# CPSC 5042: Week 2

Understanding Main Memory

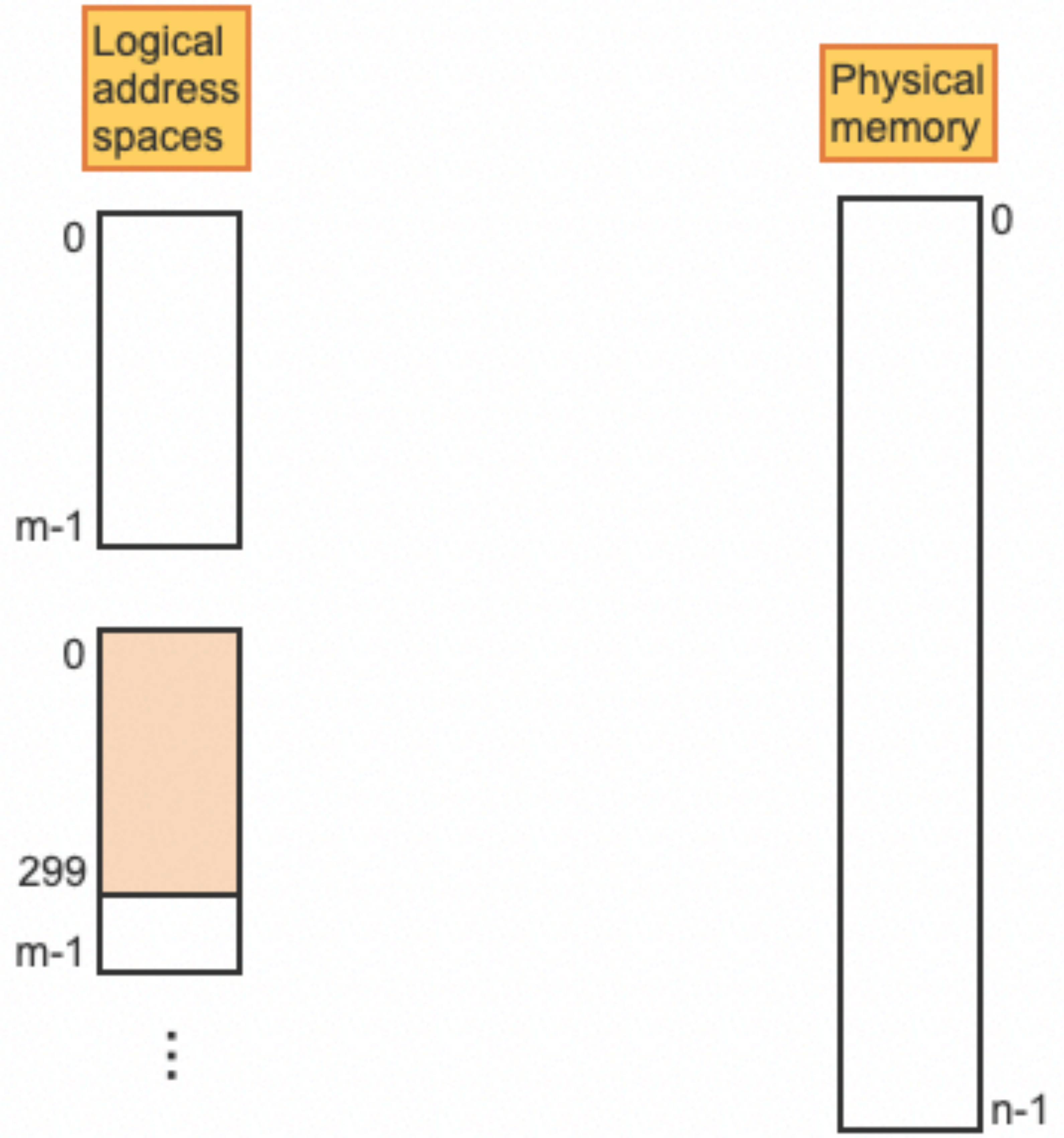
“640k ought to be enough for anybody.”

# What is Main Memory?

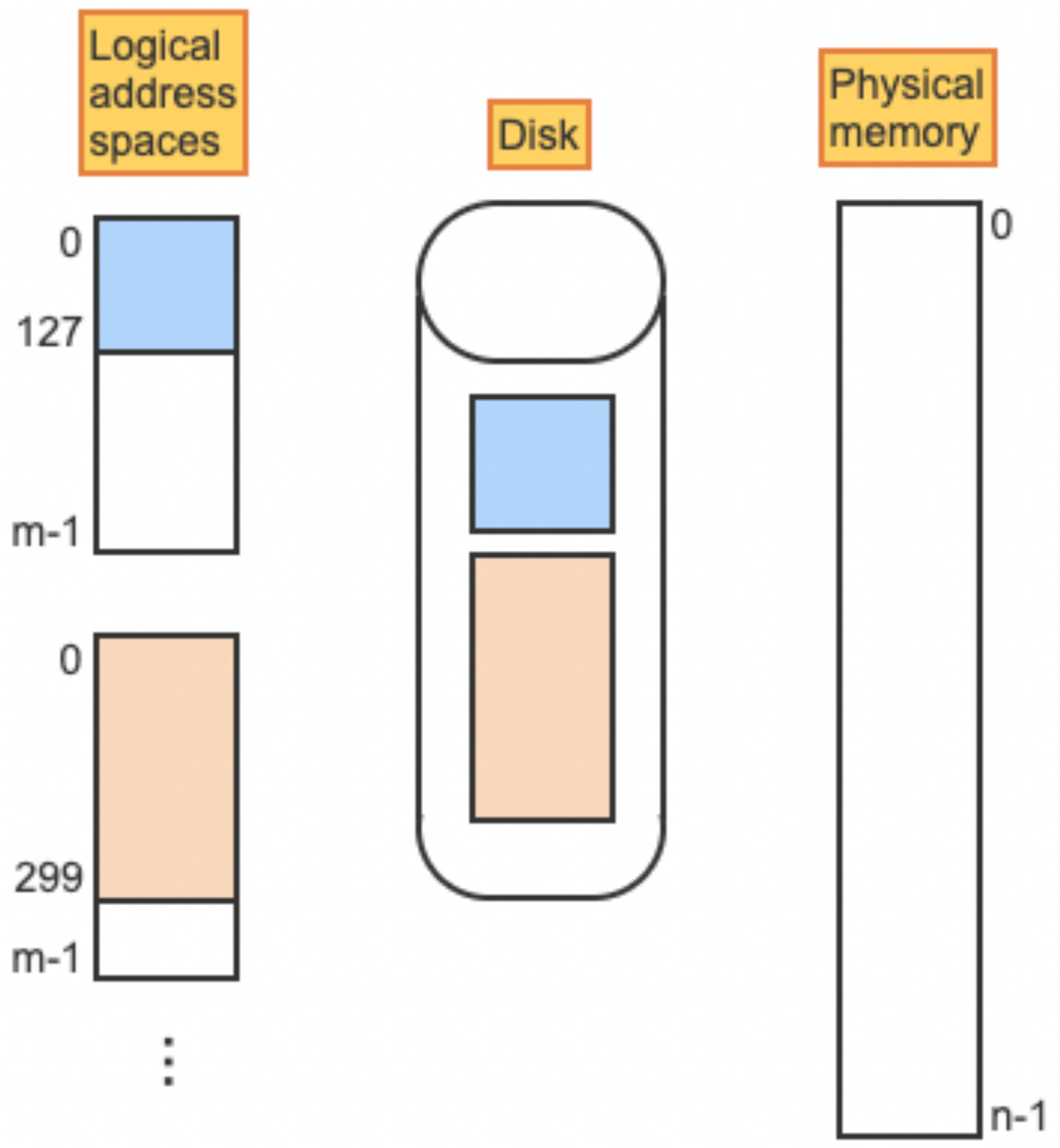
- Main Memory is the memory unit that works directly with CPU, and in which data and instructions must reside.
- Also called RAM (Random Access Memory)
- Famous Computer Scientist Jay Forrester developed it in 1947 as a part of a project in MIT called Whirlwind. At the time, they called in “coincident current magnetic core storage”. It later was just called RAM

# Logical vs Physical Memory

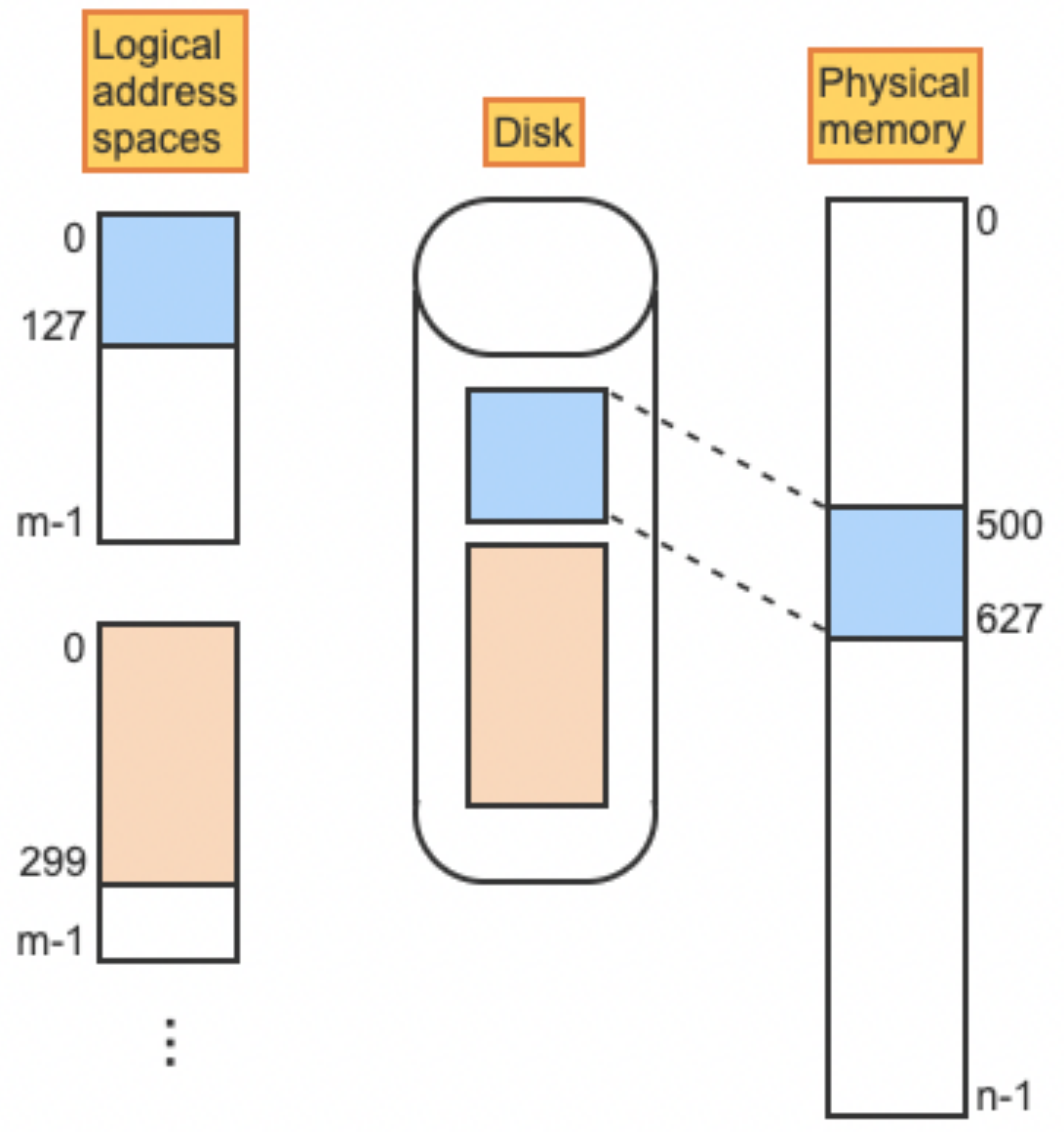

- A computer system has a single physical memory consisting of n words addressed by fixed-size physical addresses in the range [0 : n-1]



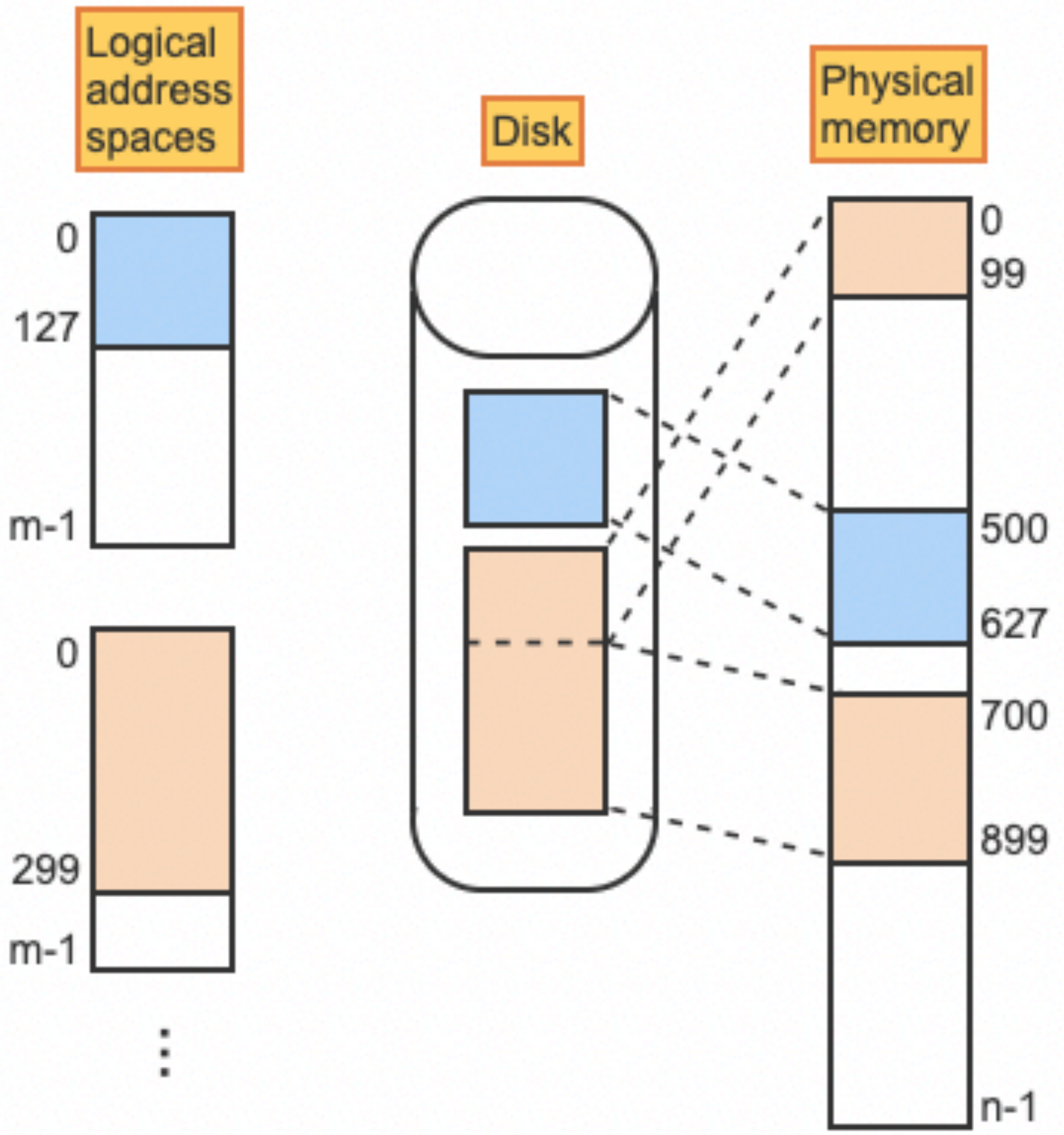

- To facilitate program development and program loading, multiple logical address spaces are provided with possibly different address ranges [0 : m-1]




- The occupied portion of each logical address space is kept on disk

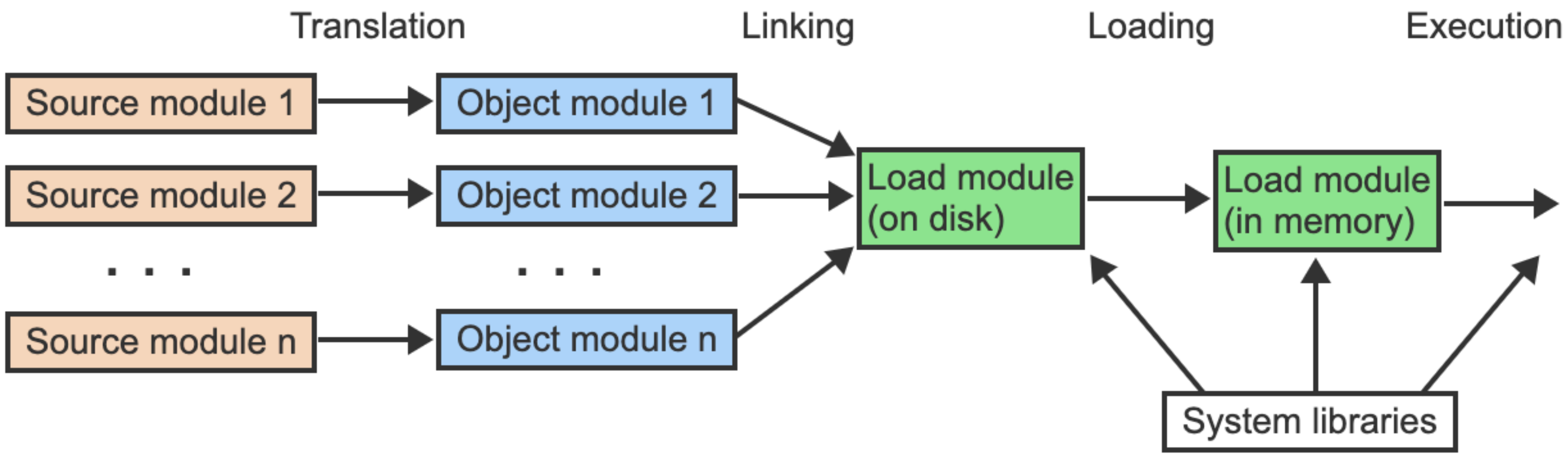



- A program may be copied to a contiguous region of physical memory when needed for execution. Logical space [0 : 127] is copied to memory locations [500 : 627]




- A program can also be copied to multiple disjoint regions of physical memory. The space [0 : 299] is mapped to the regions [0 : 99] and [700 : 899]

# Program Transformations




- A program consists of one or more source modules written in a high-level language or assembly language.
- Each module is translated separately into an object module using a compiler or assembler.
- Object modules are combined into a load module by a linker. The load module is kept on disk.
- Prior to execution, the load module is copied to physical memory by a loader.
- Various system libraries, in the form of object modules, may be included in the load module by the linker, the loader, or during execution

# Relocations & Address Binding

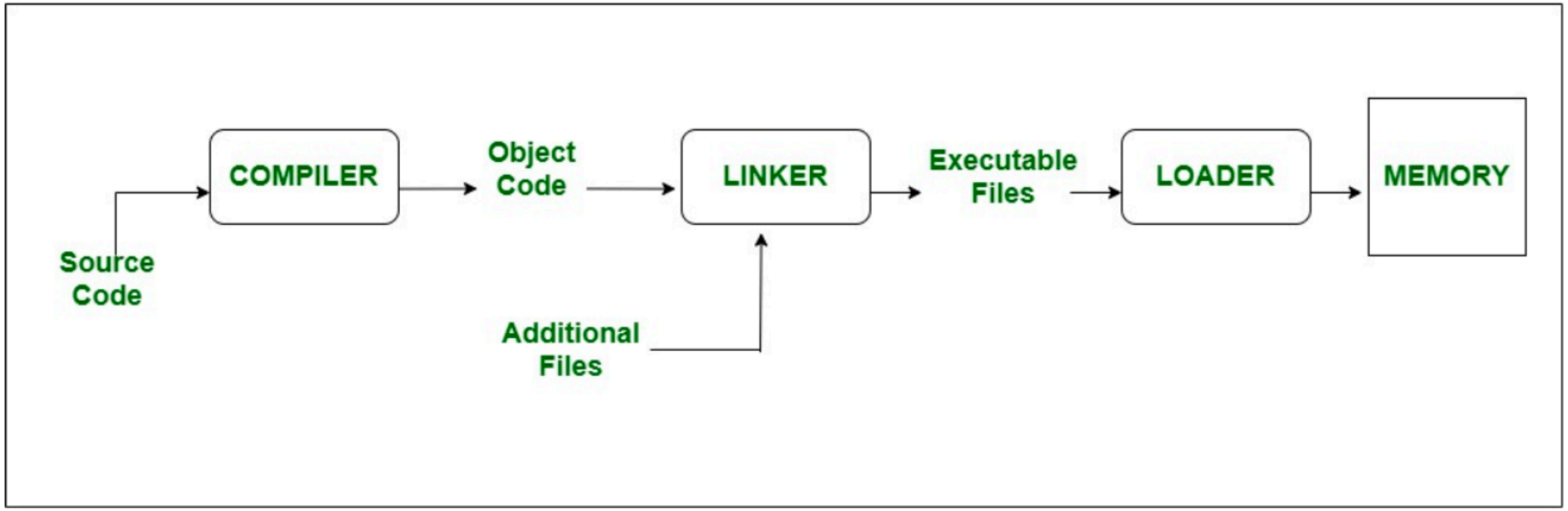
# Why do we need to relocate?

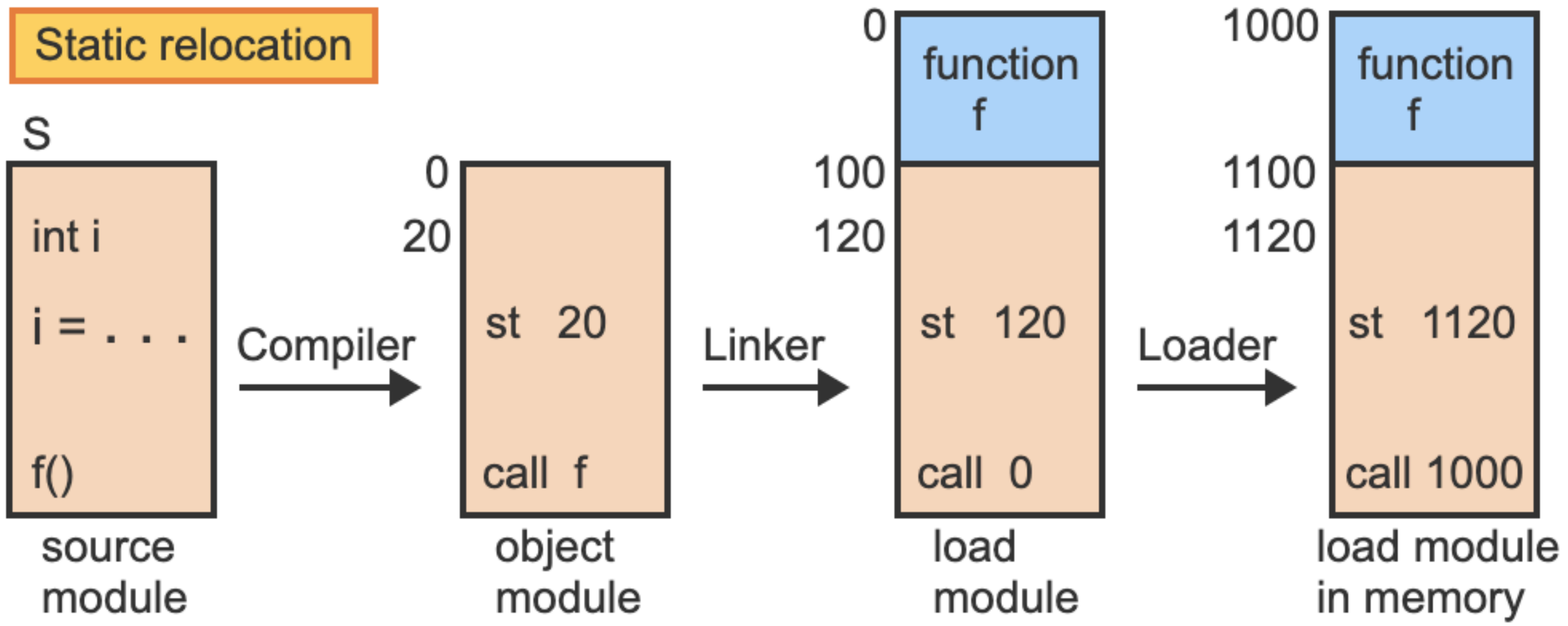
- Since the physical addresses of the program components are not known until load time,
- the compiler/assembler and the linker assume logical address spaces starting with address 0.
- During each step of the program transformation, the logical addresses of instructions and data may be changing.
- Only during the final step of program loading are all logical addresses bound to actual physical addresses in memory.

# What is relocation?

- Program relocation is the act of moving a program component from one address space to another.
- The relocation may be between two logical address spaces or from a logical address space to a physical address space.

# Taking code to execution:



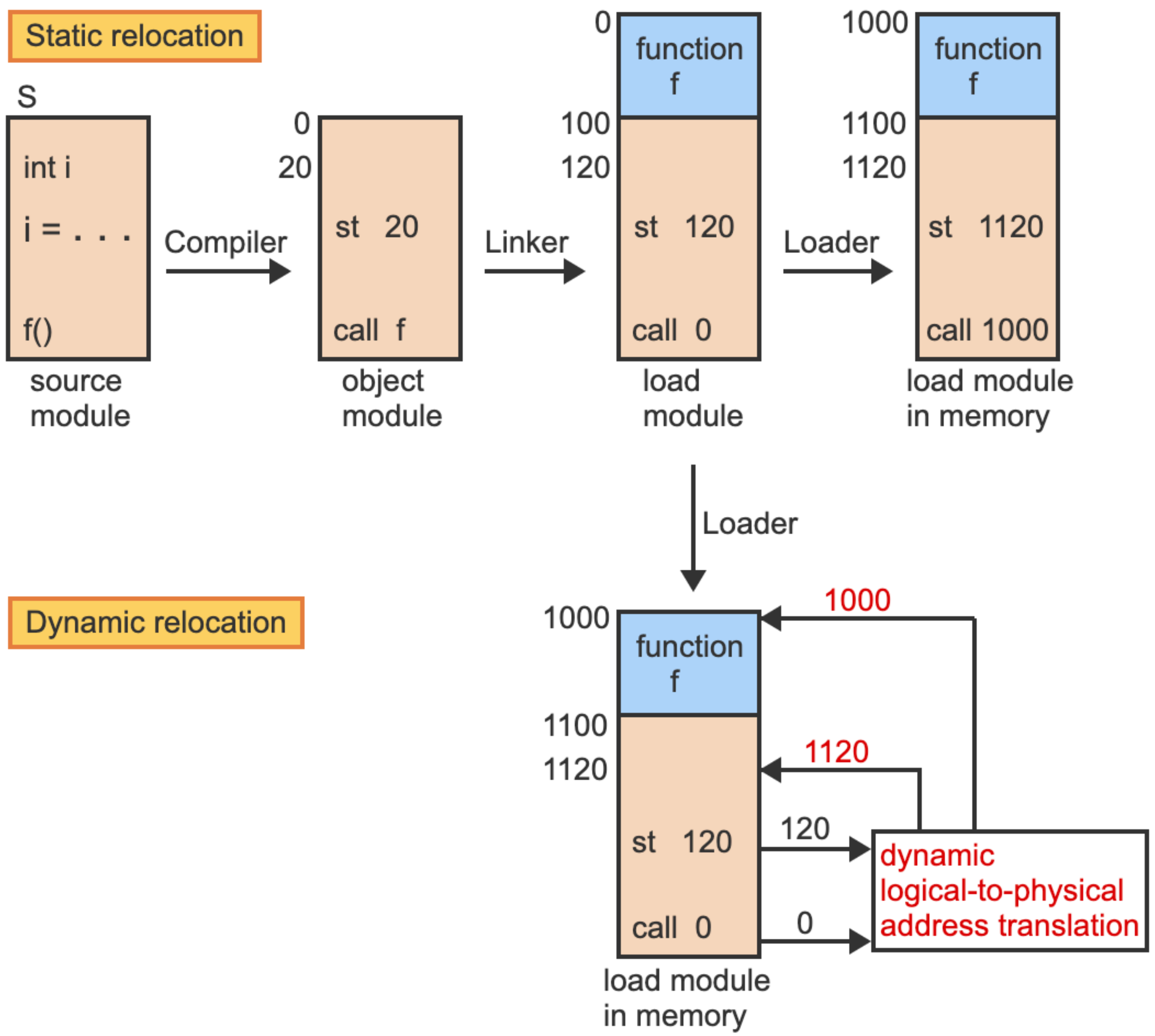



- A program consists of multiple source modules, each written in a high-level language. Module S defines a variable i, which is later assigned a value. Function f resides in a different module.

- The compiler assumes a logical address 0 and assigns i to address 20. The store instruction st then references location 20. The location of f is unknown and is kept as a symbolic external reference.

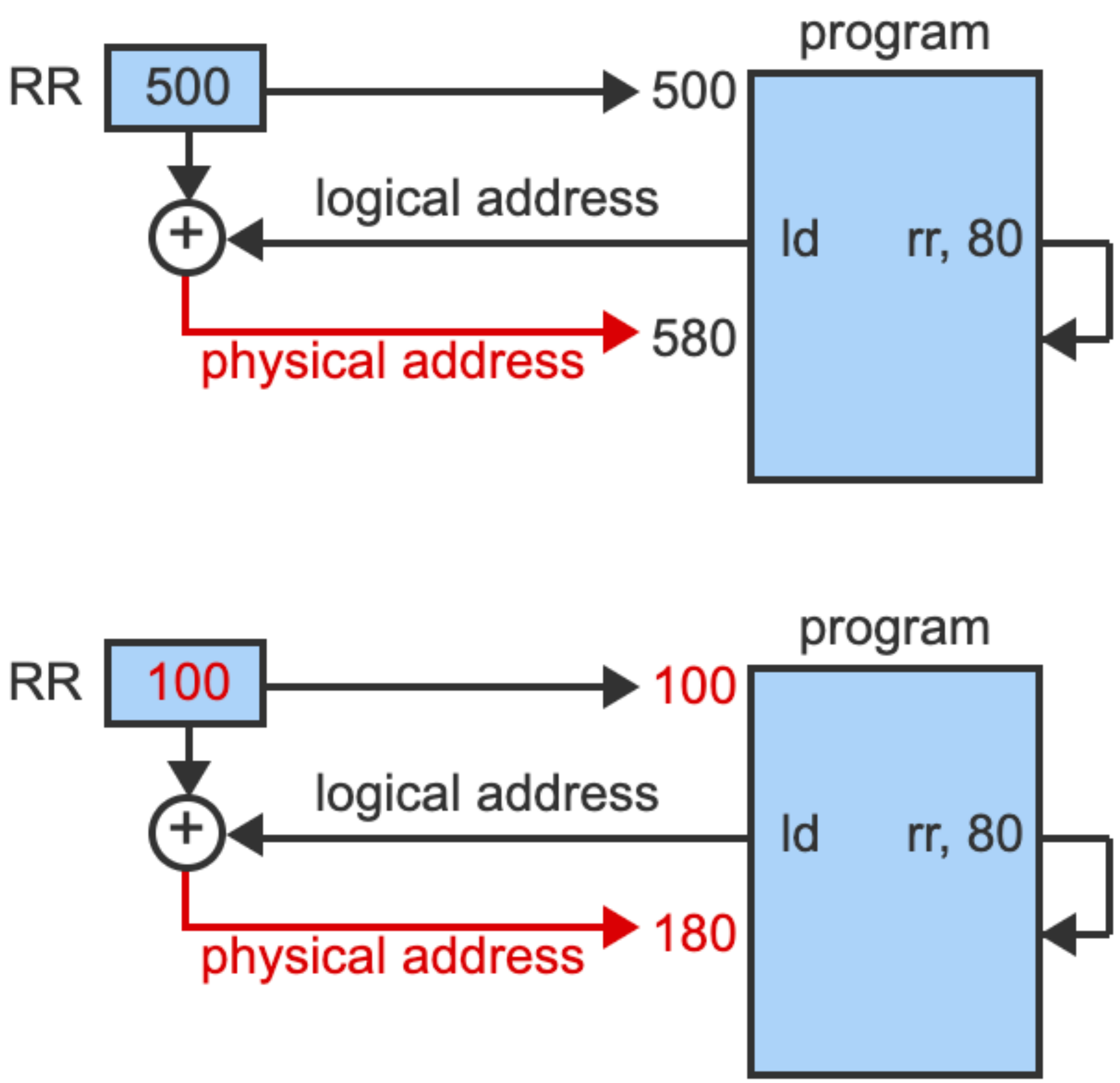
- The linker also assumes a logical starting address of 0 and combines all object modules into one load module. S is relocated to reflect the starting address 100. The call now refers to the start of f.

- **With static relocation**, the loader must adjust all logical addresses to physical addresses when copying the module into memory at the starting physical address of 1000




- With **dynamic** relocation, the load module is copied into memory also starting at address 1000 but without any changes to the addresses in the code
- Relocation is done at runtime by adding the starting address 1000 to every address when referenced

# Dynamic Relocation using Relocation Register




- When a single-module program is loaded into memory, the starting address 500 is loaded into the relocation register RR as the base address for future address calculations

- The load instruction *ld* uses a flag, RR, to indicate that the operand 80 is not an absolute value but a logical address referring to 580

    - The CPU adds 500 to 80, which results in the physical address 580

- When the program is moved to a new area of memory starting at address 100, the code does not change

    - Only RR is loaded with 100, which guarantees that correct physical addresses are generated at runtime

# Apple II+

- June 1979, 44 years ago for $1200 ($5000 today)
- 16KB, 32KB, 48KB of RAM expandable to 64KB using an expansion card

# Single User Contiguous Scheme

- Program was loaded into memory contiguously
- If Program was too big it would not run unless memory was increased.
  - Memory was “expensive back then ($6000 in today’s money) for 16KB.


# Does it fit?

| | |
|---|---|
| Operating System | 10K |
| Program 1 (40K) | 50K |
| Unused Main Memory | |

- If it doesn't fit, the Program could not load or run. The programs of "yesterday" had no concept of Virtual Memory (as everything contained in the program)

# Advantages

- Single User Continuous Scheme Advantages:
  - Jobs are allocated sequentially
    - i.e. Memory Manager is simple
    - i.e. OS has to do less
  - OS checks if there's room
    - If not, job aborted

# Disadvantages

- Single User Continuous Scheme Disadvantages:
  - Only one program at a time
    - Memory cannot be shared
  - Memory wasted
    - If program is small, extra memory lies there doing nothing

# How to fix it?

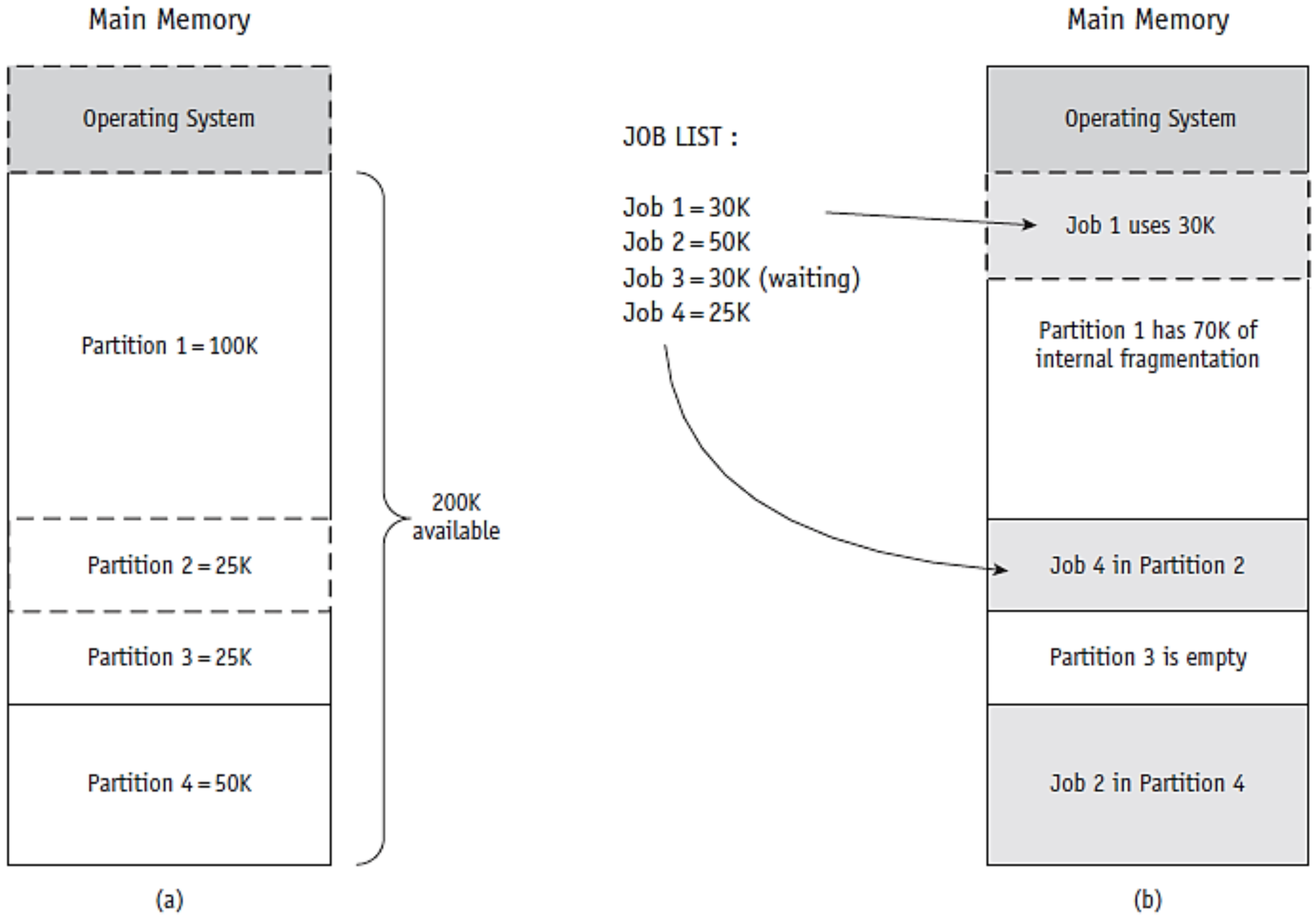
- Fixed Partitions!
- Sys Admins would split memory into partitions
  - Not necessarily same size
  - Probably in accordance to programs that ran
- e.g. 64K -> 1x32K partitions & 1x16K partitions & 2x8K partitions

# How to fix it?

- Fixed Partitions supported multiple jobs

| Partition Size | Memory Address | Access | Partition Status |
|---|---|---|---|
| 100K | 200K | Job 1 | Busy |
| 25K | 300K | Job 4 | Busy |
| 25K | 325K | | Free |
| 50K | 350K | Job 2 | Busy |

# Job 3 is starved



# Advantages/ Disadvantages

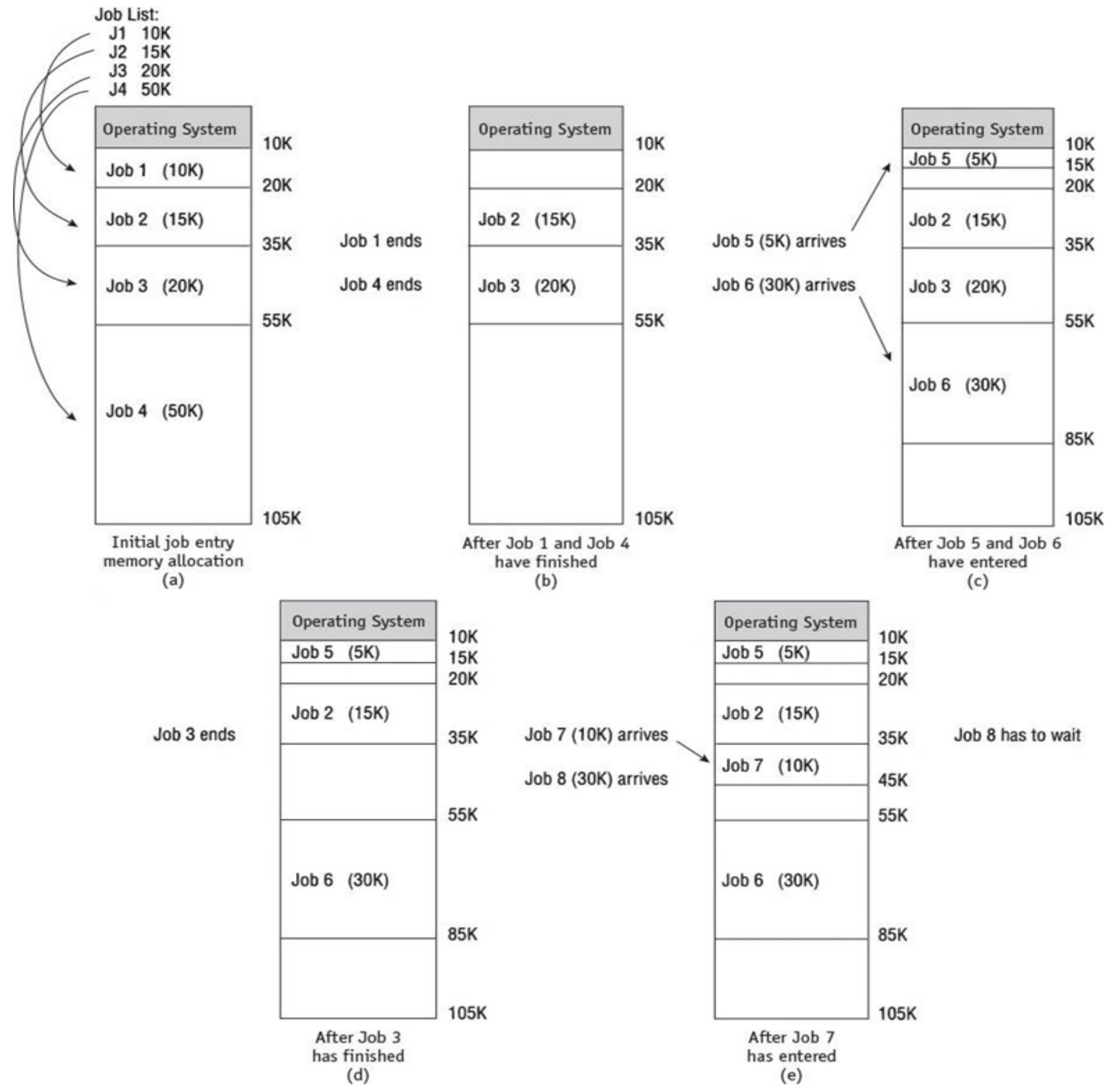
- Fixed Partitions:
    - Advantages
        - Support multi-programming
        - Protected with interfering from each other
    - Disadvantages
        - Partition size may not reflect reality
            - Sysadmin has to constantly keep updating sizes

# Is dynamic better?

# Dynamic Partitions

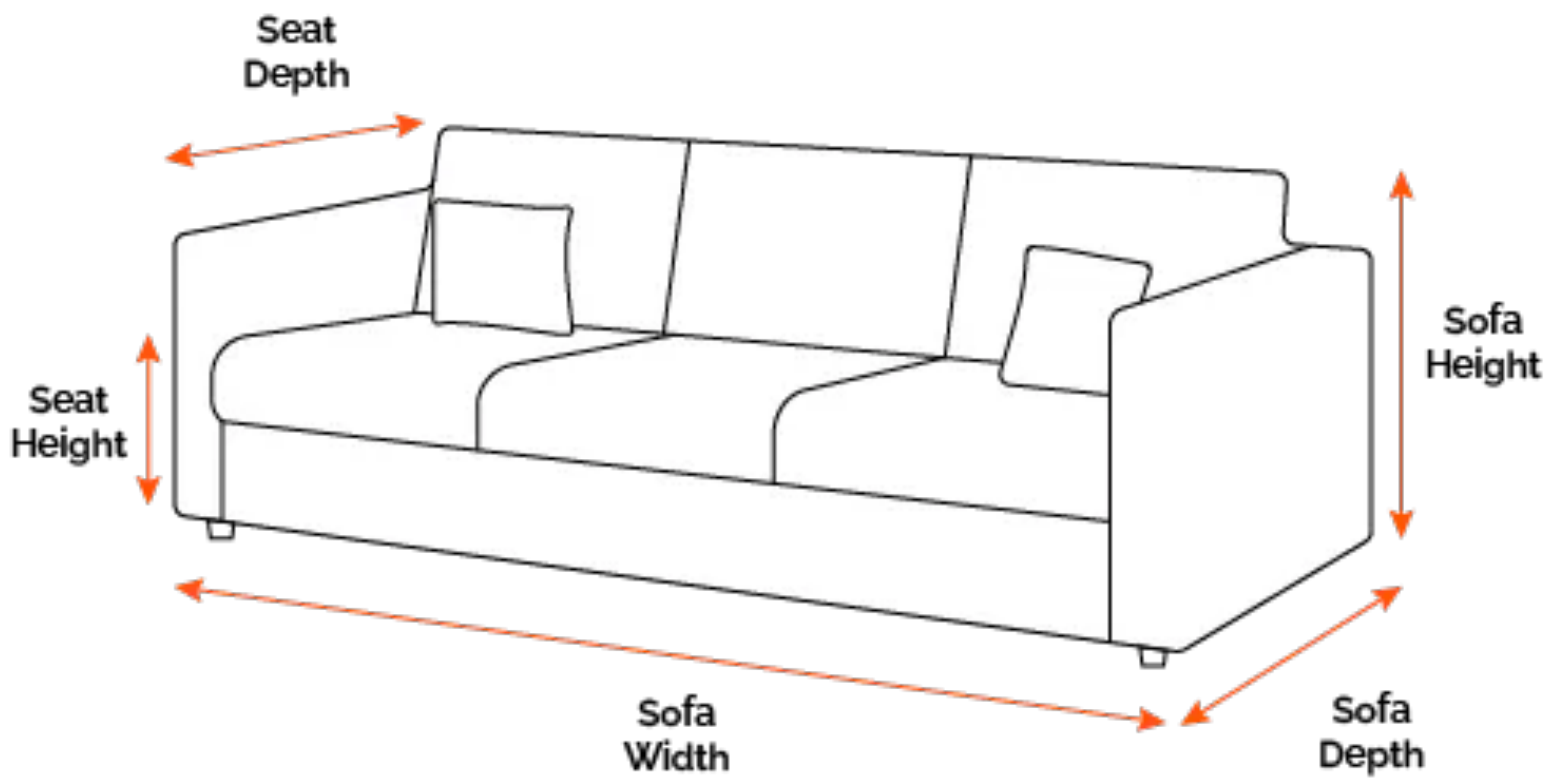
- Main Memory: Partitioned
  - One contiguous partition per job
- Advantages:
  - Better use of memory
- Disadvantages:
  - De-Fragmentation could result into more management

# Dynamic Partitions



# Internal vs External Fragmentation

# My Living Room



# First Fit Allocation

- Using a first-fit scheme,
  - Job 1 claims the first available space
  - Job 2 then claims the first partition large enough to accommodate it,
    - but by doing so it takes the last block large enough to accommodate Job 3
  - Therefore, Job 3 must wait until a large block becomes available,
    - even though there's 75K of unused memory space (internal fragmentation)
- Let's see..

# First Fit Allocation

Job List:

| Job number | Memory requested |
|---|---|
| J1 | 10K |
| J2 | 20K |
| J3 | 30K* |
| J4 | 10K |

Memory List:

| Memory location | Memory block size | Job number | Job size | Status | Internal fragmentation |
|---|---|---|---|---|---|
| 10240 | 30K | J1 | 10K | Busy | 20K |
| 40960 | 15K | J4 | 10K | Busy | 5K |
| 56320 | 50K | J2 | 20K | Busy | 30K |
| 107520 | 20K | | | Free | |
| Total Available: | 115K | Total Used: | 40K | | |

# Best Fit Allocation

Job List:

| Job number | Memory requested |
|---|---|
| J1 | 10K |
| J2 | 20K |
| J3 | 30K |
| J4 | 10K |

Memory List:

| Memory location | Memory block size | Job number | Job size | Status | Internal fragmentation |
|---|---|---|---|---|---|
| 40960 | 15K | J1 | 10K | Busy | 5K |
| 107520 | 20K | J2 | 20K | Busy | None |
| 10240 | 30K | J3 | 30K | Busy | None |
| 56320 | 50K | J4 | 10K | Busy | 40K |
| Total Available: | 115K | Total Used: | 70K | | |

# Next Fit

- Starts searching from last allocated block for next available block

# Worst Fit

- Worst-fit: allocates largest free available block
- Opposite of best-fit
- Why would it be useful though? Why is it even a strategy?
    - 

# Worst Fit

- Worst-fit: allocates largest free available block
- Opposite of best-fit
- Why would it be useful though? Why is it even a strategy?
    - Produces largest possible fragmentation!

# Best Strategy?

- Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

- Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

# Deallocation

- For Fixed Partition System, its simple as status goes from busy to free
- Dynamic is much harder as it tries to combine free areas together and make one free area

# Deallocation

- 3 cases:
  - When block to be deallocated is adjacent to a free block
  - When block is between two free blocks
  - When block to be deallocated is isolated from other blocks

# Joining two free blocks

| Beginning Address | Memory Block Size | Status |
|---|---|---|
| 4075 | 105 | Free |
| 5225 | 5 | Free |
| 6785 | 600 | Free |
| 7560 | 20 | Free |
| (7600) | (200) | (Busy)[1] |
| *7800 | 5 | Free |
| 10250 | 4050 | Free |
| 15125 | 230 | Free |
| 24500 | 1000 | Free |

What do you think will happen after 7600 changes to free?

If you think that it will be 7600 with a block size of 205, you are correct

# Joining three free blocks

| Beginning Address | Memory Block Size | Status |
|---|---|---|
| 4075 | 105 | Free |
| 5225 | 5 | Free |
| 6785 | 600 | Free |
| *7560 | 20 | Free |
| (7580) | (20) | (Busy)[1] |
| *7600 | 205 | Free |
| 10250 | 4050 | Free |
| 15125 | 230 | Free |
| 24500 | 1000 | Free |

THINK CONTIGIOUS

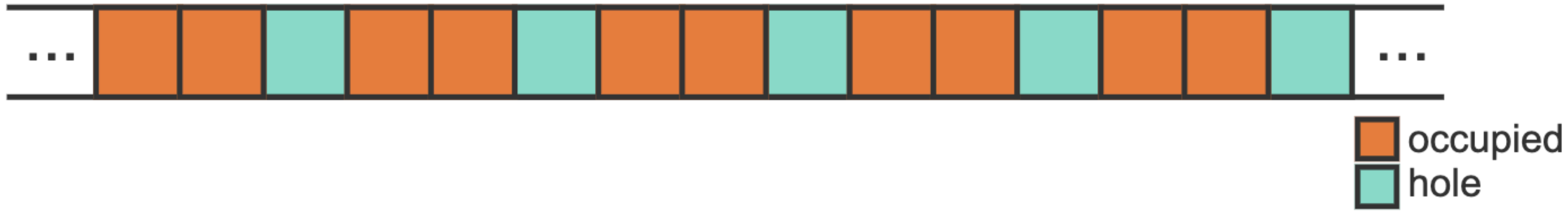
We will now have a Contiguous region at 7560 of 245 bytes

# 50% Rule

# 50% Rule

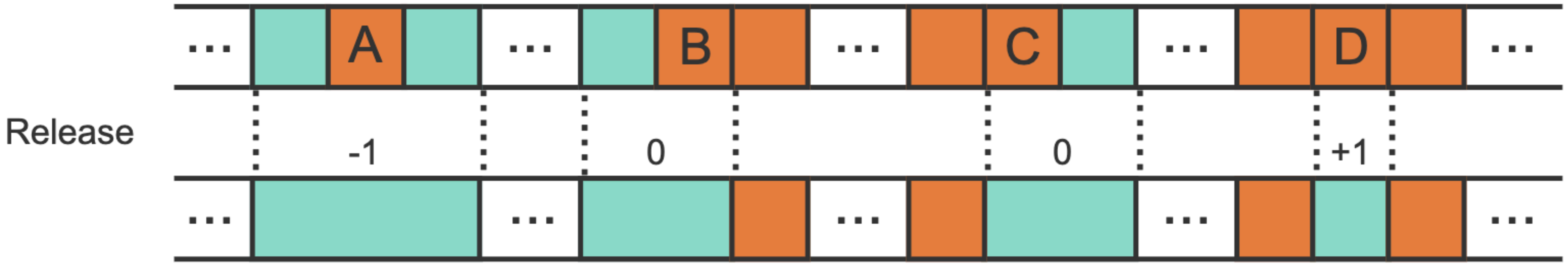
- If the probability of finding an exact match for a request approaches 0, one third of all memory partitions are holes and two thirds are occupied blocks

- Formally, n = 0.5 m, where n is the number of holes and m is the number of occupied blocks




- The 50% rule states that, on average, one hole exists for every 2 occupied blocks. The ratio cannot be changed by any allocation strategy, regardless of the block or hole sizes.

# 50% Rule Proof:

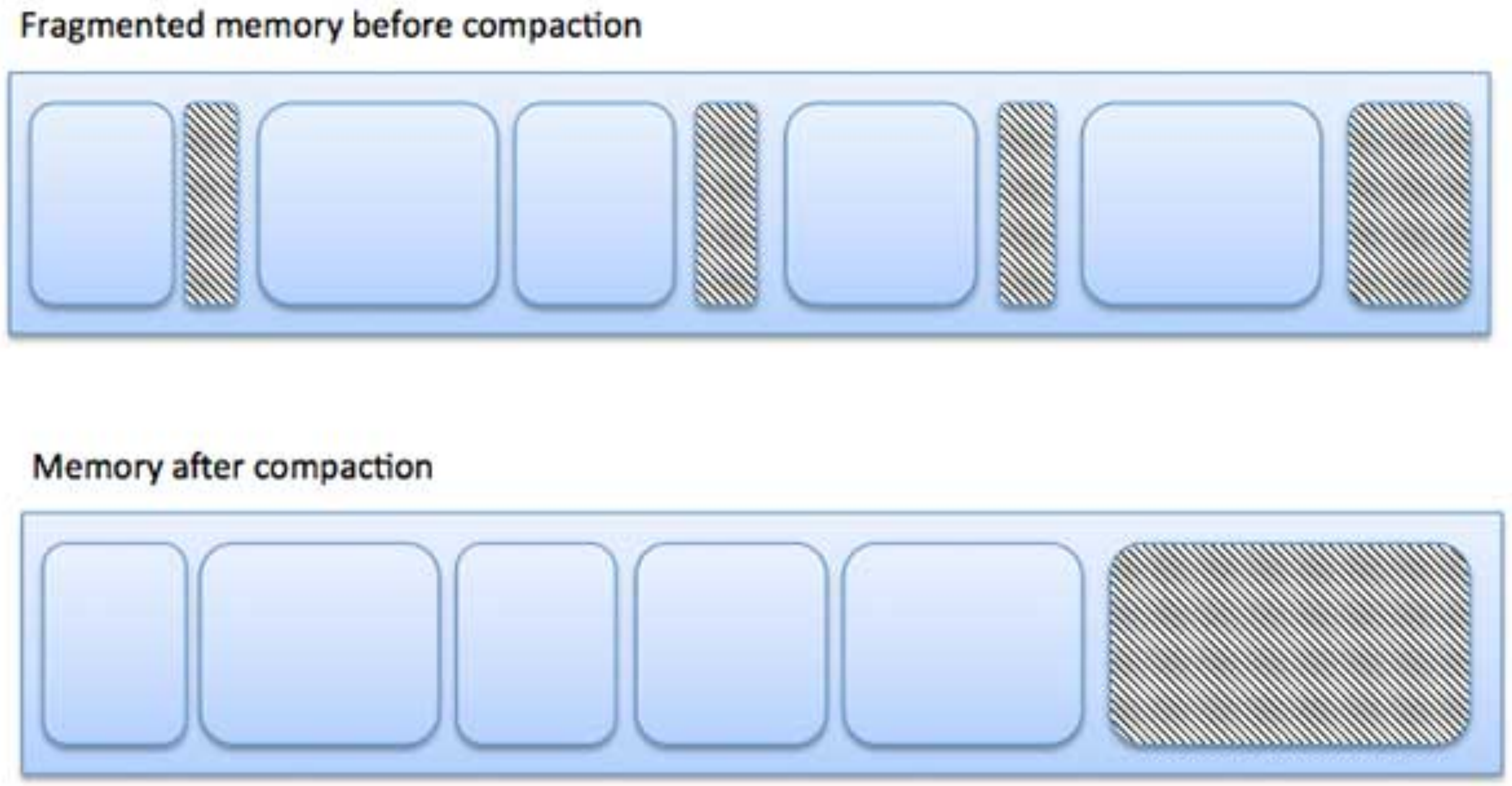


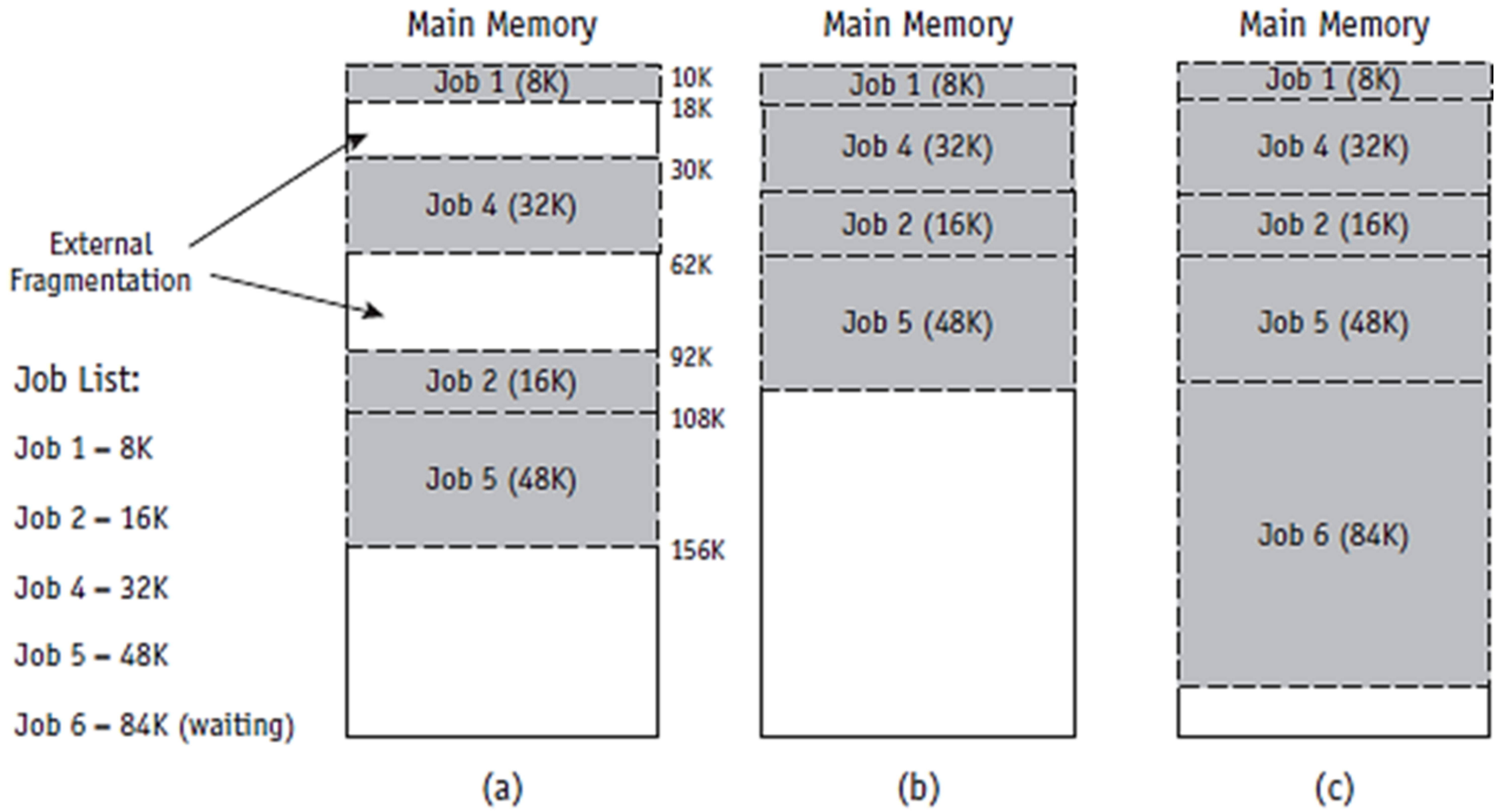

-1 + 0 + 0 + 1 = 0 $\Rightarrow$ Releases do not change # of holes

- 4 types of occupied blocks co-exist in memory:
- A has 2 hole neighbors. B and C have 1 hole neighbor each. D has no hole neighbors.
  - Releasing a block of type A decreases the number of holes by 1 by coalescing the released block with the 2 neighboring holes
  - Releasing a block of type B or C does not change the number of holes. Only the size of the existing hole is increased
  - Releasing a block of type D increases the number of holes by 1 by creating a new hole

# Relocatable Dynamic Partitions

- What if the OS Memory Manager could relocate blocks in order to eliminate fragmentation.
- Concept is similar to disk defragmentation




Main Memory
Job 1 (8K)
Job 4 (32K)
Job 2 (16K)
Job 5 (48K)
10K
18K
30K
62K
92K
108K
156K
External Fragmentation
Job List:
Job 1 – 8K
Job 2 – 16K
Job 4 – 32K
Job 5 – 48K
Job 6 – 84K (waiting)
(a)
Main Memory
Job 1 (8K)
Job 4 (32K)
Job 2 (16K)
Job 5 (48K)
(b)
Main Memory
Job 1 (8K)
Job 4 (32K)
Job 2 (16K)
Job 5 (48K)
Job 6 (84K)
(c)

# Advantages of Compaction

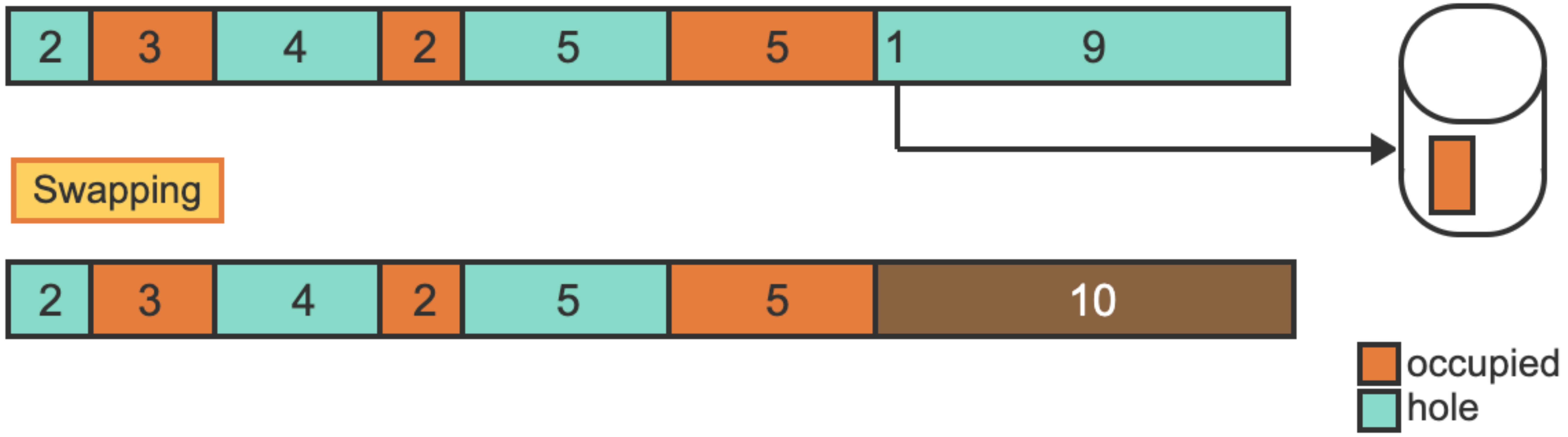
- Compacting and relocating: optimizes memory use
  - Improves throughput
- Compaction: overhead
- Compaction timing options
  - When a certain memory percentage is busy
  - When there are waiting jobs
  - After a prescribed time period has elapsed
- Goal: optimize processing time and memory use; keep overhead as low as possible

# Swapping vs Compaction

- Current State:

- Current State:

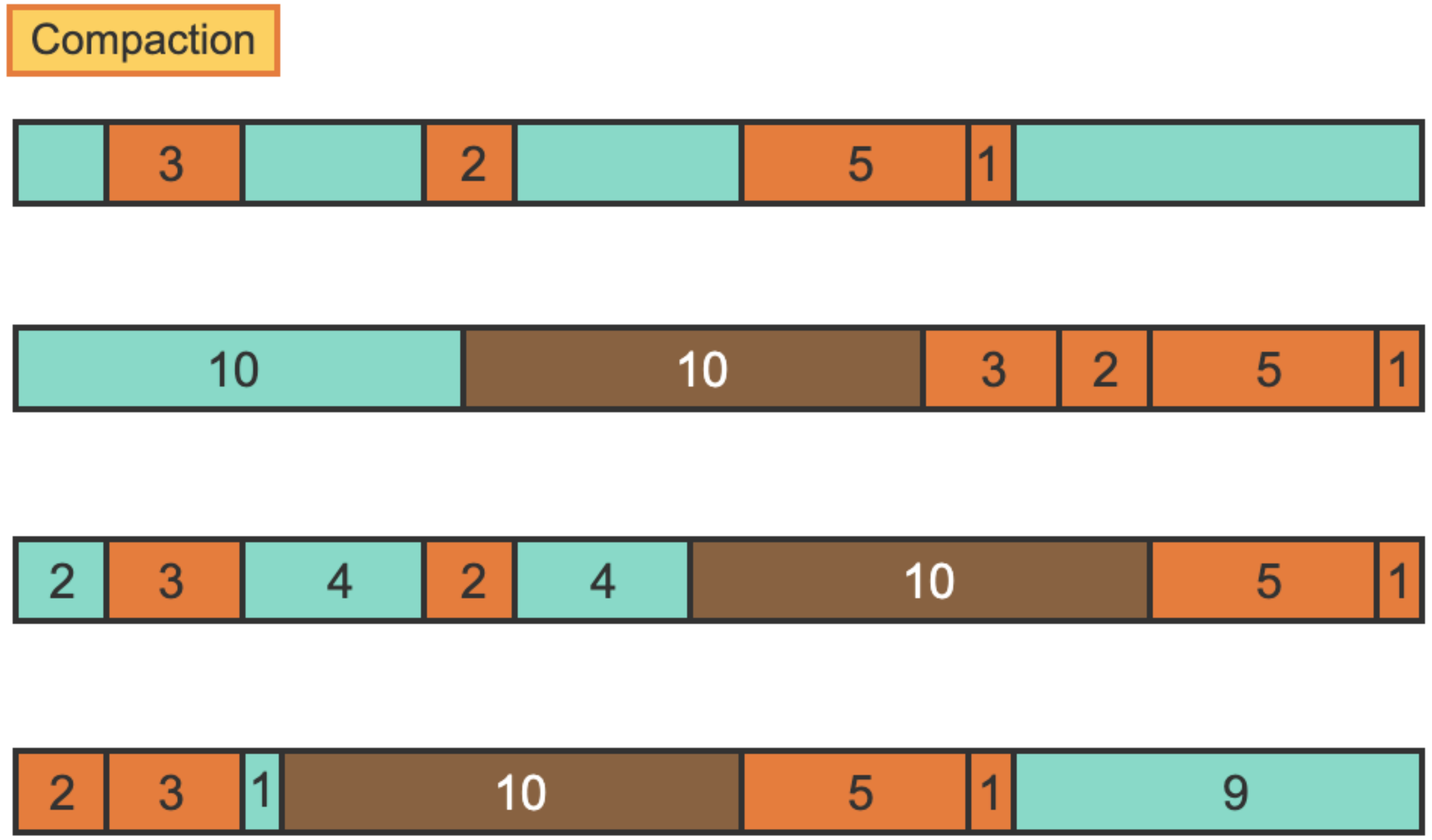

- After Swapping:



- Current State:

- Current State:


- After Compaction:



# Paging

# What is paging?

- Break physical memory into fixed size blocks: frames
- Break logical memory into same size blocks: pages

# Advantages of Paging

- Avoids External Fragmentation
- No need for compaction