Memory Management

Week 06 – Lecture 1

Memory Abstraction

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
    Prentice-Hall, Inc.

  and customised for the needs of this course.
- Additional input for the slides are detailed later

Memory (1)

- Paraphrase of Parkinson's Law, "*Programs expand to fill the memory available to hold them.*"
- Average home computer nowadays has 10,000 times more memory than the IBM 7094, the largest computer in the world in the early 1960s

Memory (2)

- We all want private, infinitely large, infinitely fast and inexpensive memory that does not lose its contents when the electric power is switched off
- The second choice is the concept of a **memory hierarchy**:
  - a few megabytes of very fast, expensive, volatile cache memory
  - a few gigabytes of medium-speed, medium-priced, volatile main memory
  - a few terabytes of slow, cheap, nonvolatile magnetic or solid-state disk storage
- It is the job of the operating system to abstract this hierarchy into a useful model and then manage the abstraction

Memory (3)

- The part of the operating system that manages the memory hierarchy is called the **memory manager**
- Its job is to efficiently manage memory:
  - keep track of which parts of memory are in use
  - allocate memory to processes when they need it
  - deallocate it when they are done

No Memory Abstraction (1)

- Early main- frame computers, early minicomputers and early personal computers had no memory abstraction
- All the physical memory was available to every program and it was not possible to have two running programs in memory at the same time
- The first program could easily erase data belonging to the second one

No Memory Abstraction (2)

- Even without memory abstraction several variations are possible:
  - The OS may be at the bottom of memory in RAM (Fig. 3-1a). It was used on mainframes and minicomputers
  - The OS may be at the top of memory in ROM (Fig. 3-1b). Is used on some handheld devices and in embedded systems
  - The device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below (Fig. 3-1c). It was used by early personal computers with BIOS in ROM and MS-DOS running in RAM, for example

No Memory Abstraction (3)

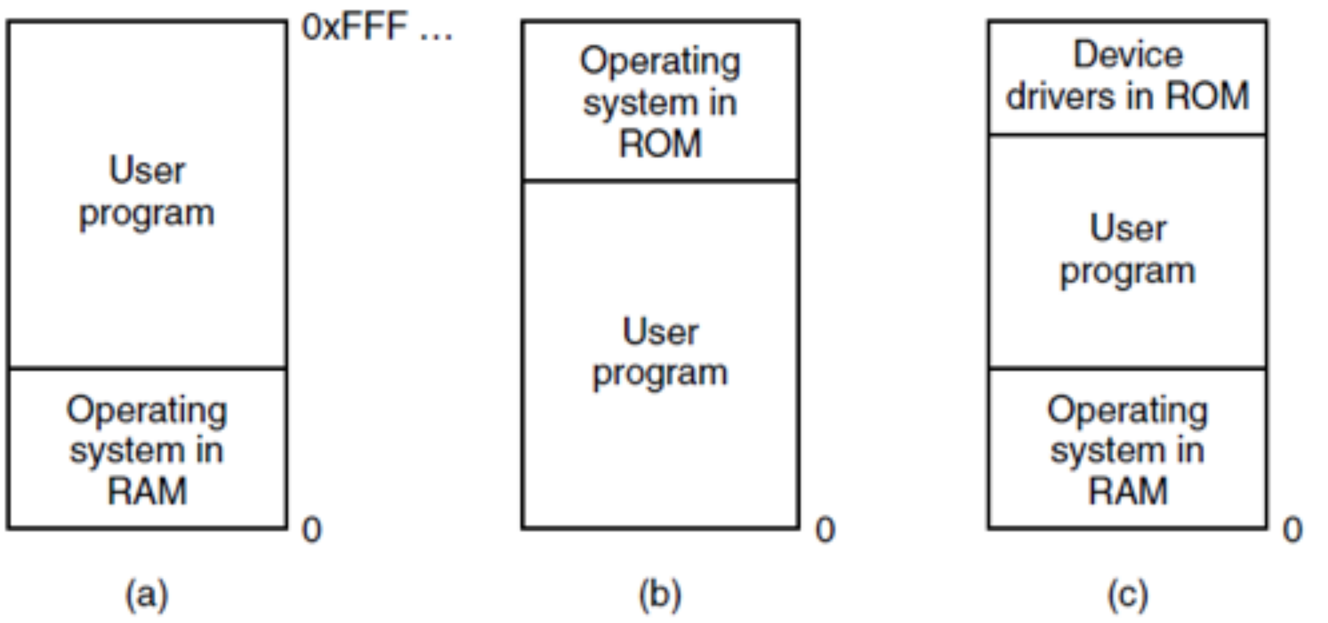

Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist

Running Multiple Programs Without a Memory Abstraction (1)

- It is still possible to run multiples programs even with no memory abstraction
- The OS has to save the entire contents of memory to a disk file, then bring in and run the next program (swapping)
- As long as there is only one program at a time in memory, there are no conflicts

Running Multiple Programs Without a Memory Abstraction (2)

- With the addition of some special hardware, it is possible to run multiple programs concurrently, even without swapping like it was done in early models of the IBM 360:
  - memory was divided into 2-KB blocks and each is assigned a 4-bit protection key held in special registers inside the CPU
  - the PSW (Program Status Word) also contained a 4-bit key
  - the hardware trapped any attempt by a running process to access memory with a protection code different from the PSW key
  - Since only the OS could change the protection keys, user processes were prevented from interfering with one another and with the OS itself

Running Multiple Programs Without a Memory Abstraction (3)

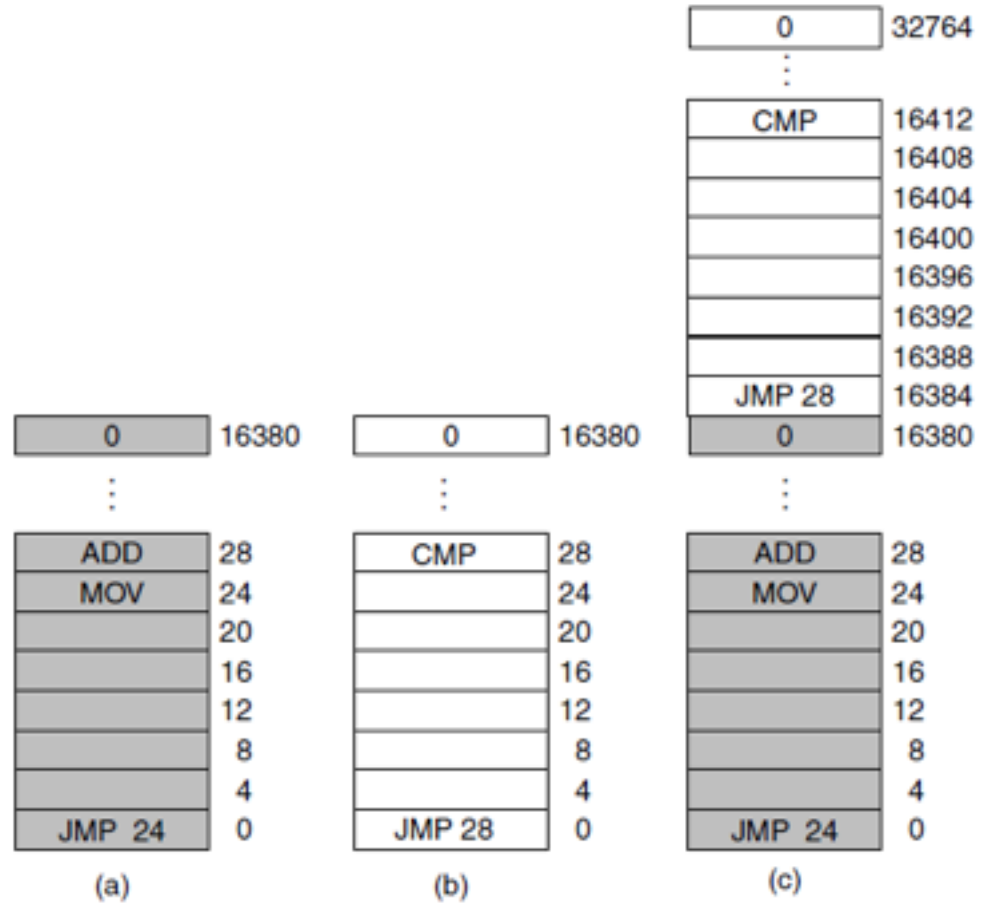

Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

Running Multiple Programs Without a Memory Abstraction (4)

- The problem of the approach described above:
  - The first program will function normally
  - However, after the OS decides to run the second program, it will execute the first instruction, which is *JMP 28*
  - Instead of jumping to the CMP instruction as expected, it will jump to the ADD instruction of the first program and most likely crush

Running Multiple Programs Without a Memory Abstraction (5)

- The solution is to use **static relocation**:
  - when a program is loaded at address 16384, the constant 16384 is added to every program address during the load process. For example, *JMP 28* becomes *JMP 16412*
- Drawbacks of such an approach are:
  - It is not a very general solution and slows down loading.
  - It requires extra information to indicate which words contain (relocatable) addresses and which do not. For example, 28 in *JMP 28* has to be relocated but an instruction like *MOV REGISTER1, 28* must not be relocated since 28 is a number, not an address

A Memory Abstraction: Address Spaces (1)

- Exposing physical memory to processes has several major drawbacks:
  - if user programs can address every byte of memory, they can intentionally or by accident trash the OS even if only one application is running
  - it is difficult to have multiple programs running at once if there is only one CPU

A Memory Abstraction: Address Spaces (2)

- Two problems have to be solved: **protection** and **relocation**
- An **address space** is a concept that helps solving both problems. It is the set of addresses that a process can use to address memory
- The address space creates a kind of abstract memory for programs to live in
- Each process has its own address space, independent of those belonging to other processes

Base and Limit Registers (1)

- **Dynamic relocation** – mapping each process' address space onto a different part of physical memory
- **Base** and **limit** registers – two special hardware registers. Base register contains the physical address of the beginning of the program. Limit register contains its length
- When these registers are used, programs are loaded into **consecutive memory locations** wherever there is room and without relocation during loading

Base and Limit Registers (2)

- Every time a process references memory, the CPU hardware automatically adds the base value to the address generated by the process
- Simultaneously, it checks whether the address offered is equal to or greater than the value in the limit register, in which case a fault is generated and the access is aborted
- Disadvantage: the need to perform an addition and a comparison on every memory reference (additions are slow unless special addition circuits are used)

Base and Limit Registers (3)

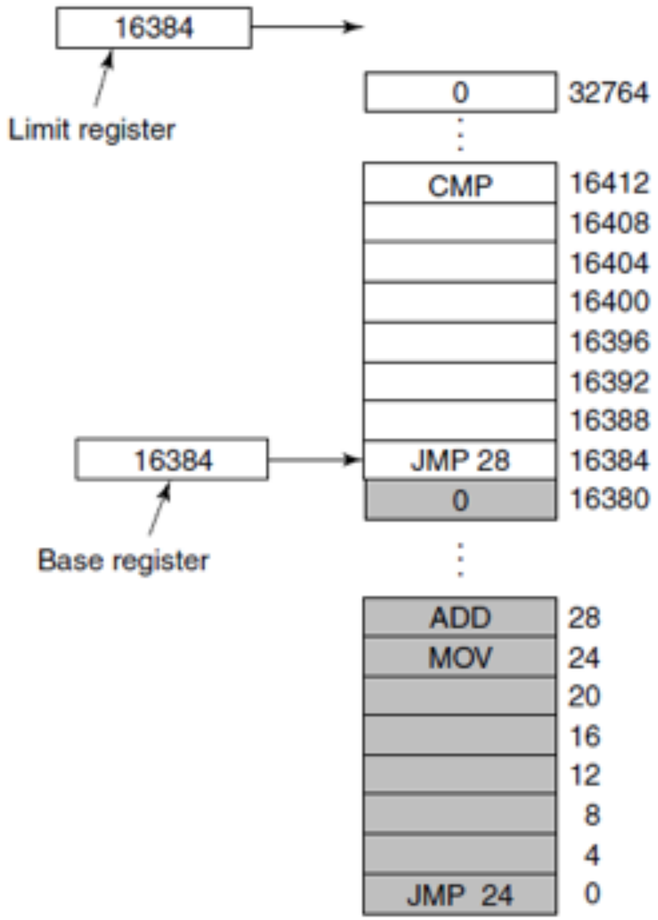

Figure 3-3. Base and limit registers can be used to give each process a separate address space.

Swapping (1)

- All the running processes may require more memory than it is physically available
- There are two approaches to deal with this problem:
  - **Swapping** – idle processes are mostly stored on disk and are brought into main memory when needed, then are put back on disk
  - **Virtual memory** – programs run even when they are only partially in main memory (will be discussed later)

Swapping (2)

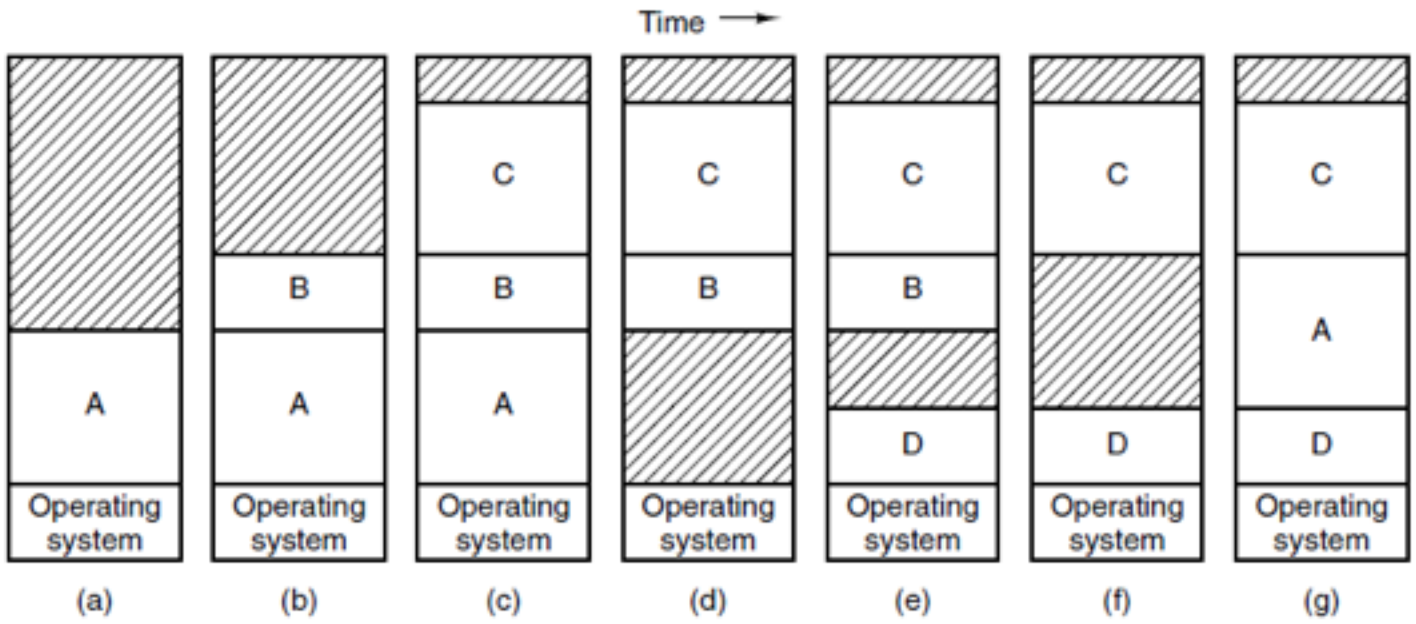

Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory

Swapping (3)

- After swapping in a process might be at a different location and addresses contained in it must be relocated, either by software or, more likely, by hardware during program execution. Base and limit registers would work fine here
- Swapping creates multiple holes in memory which are possible to combine into a big one by moving all the processes downward as far as possible
- Such a technique is known as **memory compaction**. However, it it requires a lot of CPU time, so it is usually not done

Swapping (4)

- How much memory should be allocated for a process when it is created or swapped in?
  - If processes are created with a fixed size that never changes, the OS allocates exactly what is needed
  - If processes' data segments can grow, for example, by dynamically allocating memory from a heap, a problem occurs whenever a process tries to grow

Swapping (5)

- If a hole is adjacent to the process, it can be allocated and the process allowed to grow into the hole
- If the process is adjacent to another process, the growing process will either have to be moved to a hole in memory large enough for it, or one or more processes will have to be swapped out to create a large enough hole
- If a process cannot grow in memory and the swap area on the disk is full, the process will have to suspended until some space is freed up (or it can be killed)

Swapping (6)

- If it is expected that most processes will grow as they run, it is probably a good idea to allocate a little extra memory whenever a process is swapped in or moved (Fig. 3-5a)
- If processes can have two growing segments (the data segment being used as a heap and a stack segment), the memory between them can be used for either segment (Fig. 3-5b)

Swapping (7)

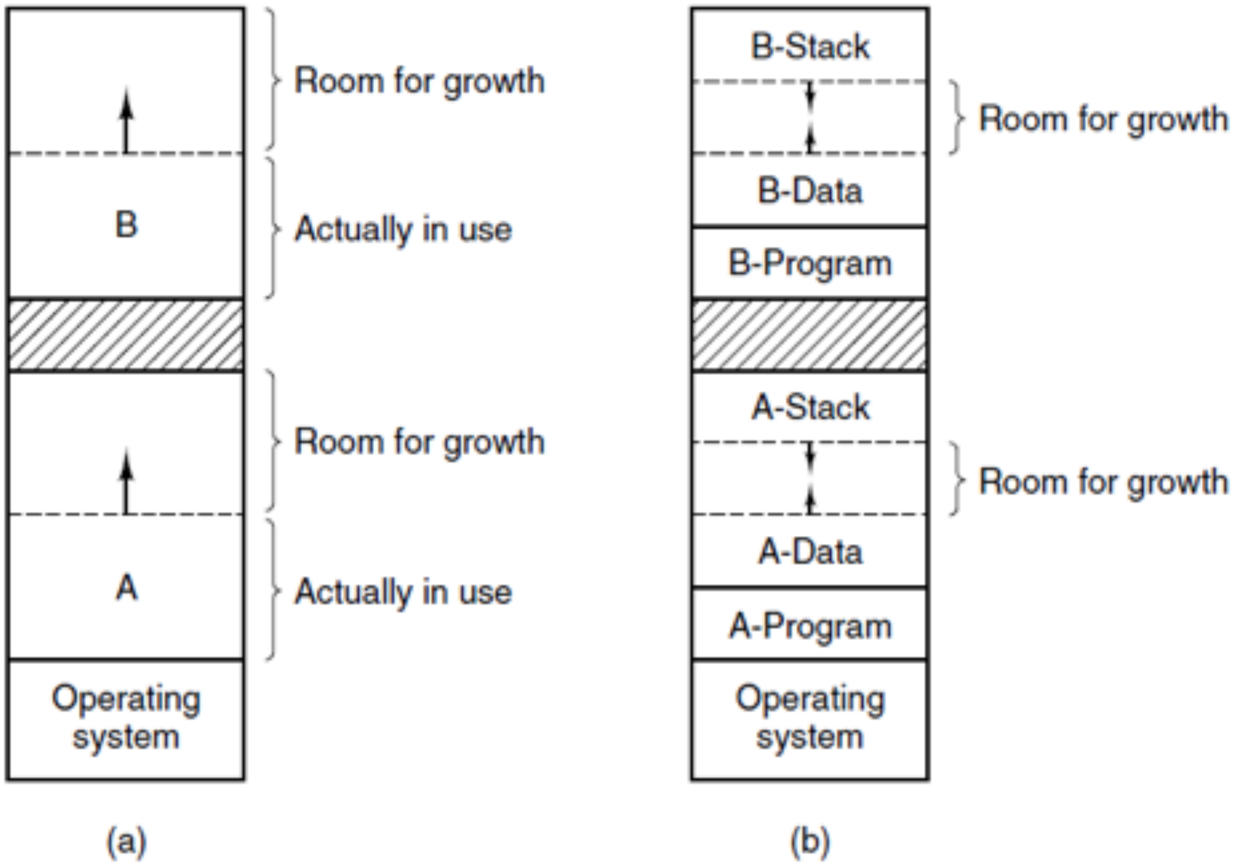

Figure 3-5. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

Memory Management with Bitmaps (1)

- In general terms, there are two ways to keep track of memory usage: **bitmaps** and **free lists**
- With a bitmap, memory is divided into allocation units as small as a few words and as large as several kilobytes
- Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied or vice versa (Fig. 3-6)

Memory Management with Bitmaps (2)

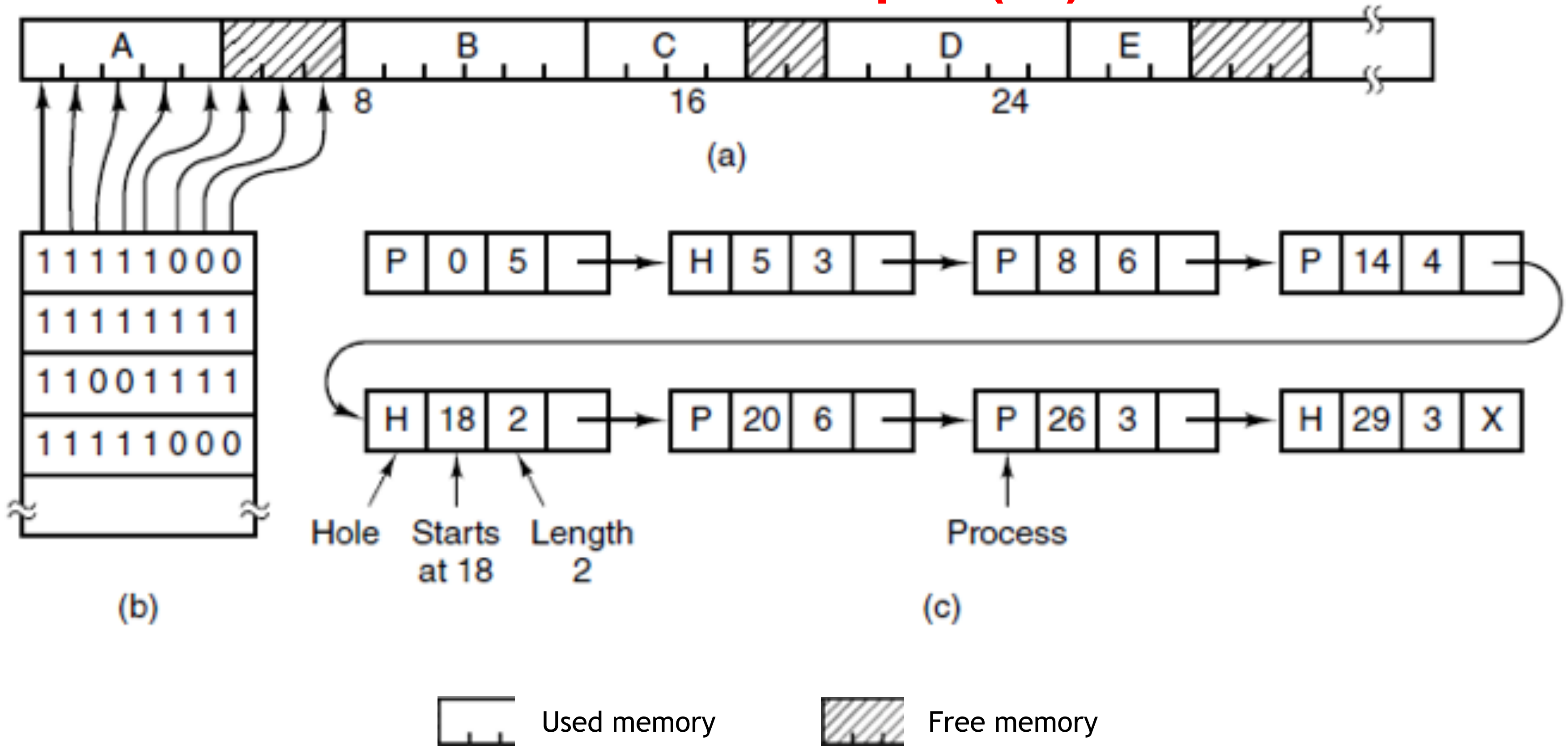

Figure 3-6. (a) A part of memory with five processes and three holes. (b) The corresponding bitmap. (c) The same information as a list.

Memory Management with Bitmaps (3)

- The size of the allocation unit is an important design issue
- The smaller the allocation unit, the larger the bitmap. With 4-bytes allocation unit 32 bits of memory will require 1 bit of the map, so the bitmap will take up only 1/32 of memory
- If the allocation unit is chosen large, the bitmap will be smaller, but memory may be wasted in the last unit of the process if the process size is not an exact multiple of the allocation unit

Memory Management with Bitmaps (4)

- A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit
- The main problem is that when it has been decided to bring a *k*-unit process into memory, the memory manager must search the bitmap to find a run of *k* consecutive 0 bits in the map
- There is an argument against bitmaps: searching a bitmap for a sequence of a given length is a slow operation since the sequence may cross word boundaries in the map

Memory Management with Linked Lists (1)

- Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments each element of which has the next structure:

H/P	**18**	**2**	→
Hole or Process	Start of the segment	Length of the segment	Pointer to the next element

Memory Management with Linked Lists (2)

- Sorting the list by address has the advantage that when a process terminates or is swapped out, updating the list is straightforward
- If we have three consequent processes (Fig. 3-7a), updating the list requires replacing P by an H
- In case when there are two consequent processes (Fig. 3-7bc) two entries are merged into one
- The last case (Fig. 3-7d) is when a process is surrounded by two holes. It may be more convenient to have the list as a double-linked list, rather than the single-linked list and it will make it easier to find the previous entry and to see if a merge is possible

Memory Management with Linked Lists (3)

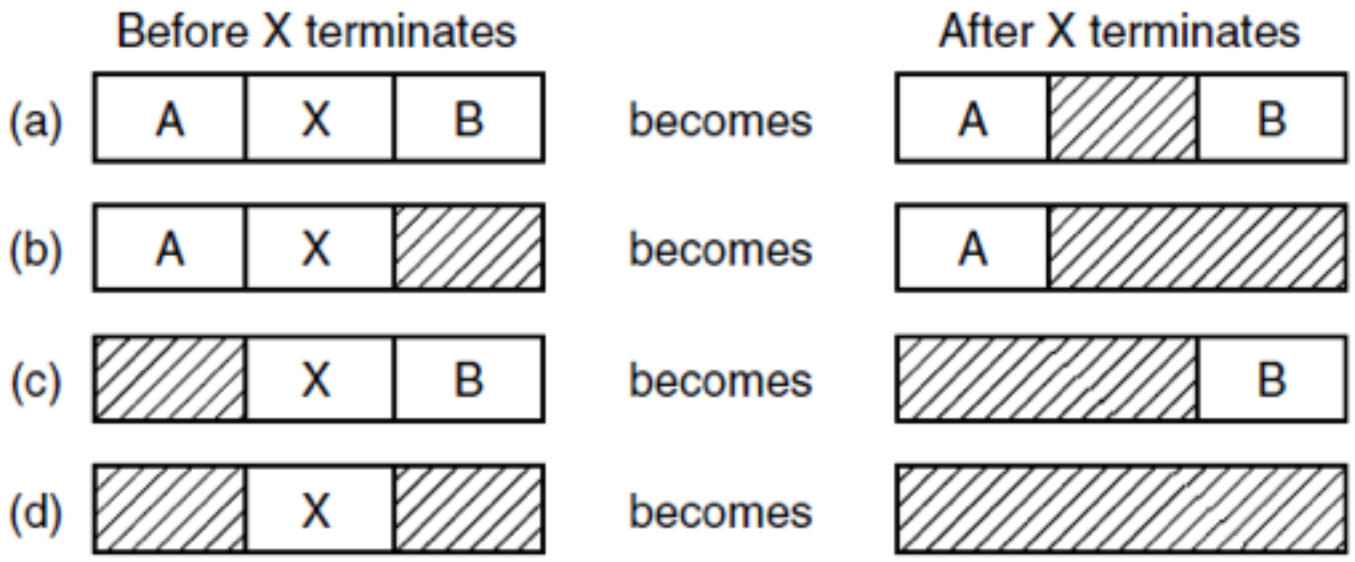

Figure 3-7. Four neighbor combinations for the terminating process, *X*.

Memory Management Algorithms (1)

- Several algorithms can be used to allocate memory for a created process:
  - **First fit** – the memory manager scans along the list of segments until it finds a hole that is big enough
  - **Next fit** – It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time

Memory Management Algorithms (2)

- **Best fit** – searches the entire list, from beginning to end, and takes the smallest hole that is adequate
- **Worst fit** – always takes the largest available hole, so that the new hole will be big enough to be useful
- **Quick fit** – maintains separate lists for some of the more common sizes requested

End

Week 06 – Lecture 1

References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
Prentice-Hall, Inc.