Agenda

Review: Processes and VAS

Memory management
- No abstraction
- Address Spaces
- Swapping

Managing free memory
- Bitmaps
- Free lists

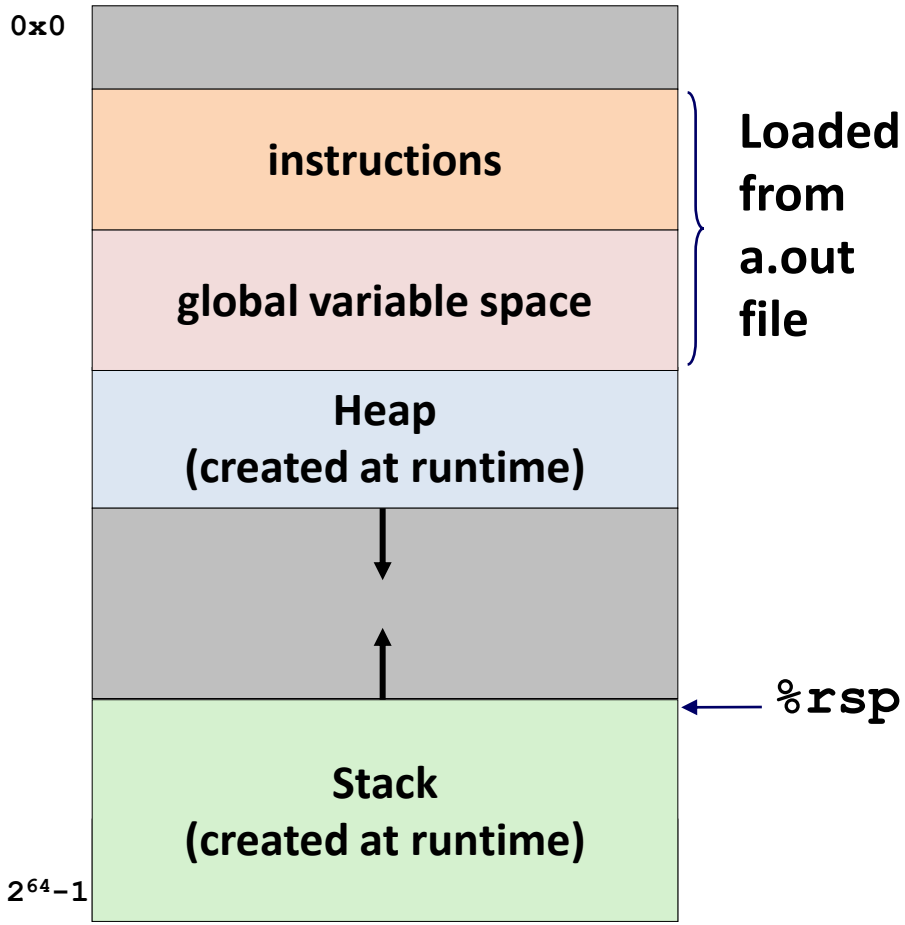
Allocation Strategies

Recall: Processes and their Virtual Address Space

A process is an abstraction for running programs

Each process has a “lone view” of the operating system, including its memory.

From the process’s point of view, its memory is large and contiguous and completely independent from other processes



Process’s virtual (logical) Address Space

OS Memory Management

OS/Processes use **main memory** (RAM) to run

Different from disk memory, where we store persistent data

Ideally programmers want memory that is

- large
- fast
- non-volatile
- cheap

Problem: There's no single type of memory that is both high capacity and high performance

Solution: Memory hierarchy

Memory Hierarchy

The **memory hierarchy** arranges memory in terms of performance and capacity

- This layered arrangement helps leverage the advantages of different types of memory while maintaining performance

The OS hides the details of memory management from the user

Aside: Normally we do not need to know the details of how the OS manages memory under-the-hood but there is a one key mistake that programmers should always avoid. Can you name it?

Memory Hierarchy

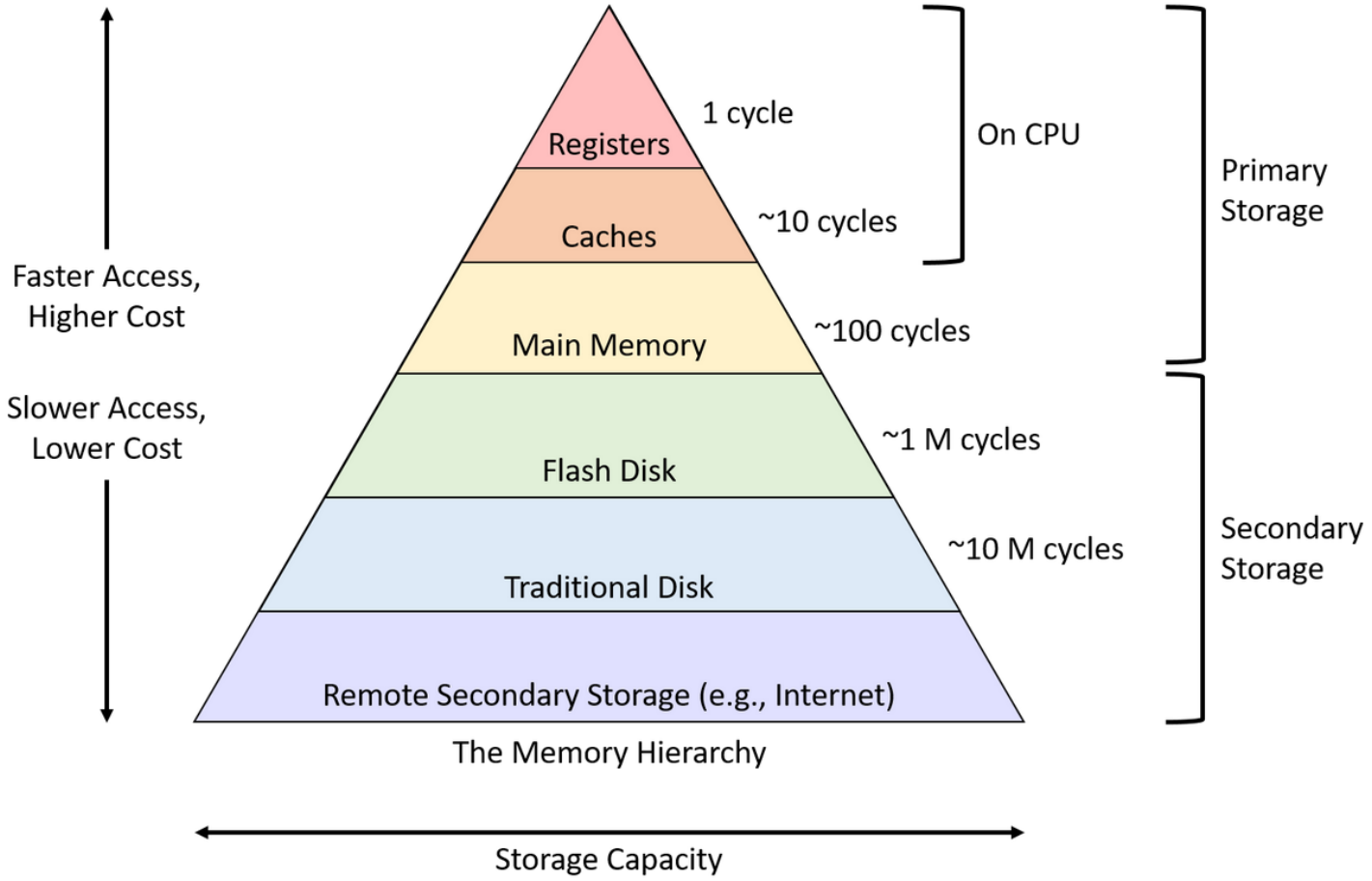


Figure 1. The memory hierarchy

From *Dive into Systems*

Memory Hierarchy: Analogy with books

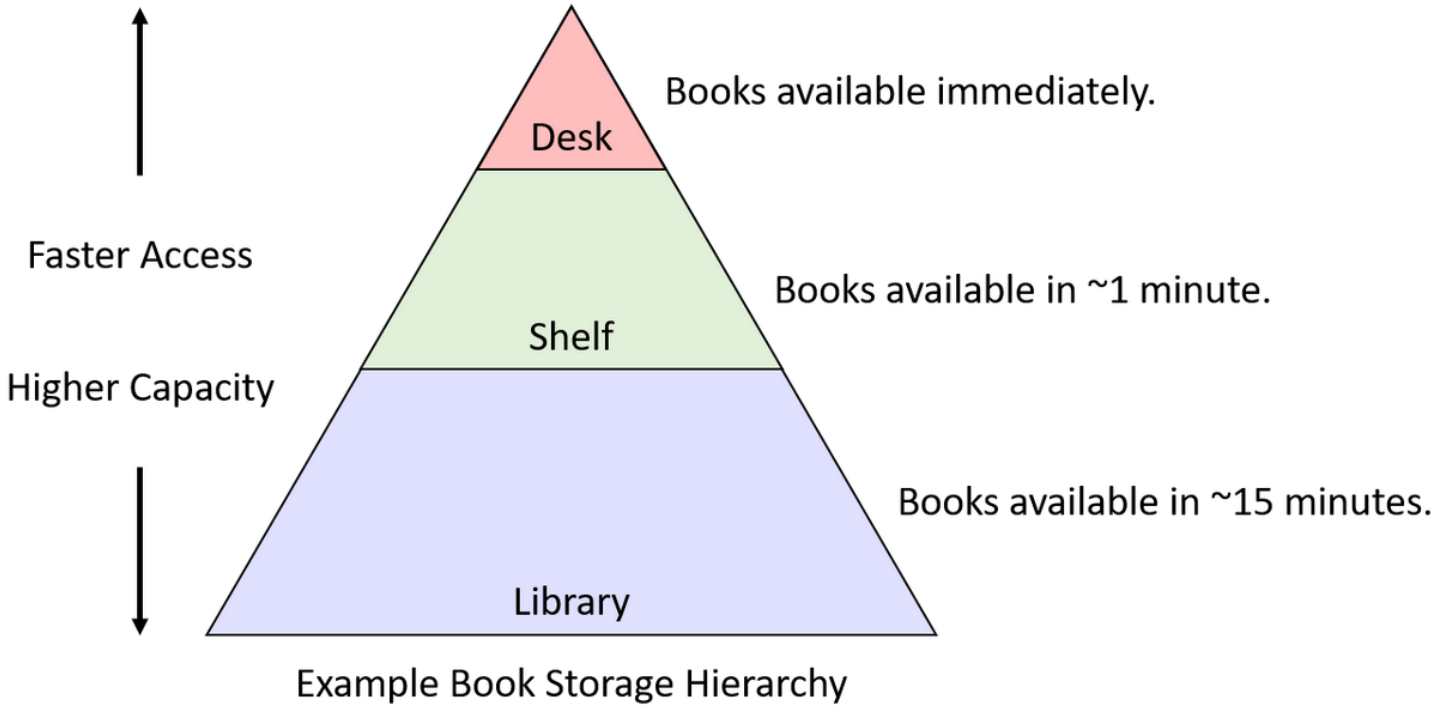


Figure 1. A hypothetical book storage hierarchy

From *Dive into Systems*

OS Memory Models

OS maintains the programmer's view of memory. Several possible approaches

1. No memory abstraction (simplest)
2. Address Spaces
3. Virtual memory

No memory abstraction (simplest)

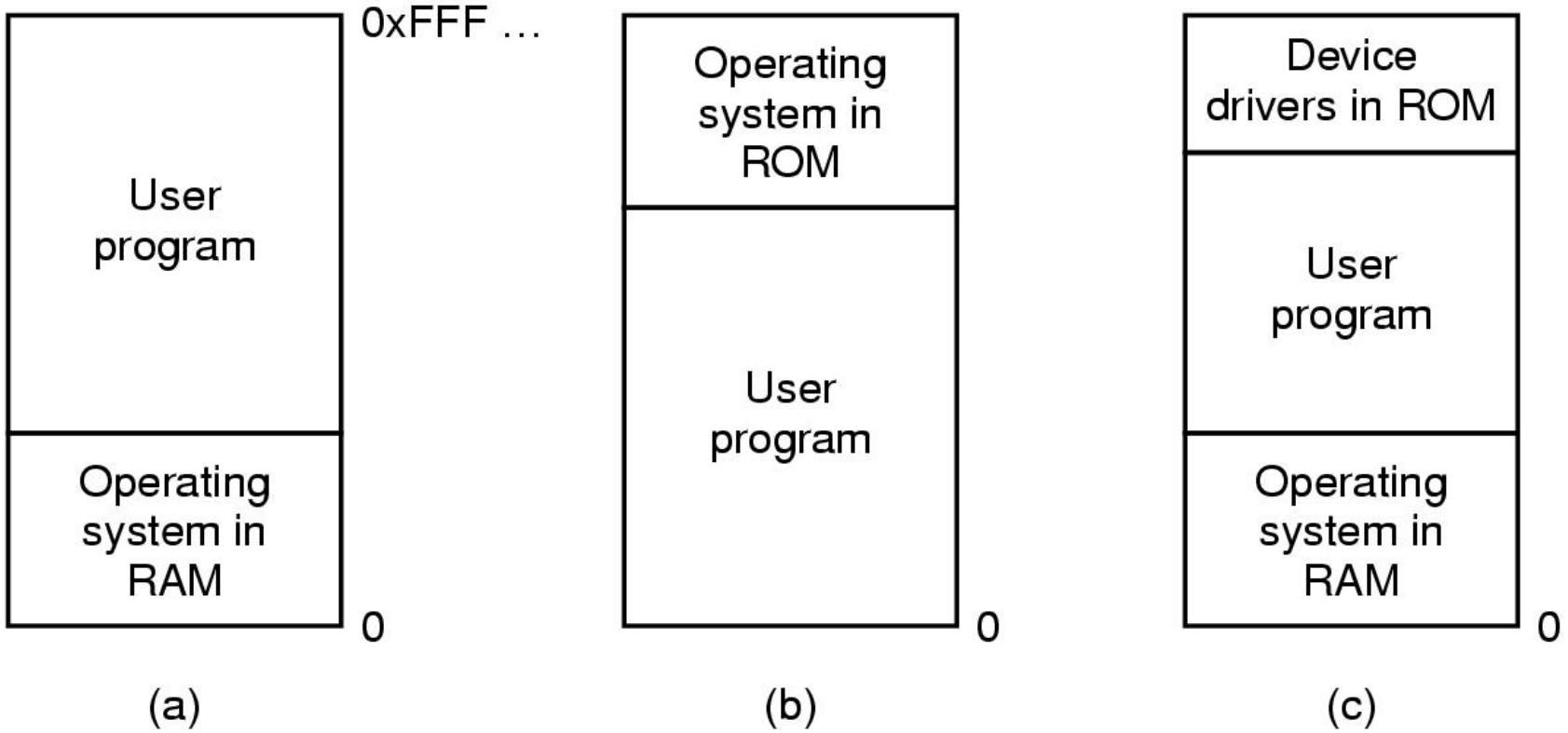
Idea: Programs reference physical memory directly

Where does the OS go?

Where does the process go?

Usual solution: OS starts at one end and user programs start at the other

No memory abstraction (simplest)



Memory management only needs to segment memory between the OS and the user process. What could possibly go wrong?

What if we want to run more than one program?

Swapping: Move programs in/out of memory when they are running (slow)

Sharing memory: Allow multiple programs to coexist in memory

- Two concerns:
  - Protection: how to protect programs from each other (bad writes)
  - Relocation: how should programs reference physical memory?

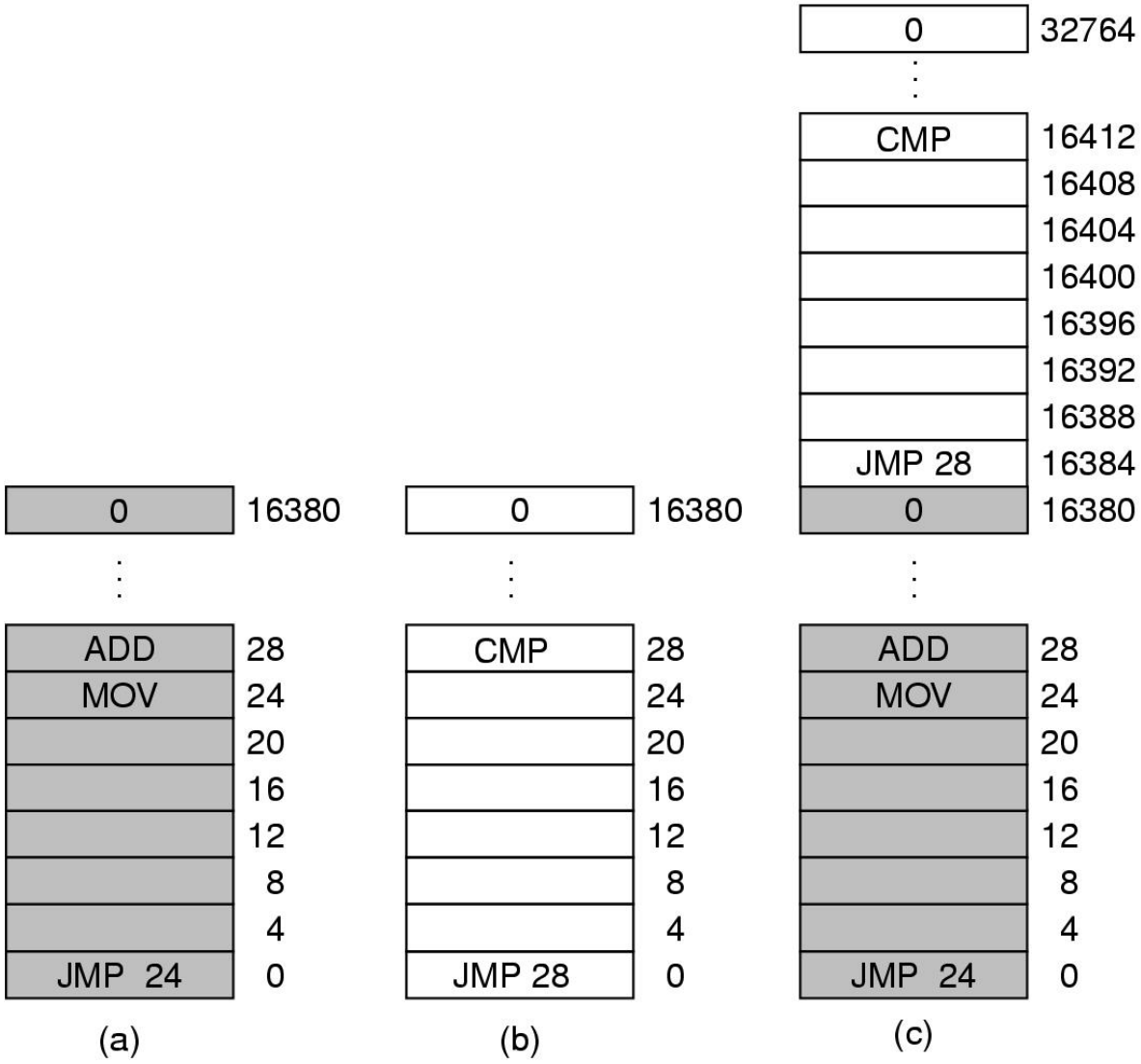
Example: Multiple Programs (No Memory Abstraction)

Imagine we have two programs

Each program has an IF statement (JMP)

How do we compute the address for the JMP command?

Do we modify the assembly based on its location in memory?



No memory abstraction: Protection and relocation

Protection: Use keys to mark which memory belongs to each process

- PSW contains the key (4 bits)
- 4-bit registers associated with blocks of memory
- If the 4-bit register does not match the key, trigger interrupt to process

Relocation: When loading a process into memory, modify addresses based on location of process in memory. This process is called **static relocation**

- Example: JMP 28 -> JMP 16412
- Can be complicated (requires per instruction rules)
- Can slow down loading programs

Memory Abstraction #1: Address Spaces

Limitations of accessing physical memory directly:

- Bad writes can trash the OS or other programs
- Supporting multiple programs is complicated and/or slow

An **address space** is a set of addresses that a process can use to address memory

Process address spaces are independent from each other

Address spaces get mapped to physical memory dynamically, as the program executes. This approach is called **dynamic relocation**.

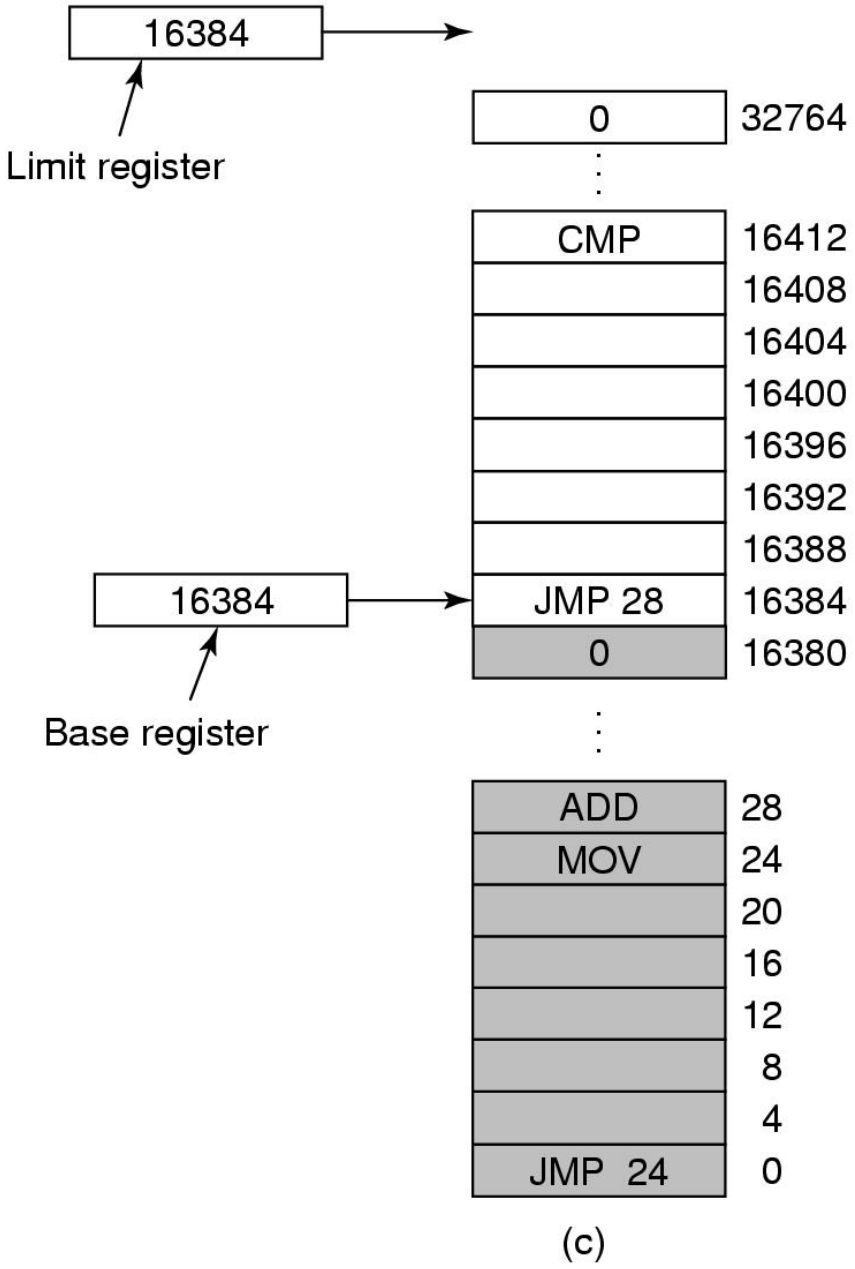
Example: Address Spaces implemented with base and limit registers

Idea: Store the base pointer for the program and treat all addresses in assembly as offsets

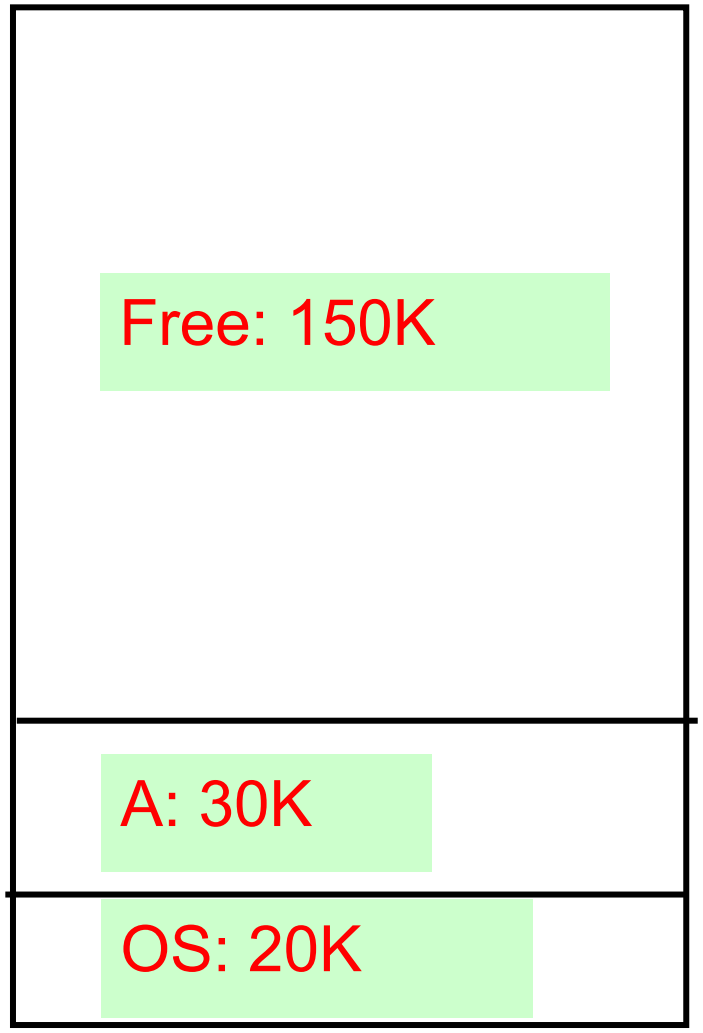
Example: If the base register has value 16384, what is the target address of JMP 28?

This technique maps compile-time (**virtual, or logical**) addresses to run-time (**physical**) addresses.

Use the limit register to test for invalid read/writes



Swapping



What do we do if physical memory isn't large enough to hold all our running processes?

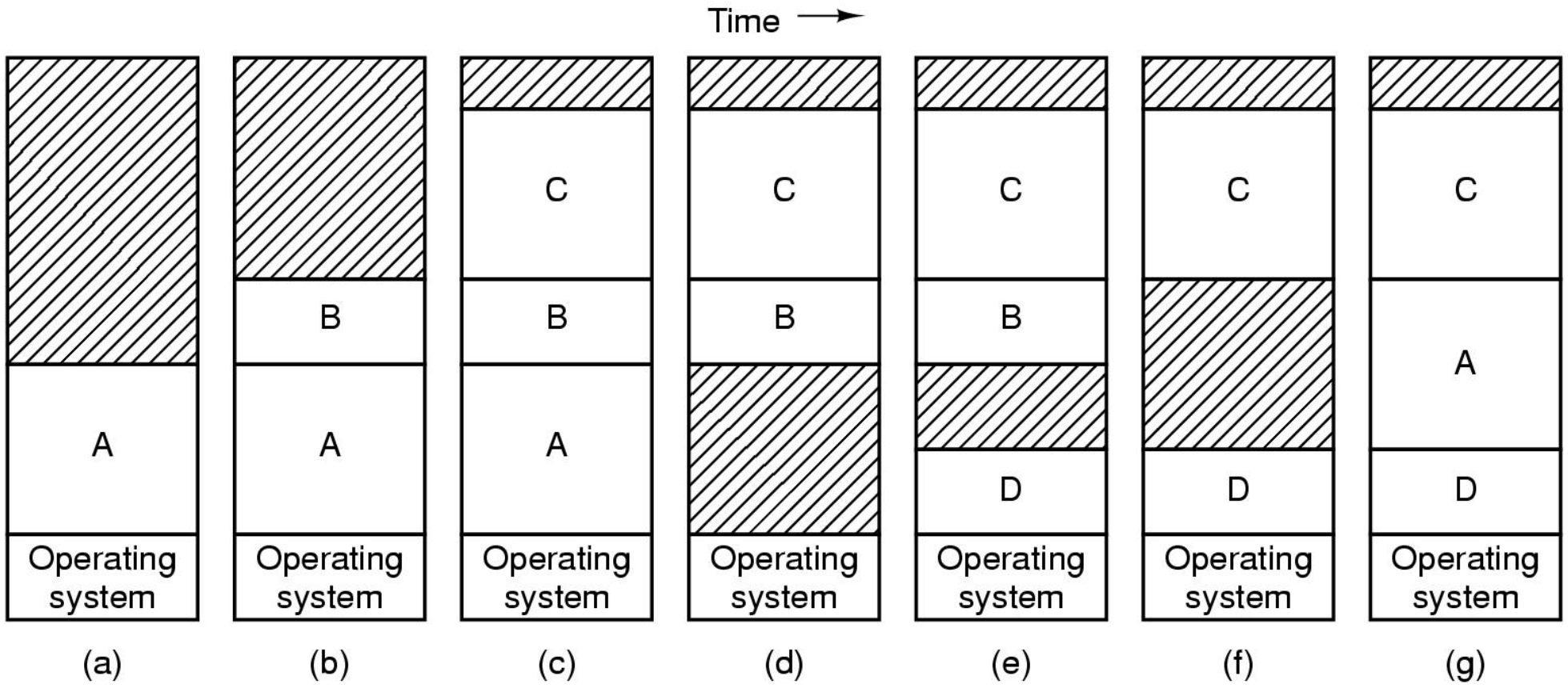
Idea: Swap programs in/out to run. The entire process is brought into main memory when running and written back out to the disk as needed.

If we run out of memory

- Processes will need to be moved to a hole with sufficient space, or
- Swapped out of memory until a big enough hole opens up

Swapper decides which processes should be in main memory

Example: Swapping



Memory allocation changes as
- processes come into memory
- leave memory

Shaded regions are unused memory. Memory can be compacted (slow)

Example: Swapping

- New process C requests 20K
- New process E requests 20K
- C exits

Swapping

Need free space for dynamic allocation of memory (heaps) within the space allocated to a process

- stack grows downwards and heap grows upwards, with fixed space for compiled code

OS must keep track of memory that is free

- Bitmaps (arrays)
- Linked lists

Classical tradeoffs: space required vs time for (de)allocation

Managing Free Space: Bitmap

A **bitmap** is a block of bits

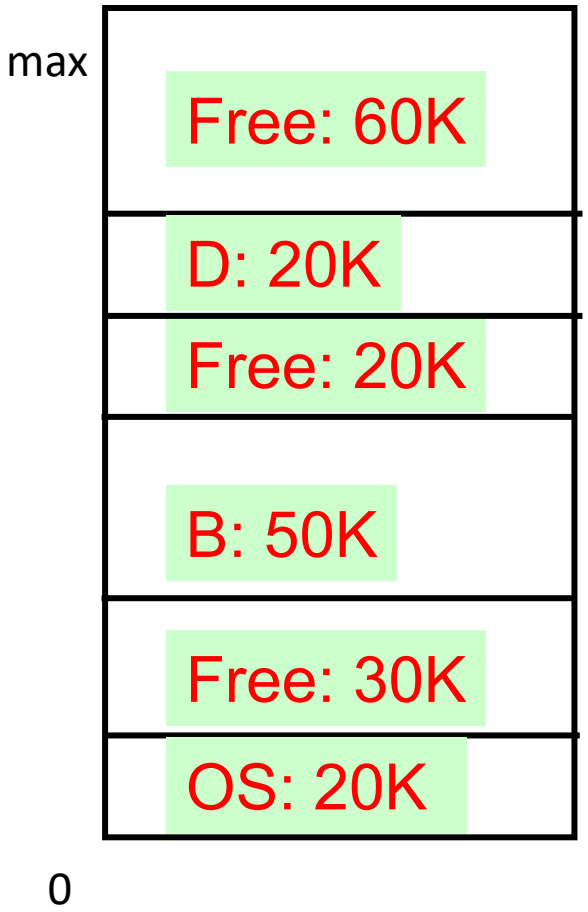
Idea: Let's segment memory into chunks of equal size. For each chunk, a bit is 1 if the memory is in use; 0 otherwise

- e.g maintain a vector bit field where the i-th bit tells whether the i-th chunk is free

Example: Bitmap

Suppose we segment memory into 10K chunks.
Give a bitmap for this block of memory

Managing Free Space: Linked Lists



Idea: Use a linked list, sorted by address, to indicate blocks of free/used memory

Each record has

- Process ID/ Free (H: hole)
- Start location
- Size
- Pointer to Next record

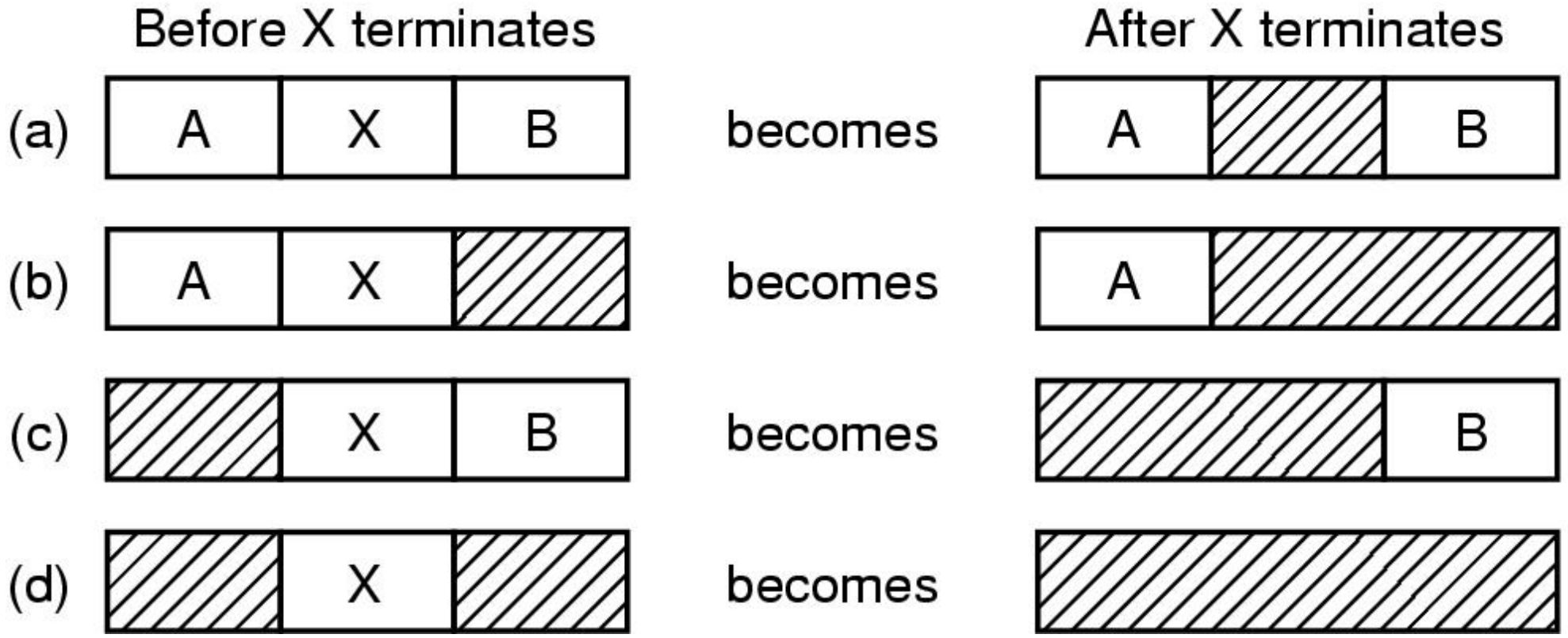
Example: Linked List

Suppose we have 10K chunks. What is the linked list corresponding to this memory?

Managing Free Space: When a process terminates

We can combine neighboring empty regions when we delete

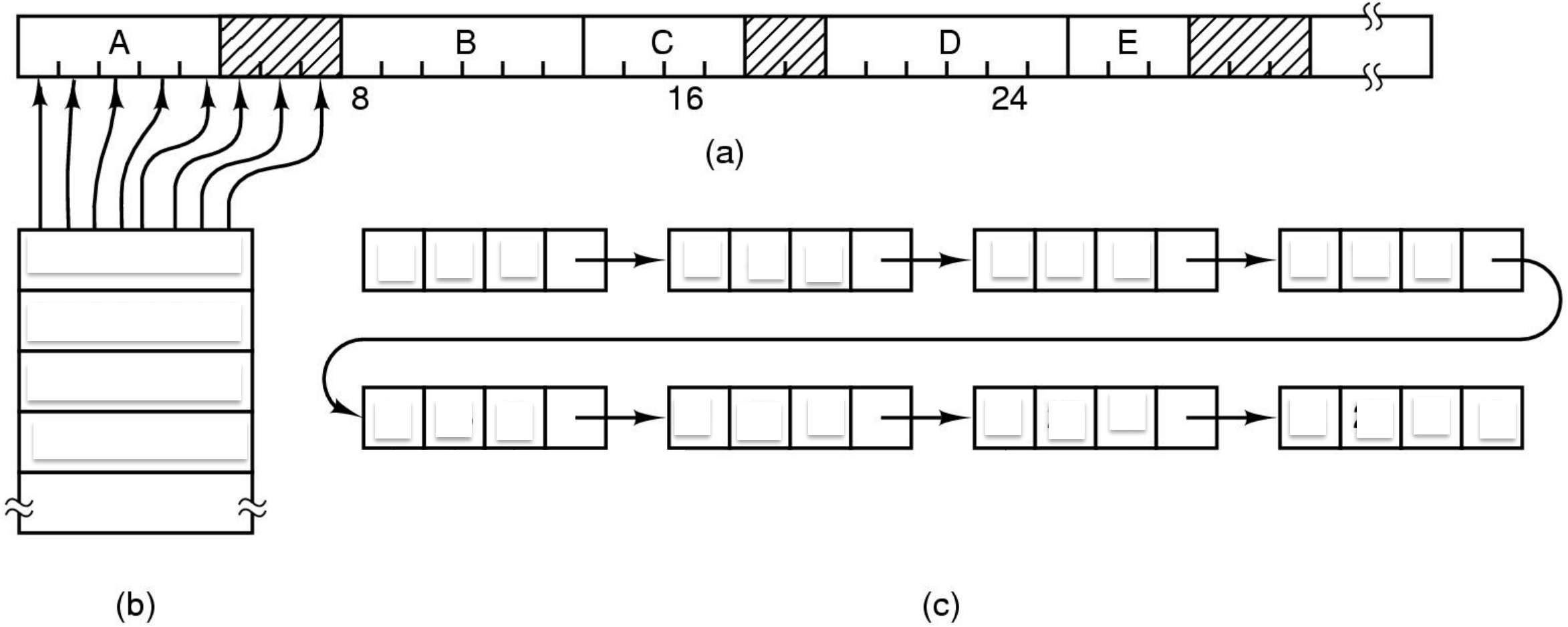
Four neighbor possibilities for the terminating process X



Doubly-linked list can help here...

Exercise: Managing free memory
Fill in the bitmap and list values

Part of memory with 5 processes, 3 holes
- tick marks show allocation units
- shaded regions are free



Managing free space

Bitmaps

- Chunk size is important (too big and we waste memory; too small and the bitmap is too large)
- Finding contiguous free space can be expensive

Linked list

- Space efficient
- If the list is sorted by address, combining free memory is easier
- If the list is sorted by size, finding open spots is easier

Allocation Strategies

Problem: When we create a new process, where should we put it?

Different strategies:

First fit: Find first free chunk big enough
Best fit: Find best fitting hole
Worst fit: Find largest hole
Quick fit: Use multiple lists with common allocation sizes

Challenges:

- Fragmentation occurs as memory gets cut into small, non-contiguous chunks that cannot be used
- Searching the list for free space needs to be fast
- Combining free chunks should also be fast

Example: Allocation Strategy

Suppose a new process requests 15K, which hole should it use?

- First-fit:
- Best-fit:
- Worst-fit:

Best-fit vs. First-fit vs. Worst-fit

All can leave many small and useless holes.

To shorten search time for First-Fit, start the next search at the next hole following the previously selected hole – called **Next-Fit**.

Separate lists for holes and processes can make searching for holes faster. Holes can be sorted by size to F-F and B-F faster.

In practice, F-F is usually better than B-F, and F-F and B-F are better than W-F.

Example

Assume holes of 20K and 15K, requests for 12K followed by 16K

First-fit

Best-fit

Example: Allocation strategies

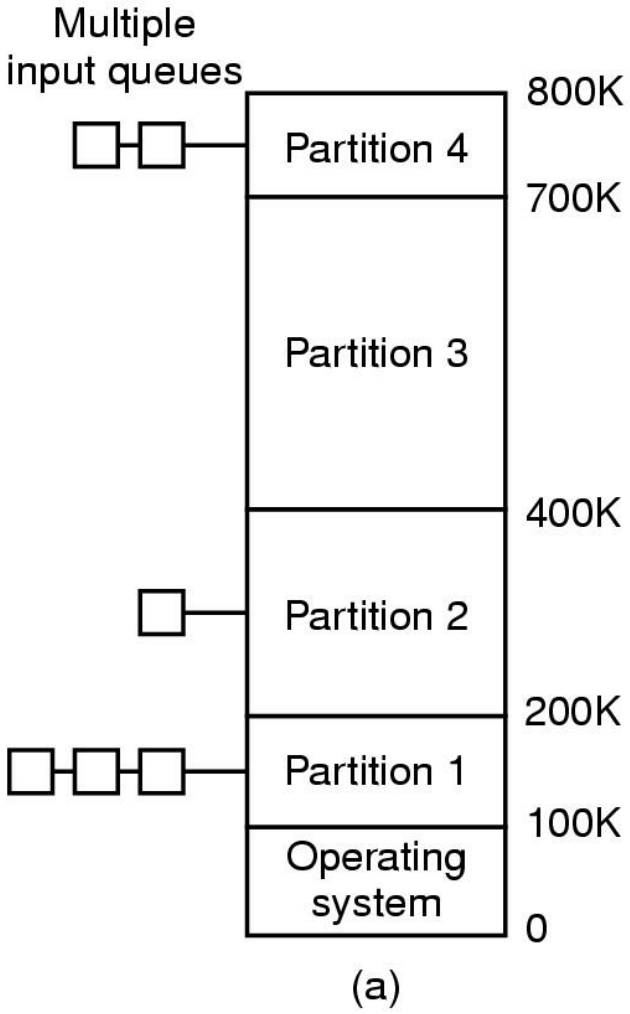
Assume holes of 20K and 15K, requests for 12K, followed by 14K, and 7K

First-fit

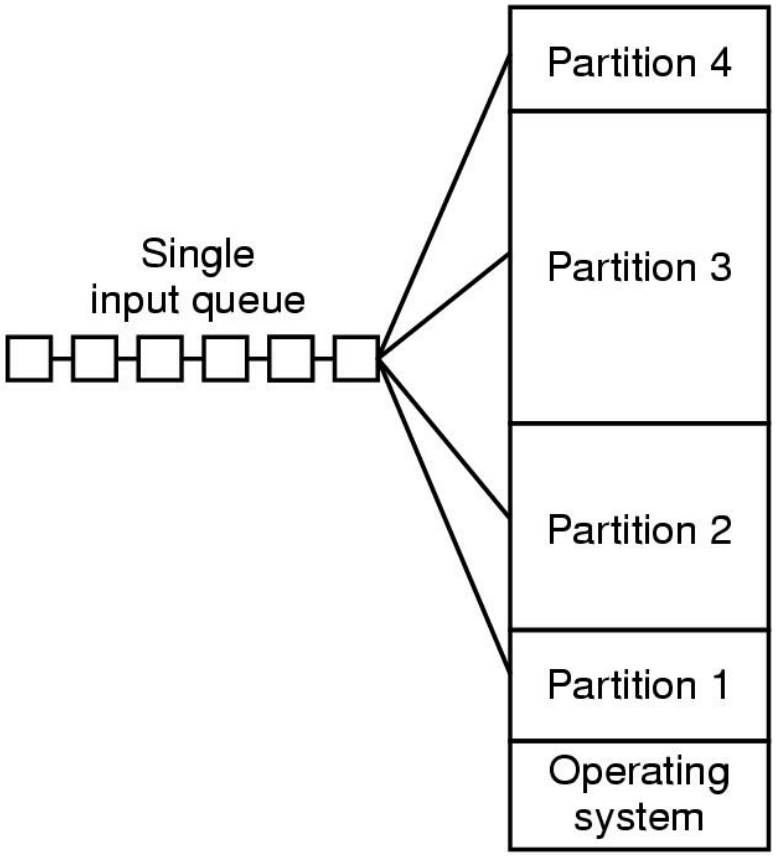
Best-fit

Extras/Unused

Multiprogramming with Fixed Partitions



(a)



(b)

Design: Fixed memory partitions

a) separate input queues for each partition

b) single input queue

Fixed Partitions: Advantages/Disadvantages

+ Simple

- Partition sizes limit the available memory for each process.

- A process is either entirely in main memory or entirely on backing store (i.e., swapped in or swapped out).

Design: Should there be a queue per partition or one global queue?

Fixed Partitions: Fragmentation

Fixed partitions waste memory

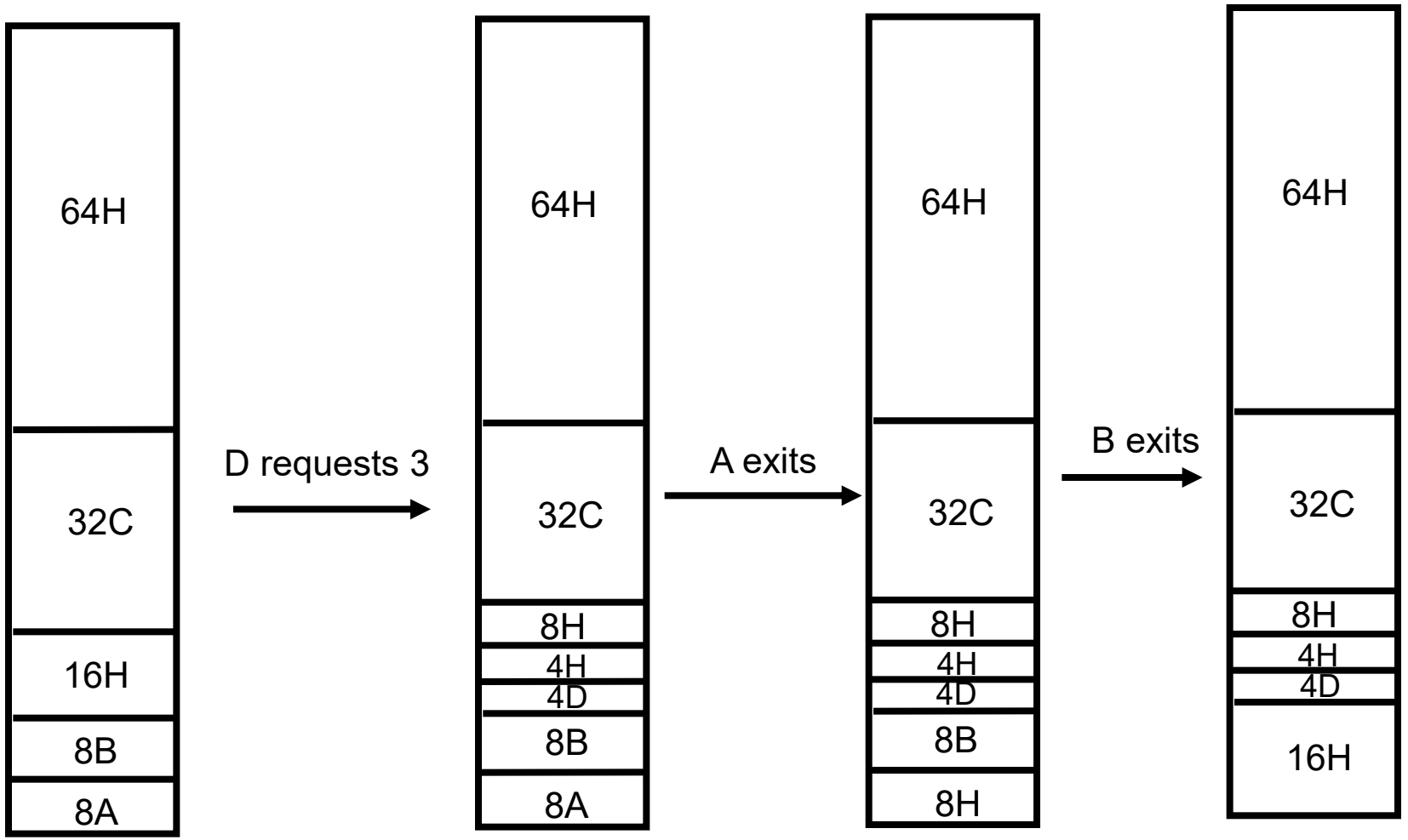
Internal fragmentation: memory which is internal to a partition, but not used.

External fragmentation: a partition is unused and available, but too small for any waiting job.

Buddy Systems

- Allocation algorithm that forms basis of Linux MM
- Suppose we have 128 units (128 pages or 128K)
- Each request is rounded up to powers of 2
- Initially a single hole of size 128
- Suppose, A needs 6 units, request rounded up to 8
- Smallest hole available: 128. Successively halved till hole of size 8 is created
- At this point, holes of sizes 8, 16, 32, 64
- Next request by B for 5 units: hole of size 8 allocated
- Next request by C for 24 units: hole of size 32 allocated

Buddy Systems



Memory Management Strategies

1. Fetch Strategy:
   Determine when to load and how much to load at a time. E.g., demand fetching, anticipated fetching (pre-fetching).
2. Placement (or allocation) Strategy:
   Determine where information is to be placed.
   E.g., Best-Fit, First-Fit, Buddy-System.
3. Replacement Strategy:
   Determine which memory area is to be removed under contention conditions.
   E.g., LRU, FIFO.