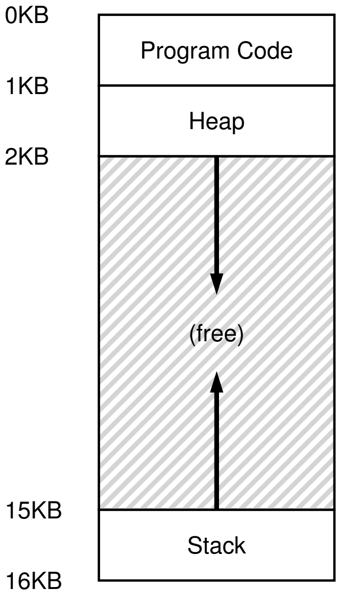


CSL 301
OPERATING SYSTEMS

Lecture 7
Address Space

Instructor
Dr. Dhiman Saha

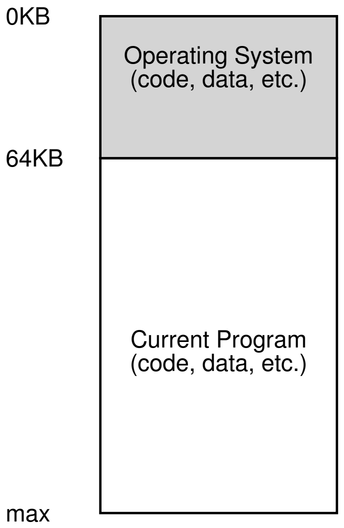
Virtualization of Memory

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?

A Historical Perspective

- Early Systems
- The OS $\equiv$ a set of routines
- Only one running program
- Few illusions!


0KB
Operating System (code, data, etc.)
64KB
Current Program (code, data, etc.)
max

Multiprogramming

- Multiple processes
- Ready to run at a given time
- The OS had a new task
- Switch between them

Goals

- Effective utilization
- Efficiency

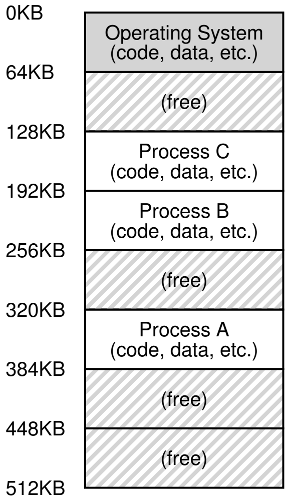
And Time Sharing

- Interactivity
- Many users concurrently using a machine

Can you suggest one possible solution?

- How did we do this in case of CPU virtualization?
- Can we follow the same approach?
- What are the issues?

The physical memory allocated to a process could be non-contiguous too


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)

Address Space

Task

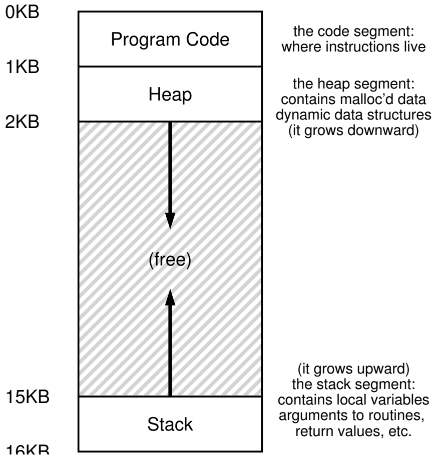
To create an easy to use abstraction of physical memory

Address Space

The running program's view of memory in the system.

- A process has a set of addresses that map to bytes
- This set is called address space
- How can we provide a private address space?

What is in the address space?



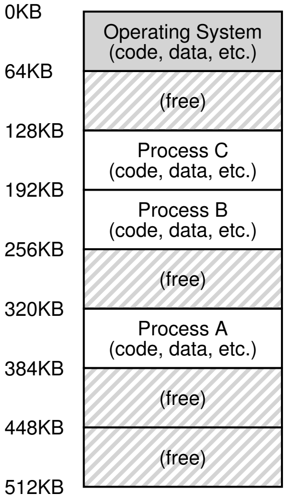
What goes where?

```
int x;
int main(int argc, char* argv[]) {
int y;
int *z = malloc(sizeof(int));
}
```

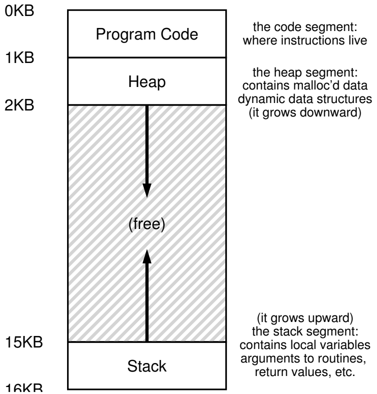
- x $\longrightarrow$ code
- main $\longrightarrow$ code
- y $\longrightarrow$ stack
- z $\longrightarrow$ heap

How are code, stack, heap arranged in this address space? Why?

Process A tries to perform a load at address 0



Actual View



View of Process A

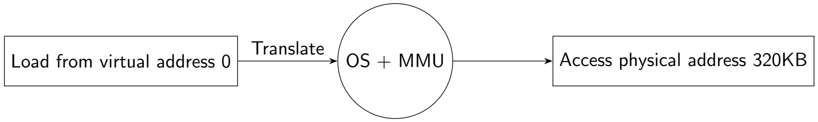
The Crux: How to Virtualize Memory?

The Core Problem

How can the OS build this private, clean-looking address space for multiple processes on top of a single, messy physical memory?

The Answer: Virtualization!

- The address space seen by the program is **virtual**.
- When a program tries to access an address (e.g., address '0'), the OS, with hardware help, translates this **virtual address** to a **physical address**.



Address Translation

- Virtual addresses (VA) to physical addresses (PA)
- CPU issues loads/stores to VA
- Memory hardware accesses PA
- OS allocates memory and tracks location of processes
- Memory Management Unit (MMU): memory hardware that does the translation
- OS makes the necessary information available

Isolation

- One of the primary things that the OS achieves by abstracting out the physical memory
- What is the implication?

HW Short Notes

- Microkernel
- Monolithic Kernel

Goals of a Virtual Memory System

- **Transparency**
  - The process should be completely unaware that memory is being virtualized.
  - It should behave as if it has its own private memory. The OS and hardware handle the magic behind the scenes.

Goals of a Virtual Memory System

- **Transparency**
  - The process should be completely unaware that memory is being virtualized.
  - It should behave as if it has its own private memory. The OS and hardware handle the magic behind the scenes.
- **Efficiency**
  - Virtualization should be as fast as possible (time) and not consume too much memory for its own data structures (space).
  - This relies heavily on hardware support (e.g., TLBs).

Goals of a Virtual Memory System

- **Transparency**
  - The process should be completely unaware that memory is being virtualized.
  - It should behave as if it has its own private memory. The OS and hardware handle the magic behind the scenes.
- **Efficiency**
  - Virtualization should be as fast as possible (time) and not consume too much memory for its own data structures (space).
  - This relies heavily on hardware support (e.g., TLBs).
- **Protection**
  - Protect processes from each other. A bug in Process A should not corrupt Process B.
  - Protect the OS from user processes.
  - This provides the fundamental principle of **isolation**.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
printf("location of code  : %p\n", main);
printf("location of heap  : %p\n", malloc(1));
int x = 3;
printf("location of stack : %p\n", &x);
return 0;
}
```

```
location of code  : 0x55658716e6fa
location of heap  : 0x556587aa2670
location of stack : 0x7ffdcd2c26b4
```

Memory API

`malloc()` and `free`

Are these system calls?

- `malloc` library manages space within virtual address space
- Depends on some other system calls
- `brk/sbrk`
- Alternatively, `mmap()`

- Common mistakes while handling memory
- Using tools like `valgrind` and `purify`