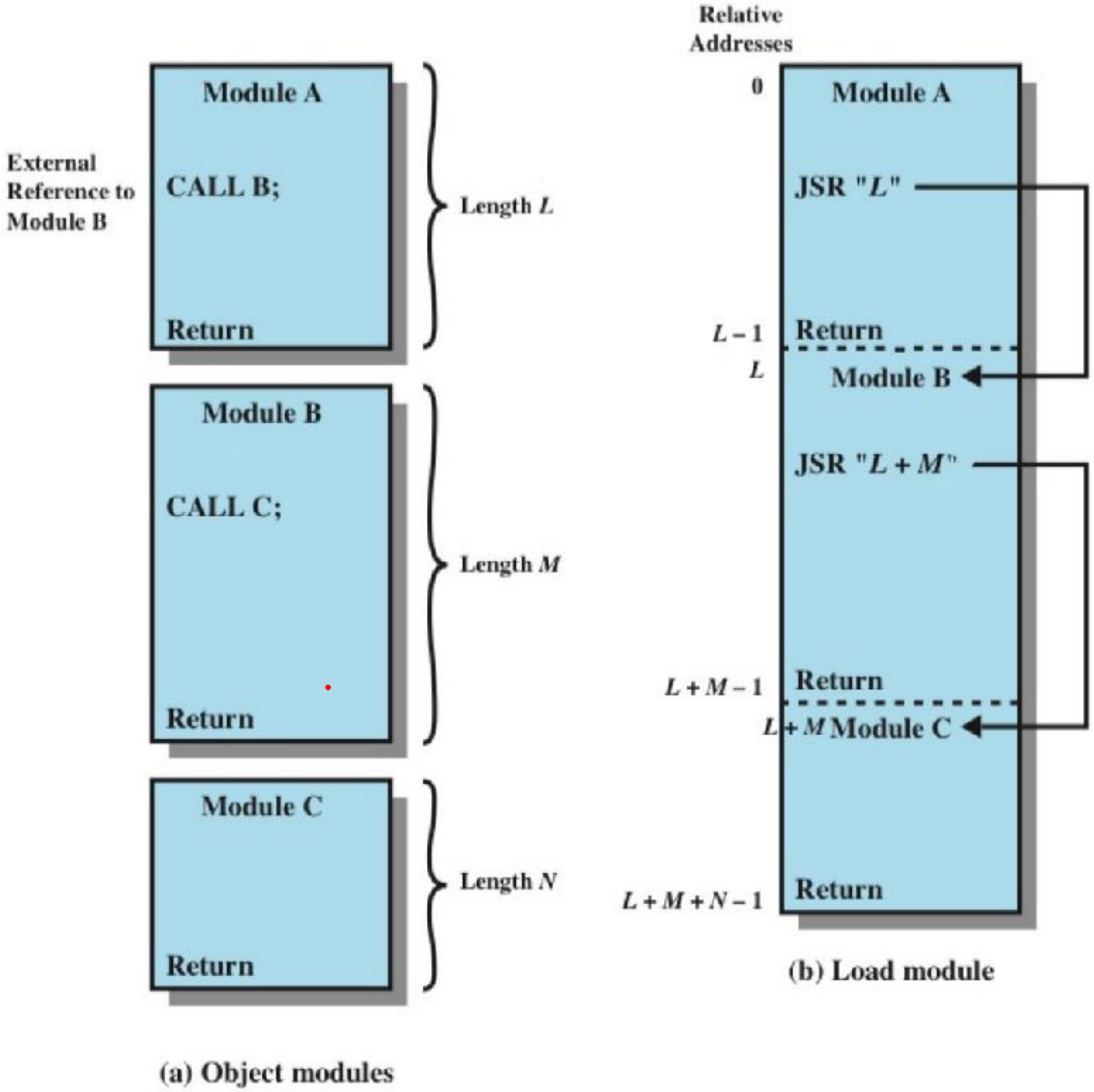


Linking and Loading



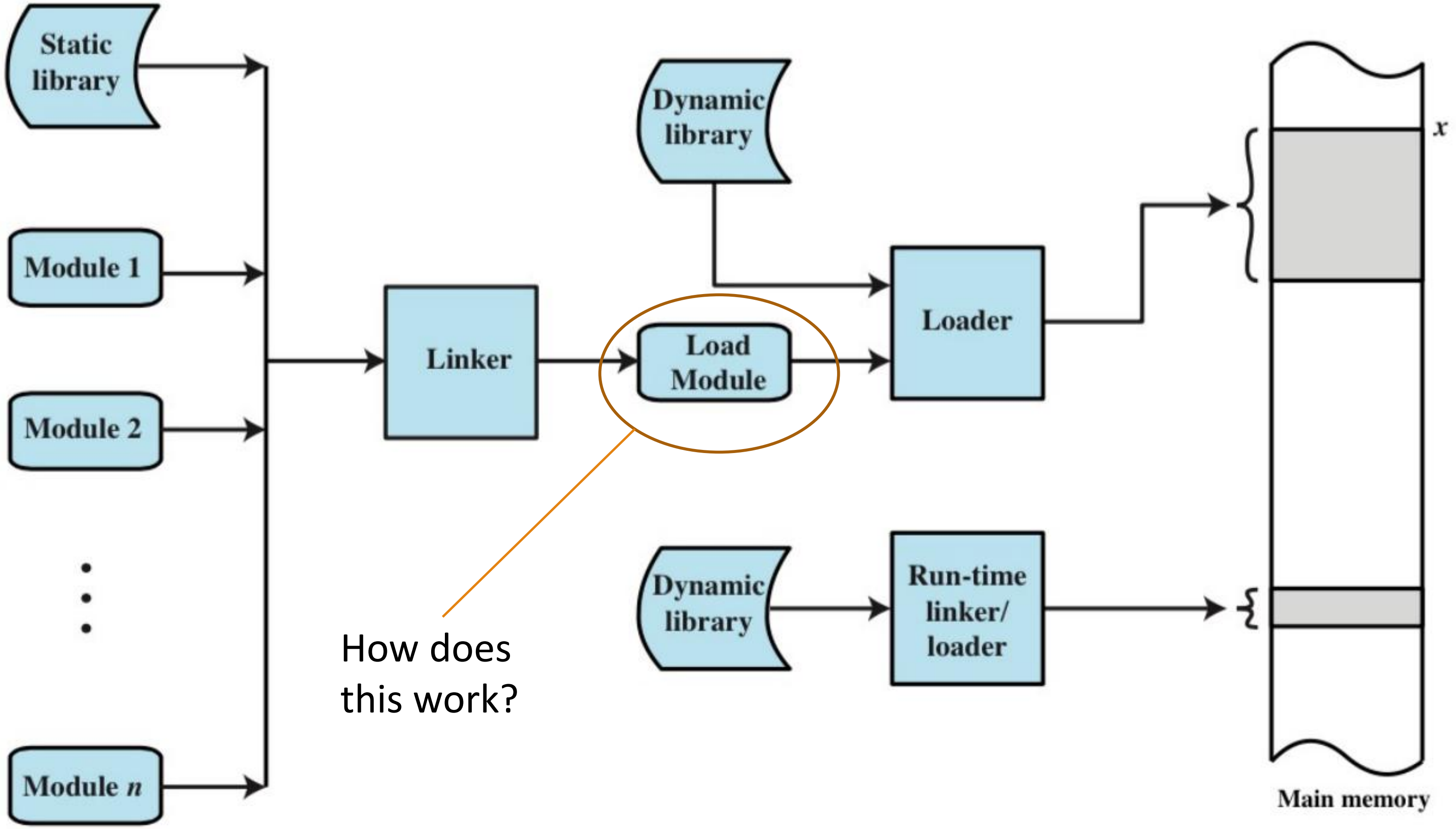
How does this work?

Linking



External Reference to Module B
Module A
CALL B;
Return
Length L
Module B
CALL C;
Return
Length M
Module C
Return
Length N
(a) Object modules
Relative Addresses
0
Module A
JSR "L"
$L-1$
Return
L
Module B
JSR "$L+M$"
$L+M-1$
Return
$L+M$
Module C
$L+M+N-1$
Return
(b) Load module

Linking and Loading

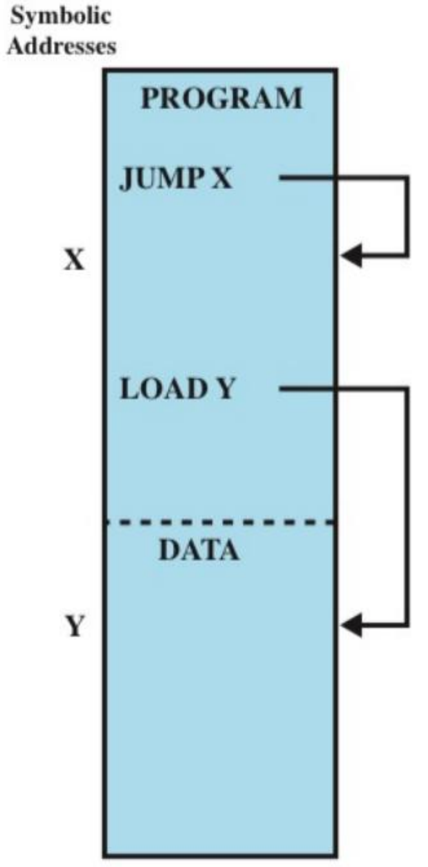


How does this work?

Types of load modules
1. Object module

Modules are loaded via symbols like “X,” or “Y”

Abstraction layer for the user



(a) Object module

Types of load modules
2. Absolute load module

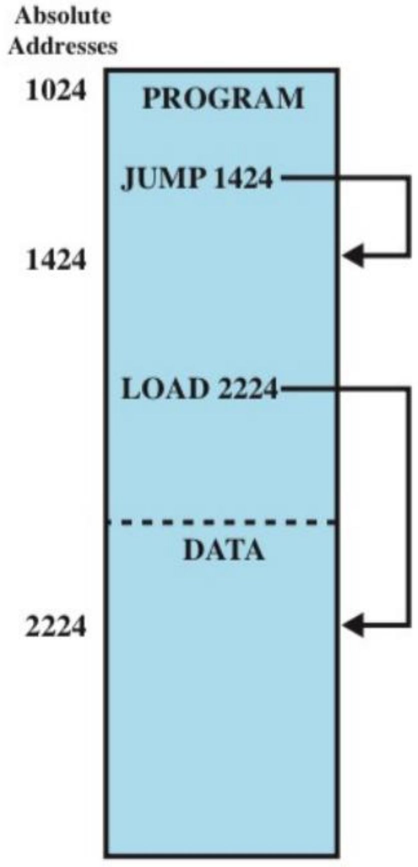
Physical addressing of modules

Let's say module "X" is located in memory location 1424

We can translate the logical module to an absolute address

X is 400 memory locations away from the start of the process

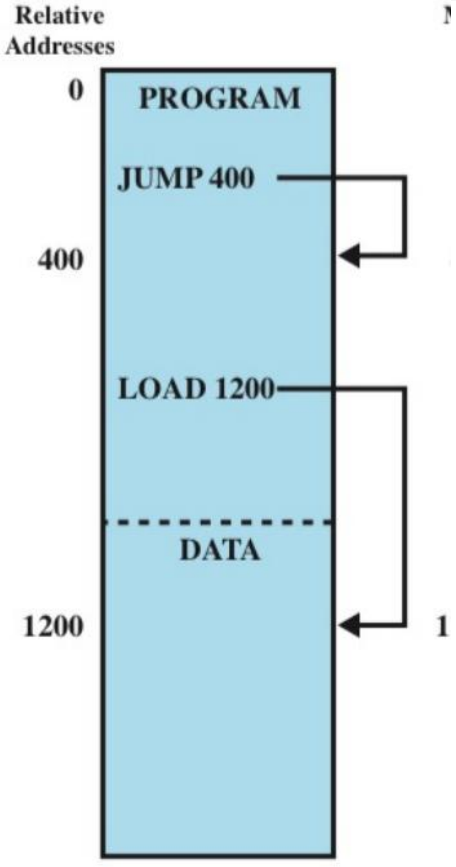
Modules are contiguous because of the linking step



(b) Absolute load module

Types of load modules
3. Relative load address

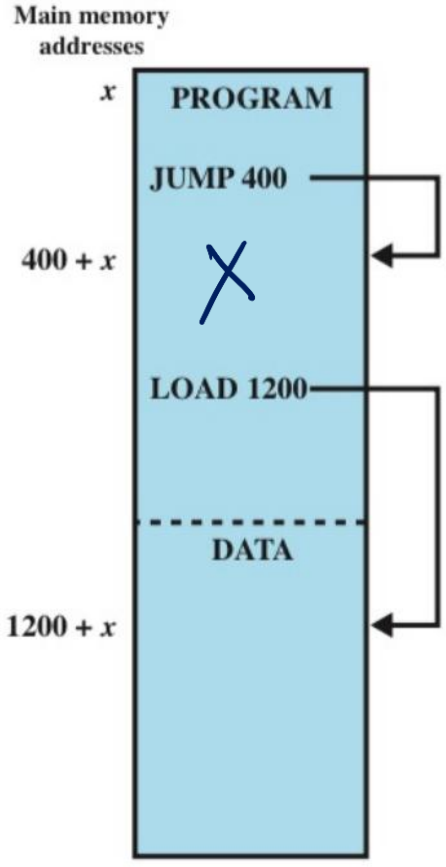
- All memory locations are represented as an offset from the beginning of the file
- Start of the file is 0



(c) Relative load module

Types of load modules
4. Relative load module at an arbitrary position

- Represent the start of the program as any memory location “x”



(d) Relative load module loaded into main memory starting at location x

Partitioning summary

- Partitioning schemes all result in some degree of fragmentation
  - Fixed-sized partitioning → internal fragmentation
  - Variable-sized partitioning → external fragmentation
- What was the next evolution in memory allocation?

Paging

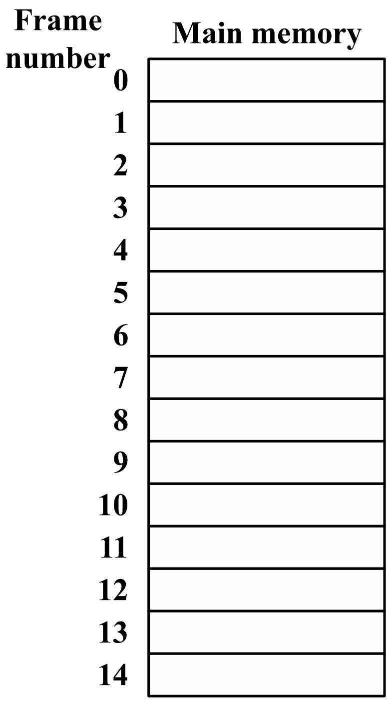
Paging

- Key idea:
  - Partition memory into equal fixed-size chunks that are relatively small
  - Process is also divided into small fixed-size chunks of the same size
- Turns out we can have very little fragmentation under paging

Pages	Frames
• Chunks of a process	• Available chunks of memory

Assignment of Process Pages to Free Frames

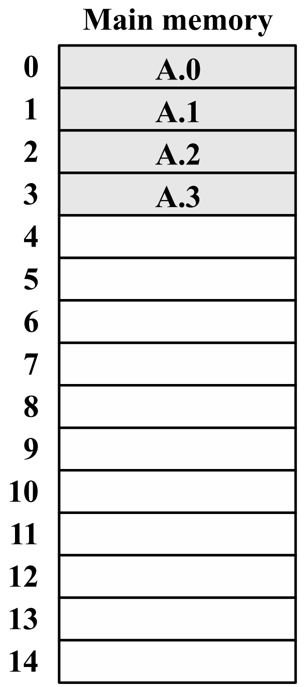
1) picture a section of memory with 15 *frames* (i.e., chunk) of free memory



(a) Fifteen Available Frames

Assignment of Process Pages to Free Frames

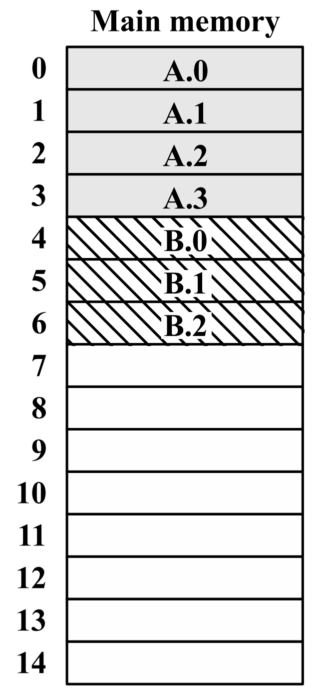
2) Process A is loaded from secondary memory into primary memory.



(b) Load Process A

Assignment of Process Pages to Free Frames

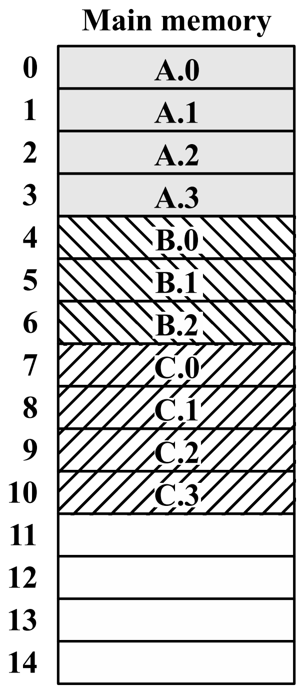
3) Process B is loaded from secondary memory into primary memory.



(c) Load Process B

Assignment of Process Pages to Free Frames

4) Process C is loaded from secondary memory into primary memory.

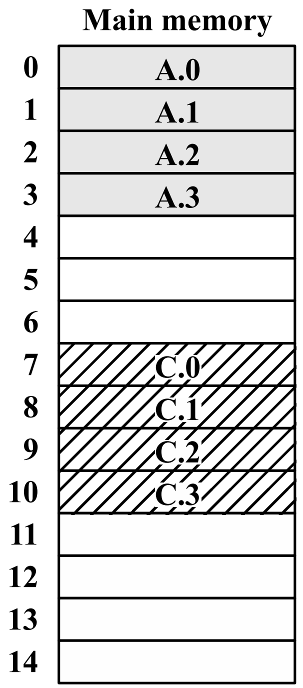


(d) Load Process C

Assignment of Process Pages to Free Frames

5) Process B is suspended and swapped out of main memory

What if we wanted to load Process D, which is five pages long?

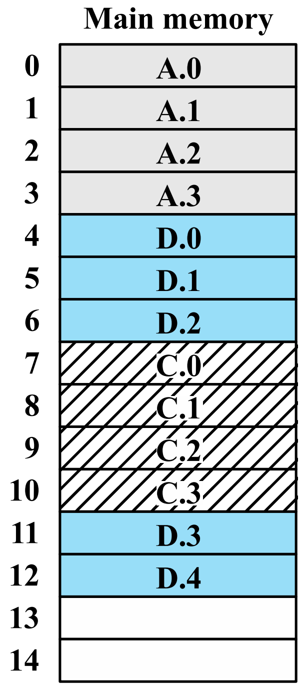


(e) Swap out B

Assignment of Process Pages to Free Frames

How does page size affect internal fragmentation?

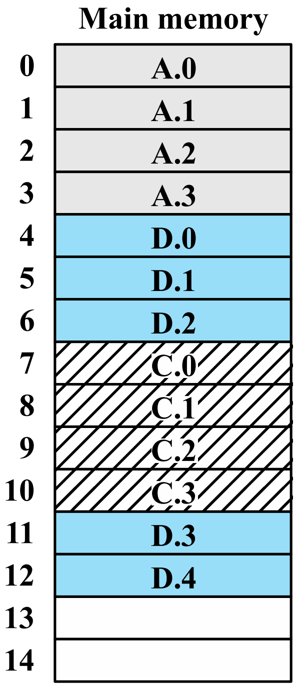
- Let’s say we have a process image that is 517 MB
- If the page size is 20MB:
  - 517 % 20 = 17
  - Leads to internal fragmentation of 3
- If the page size is 2MB
  - 517 % 2 = 1
  - Leads to internal fragmentation of 1



(f) Load Process D

Assignment of Process Pages to Free Frames

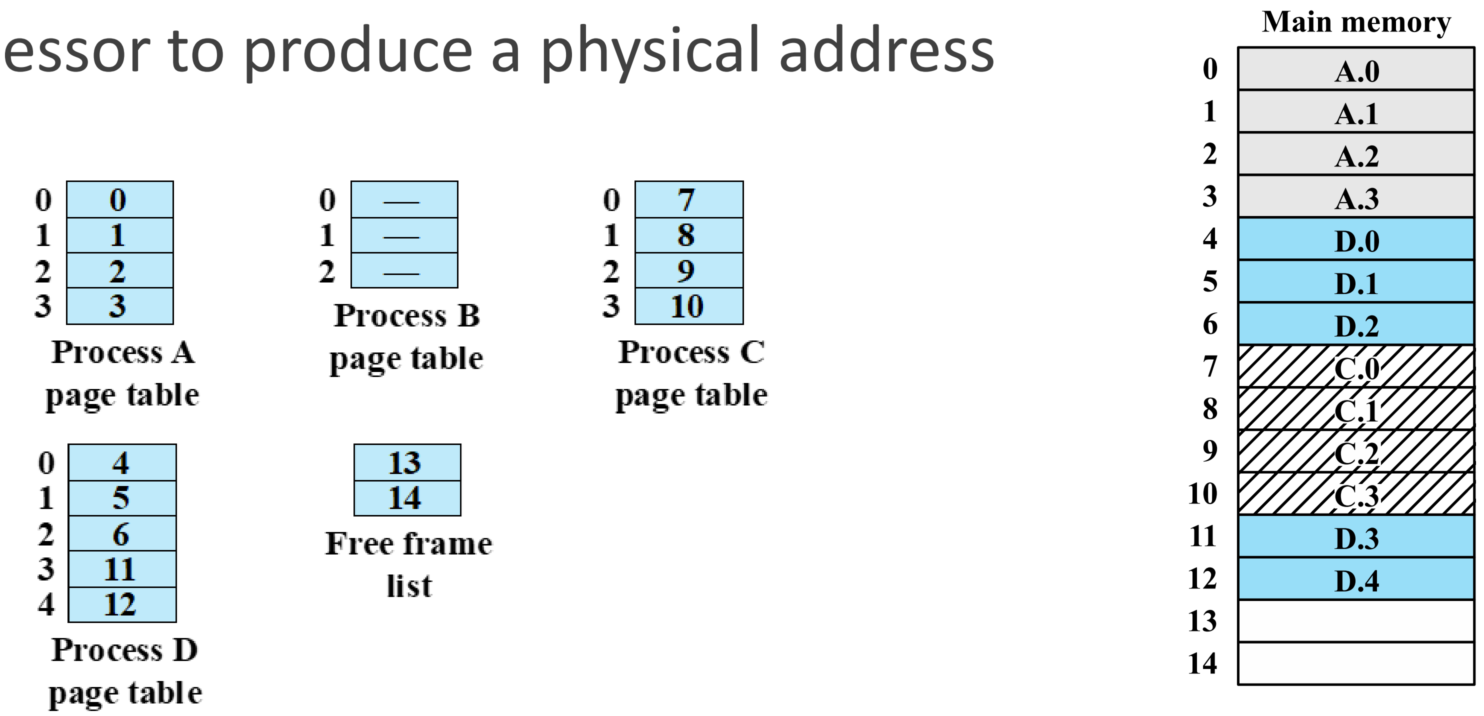
- Would a simple relative addressing in this scenario work?
- No! Frames containing Process C are in the way of Process D
- Requires the use of a *page table*



(f) Load Process D

Page Table

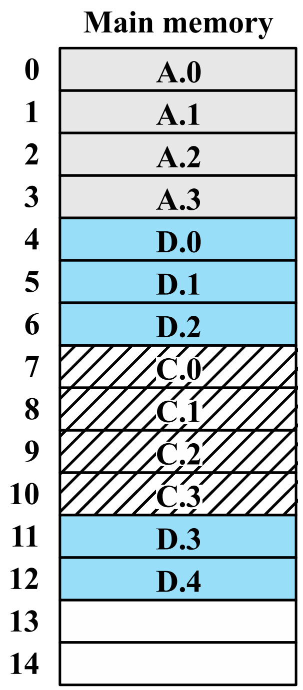
- Maintained by operating system for each process
- Contains the frame location for each page in the process
- Used by processor to produce a physical address



(f) Load Process D

Note on Page Tables

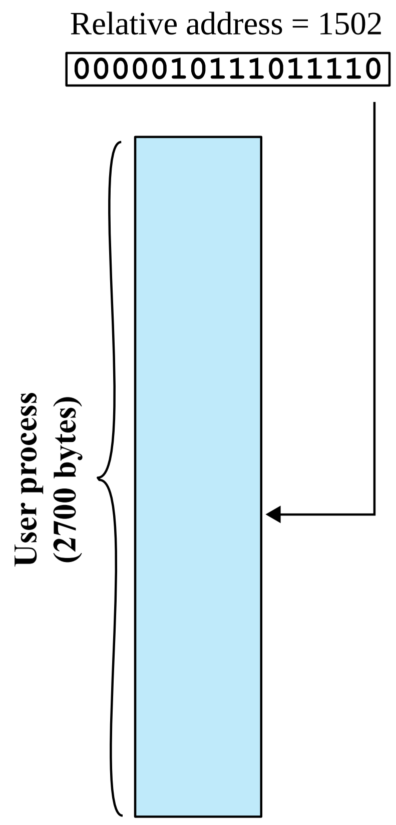
- Notice that this is essentially a fixed-width partitioning scheme
- Differences from typical fixed-width partitioning:
  - Partitions are very small
    - Thus, internal fragmentation is relatively small
  - Process images do not need to be contiguous partitions of memory



(f) Load Process D

Relative to Physical Addresses With Paging

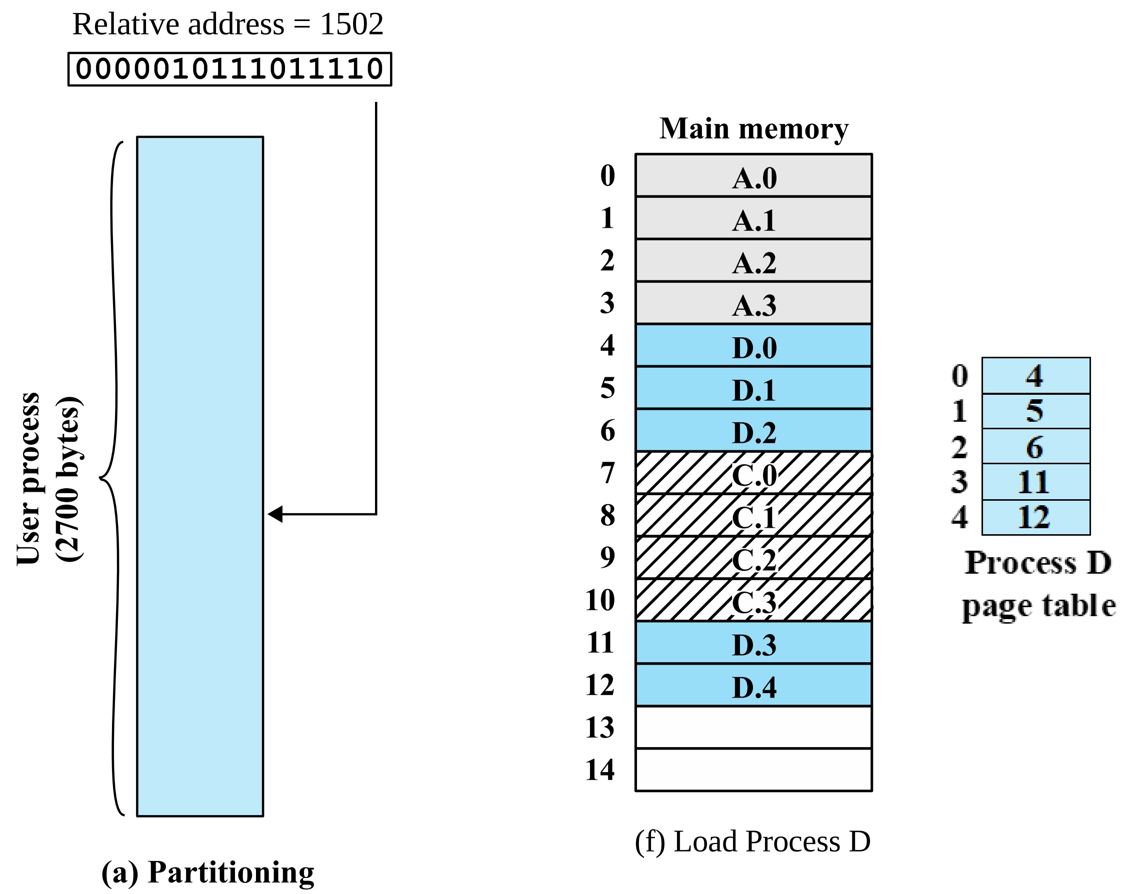
- Review: under a non-paging partitioning scheme, a relative address is just an offset from the beginning of the process image



(a) Partitioning

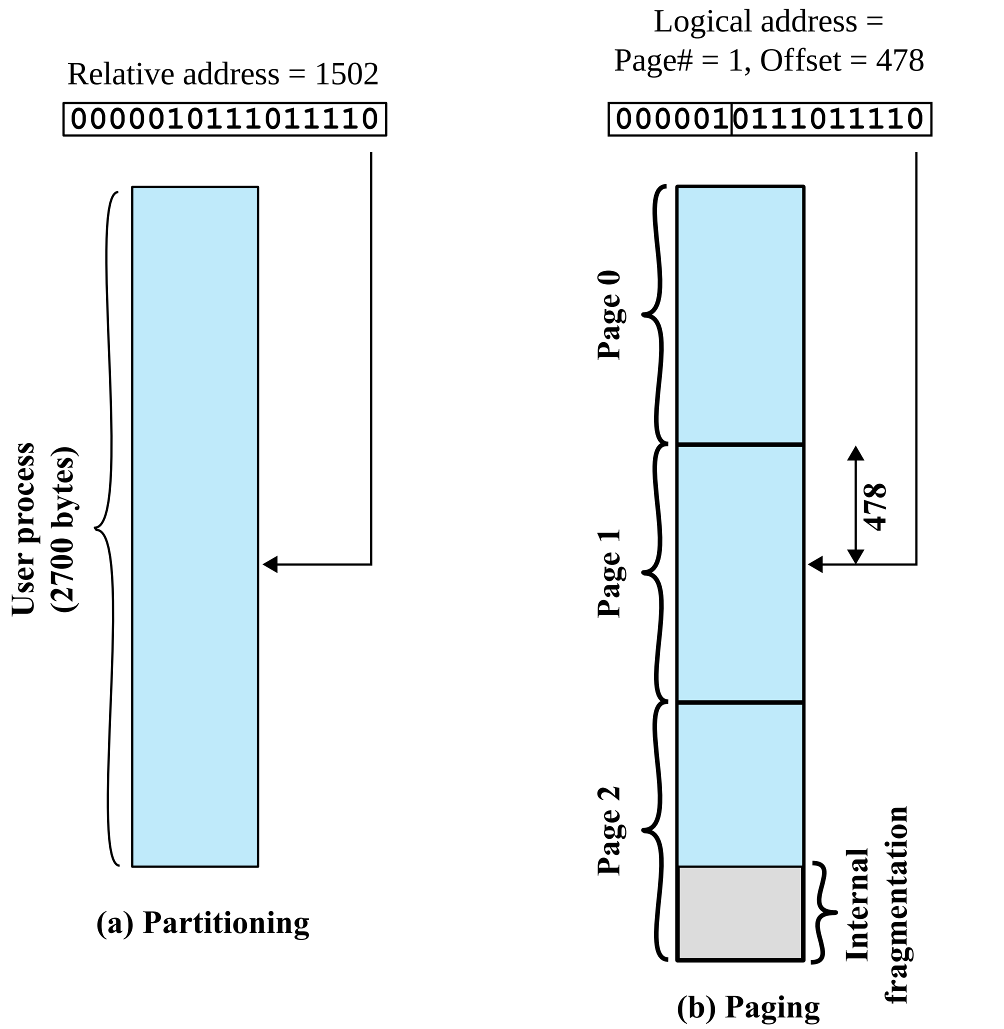
Relative to Physical Addresses With Paging

- Review: under a non-paging partitioning scheme, a relative address is just an offset from the beginning of the process image
- How could we make this work for a paging system?



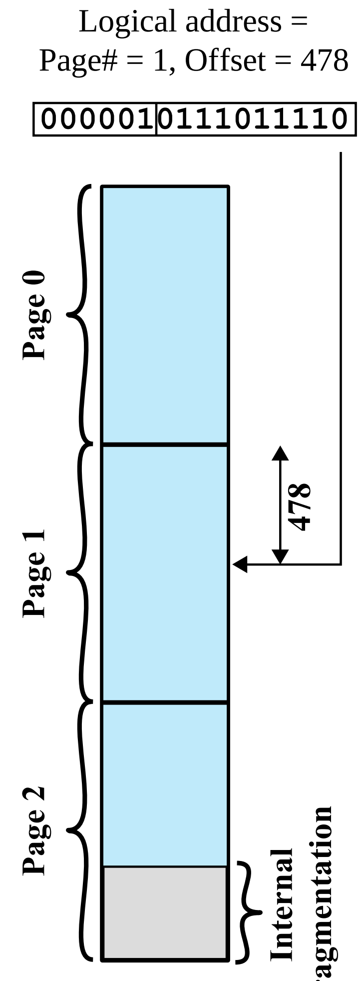
Relative to Physical Addresses With Paging

- Paging systems use a two part address:
  - Part one is the page of the address
  - Part two is the offset from the beginning of the page



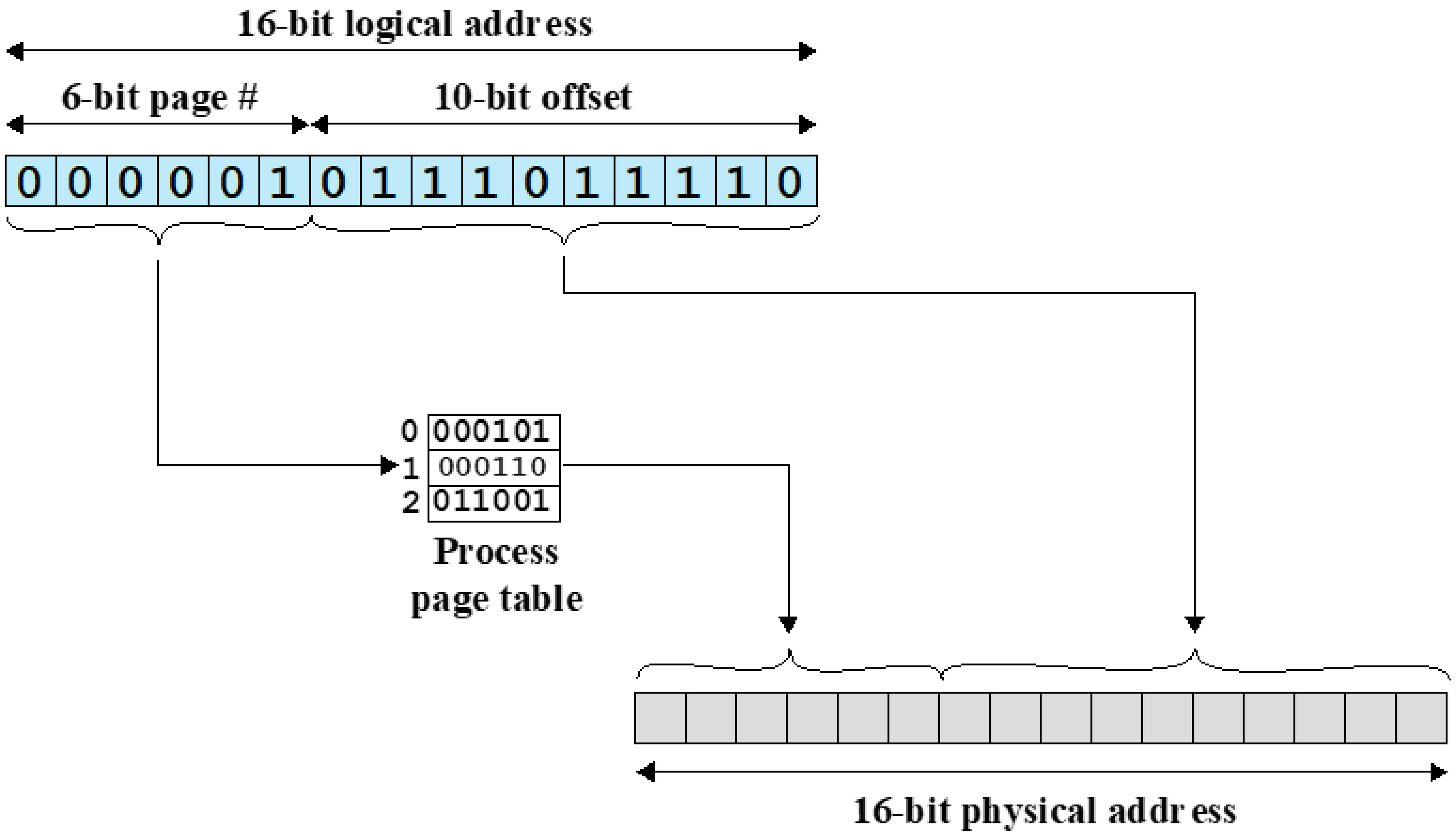
Relative to Physical Addresses With Paging

- The whole paging address would be broken down into:
  - n bits for page number
  - m bits for offset
- The page number is the left n bits of the relative address



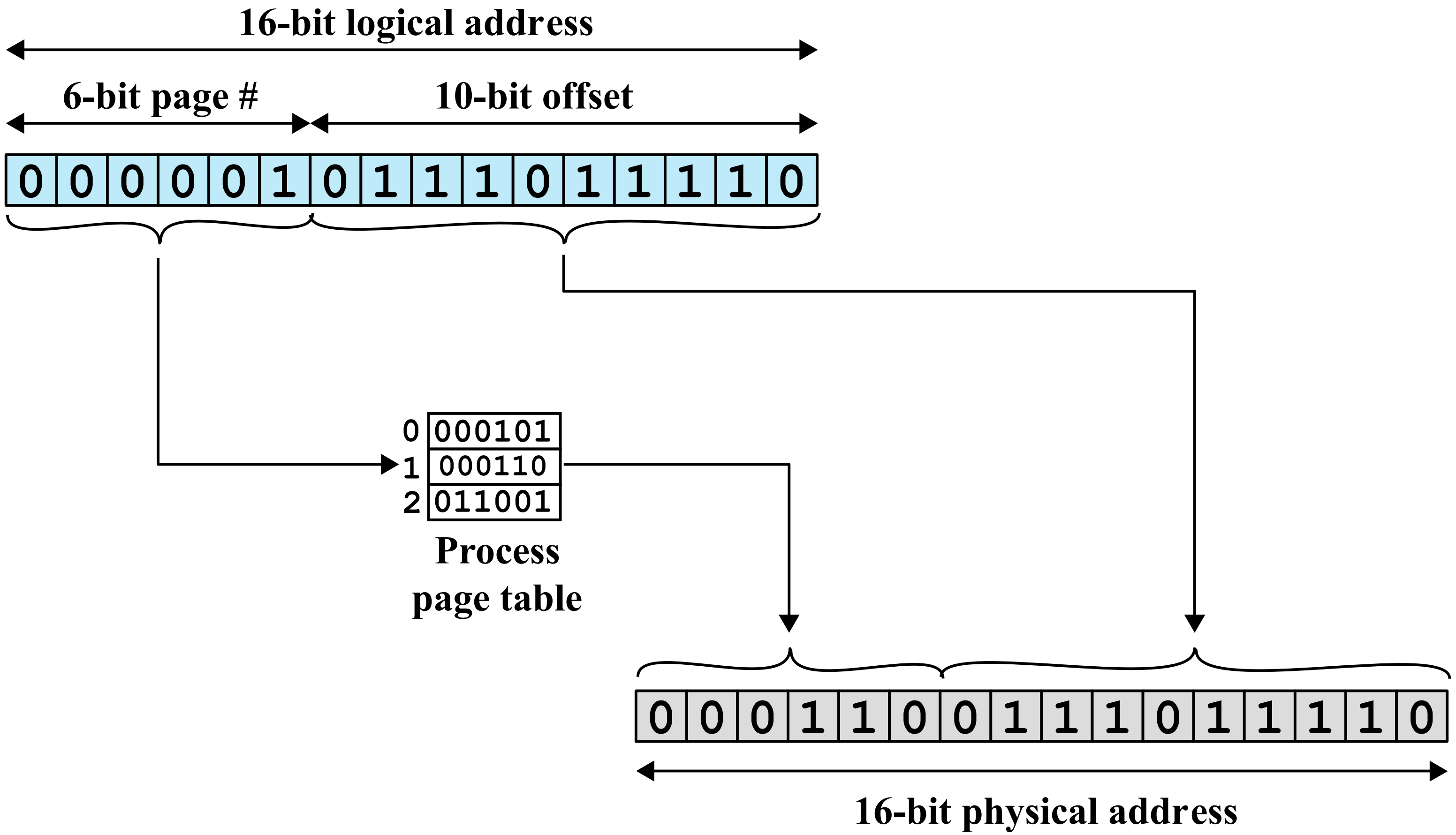
(b) Paging

Paging Translation Example



(a) Paging

Paging Translation Example



(a) Paging

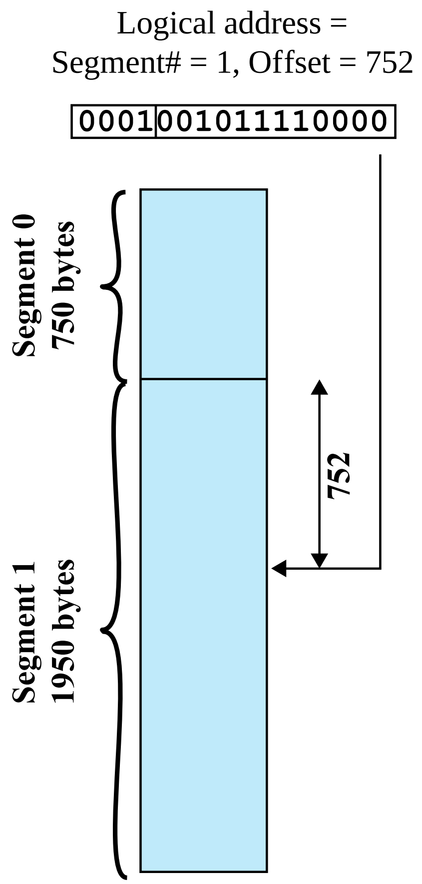
Segmentation

Segmentation

- Paging offers a non-contiguous *equal-sized fixed-width* partitioning scheme
- *Segmentation* is a non-contiguous *dynamic-sized* partitioning scheme
  - Provided as a convenience for organizing programs and data

Segmentation

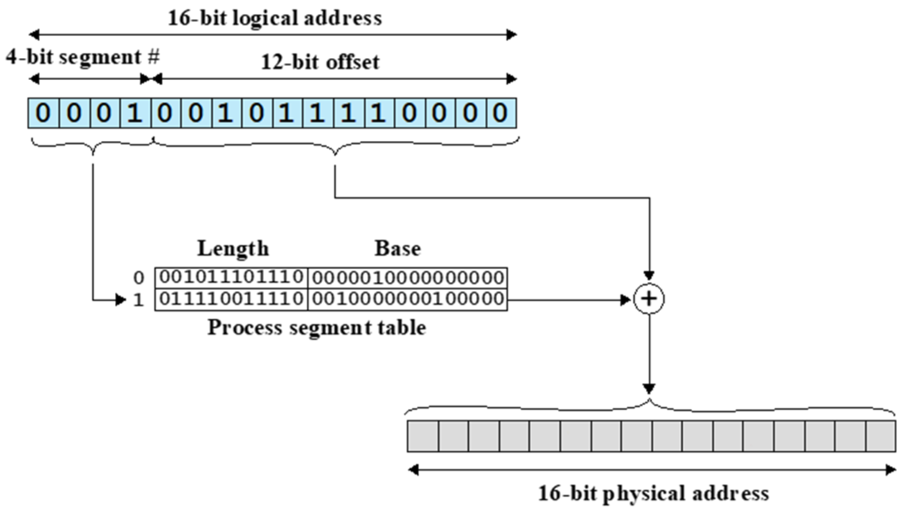
- A program can be subdivided into segments
  - May vary in length
  - There is a maximum length
- Addressing consists of two parts
  - Segment number
  - An offset
- Similar to dynamic partitioning
- Eliminates internal fragmentation
- Usually visible to programmer
- The principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation



(c) Segmentation

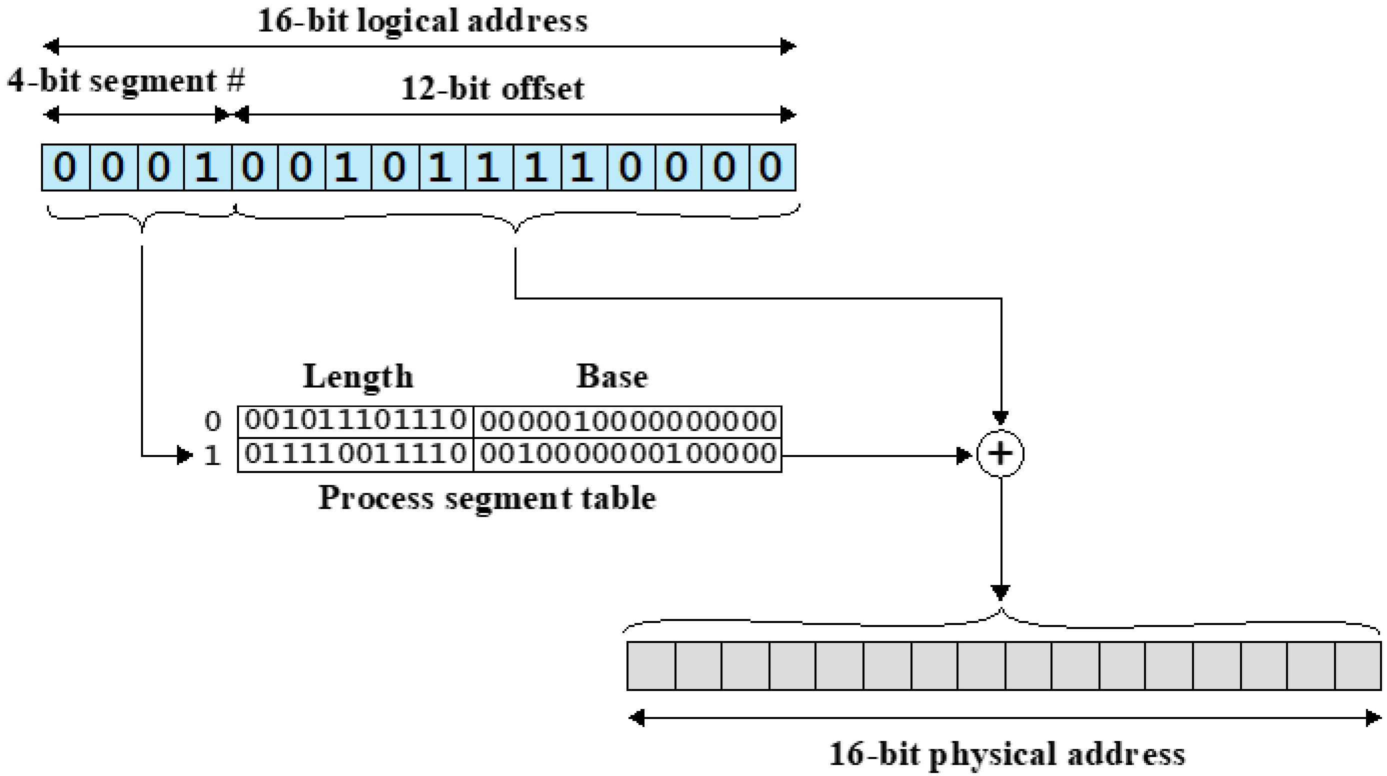
Translating Segment Addressing

- Extract the segment number as the leftmost n bits of the logical address
- Use the segment number as an index into the process segment table to find the starting physical address of the segment
- Compare the offset, expressed in the rightmost m bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid
- The desired physical address is the sum of the starting physical address of the segment plus the offset



(b) Segmentation

Segmentation Translation Example



(b) Segmentation

Exercise

- Consider a simple segmentation system that has the following table:

Segment number	Starting Address	Length (bytes)
0	660	248
1	1752	422
2	222	198
3	996	604

- For each of the following logical addresses (segment number, offset) determine the physical address or indicate if a segment fault occurs
  - 0, 198
  - 2, 156
  - 1, 530
  - 3, 444
  - 0, 222