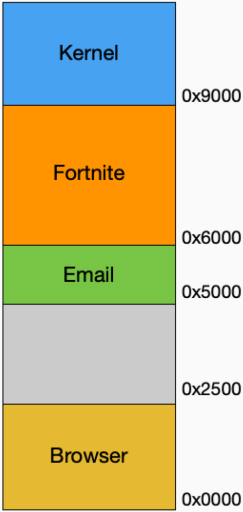
Virtual Memory & Paging

Azza Abouzied

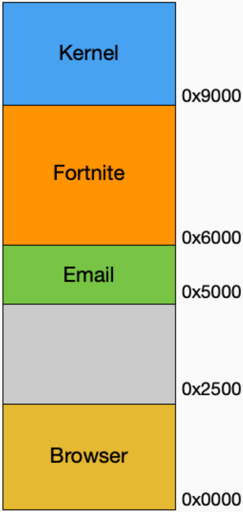
From Last Class: Why “Virtual” Memory?

Can we keep everything in memory?



Life would be great if we didn't have to pay 0.15ms to access disk. Let's keep everything in memory!

Problem 1: The Jump Issue



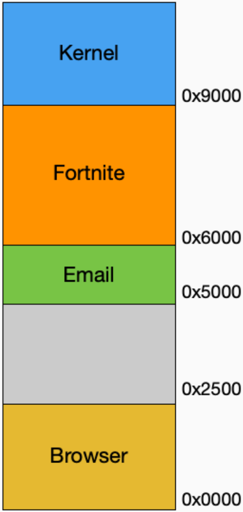
What do the following mean?

Browser: jmp 1000

Fortnight: jmp 1000

space

Problem 1: The Jump Issue



What do the following mean?

Browser: jmp 1000

Fortnight: jmp 1000

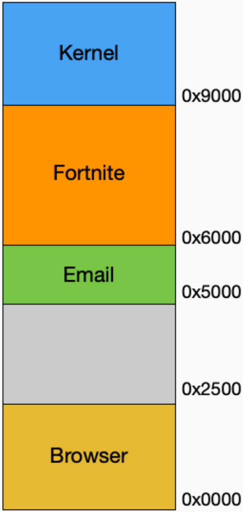
space

A work around: static relocation

When loading at adr 0x7000, add 0x7000 to every address in the executable code!

- Slow loading
- Programs need to define what is relocatable and what is not.

Problem 2: The Protection Issue



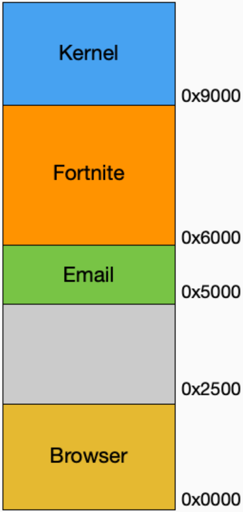
Writes external to my memory

Email's address offsets can range from 0 to 0x1000.

Email writes to address offset 0x1001!

space

Problem 2: The Protection Issue



Writes external to my memory

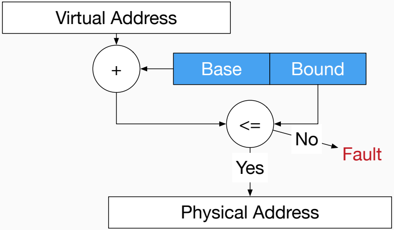
Email's address offsets can range from 0 to 0x1000.

Email writes to address offset 0x1001!

space

A work around: address space & base+bound

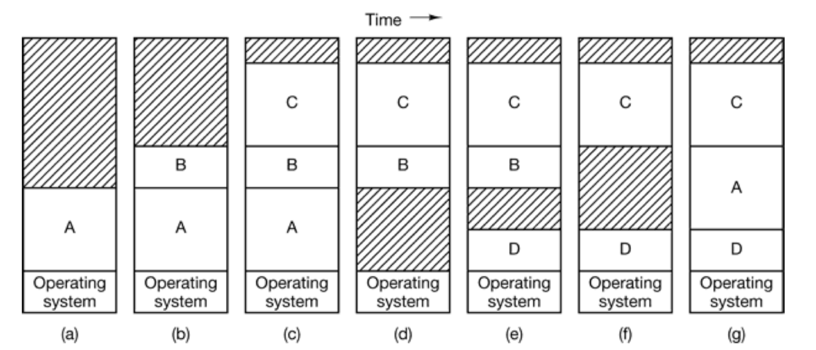
space



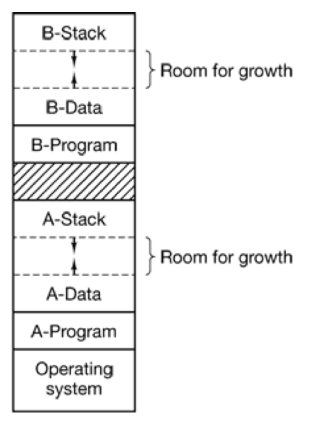
Problem 3: More processes than can fit in memory!

A work around: Swapping

Bring in entire process from disk, run it then put it back on disk



Problem 4: A process grows!



Work arounds?

Give more than needed!

space

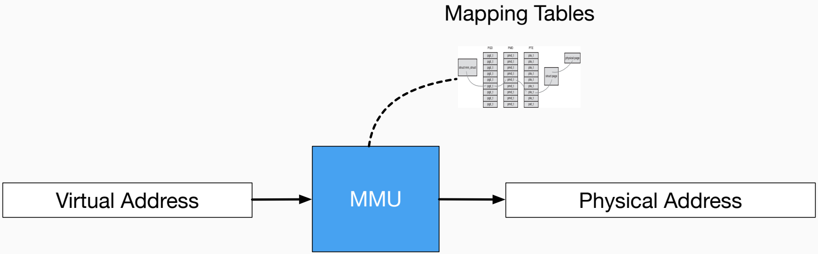
Problem 5: What if a process just needs more memory than your entire RAM?

The seeds of virtual memory

What do you want from your virtual memory system?

1. Simplicity: Processes get a *flat linear address space*. Access addresses from 0 to 2^{32} or 0 to 2^{64} depending on architecture!
2. Flexibility or Deception: Processes need to move in and out of memory with partial parts in memory and other parts on disk. Satisfy processes that require more memory than you have!
3. Efficiency: 80/20 rule. Most of what you need is already on memory. Occasionally, you will go to disk to get the rest!

The seeds of virtual memory



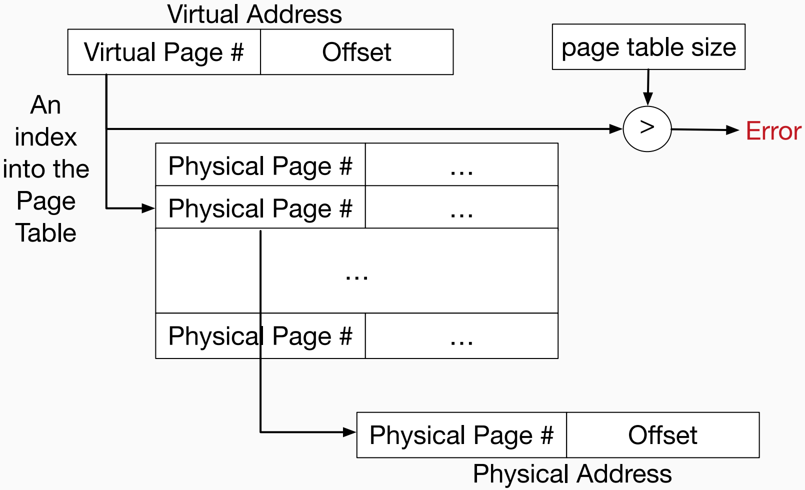
The kernel’s job is set up these mapping tables *for each process*. Hardware handles the mapping table lookup on every virtual address operation.

But what is the right granularity of mapping? a byte-to-byte? a whole segment?

Most operating systems opt for paging. Each virtual address maps to a page address and an offset within it. The mapping tables are called page tables.

How does paging work?

The simplified view of paging



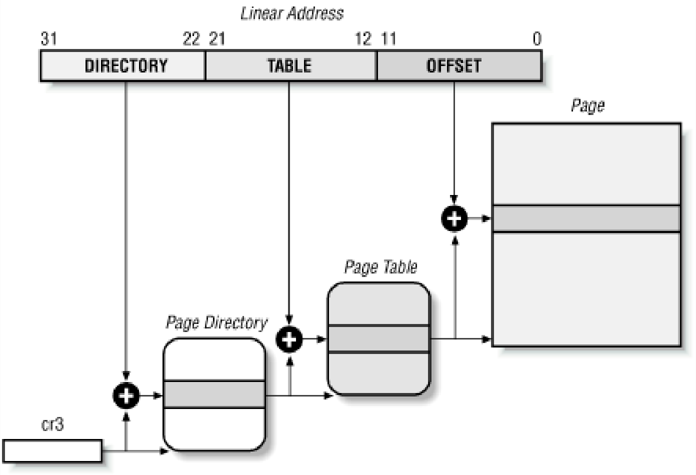
What are the right numbers for?

Page size: On 32-bits, this is typically 4KB. Can also be 8KB on 64-bit architectures.

Bits in the offset: If a page is 4KB we have 4096 unique bytes we should be able to address so $2^{12} = 4096$. 12 bits.

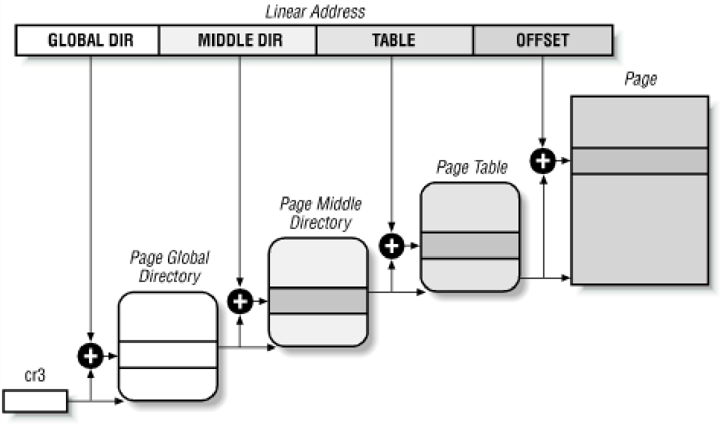
Page table size: If you want to support flat addresses on 32- or 64-bit machines then you need to map every virtual address to a physical address. On 32-bit: $32 - 12 = 20$, the table should have 2^{20} entries. With 2^{52} entries on 64-bit with 4KB pages, you can't fit the page table for a process in memory!

Multi-level paging



How does multi-level paging help keep the size of the page tables small?

Multi-level paging



Linux uses three-level paging. How does it help keep the size of the page tables small?

Back to Basics: How a user process gets memory?

Basic rules

1. Process requests for memory are not urgent.
2. The kernel defers allocating dynamic memory.
3. The kernel must catch all addressing errors of a user process.

When a user process asks for memory, it doesn't get additional page frames; instead, it gets the right to use a new range of linear addresses, which become part of its address space. This interval is called a memory region (area).

Back to Basics: When do we get new memory regions?

1. On process creation. Not all executable code needs to be loaded right away

2. Same process loads a new program: old memory regions released, new ones assigned.

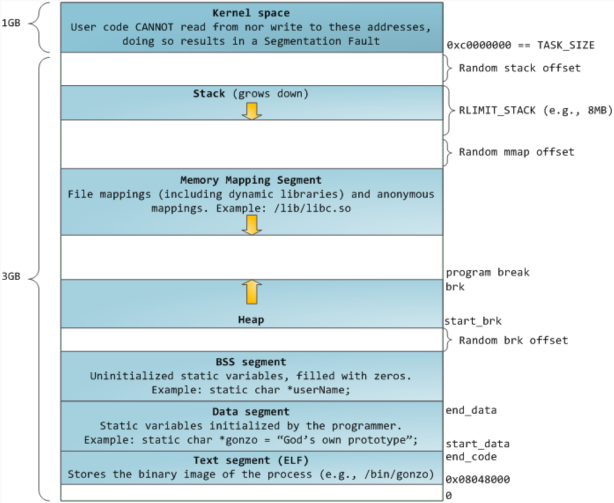
3. On **memory mapping** a file: mmap()

4. On pushing data to the user mode stack until the stack is full and that memory region needs to be expanded.

5. On creating a shared memory region to share data with other processes.

6. On expanding the heap through malloc()

Back to Basics: Memory Layout of Memory Regions



Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

Stack (grows down)

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

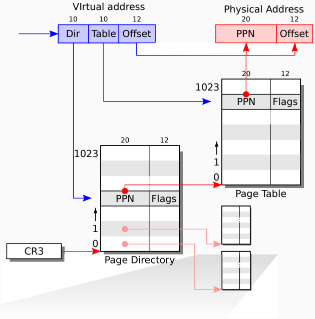
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;

Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = “God’s own prototype”;

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

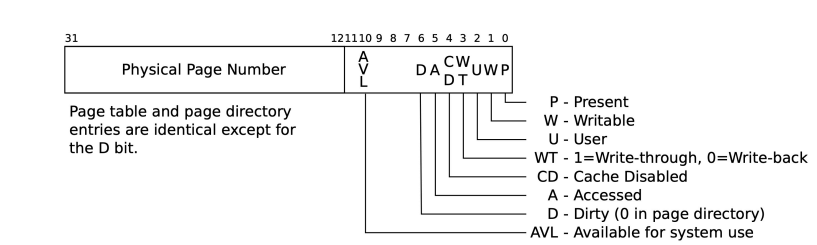
What does this mean for multi-level paging? Entire ranges of virtual addresses are not used and thus top-level pointers in the directory table point to NULL!

The Page Tables



1. What is the size of a page frame?
2. What is the size of page table in bytes? Why?
3. How many entries are in each page table?
4. What is the size of each page table entry?
5. Assume a process uses all virtual addresses, how much space is used by the page tables?
6. How many bits for the physical page number?
7. What happens with the remaining bits?

The Page Table Entry on x86



Page table and page directory entries are identical except for the D bit.

P - Present
W - Writable
U - User
WT - 1=Write-through, 0=Write-back
CD - Cache Disabled
A - Accessed
D - Dirty (0 in page directory)
AVL - Available for system use

x86 paging hardware understands flags in bits 0 to 8; bits 9 to 11 can be used by the operating system.

The Page Fault

When a page is accessed in a way that does not match its protection bits or the **present bit is 0**, paging hardware triggers a fault.

When the present bit is 0, the kernel has full control on how to use the remaining bits of the PTE.

Demand paging

When is Present bit = 0?

1. The page was never accessed by the process. The PTE has 0s.
    - the missing page maps to a file → load from file.
    - it is an *annonymous region* → allocate a free page.
2. The page was already accessed by the process, but its content is temporarily saved on disk.
    - the PTE is not filled with zeros → *It was swapped out, swap it back in*
    - the PTE stores the location on the disk of the swapped-out page.

Swapping

The kernel uses swap space on disk as an extension of RAM.

- It is transparent to the programmer.
- Expands address space available to a process.
- Expands space available to load processes.

Generally not a good thing! We want to keep swapping to a minimum. Why?

Do not confuse “standard swapping” with this “swapping” or “paging”.

Key Swapping Design Questions

1. Which kind of page to swap out?
2. How to distribute pages in the swap areas?
3. How to select the page to be swapped out?
4. When to perform page swap-out?

What to swap out?

Swapping applies only to:

1. Pages belonging to an anonymous memory region (for instance, a User Mode stack, heap) of a process

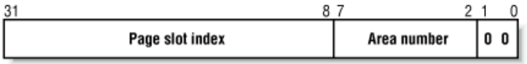
2. Modified pages belonging to a private (file) memory mapping of a process

3. Pages belonging to a shared memory region used for inter-process communication.

Why not memory-mapped files in general?

How to distribute pages in the swap areas?

- A swap area is broken into slots, the PPN tells us which slot.



- Store pages in contiguous slots to reduce disk seeks.
- Multiple swap areas across several storage devices.

How to select the page to be swapped out?

- Steal pages from the process having the largest number of pages in RAM.
- Typically swap out least recently used pages

When to perform page swap-out?

- When free memory falls below a pre-defined threshold.

Page Replacement Policies!

Questions?