9. Address Spaces

One way to implement time sharing would be

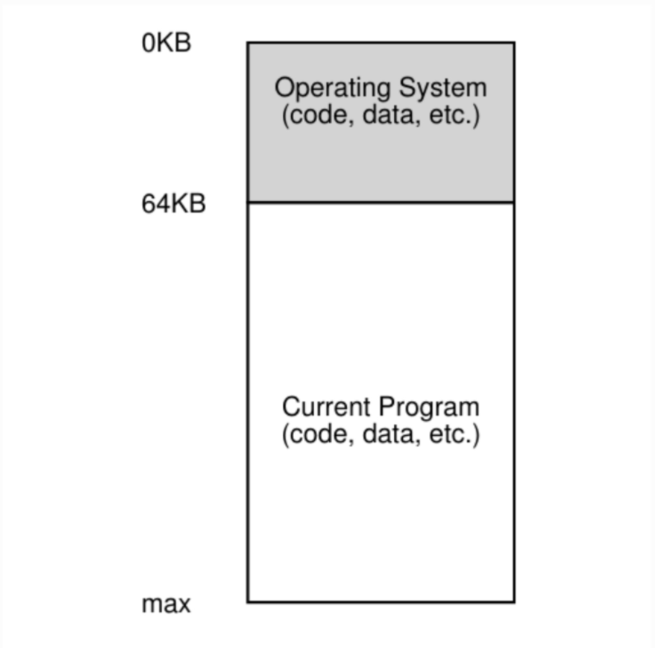

Figure 13.1: **Operating Systems: The Early Days**

- to run one process for a short while, giving it full access to all memory
- then stop it, save **all (physical memory)** of its state to some kind of disk
- load some other process's state
- run it for a while

=> implement some kind of "crude" sharing of the machine

Drawback : too slow, particularly as memory grows
saving the entire contents of memory to disk is brutally non-performant

=> rather do leave processes in memory while switching between them, allowing the OS to implement time sharing efficiently

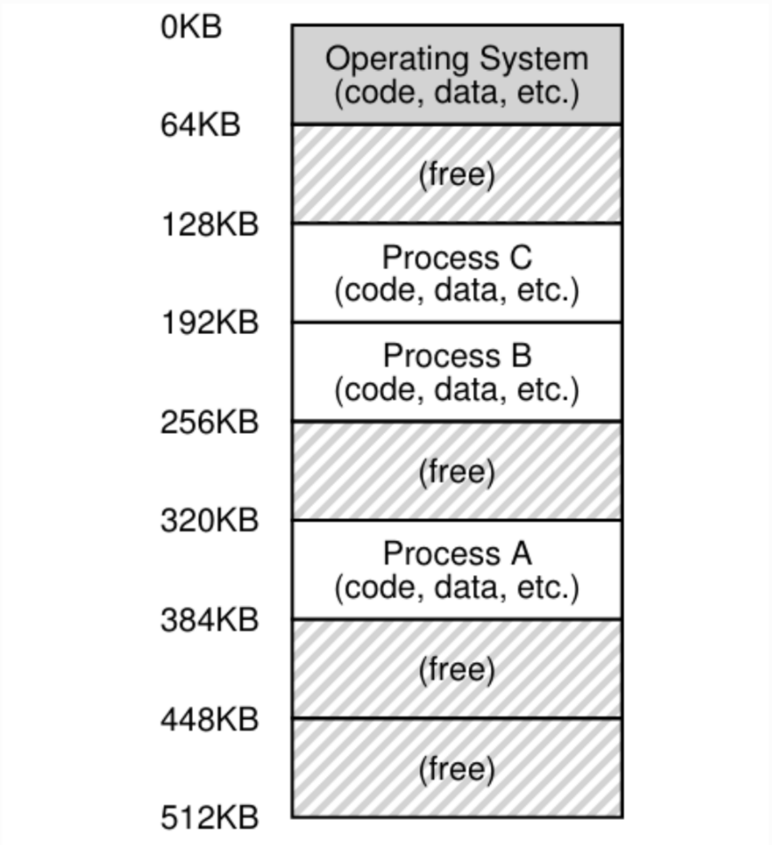
0KB
64KB
128KB
192KB
256KB
320KB
384KB
448KB
512KB
Operating System (code, data, etc.)
(free)
Process C (code, data, etc.)
Process B (code, data, etc.)
(free)
Process A (code, data, etc.)
(free)
(free)

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

allowing multiple programs to reside concurrently in memory
=> makes **protection** an important issue

Address space

: running program's view of memory in the system

- contains all of the memory state of the running program : code / stack / heap / etc...
  - stack :
    - keep track of where it is in the function call chain
    - allocate local variables & pass parameters & return values to and from routines
  - heap : dynamically-allocated, user-managed memory (ex. malloc() in C, new in C++ and java)

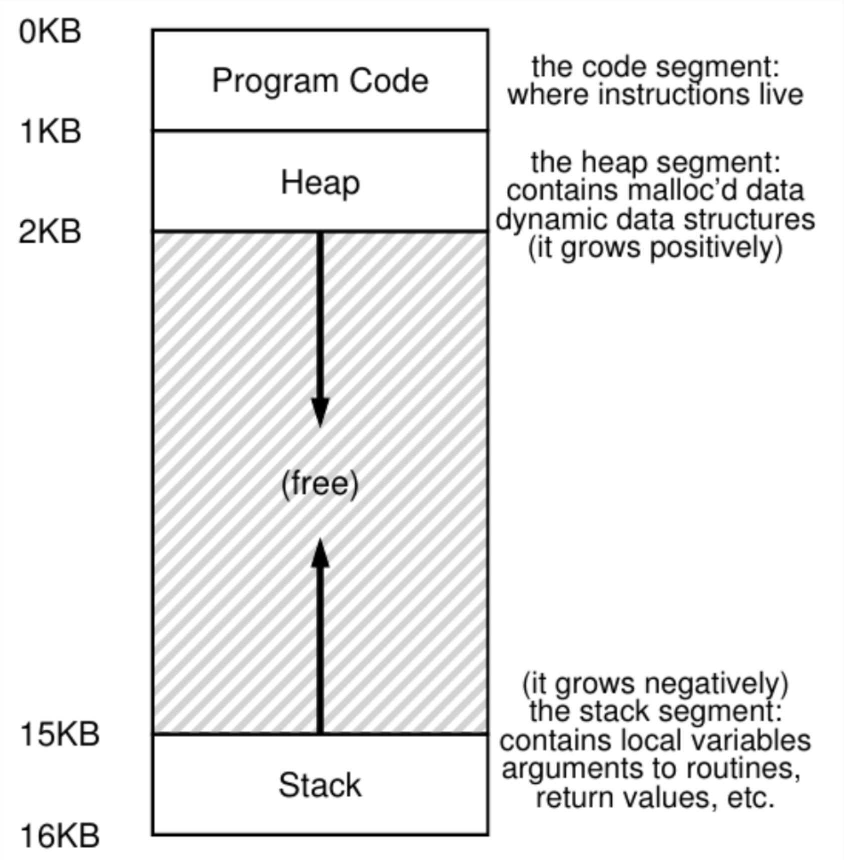
0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
(it grows negatively) the stack segment: contains local variables arguments to routines, return values, etc.

Figure 13.3: **An Example Address Space**

- Code : static => easy to place in memory => top
- heap & stack
  - may grow & shrink while the program runs
  - heap : at the top -> downward
  - stack : at the bottom -> upward

  => so that we can allow both grow

When multiple threads co-exist in an address space, this way to divide the address space won't work

when we describe the address space : it's about "running" program.

The program really isn't in memory at physical address 0 through 16KB; rather it is loaded at some arbitrary physical addresses
=> how each process is loaded into memory at a different address?

Any address you see as a programmer of a user-level program is a virtual address.

HOW TO Virtualize MEMORY?

How can the OS build this abstraction of a private, potentially large address space for multiple running processes (all sharing memory) on top of a single, physical memory?

Virtualizing memory

the running program thinks it is loaded into memory at a particular address (say 0) and has a potentially very large address space.
The reality is quite different.

Goals of VM systems

1. Transparency (hard to notice)
: The OS should implement virtual memory in a way that is invisible to the running program.
=> the program shouldn't be aware of the fact that memory is virtualized

2. Efficiency
: the OS should strive to make the virtualization as efficient as possible both in terms of time & space(not using too much memory for structures needed to support virtualization)

3. Protection
: the OS should make sure to protect processes form one another as well as the OS itself from processes
=> When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect anything outside its address space
=> isolation among processes. each process should be running in its own isolated cocoon

Microkernels
take isolation even further, by walling off pieces of the OS from other pieces of the OS
=> provide greater reliability that typical monolithic kernel designs.

Lazy Allocation
even if you ask the OS to allocate a certain amount of memory with malloc, the system doesn’t immediately give you that physical memory. Instead, it reserves virtual address space.

Only when your program actually touches (reads or writes) those pages, the OS maps them to real physical memory.

This lazy allocation is more efficient because unused memory pages don’t waste RAM.

"Random Access" just means:
Any memory location can be **accessed directly and quickly**, regardless of its position.

This contrasts with **sequential access** devices like hard drives or tapes, where you must go through data in order.