Chapters 7 & 8

Memory Management

Part 0: Introduction

Review: Multiprogramming

Multiprogramming: switching Between multiple ready tasks

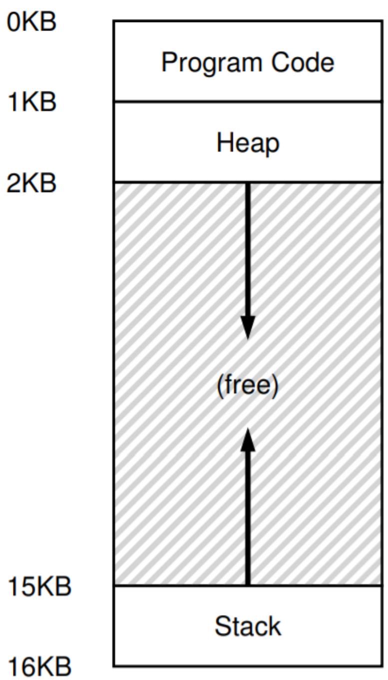
We've discussed:

- Concurrency
- CPU scheduling

And now:

- Memory Management

Review: Memory Model of a Process



Memory:

1. addresses start at ∅
2. both linear and adjacent
3. memory is infinite

<u>The Crux of the Problem:</u> How does the OS provide a private, potentially large address space on top of a single, unified, physical memory?

RAM

Requirements

1. Transparent: process does not know it is there
   memory management
2. Process Isolation/Protection: independent processes can not read or write to each others memory
   getting both is hard
3. Allow for sharing, when wanted
4. Efficiency

A First Attempt

Idea: only the os & the currently running ~~process~~ are in RAM

→ all other processes are swapped out to disk

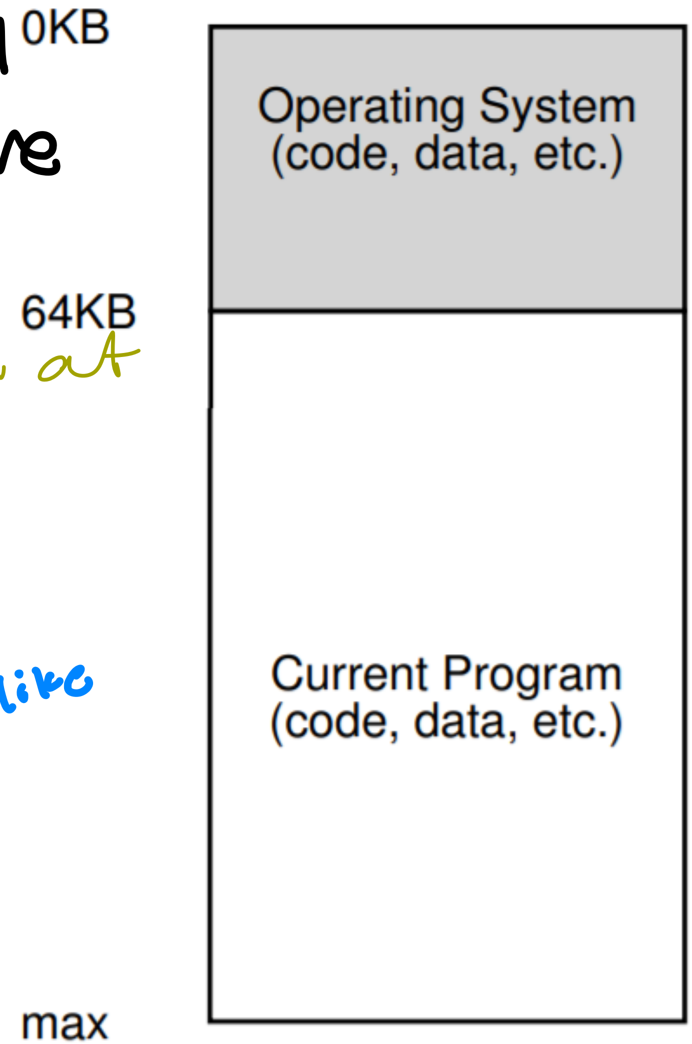
→ multiple processes are in RAM at the same time

meh Transparent? ~ meh.

yes Protection? ✓

no Efficient? X - bc disk is really slow context switch wld be like a mil cycles

no Sharing? X



The Road Ahead

1. Partitioning
2. segmentation
3. Paging

Aside: Every Address you see is Virtual

You've likely printed the address of a variable before…

…this is **never** the *true physical address* in RAM…

=> virtual address : address into the abstraction

```c
int main(int argc, char *argv[]) {
        printf("location of code : %p\n", main);
        printf("location of heap : %p\n", malloc(100e6));
        int x = 3;
        printf("location of stack: %p\n", &x);
        return x;
}
```

Part 1: The Basics & Partitioning

Terms

Address Space: memory locations that can be accessed by the process

Address Translation:

virtual address $\longmapsto$ physical address

mapping

Simple Example

x = x + 3

Becomes:

lw

4

```
128: movl 0x0(%ebx), %eax ;load 0+ebx into eax
```

1

addi

```
132: addl $0x03, %eax ;add 3 to eax register
```

2

sw

```
135: movl %eax, 0x0(%ebx) ;store eax back to mem
```

3 5

5

Memory Partitioning

Basic setup:

1. Divide RAM into "chunks"/partitions
   - static/fixed size
   - dynamic/variable sized
2. load a process into a big enough partition

→ one process per partition
→ one partition per process

Implementing

Two registers:

1. Base Register → stores start addr of the partition
2. Bound /limit → stores the end addr

Address translation:

physical address = virtual address + Base Register

→ e/a proc gets an address space that starts at 0

Translation Example

A process:

- An address space of 4 KB → size → 4096
- Loaded at physical address 16 KB

2^{10}

Virtual Address	Physical Address
0	16 KB
1 KB	17 KB
3000	$16 \cdot 1024 + 3000$ — 19384
4400	FAULT out of Bound

Fixed vs Dynamic

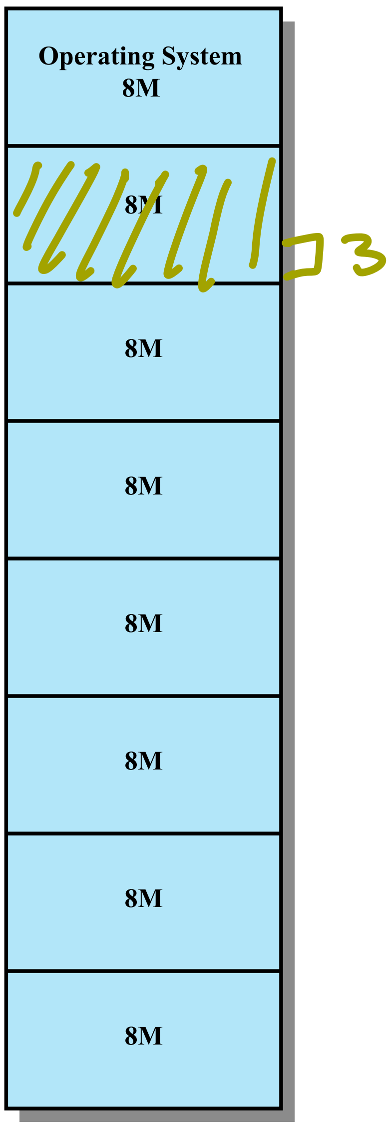
Fixed Partitioning:

- Partitions cannot change in size or #
- May be of the same or diff size

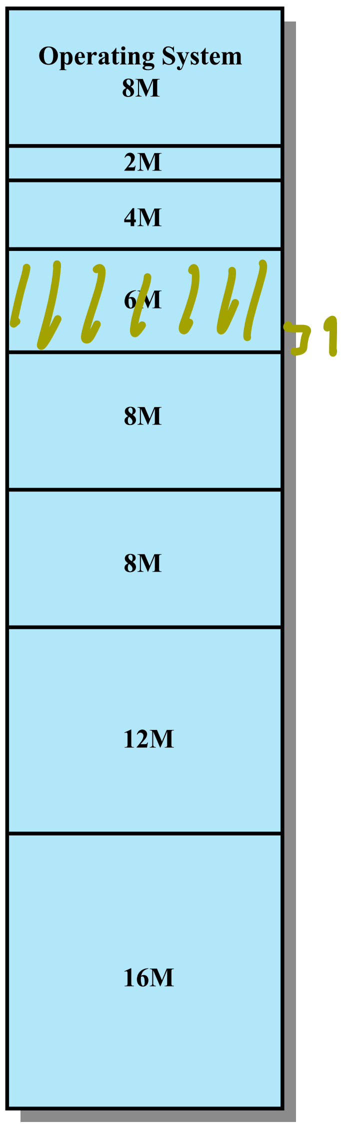
Dynamic Partitioning:

- Variable size and # of partitions
- Partitions are made on allocation requests

Example: Fixed Partitioning



(a) Equal-size partitions



(b) Unequal-size partitions

Allocate a **5**MB process:

a. Into (a)

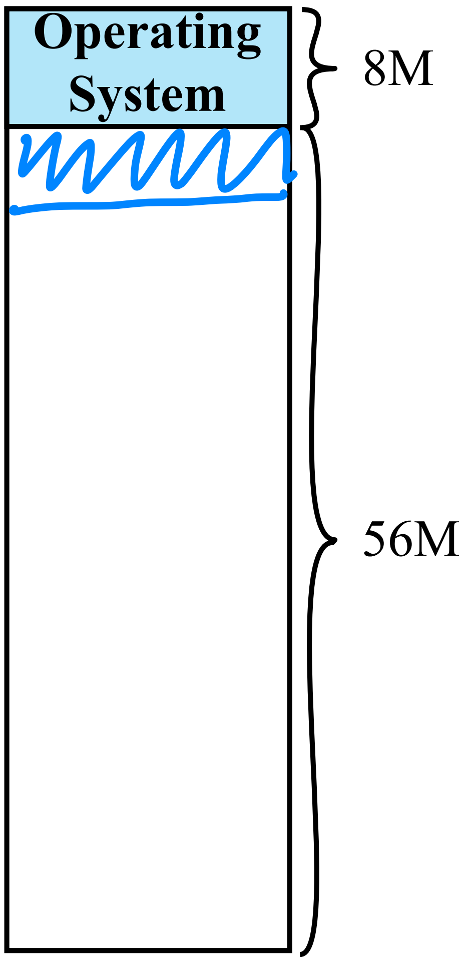
→ 3 MB wasted due to over allocation

b. Into (b)

→ 1 MB wasted

Internal fragmentation

Example: Dynamic Partitioning



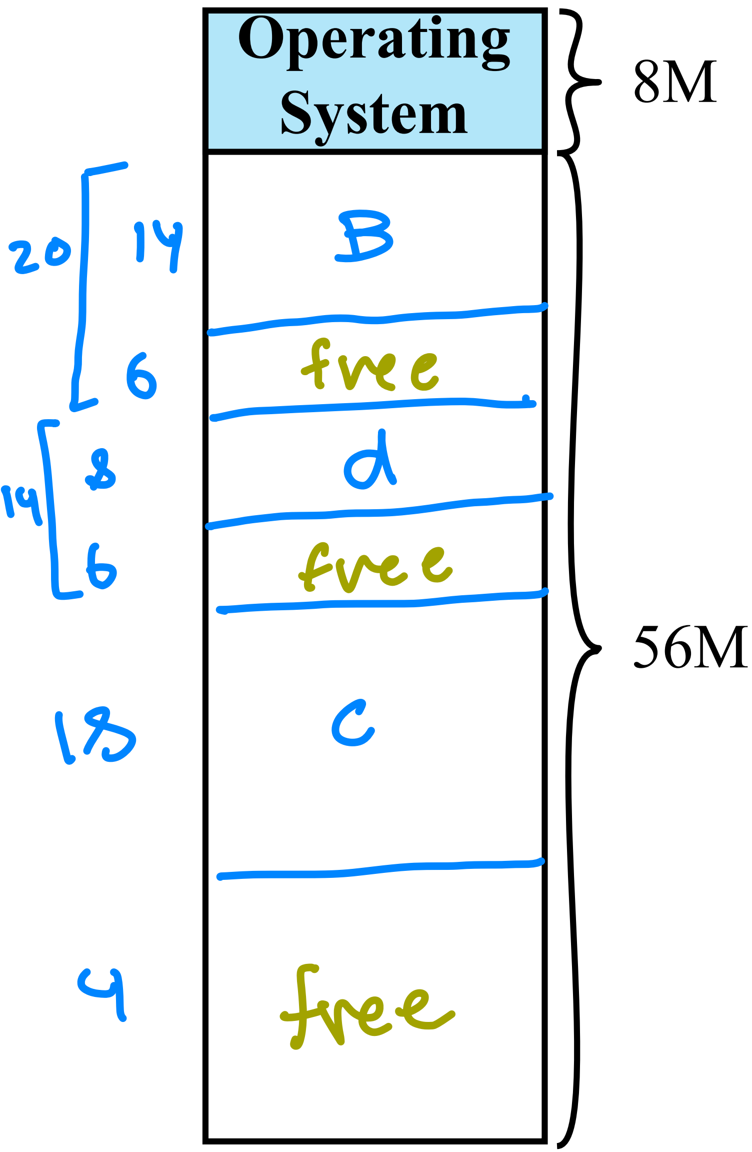
(a)

Allocate a **5**MB process.

→ the exact size needed

→ no internal fragmentation

Why Dynamic Partitioning Isn't So Great

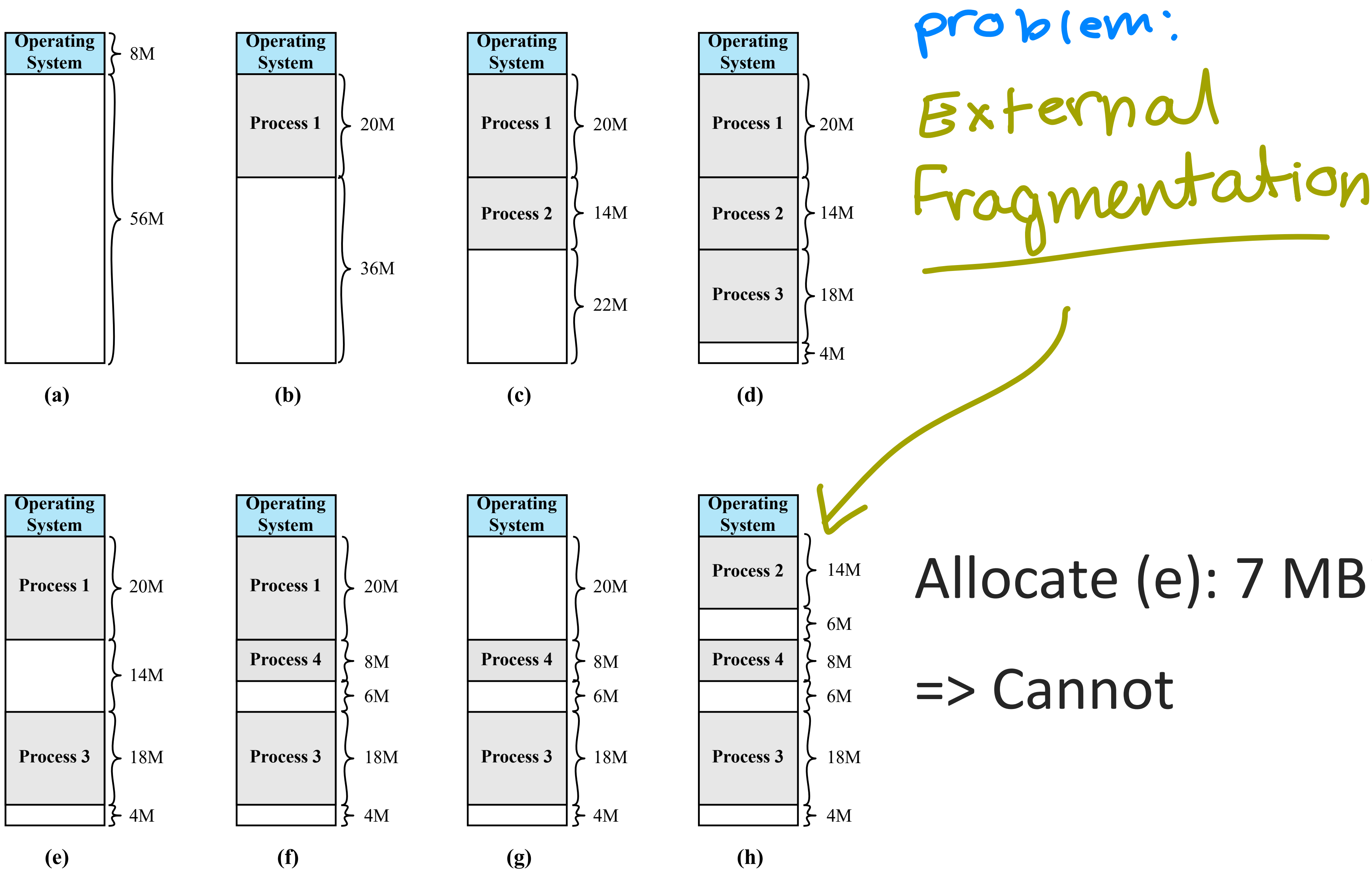


1. Allocate (a): 20 MB
2. Allocate (b): 14 MB
3. Allocate (c): 18 MB
4. Free (b)
5. Allocate (d): 8 MB
6. Free (a)
7. Re-allocate (b): 14 MB
8. Allocate (e): 7 MB

end up w/ holes in mem

no space!

Computer Graphics Version



Allocate (e): 7 MB?

=> Cannot

More Terms

Placement Algorithm: where in RAM to allocate a request

Internal Fragmentation: Memory w/ in a partition is wasted

External Fragmentation: memory between partions is wasted

solutions:

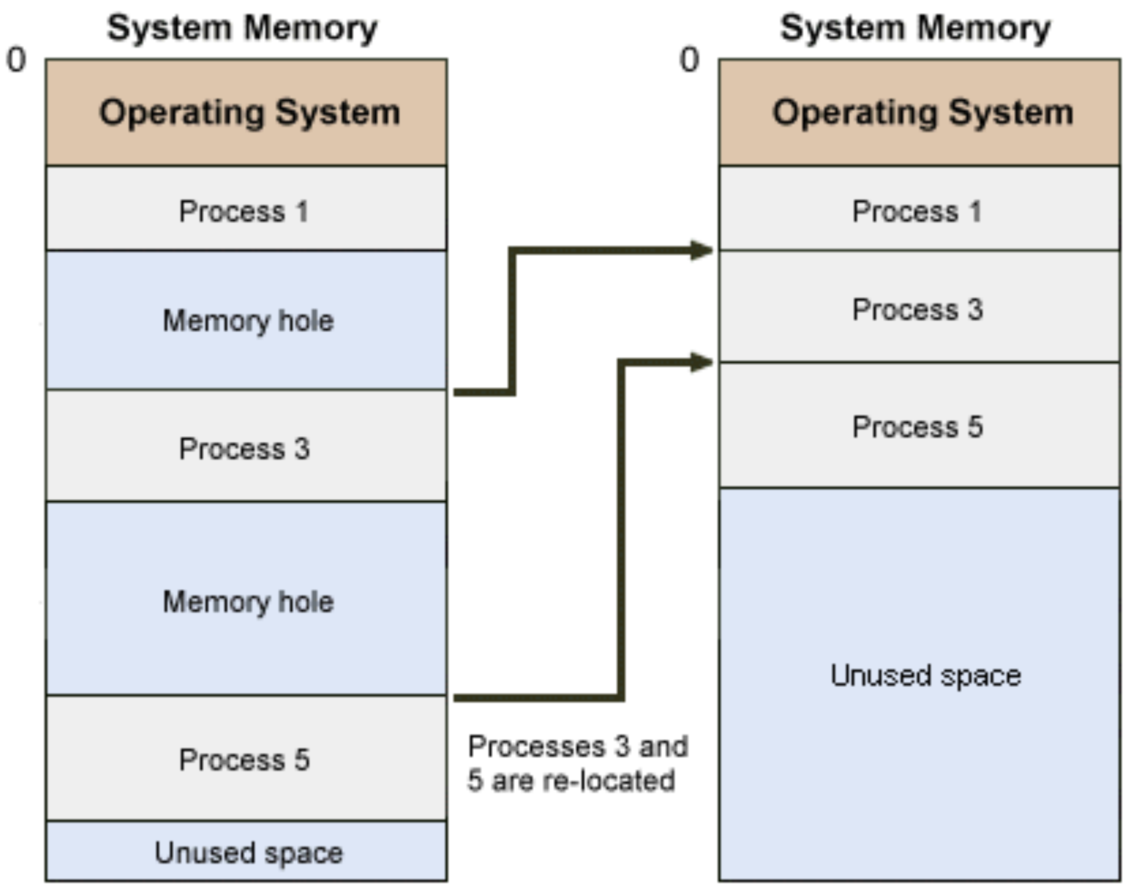
a) compaction

b) use a different placement algorithm

Compaction Example

Compaction: move partitions in physical mem to make them contiguous

$$$



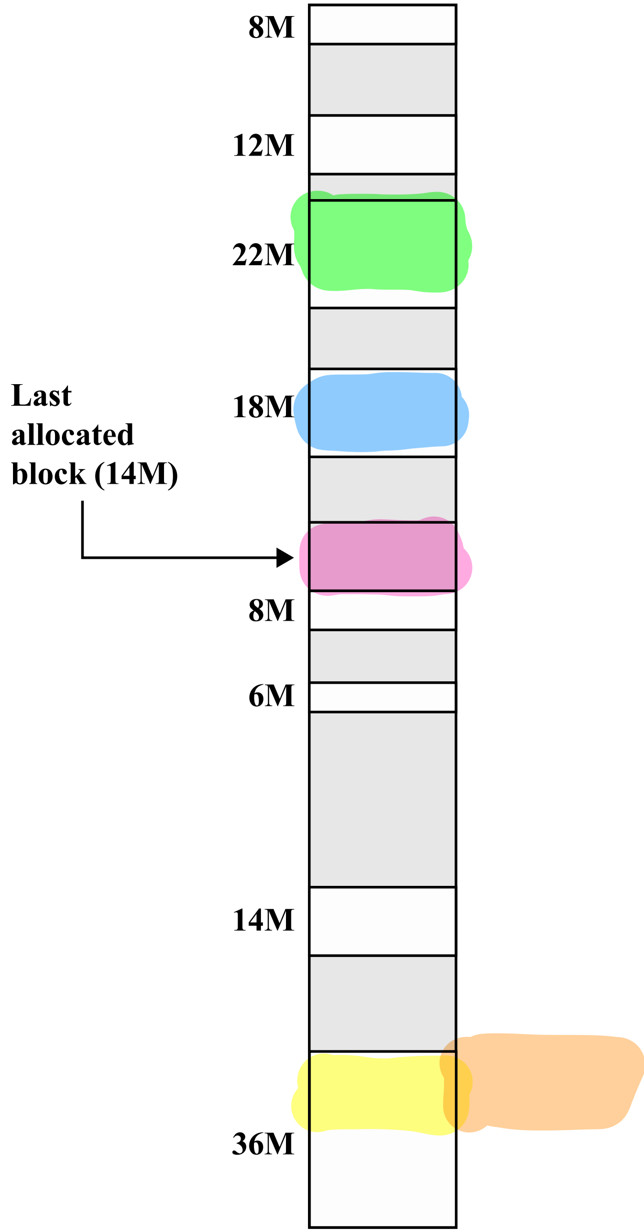
← slow

~only 10,000s of cycles

Basic Placement Schemes

1. Best fit: choose the block closest in size
2. worst fit: choose largest block
3. First fit: chooses first block that is big enough
4. Next fit: Choose the next big enough block

Placement Example



Allocate a 16M using:

a) Best Fit

b) Worst Fit

c) First Fit

d) Next Fit

does the same thing