Lecture 7: Introduction to virtual memory

Mythili Vutukuru

IIT Bombay

Why virtualize memory?

- Because real view of memory is messy!
- Earlier, memory had only code of one running process (and OS code)
- Now, multiple active processes timeshare CPU
  - Memory of many processes must be in memory
  - Non-contiguous too
- Need to hide this complexity from user

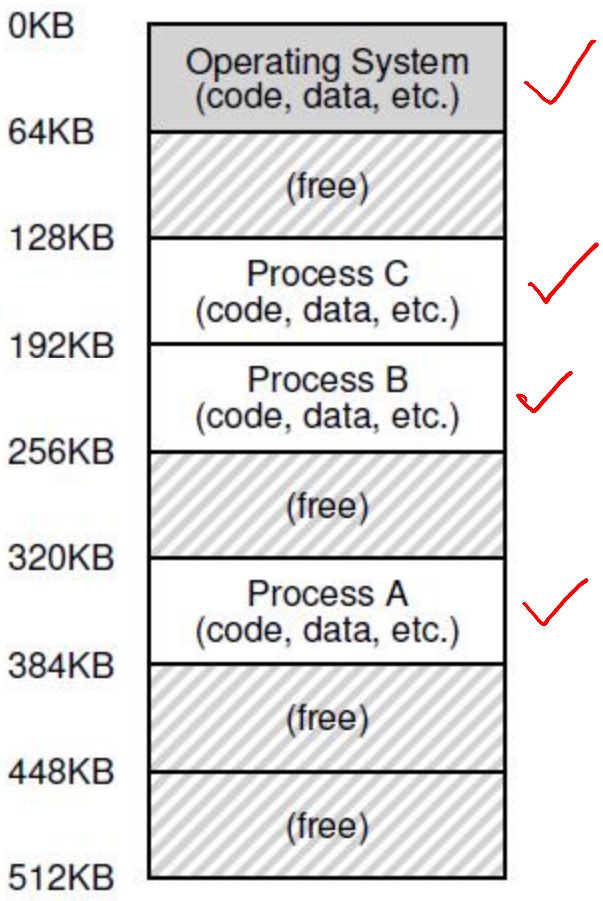


Figure 13.2: **Three Processes: Sharing Memory**

Abstraction: (Virtual) Address Space

- Virtual address space: every process assumes it has access to a large space of memory from address 0 to a MAX
- Contains program code (and static data), heap (dynamic allocations), and stack (used during function calls)
  - Stack and heap grow during runtime
- CPU issues loads and stores to virtual addresses

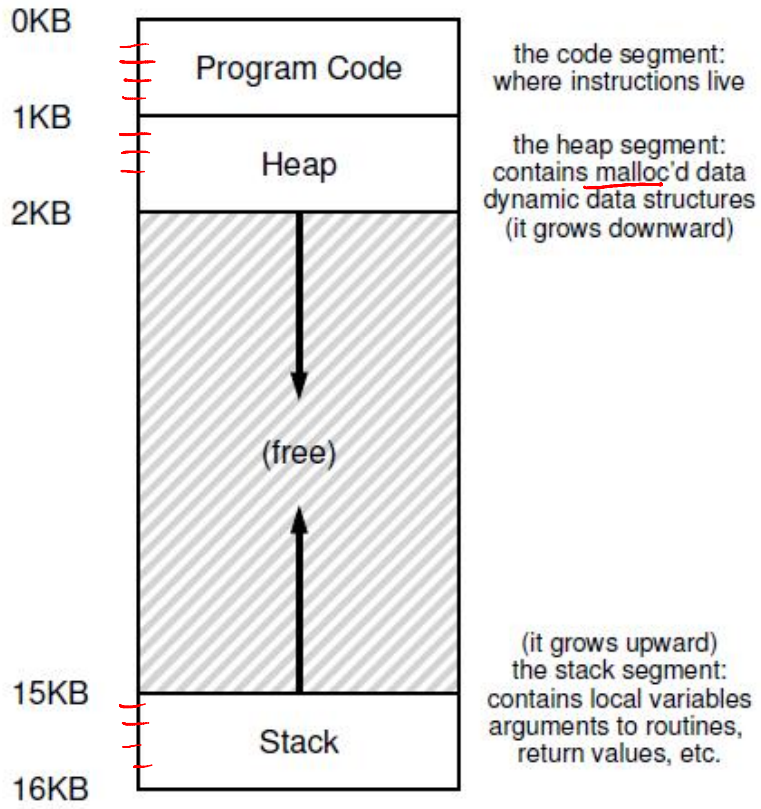
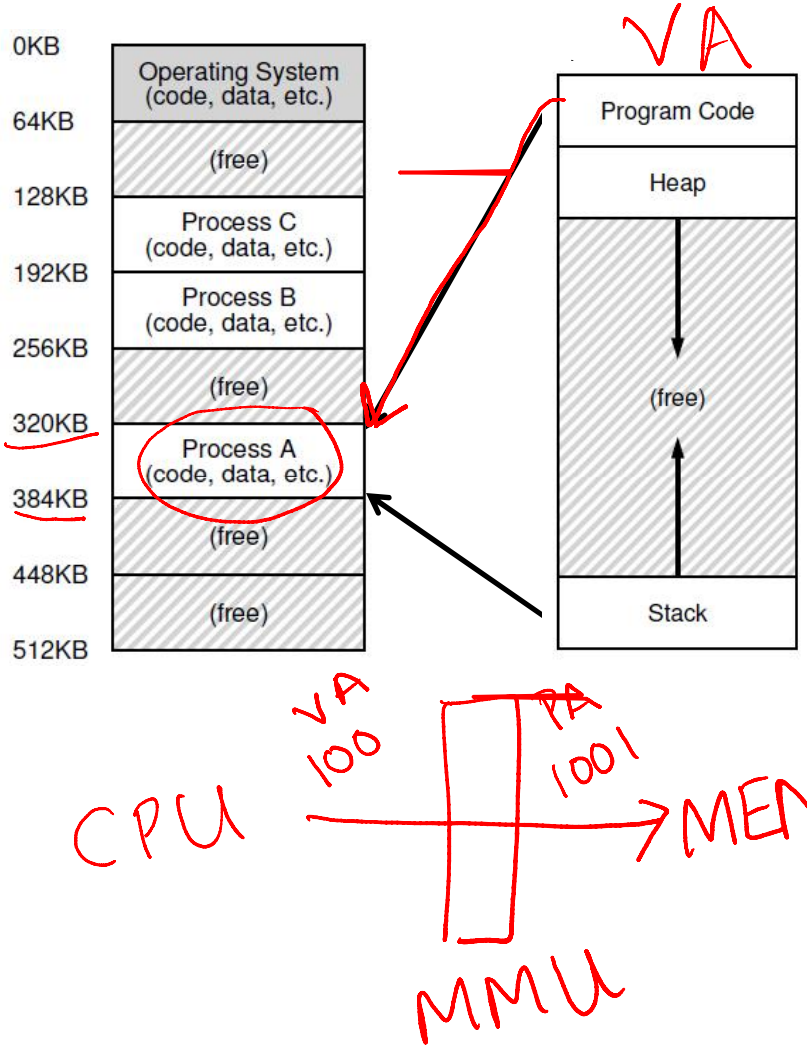


Figure 13.3: **An Example Address Space**

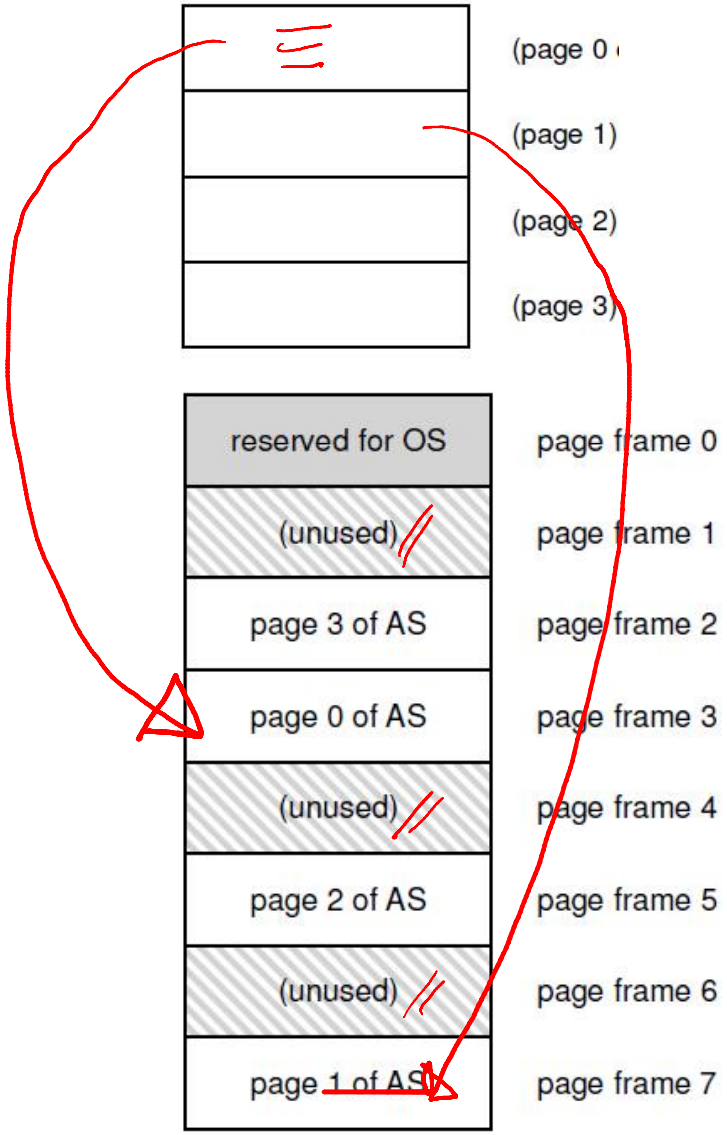
How is actual memory reached?

- Address translation from virtual addresses (VA) to physical addresses (PA)
  - CPU issues loads/stores to VA but memory hardware accesses PA
- OS allocates memory and tracks location of processes
- Translation done by memory hardware called Memory Management Unit (MMU)
  - OS makes the necessary information available



Example: Paging

- OS divides virtual address space into fixed size pages, physical memory into frames
- To allocate memory, a page is mapped to a free physical frame
- Page table stores mappings from virtual page number to physical frame number for a process (e.g, page 0 to frame 3)
- MMU has access to page tables, and uses it to translate VA to PA



Goals of memory virtualization

- Transparency: user programs should not be aware of the messy details
- Efficiency: minimize overhead and wastage in terms of memory space and access time
- Isolation and protection: a user process should not be able to access anything outside its address space

How can a user allocate memory?

- OS allocates a set of pages to the memory image of the process
- Within this image
  - Static/global variables are allocated in the executable
  - Local variables of a function on stack
  - Dynamic allocation with `malloc` on the heap

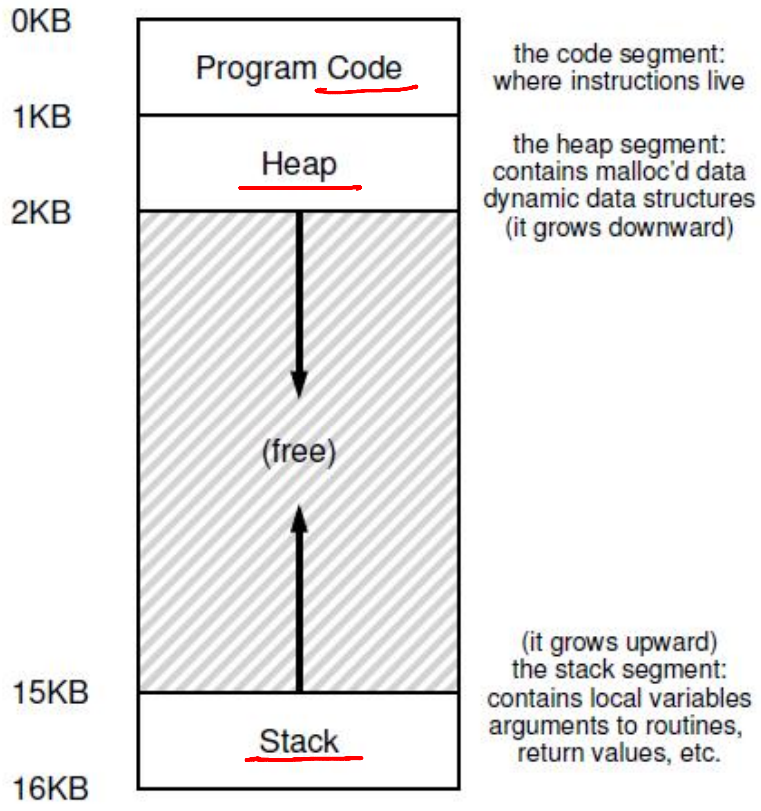
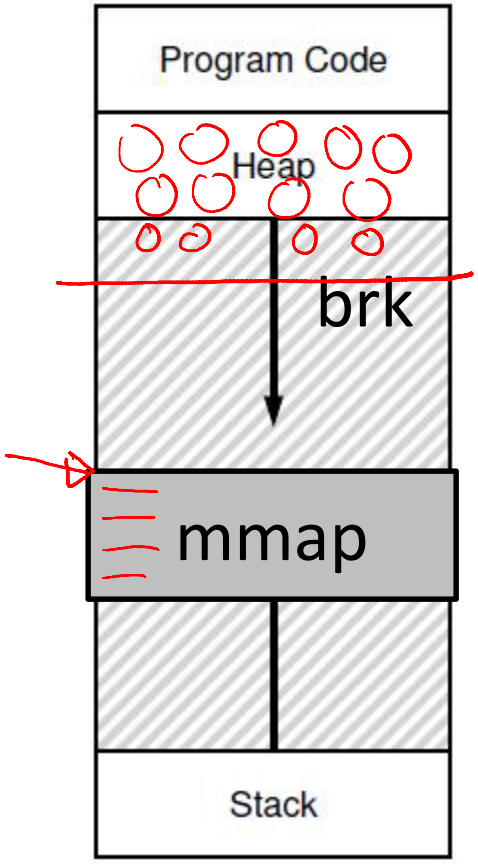


Figure 13.3: **An Example Address Space**

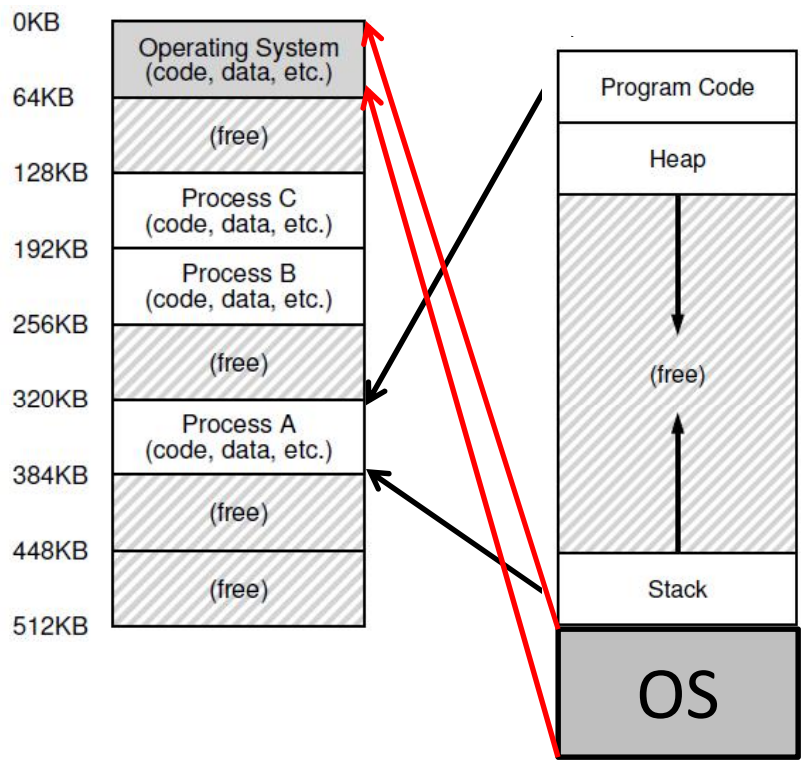
Memory allocation system calls

- `malloc` implemented by C library
  - Algorithms for efficient memory allocation and free space management
- To grow heap, libc uses the `brk/sbrk` system call
- A program can also allocate a page sized memory using the `mmap()` system call
  - Gets "anonymous" page from OS



A subtle point: what is the address space of the OS?

- OS is not a separate process with its own address space
- Instead, OS code is part of the address space of every process
- A process sees OS as part of its code (e.g., library)
- Page tables map the OS addresses to OS code



How does the OS allocate memory?

- OS needs memory for its data structures
- For large allocations, OS allocates a page
- For smaller allocations, OS uses various memory allocation algorithms (more later)
  - Cannot use libc and `malloc` in kernel!