MEMORY MANAGEMENT

Lec Shakib, CSE, MIST

Ch- 3.1 to 3.4

Modern OS | Tanenbaum 4th Edition

Table of Content

MAIN MEMORY

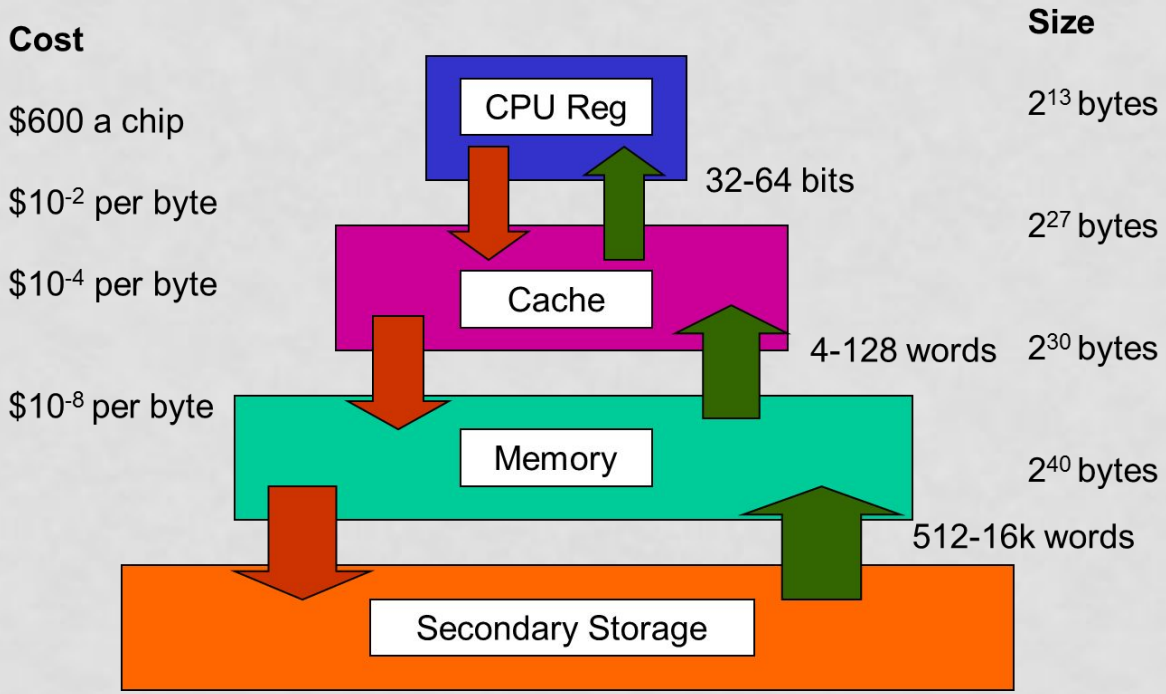
- A program resides on a disk as binary executable file.
- To be executed the program must be brought into RAM.
- The CPU fetches instructions from RAM according to the value of the Program Counter.
- The instruction operands may be needed to be fetched from RAM.
- After execution the results may be stored back to RAM.

MEMORY MANAGEMENT

- Ideally programmers want memory that is
  - private
  - large
  - fast
  - non volatile
  - Cheap

MEMORY MANAGEMENT

- But in real world…



- It is the job of the operating system to
  - **abstract** this hierarchy into a useful model
  - and then **manage** the abstraction.

MEMORY MANAGEMENT

- The part of the operating system that manages the memory hierarchy is called the *memory manager*.
  - Keep track of used memory
  - Allocate memory to processes
  - Deallocate it when they are done

OBJECTIVE

- In this chapter we will study
  - how operating systems create abstractions from memory
  - and how they manage them.
- The focus will be on the **programmer's model** of main memory
- Memory Abstraction: the view/illusion of the memory presented to the programmer by the OS
- We will investigate several memory management models.

NO MEMORY ABSTRACTION

- The simplest memory abstraction is to have no abstraction at all.
- Every program simply saw the physical memory
  - MOV REG1, 1000
  - move the contents of physical memory location 1000 to REGISTER1
- Model of memory presented to the programmer is simply physical memory
- Not possible to have 2 running programs in memory at the same time
- One program might interfere with the other program, which leads to program crash.

BUT REALLY WANT TO RUN MORE THAN ONE PROGRAM

- Could swap new program into memory from disk and send old one to disk
- At each instance, there is only 1 program in the memory.
- Not really concurrent
- Swapping in and out entire programs is slower.
- *How to run multiple programs concurrently without swapping between them?*

IBM 360's PROTECTION KEY IDEA

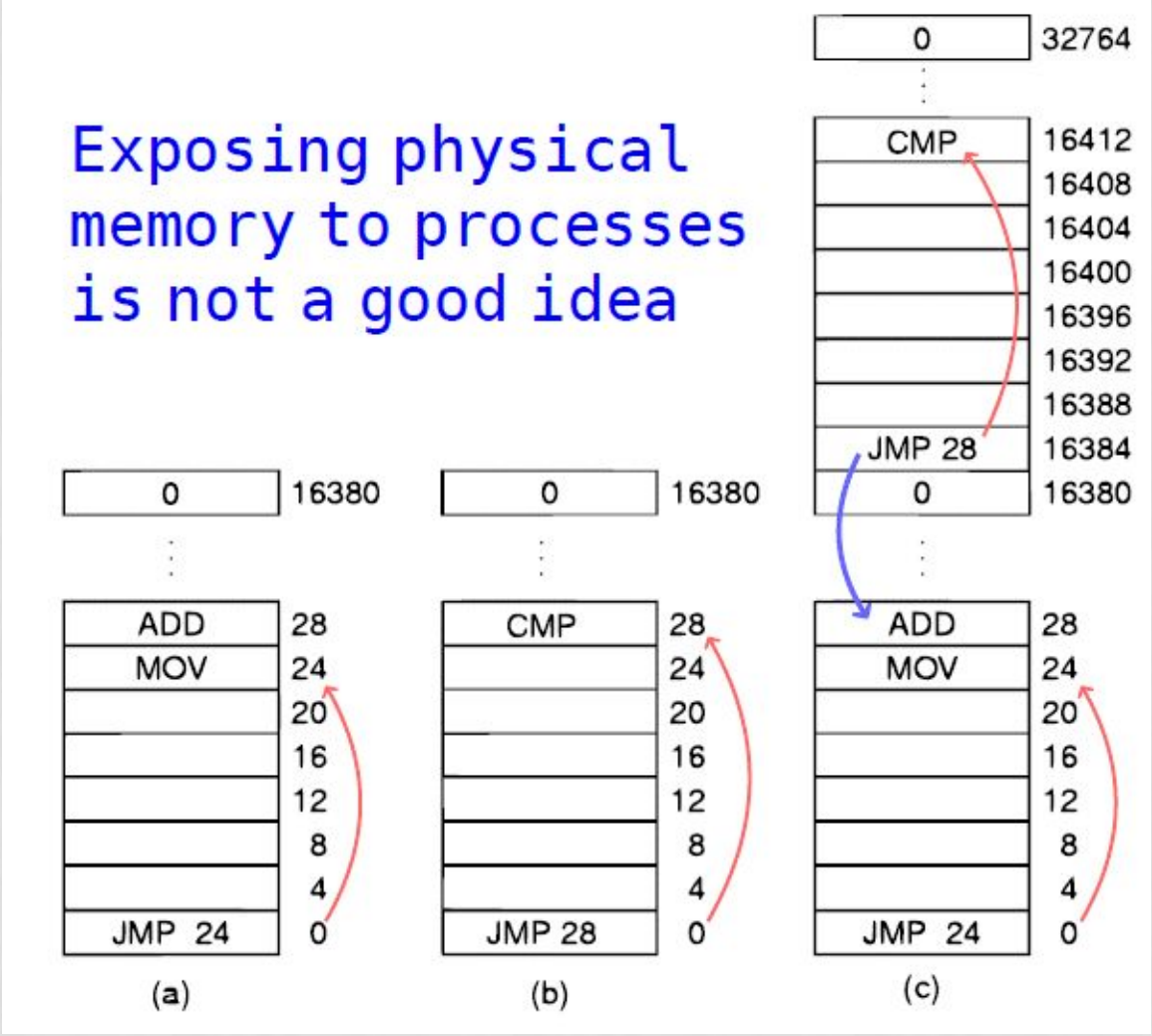
- IBM 360 –
  - divide memory into 2 KB blocks,
  - associate a 4 bit protection key with each block.
  - Keep keys in registers.
- PSW (Program Status Word) for a program also contains a 4-bit key.
- Relocate programs from disk into memory.
- OS prevent programs from accessing blocks with a different protection key.

  (PSW Key ≠ register key of block)

However there is a problem

PROBLEM WITH SIMPLE RELOCATION

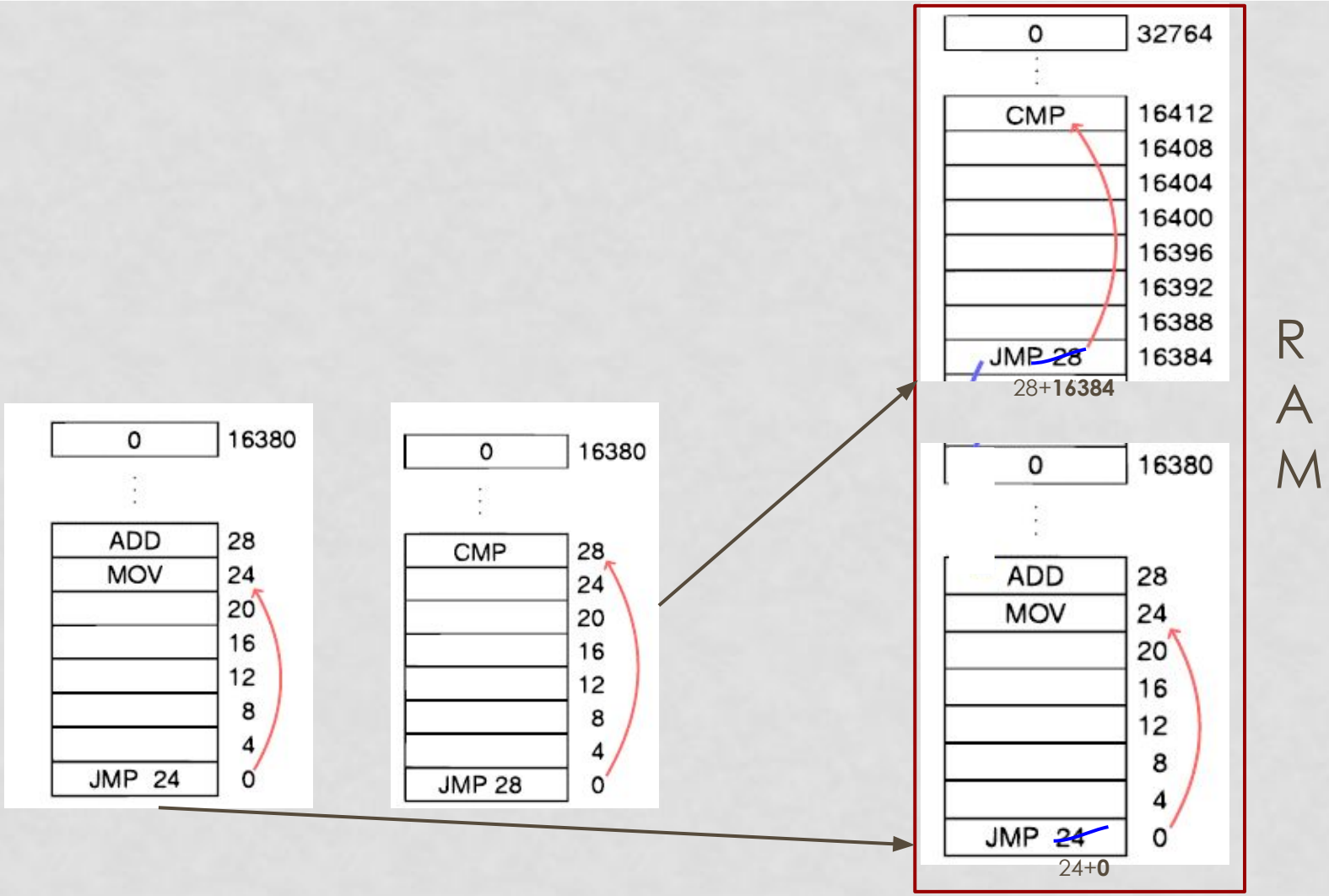
- When relocated, JMP 28 in program (b) cause program counter to go to ADD instruction in location 28, instead of CMP.
- Program crashes



IBM 360's STATIC RELOCATION IDEA

- Static relocation – load first instruction of program at memory physical address X, and add X to every subsequent address during loading
  - This is too slow and
  - Not all values can be modified
    - ADD register 1, 28 can't be modified [28 is not an address in this instruction, rather a constant value]

IBM 360's STATIC RELOCATION IDEA



PROBLEM WITH EXPOSING PHYSICAL MEMORY

- exposing physical memory to processes has several major drawbacks.
  - if user programs can address every byte of memory, they can easily trash the operating system intentionally or by accident
  - So it is difficult to have multiple programs running at once

A MEMORY ABSTRACTION: ADDRESS SPACE

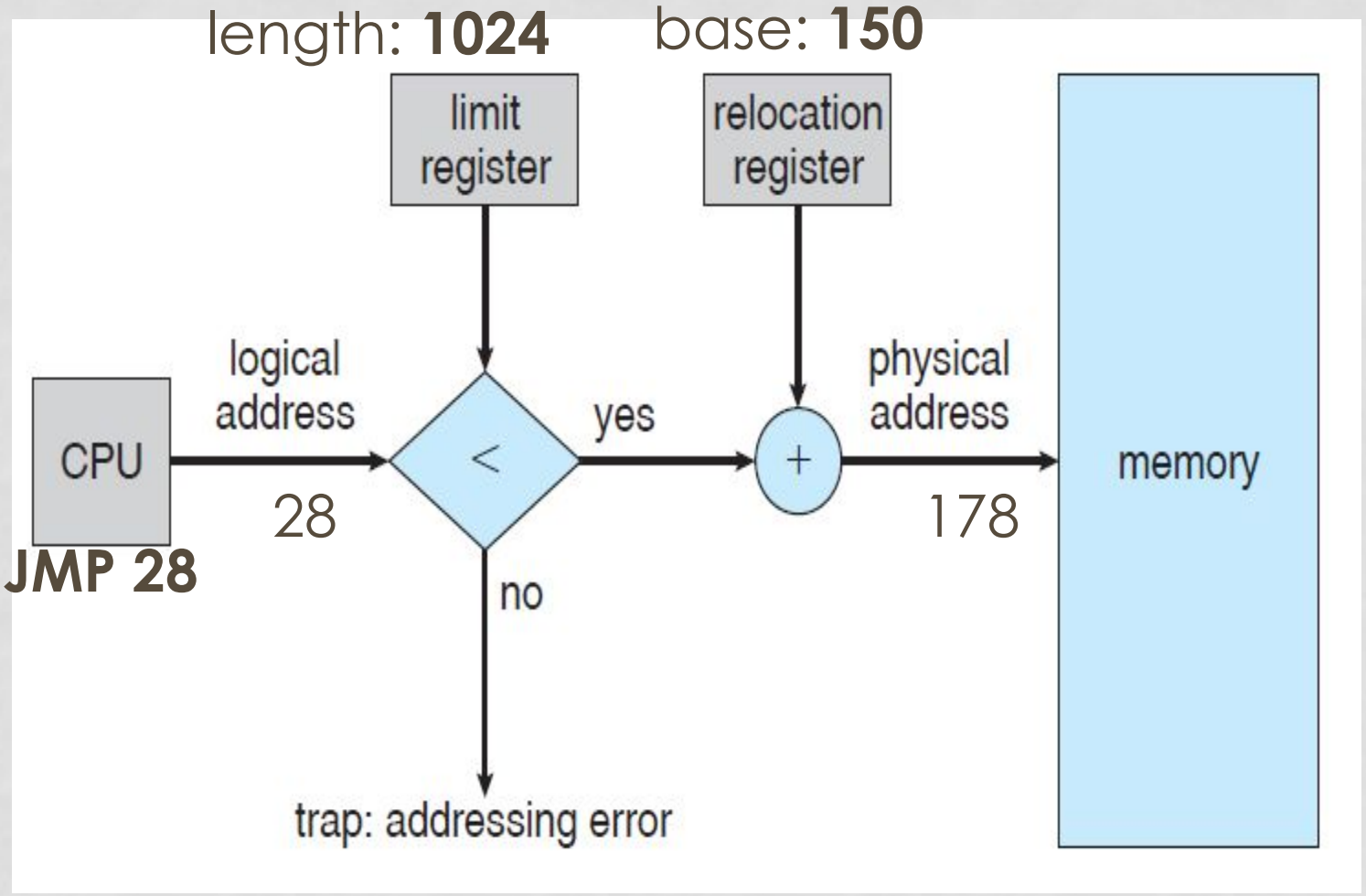
- From IBM 360's primitive solution, we learned that Two problems have to be solved if we are to run multiple programs in memory:
  - **Protection** (prevent interference of programs)
  - **Relocation** (transfer programs to memory)

- Solution: Create abstract memory space for program to exist in
  - Each program has its own independent set of addresses
  - The addresses are different for each program
  - Call it the **address space** of the ~~program~~ process.
  - i.e., private address 28 of one process in RAM, is different from private address 28 of another process in RAM.

BASE AND LIMIT REGISTERS

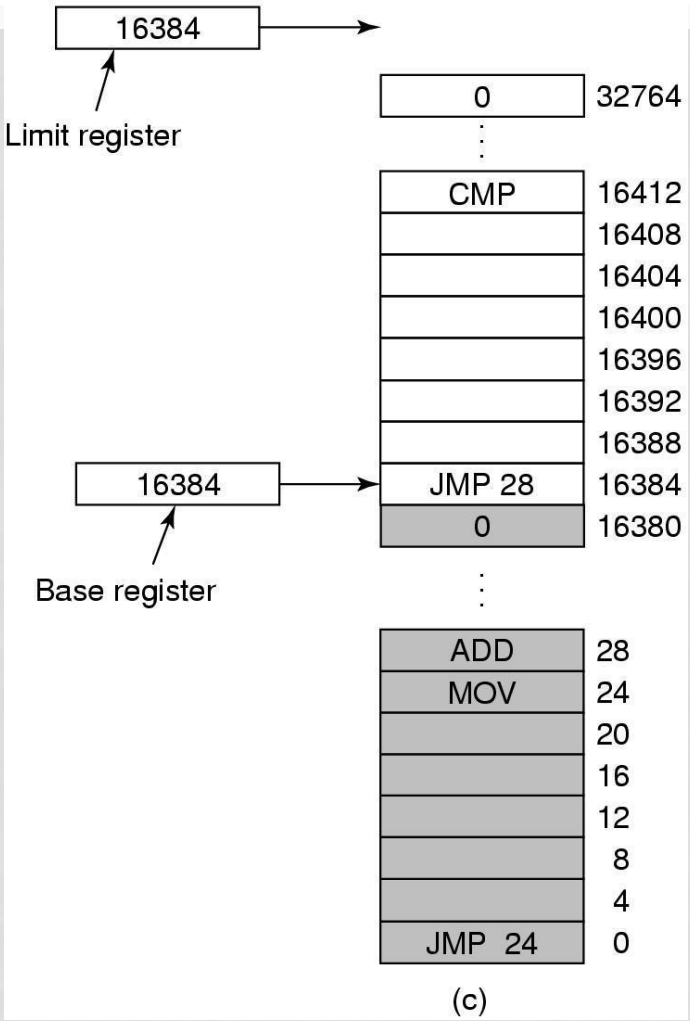
- A simpler version of dynamic relocation
- Equip CPU with two special registers
- Base contains beginning address of process
- Limit contains length of process
- Whenever a process references a memory address, base register adds base address to the address generated by process. Checks to see if address is larger then limit. If so, generates fault

BASE (relocation) AND LIMIT REGISTERS



BASE AND LIMIT REGISTERS

- Add 16384 to JMP 28.
- Hardware adds 16384 to 28 resulting in JMP 16412



Difference between static relocation & base+limit registers

Static relocation **translates** a program's addresses **permanently** when the program is loaded into memory.

But dynamic relocation uses base and limit registers to **translate** a process's addresses **on the fly during execution**, **allowing it to be moved into the memory**. During execution, moving the process into a different location of the main memory just requires the base register value to be updated.

BASE AND LIMIT REGISTERS

- Disadvantage-addition and comparison have to be done on every instruction.

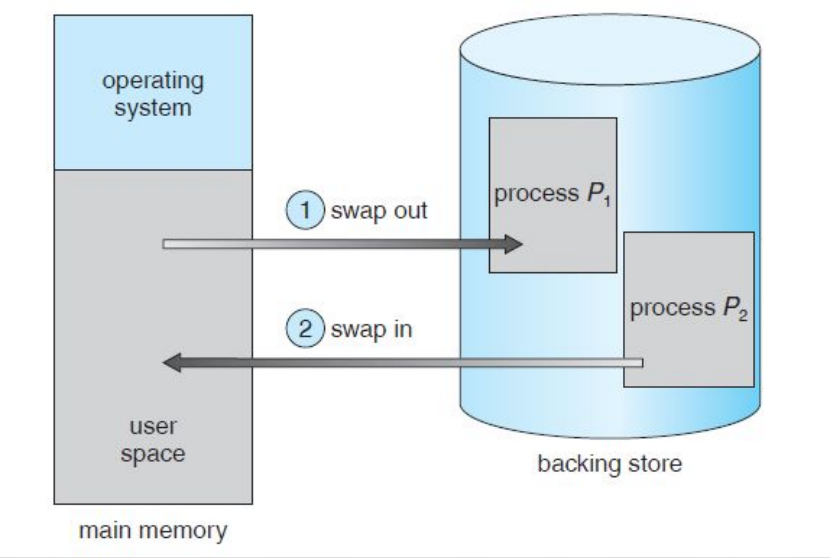
HOW TO RUN MORE PROGRAMS THAN IT CAN FIT IN MAIN MEMORY AT ONCE

- In practice, the total amount of RAM needed by all processes is much more than can fit in memory.

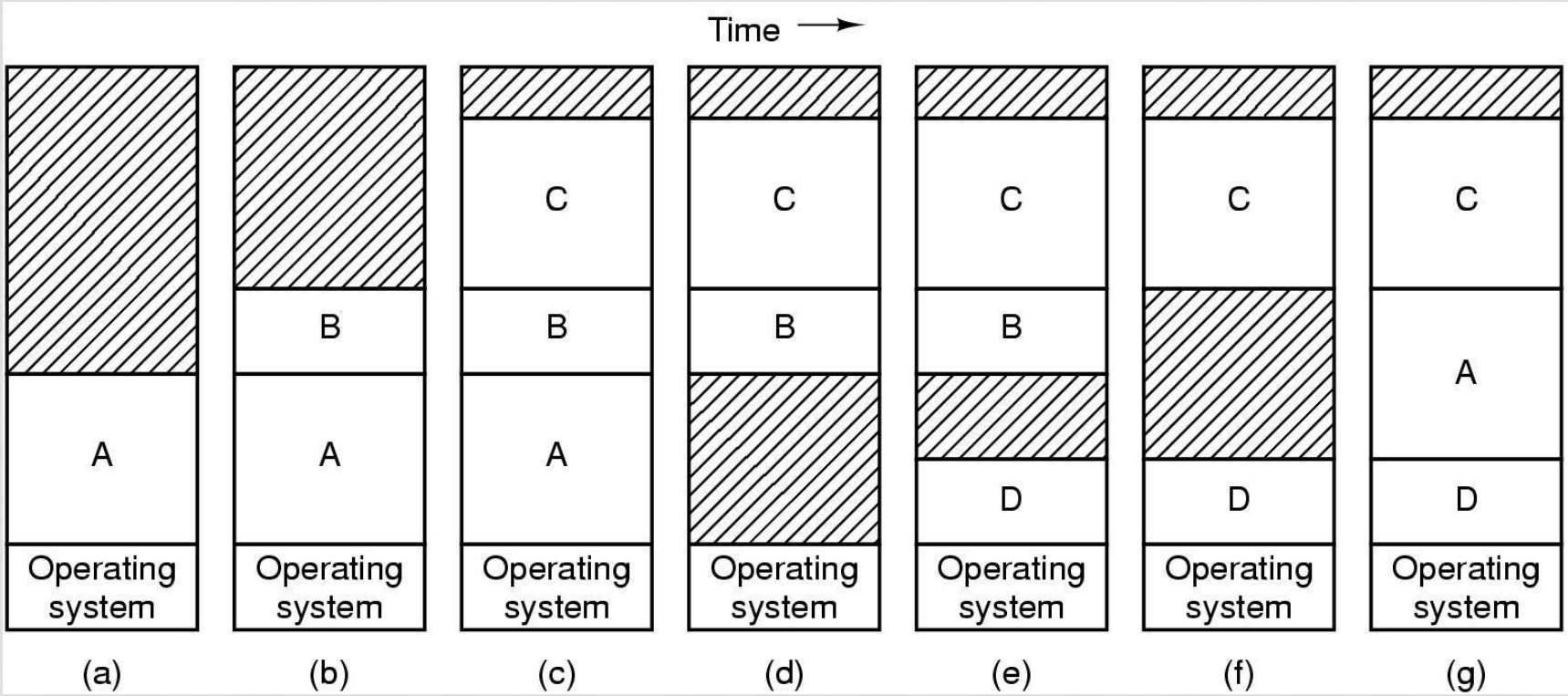
- Can't keep all processes in main memory
  - Too many (hundreds)
  - Too big (eg 200MB program)
- Two approaches to dealing with memory overload
  - Swap-bring entire process in and run it for a while
  - Virtual memory – allow process to run even if only part of it is in main memory

SWAPPING

- Bringing in each process in its entirety,
- running it for a while
- then putting it back on the disk
- But, unlike the swapping that we saw in "no memory abstraction", here multiple processes can be swapped in the memory together.



SWAPPING



Processes coming and going out of memory over time

SWAPPING

- Memory allocation changes as
  - processes come into memory
  - leave memory
- Shaded regions are unused memory
- Hole – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about: a) allocated partitions b) free partitions (hole)

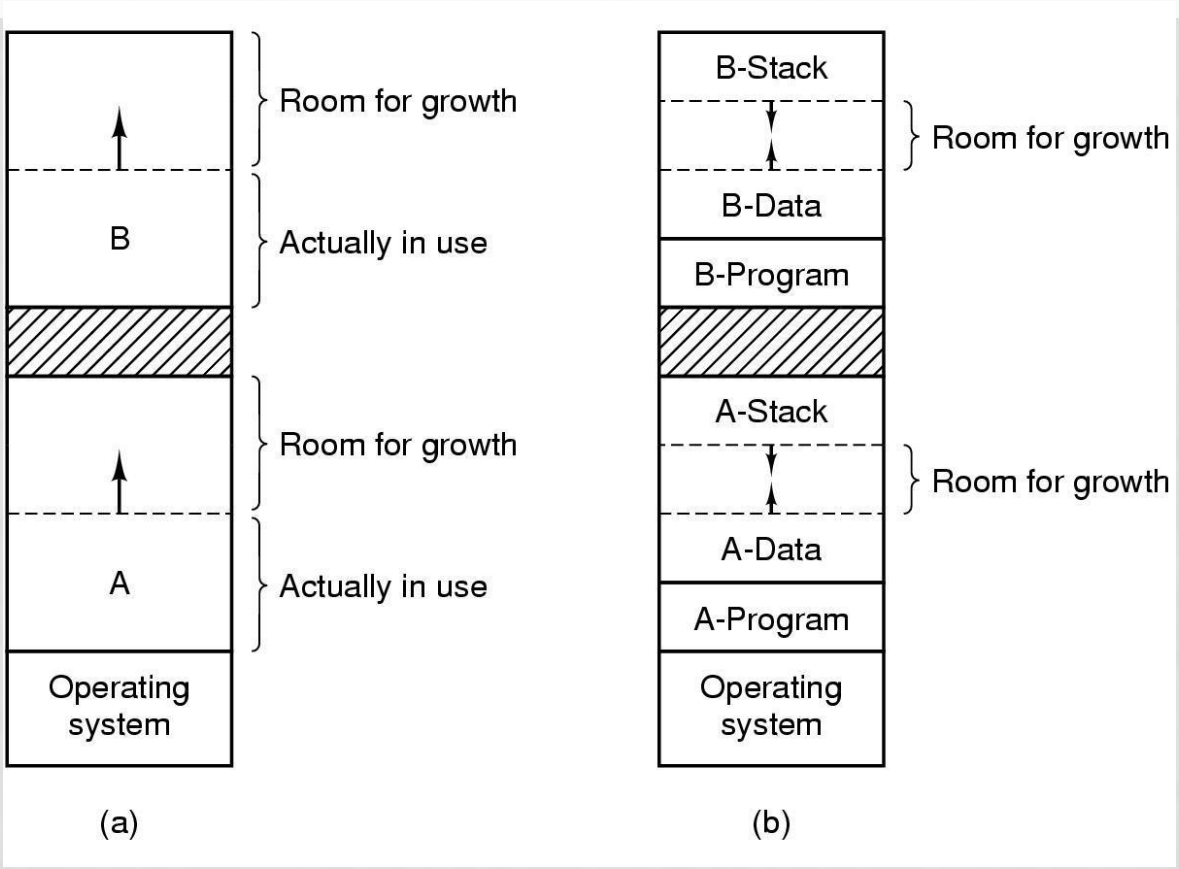
SWAPPING

- When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible.
- This technique is known as memory compaction. This technique requires a lot of CPU time.

PROCESSES GROW AS THEY EXECUTE

- Stack (return addresses and local variables)
- Data segment (heap for variables which are dynamically allocated and released)
- Good idea to allocate extra memory for both
- When program goes to disk, don't bring holes along with it!!

2 WAYS TO ALLOCATE SPACE FOR GROWTH

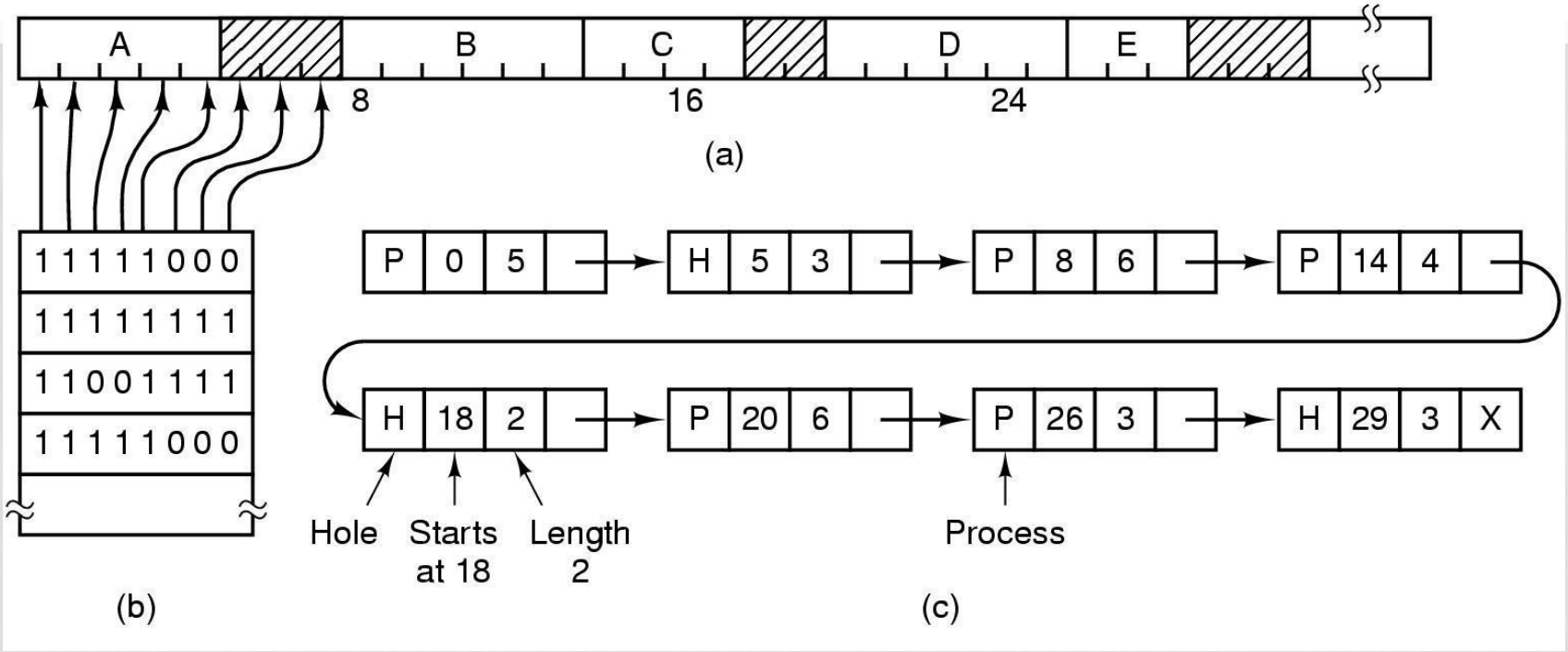


- Allocating space for growing data segment
- Allocating space for growing stack & data segment

MANAGING FREE MEMORY

- Operating system maintains information about
  - Allocated memory
  - Free memory (holes)
- Two techniques to keep track of memory usage
  - Bitmaps
  - Linked lists

BITMAPS



(a) Picture of memory

(b) Each bit in bitmap corresponds to a unit of storage (eg. bytes) in memory

(c) Linked list: P - process, H - hole

BITMAPS

- memory divided into allocation units.
- Each allocation unit is represented by a bit in the bitmap which is 0 if the unit is free and 1 if it is occupied.
- The smaller the size of a allocation unit, the larger the bitmap.
- Larger size of an allocation unit means, higher chance that a process size is not an exact multiple of the allocation unit, some memory will be wasted.

  i.e., how can we represent a process of 18 bytes with a bitmap of 4 bytes for each bit?

  18/4 not divisible exactly

BITMAPS

- Q) There are a total 4 processes currently in memory- sizes 12 KB, 110 KB, 30 KB, 280 KB. Total memory size is 1 MB. We use a bitmap to keep track of the memory, where every bit represents 2KB. Assume that every process size is an exact multiple of the bit allocation unit.

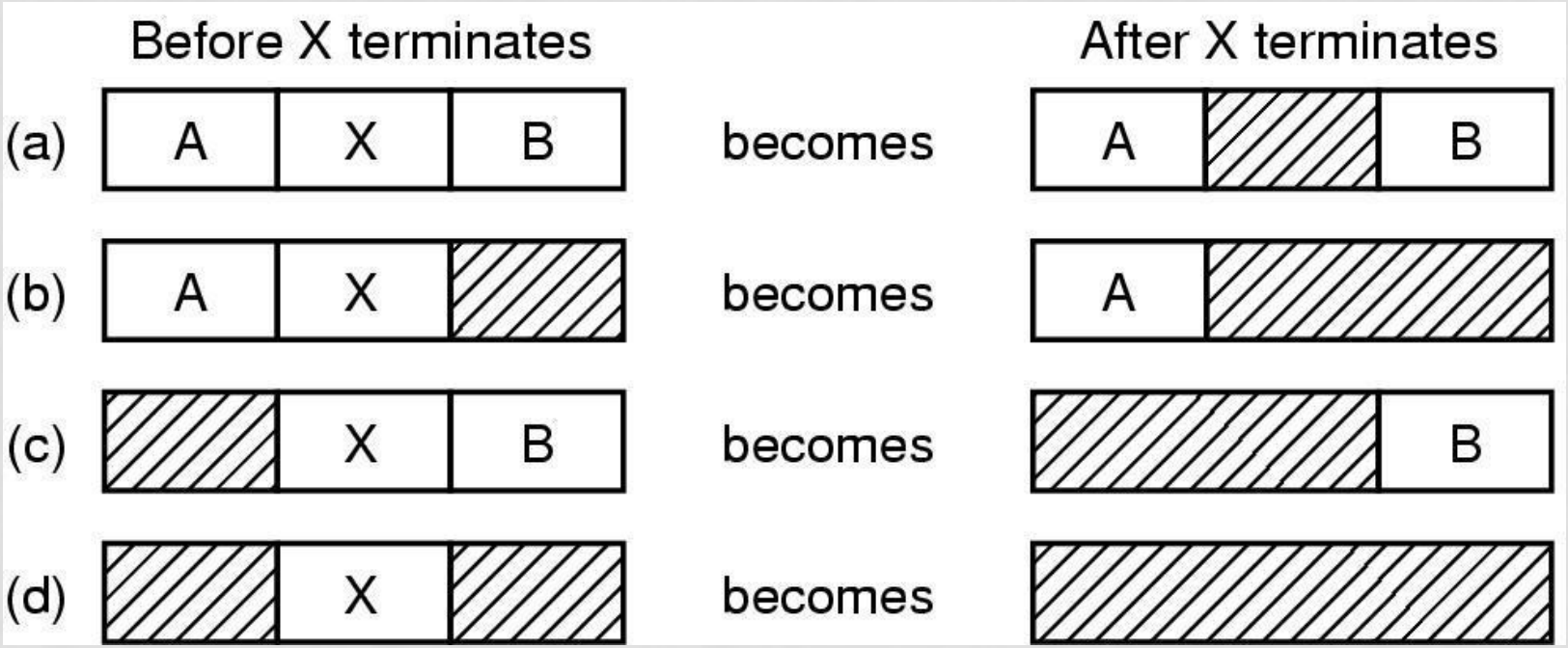
  Find-

  - How many 0s in the bitmap? What is the size of bitmap in bytes?
  - **total hole size= 1M-12K-110K-30K-280K = 1024K-12K-1K-3K-28K = 592KB**
  - **# of 0s = 592K/2K = 296**
  - **Total # of bits = 1M/2K = (1*1024)K/2K = 512**
  - **So, bitmap size in bytes = 512/8 = 64 Bytes**

BITMAPS

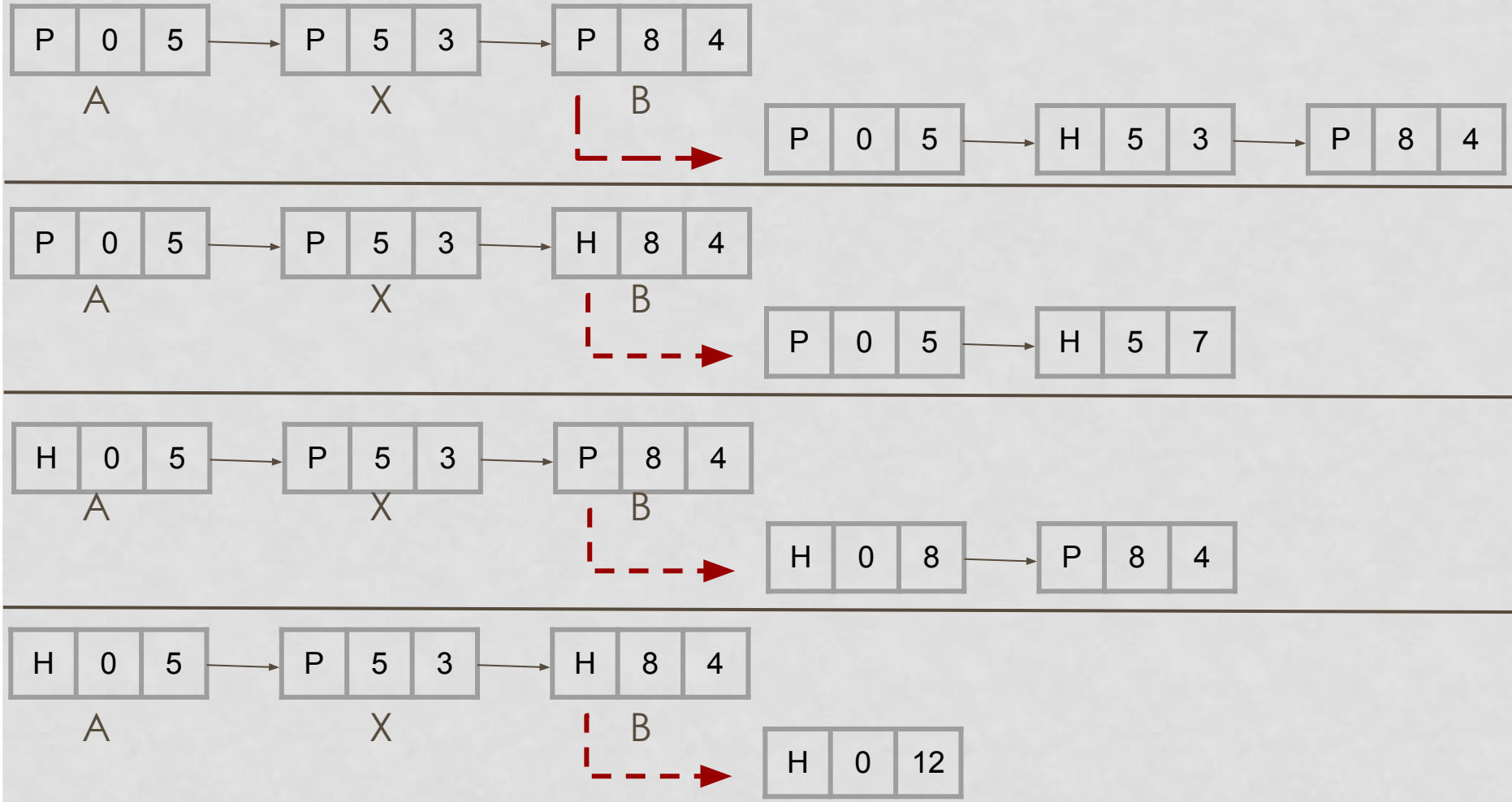
- The good – compact way to keep track of memory
- The bad – need to search memory for k consecutive zeros to bring in a file k units long. This is slow.
- Units can be bits or bytes or …….

LINKED LISTS



- Four neighbor combinations for terminating process X.
- Scenarios (a) (b) (c) (d) shown as linked lists in the next slide

LINKED LISTS



LINKED LISTS

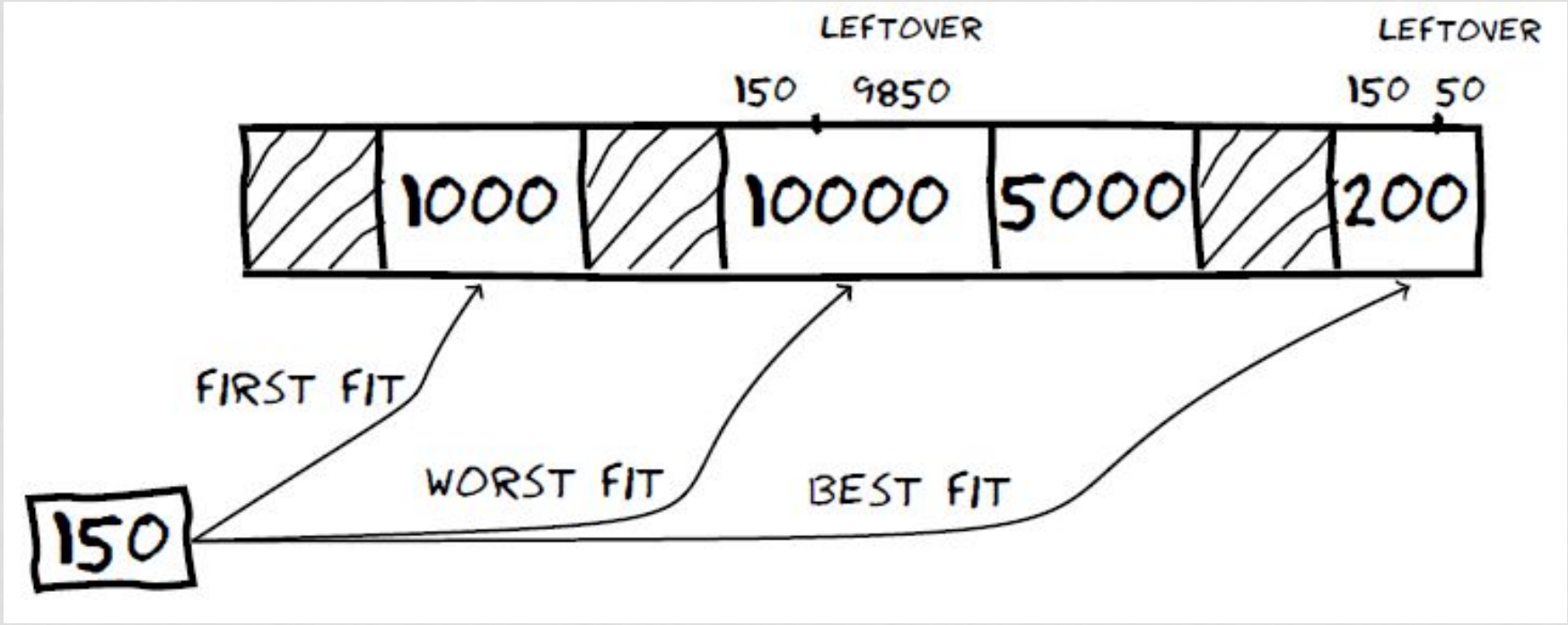
- Consecutive holes represented as a single node

LINKED LISTS

- Might want to use doubly linked lists to merge holes more easily
- Algorithms to fill in the holes in memory
  - Next fit
  - Best fit
  - Worst fit
  - Quick fit

LINKED LISTS

- Memory allocation algorithms : First fit, next fit, best fit, worst fit, quick fit



THE FITS

- First fit-fast
- Next fit-starts search wherever it is
  - Slightly worse
- Best fit-smallest hole that fits
  - Slower, results in a bunch of small holes (i.e. worse algorithm)
- Worst fit-largest hole that fits
  - Not good (simulation results)
- Quick fit- keep list of common sizes
  - Quick, but can’t find neighbors to merge with

THE FITS

- Conclusion – the fits couldn't out-smart the unknowable distribution of hole sizes.
- The extra work to deal with something which you can't predict failed to produce good results