Lecture 7 (Memory Managment & Paging)

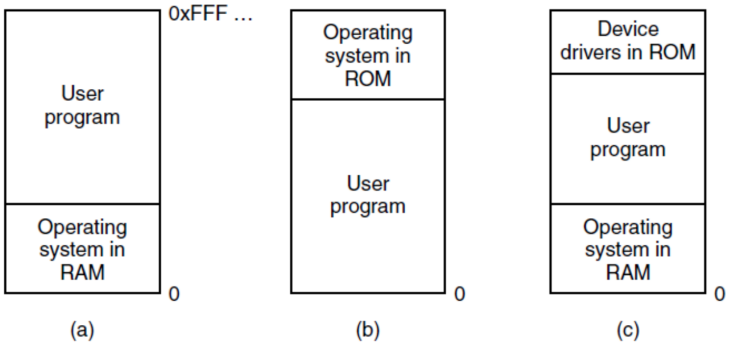
Memory hierarchy:

- Cache (very fast, expensive, volatile)
- Main memory (medium speed, medium price, volatile)
- Magnetic or solid-state disk storage (slow, cheap, nonvolatile)

Memory manager - part of operating system that manages the memory hierarchy. Keep track of using parts, allocate memory between processes, deallocate

No memory abstraction:

Impossible to have two or more running programs at the same time. OS can be at: bottom of memory in RAM(mainframes & microcomputers); top of ROM(handheld & embedded systems); device drivers at the top of ROM and rest of the system at the RAM(early personal computers).



Running multiple programs without memory abstraction: OS needs to save the entire content to the disk and run next program. As long as only one program in the memory at a time then no conflicts - **swapping**.

Static relocation: when program is loaded at address A, the constant A added to every program address during the load process. (not very general solution,

required extra info)

Memory abstraction

Two problems: protection(how address memory to not crush the whole system) & relocation(how to share memory & CPU between processes)

Address space - set of addresses that a process can use to address memory. Every process have its own address space. (abstraction for memory)

Dynamic relocation - mapping each process' address space onto different part of physical memory.

Registers of CPU: **Base register** contains physical address of beginning of program, **limit register** contains its length. With use of this 2 registers programs loaded to **consecutive memory locations** (without relocation during loading).

Every time when process referencing memory, CPU hardware adds base value to address & checks whether it's greater or equal than limit value, to abort process if true.

Disadvantage of base&limit registers: slow addition

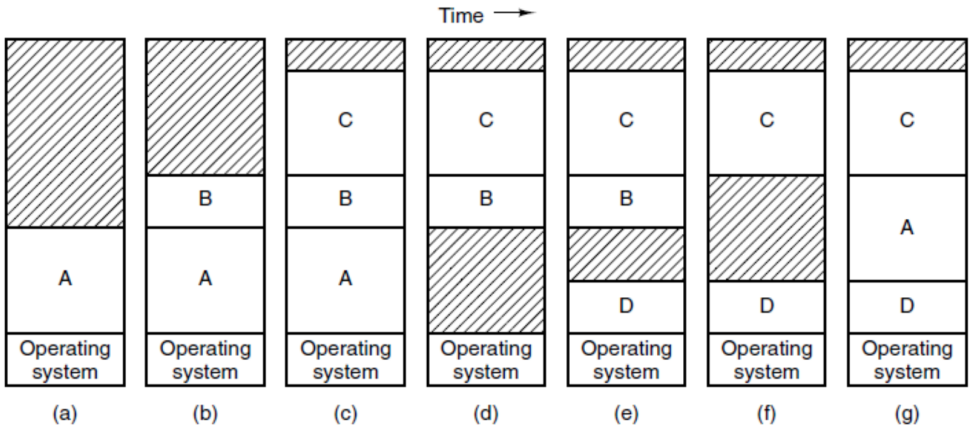
Running process may require more memory than physically available.

Approaches to solve:

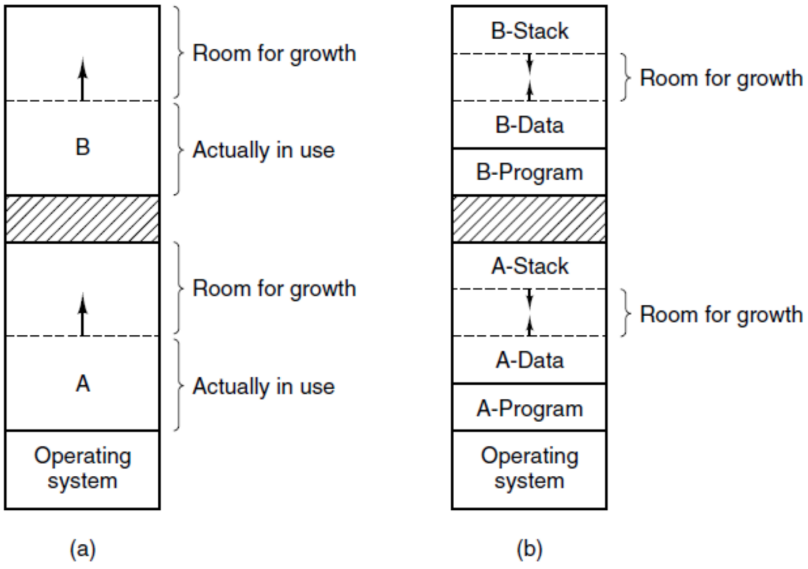
- **Swapping** - idle processes stored at the disk, brought to the memory when they needed
- **Virtual memory** - programs run even when they are only partially in main memory

---

Swapping



Memory compaction - combining holes after swapping to the one big hole by moving other processes downwards.

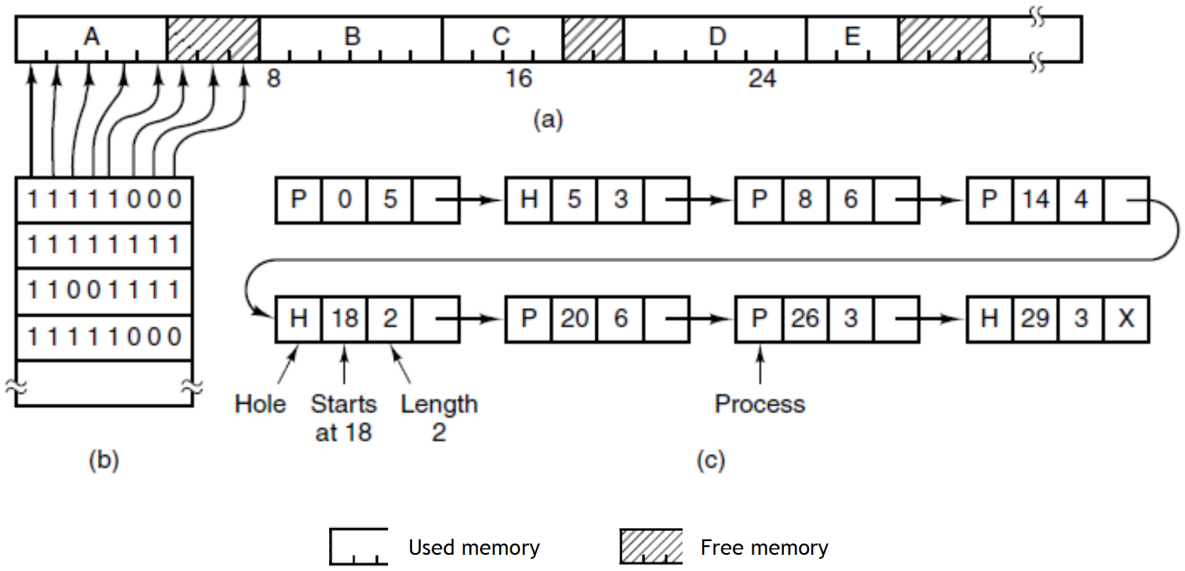


(a) allocating for growing data segment (b) allocating for growing stack & data segment

Memory Management with Bitmaps

Two ways to keep track of memory usage: **bitmaps** & **free lists**.

With a **bitmap**, memory is divided into allocation units as small as a few words and as large as several kilobytes.



(a) memory (b) bitmap (c) list

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.

memory of 32n bits ⇒ n map bits for allocation unit = 4 bytes = 32 bits

The size of the bitmap depends only on the size of memory and the size of the allocation unit

k-unit allocation process ⇒find k consecutive 0

Memory Management with Linked List

Linked list of allocated and free memory segments each element consist of: H/P, start, length,pointer to next element.

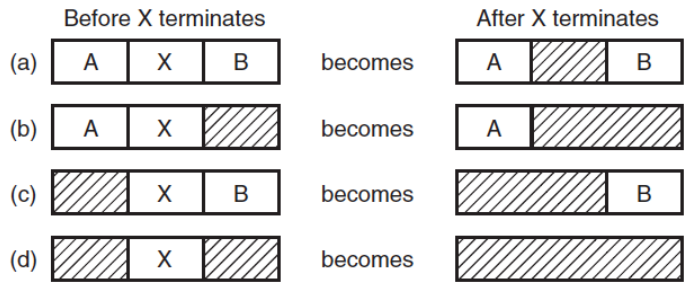


Figure 3-7. Four neighbor combinations for the terminating process, *X*.

Memory management algorithms(allocate memory for created process):

- First fit (find first big enough hole for process in the list of segments) - fast
- Next fit (work as first fit except that it starts from place where it stop at last time)
- Best fit (search whole list to find smallest acceptable hole)
- Worst fit (always takes a largest possible hole)
- Quick fit (maintains separate lists for some of the more common sizes requested)

Virtual memory

Overlay systems: **Overlay -** little pieces in which programs are splitted. Program started → overlay manager loaded into memory → load & run overlay 0 → load overlay 1 above overlay 0 (if space) or on top of overlay 0 (if no space)→...

Virtual memory - each program has its own program space, which is broken up into chunks called **pages.** Each page is a contiguous range of addresses and is mapped onto physical memory. If page not in physical memory while executing → OS get missing part → re-execute the program with missing part.

Paging

Paging - memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory. On any computer, programs reference a set of memory

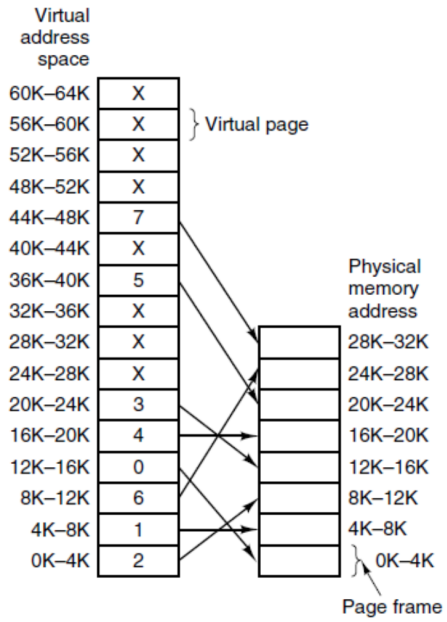
addresses which can be generated using indexing, base registers, segment registers, and other ways

Program-generated addresses are called **virtual addresses** and form the **virtual address space**

Virtual address = 4-bit page number(index in page table), 12-bit offset(location difference between the byte address you want and the start of the page) for 4k page.

MMU(memory management unit) - maps virtual addresses onto physical memory address.

Page frames - corresponding units in the physical memory. **Page table** - table which represents relation between virtual addresses and physical memory addresses.



4K= 4096, 8192→24576, 20500=(20480+20)→(12288 + 20) = 12308

Present/absent bit keep track of presenting of page in physical memory.

Page fault - situation when the program references an unmapped address, the MMU causes the CPU to trap to the OS. The OS swaps the page that is needed with a little used frame, changes the map and restarts the instruction.

Page table entry structure:

- page frame number
- present/absent bit: if 0 ⇒ not in memory access it leads to page fault, if 1 ⇒valid entry
- Protection bits (r/w — one bit or r/w/x — three bits): r - read, w - write, x - execute
- Modified bit (dirty): keep track of page usage. 1 - the processor writes to this memory.
- referenced bit: 1 - referenced for r/w
- caching bit: caching to be disabled for the page, important for pages that map onto devices

Two issues: mapping should be fast & large page table

TLB - translation lookaside buffer - associative memory - small hardware device for mapping virtual addresses to physical addresses without going through the page table. Typically small number of pages are read. Each entry contain: present/absent bit, page number, M bit, protection, page frame

check if virtual page number in TLB → if valid match & protection doesn't violate → page frame taken directly from TLB.

if not in TLB → ordinary page table lookup → removes one entry from TLB & replaces it with the page table entry just looked up → when entry deleted from TLB → only the M bit is copied back into the page table.

Software TLB management: The main gain is a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance.

Misses: soft miss(referenced page not in TLB) & **hard miss**(page not in TLB neither in memory)

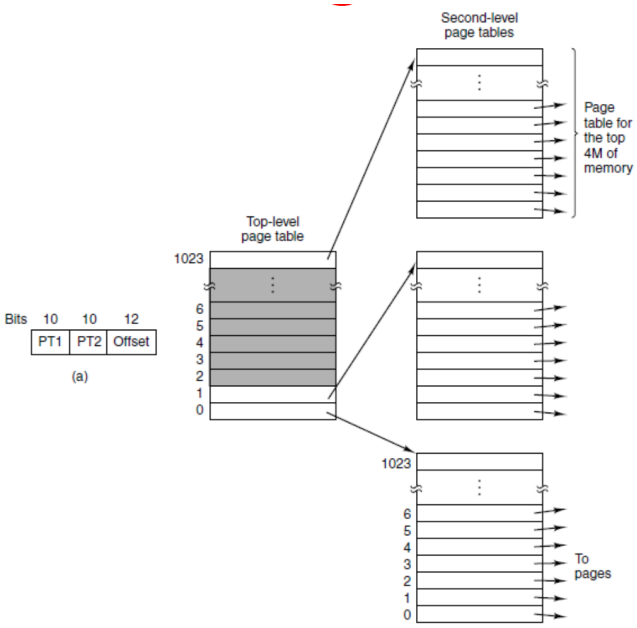
Page memory walk - process of looking up the mapping in the page table hierarchy.

Unsuccessful page memory walk:

- Minor page fault(page in memory but not in process' page table)
- Major page fault(page needs to be brought from disk)

- Segmentation fault(access invalid address, no mapping needs to add to TLB)

Multilevel page table:



The entry located by indexing into the top level page table yields the address or the page frame number of a second-level page table. The PT2 field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

Page directory - top-level page table, contains pointers to page tables.

Inverted page table - one entry per page of virtual address space, so the size is proportional to physical memory. Entry contains: pair (process ID, virtual page address) and refers to a page frame