Topic 13: Sharing Memory via Segmentation

Reading: 8.1 - 8.2.1
Next reading: 8.2.2 - 8.3

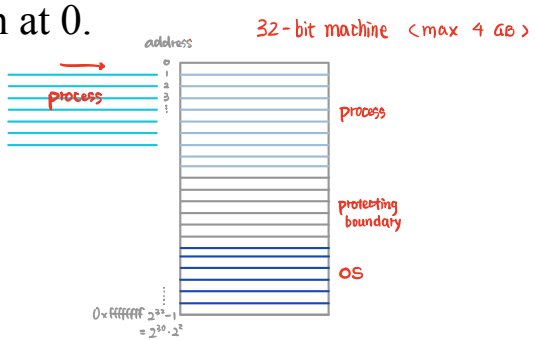
So far we know how one process uses memory. What if several processes want to share the computer's memory?

Issues in sharing memory:

- **Transparency**:
  - Want to let several processes coexist in main memory.
  - No process should be aware that memory is shared. Each must run regardless of the number and/or locations of processes.
- **Safety**: independent processes should not be able to corrupt each other.
- **Efficiency**: (both of CPU and memory) shouldn't be degraded badly by sharing. After all, the purpose of sharing is to increase overall efficiency.

Simple uniprogramming:

- Highest memory holds OS.
- Process is allocated memory starting at 0, up to the OS area.
- When loading a process, just bring it in at 0.



- Examples: early batch monitors where only one job ran at a time and all it could do was wreck the OS, which would be rebooted by an operator. Early personal computer OSes also operated in a similar fashion.

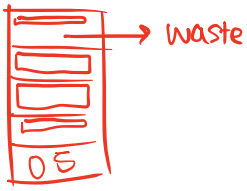
Static Software Relocation.

- Because several processes share memory, we can't predict in advance where a process will be loaded into memory.
- *Relocation* adjusts a program to run in a different area of memory. add x to address
  - *Linker* is an example of *static relocation* used to combine modules into programs. Compiler doesn't know memory addresses at compile-time; assumes object file starts at address 0.
  - Relocation techniques can also be used to allow several programs to share one main memory.

Simple multiprogramming with static software relocation, no protection:

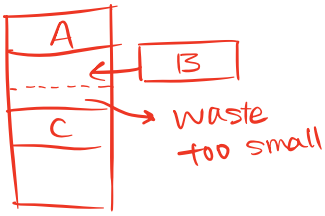
- Highest memory holds OS.
- Processes allocated memory starting at 0, up to the OS area.
- When a process is loaded, relocate it so that it can run in its allocated memory area (just like linker: linker combines several modules into one program, OS loader combines several processes to fit into one memory; only difference is that there are no cross-references between processes).

- Memory divided into *partitions,* each of which contains a process:



EASY

  - **Fixed partitions**: allocated at boot-time. All partitions are the same size. Leads to *internal fragmentation* (wasted space within partitions).



  - **Dynamic partitions**: allocated at run-time. Partitions have different sizes. Leads to *external fragmentation* (wasted space between partitions)

- Processes may share a partition by being copied out to disk (*swapped*).

- Problems:

  - Processes may destroy one another and/or the operating system (no protection between partitions).

  - Only one partition per process.

  - Processes cannot be easily moved to compact memory or swap (must swap back to same partition).

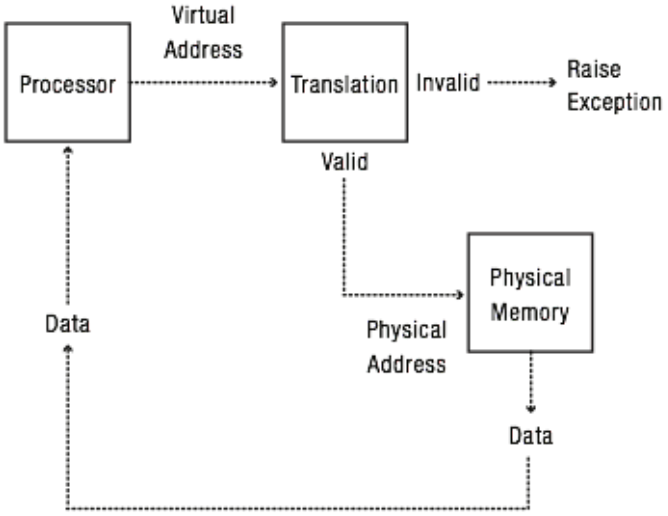
- Example: IBM OS/360

  - OS/MFT - fixed partitions

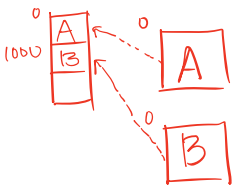
  - OS/MVT - variable partitions

  - Protection via 4-bit code for each 4-Kbyte chunk of memory plus 4-bit key for each process.

Dynamic memory relocation: instead of changing the addresses of a program when it's loaded, change the addresses dynamically during every reference. (Figure 8.1)



- Can't change addresses dynamically in software because difficulty in finding all pointers.
- Under dynamic relocation, each program-generated address (called a *logical* or *virtual* address) is translated in hardware to a *real* or *physical* address. This happens as part of each memory reference.



- Dynamic relocation leads to two views of memory, called *address spaces*. With static relocation we force the views to coincide. In some systems, there are several levels of mapping.