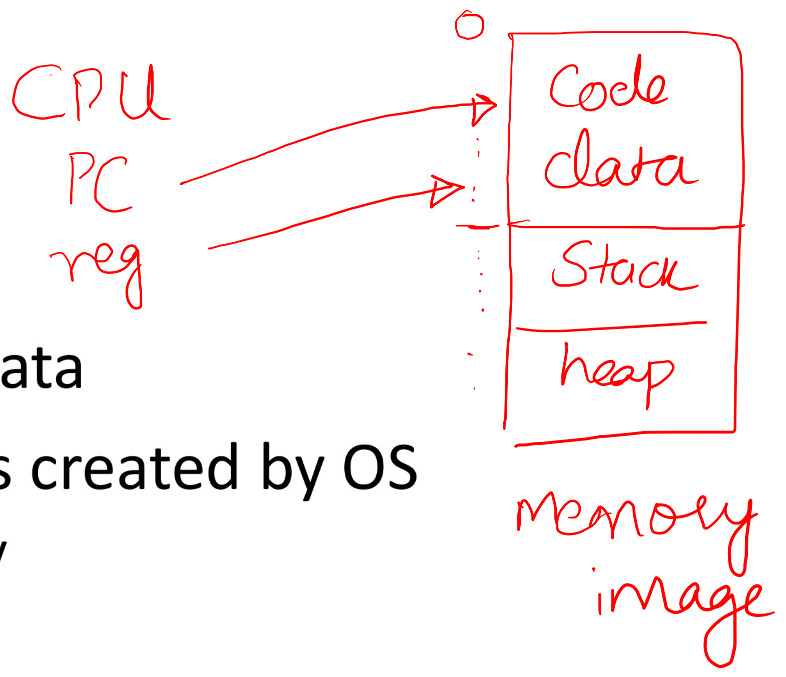
Design and Engineering of Computer Systems

Lecture 11: Memory management in OS

Mythili Vutukuru

IIT Bombay

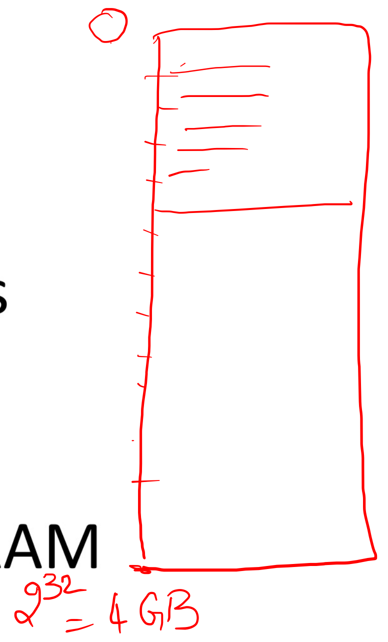
Recap: Running a program



- User program executable = code + compile-time data
- When program is run, memory image of process is created by OS
  - Code + data from executable loaded into main memory
  - Extra memory allocated for stack and heap
- Code+data in memory image assigned virtual addresses starting from 0
- CPU begins executing process, fetches code/data using virtual addresses
- OS knows actual memory locations of program code+data
  - Maintained in a data structure called page table
  - Used to translate virtual addresses to physical addresses
- We will understand this memory management done by OS in more detail

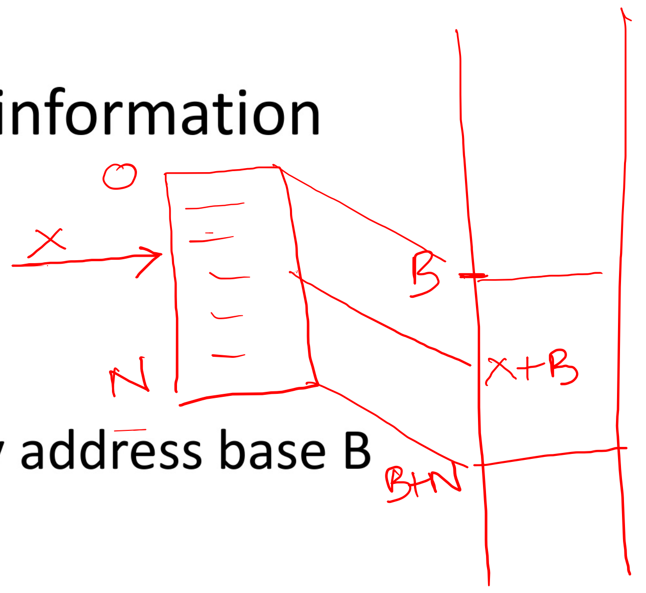
Virtual address space

- Virtual address space: set of virtual addresses available to a process
  - 0 to a max value (2^32 = 4GB in 32-bit OS)
  - Process can access code/data in its virtual address space
- Physical address space of a system: physical memory addresses in RAM
- Why do we need virtual addresses?
  - Physical address of code/data not known at compile time
  - Memory allocated to a process can be non-contiguous, hide this detail from user
  - Can control which memory a process can "see", useful for isolation
- Addresses in CPU registers and pointer variables = virtual addresses
  - User only sees virtual addresses, not physical addresses
- But memory hardware can access data using physical addresses only



Memory allocation and address translation

- OS allocates physical memory to a process, has translation information
  - It knows which virtual address maps to which physical address
  - Memory allocation method determines address translation logic
- Simplest form of memory management: base and bound
  - Place entire memory image [0,N] contiguously starting at memory address base B
  - Virtual address X translated to physical address B+X
  - Access to virtual addresses beyond N will not be permitted
- Who does address translation? A piece of hardware called Memory Management Unit (MMU)
  - Every memory access by CPU is translated by MMU before reaching RAM
  - OS provides information to MMU (e.g., base and bound) for translation/error checking

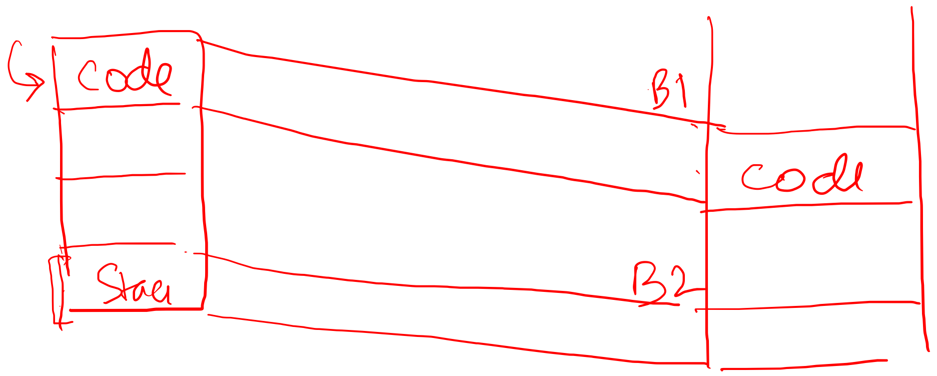


Role of OS vs MMU

$$\begin{tikzcd}
CPU \arrow[r, "VA"] & MMU \arrow[r, "PA"] \arrow[d, leftrightarrow] & RAM \\
 & OS &
\end{tikzcd}$$

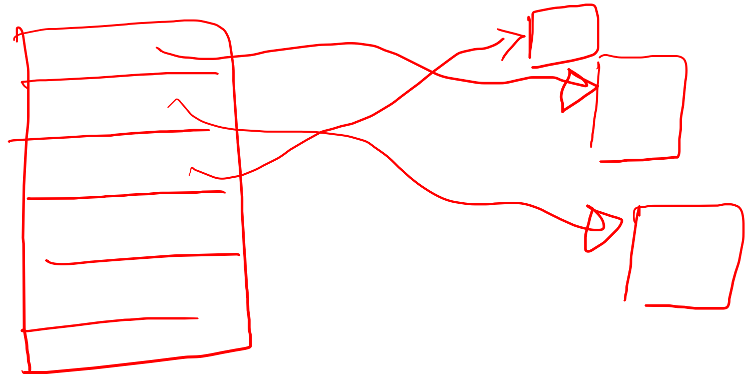
- OS allocates memory, builds translation information of process
  - But OS does not do the actual address translation on every memory access
- When process is context switched in, translation information is provided to MMU
  - Once user code starts running on CPU, OS is out of the picture (until a trap)
- CPU runs process code, accesses code/data at virtual addresses
  - Virtual addresses translated to physical addresses by MMU
  - RAM is accessed using physical addresses
- MMU raises a trap if there is any error in the address translation
  - CPU executes trap instruction, OS code runs to handle the error
- OS gives new information to MMU on every context switch

Segmentation



- Older way of memory management: generalized base and bounds
- Each segment of the program (code, data, stack,..) is placed separately in memory at a different base
    - Every segment has a separate base and bound
- Virtual address = segment identifier : offset within segment
- Physical address = base address of segment + offset within segment
    - Bound of a segment checked for incorrect access
- Multiple base, bound values stored in MMU for translation
- MMU throws a segmentation fault if a segment accessed beyond bound
    - Program fault, traps to OS to handle error, may terminate process

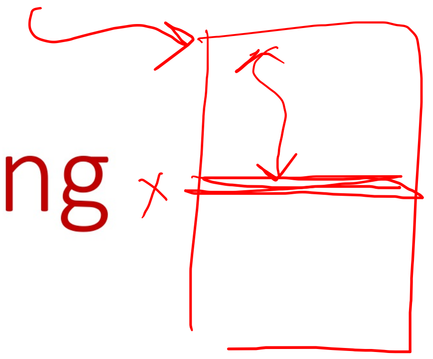
Paging



- Paging: widely used memory management system today
  - Virtual address space divided into fixed size pages
  - Each page is assigned a free physical memory frame by OS
- Memory allocation is at granularity of fixed size pages (e.g., 4KB)
  - Avoids external fragmentation (no wastage of space between pages)
  - Internal fragmentation (space may be wasted inside partially filled page)
- Page table maps logical page numbers to physical frame numbers
  - One page table per process
  - Maintained by OS, part of PCB of process
- Location of page table of currently running process known to MMU
  - Written into special CPU register, updated on context switch/page table change

0 → . . .
1 → . . .
2 → . . . .

Address translation using paging

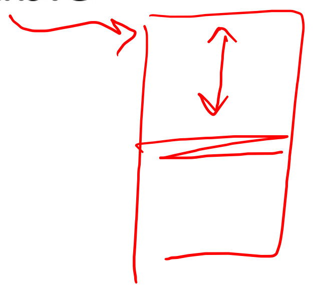


addr = Page# | offset

frame

- Address translation performed by MMU
  - Virtual address accessed by CPU = page number | offset within page
  - Find frame number corresponding to page number by looking up page table
  - Physical address = physical frame number | offset within page
- Example: 32-bit CPU, 4 KB pages
  - 32 bit virtual address = 20 bit page number | 12 bit offset within page
  - Page table maps 2^20 pages to physical frame numbers
  - Physical address = Physical frame number | 12 bit offset within page

$$\underbrace{20 \mid 12}_{32}$$



$$2^{12} = 4\text{ KB}$$

- Overhead of paging: before doing actual memory access, MMU must do extra memory access for page table mapping
  - How to avoid this?

$$\begin{tikzcd}
\text{CPU} \arrow[r, "\text{VA}"] & \text{MMU} \arrow[r, "\text{PA}"] \arrow[r, bend right] & \text{RAM}
\end{tikzcd}$$

$$\begin{tikzcd} CPU \arrow[r] & VA \arrow[r] & MMU \\ & & TLB \arrow[d, bend right, "PA"'] \\ & & RAM \end{tikzcd}$$

Translation Lookaside Buffer (TLB)

$VA \rightarrow PA$

- Overhead of memory translation: every memory access preceded by extra memory accesses to read page table
- To reduce this overhead, MMU caches the most recent translations in translation lookaside buffer (TLB)
  - Small cache within MMU to store page number to frame number mappings
  - LRU policy to evict entries if TLB is full (locality of reference)
- TLB only caches address translations, not actual memory contents
  - Different from CPU caches that cache actual memory contents
  - If TLB hit, physical address is ready, fetch memory contents in one memory access
  - If TLB miss, extra memory access for page table access also needed
- TLB flush on context switch: mappings cached in TLB change

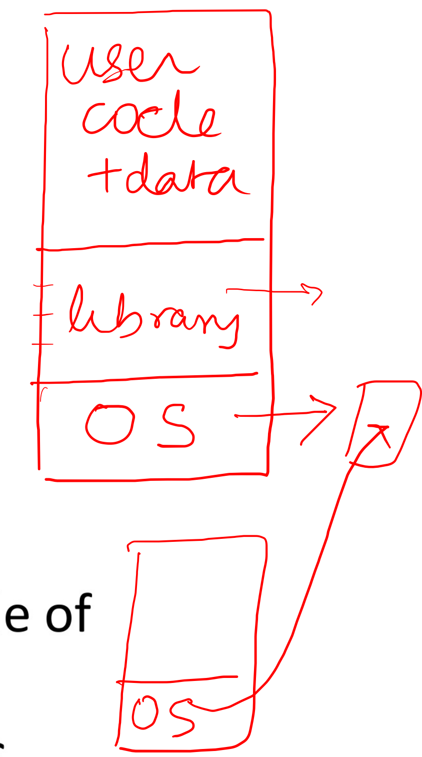
$$\begin{tikzcd}\text{CPU} \arrow[r, "\text{VA}"] & \text{cache} \arrow[r] & \text{MMU} \arrow[r] \arrow[rr, bend left] & \text{TLB} \arrow[r] & \text{RAM}\end{tikzcd}$$

What happens on a memory access?

- CPU has requested data (or instruction) at a certain memory address
  - The address is looked up in various levels of CPU caches (L1, L2, L3)
  - If address found, cache hit, data item is returned to CPU
  - If cache miss, CPU must fetch data from main memory via MMU
  - MMU looks up TLB to find frame number corresponding to page number
  - If TLB hit, physical address is found, main memory is accessed to fetch data (data cached in CPU for future access)
  - If TLB miss, MMU first accesses page table in main memory, computes physical address (translation added to TLB cache), then accesses main memory again to fetch data (data cached in CPU for future access)
- High cache hit rates and high TLB hit rates are important for good performance of the system

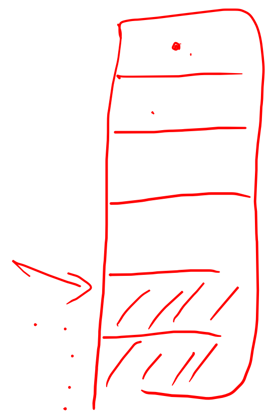
Revisiting process virtual address space

- What should virtual address space/page table of process have? Any memory that the process needs to access during its execution
  - Its own memory image: code, data, stack, heap
  - Other common memory it needs to access: shared language libraries, OS
  - Why? MMU only allows access to physical memory that is mapped in page table of process at some virtual address
- OS binary image (kernel code, data) is mapped into the virtual address space of every process
  - OS code/data assigned virtual addresses (high addresses not used by user code)
  - OS virtual addresses are mapped to physical addresses of OS via page table
  - There is only one copy of OS code/data in RAM (loaded during system bootup)
  - Page tables of all processes have same mappings to same OS physical addresses
- Why is this done? Easy to jump to OS code during a trap



Isolation and security

- How is OS code/data protected from illegal access by user?
- Page table has permissions for every memory page
  - Whether read/write or read-only (code pages are read-only)
  - Whether page can be accessed in user mode or kernel mode
- Page table mappings for OS code are protected to allow access only in kernel mode
  - User program in user mode cannot jump to high virtual addresses of OS code
  - CPU in kernel mode (after trap instruction) can access OS code/data
- MMU ensures that user programs cannot access any memory beyond what is visible in the user-accessible part of virtual address space



Summary

- In this lecture:
  - Memory management and address translation by OS, MMU
  - Segmentation, paging
  - OS is part of virtual address space of all processes
- Try to understand the virtual address of processes in your system.
  - In Linux systems, information about the virtual address space of a process can be found under /proc/PID/maps. You can see mappings for user code and data, shared libraries, and so on.