CMPT 300
Operating System I
Memory Management – Chapter 9

Dr. Hazra Imran

Summer 2022

Learning Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation

many process

available memory as required.

data & instruction → main memory

'memory mangmt'

Outline

- Introduction
- Virtualization
- Registers
- Segmentation
- Swapping
- Fragmentation
- Free Space Management
- Paging

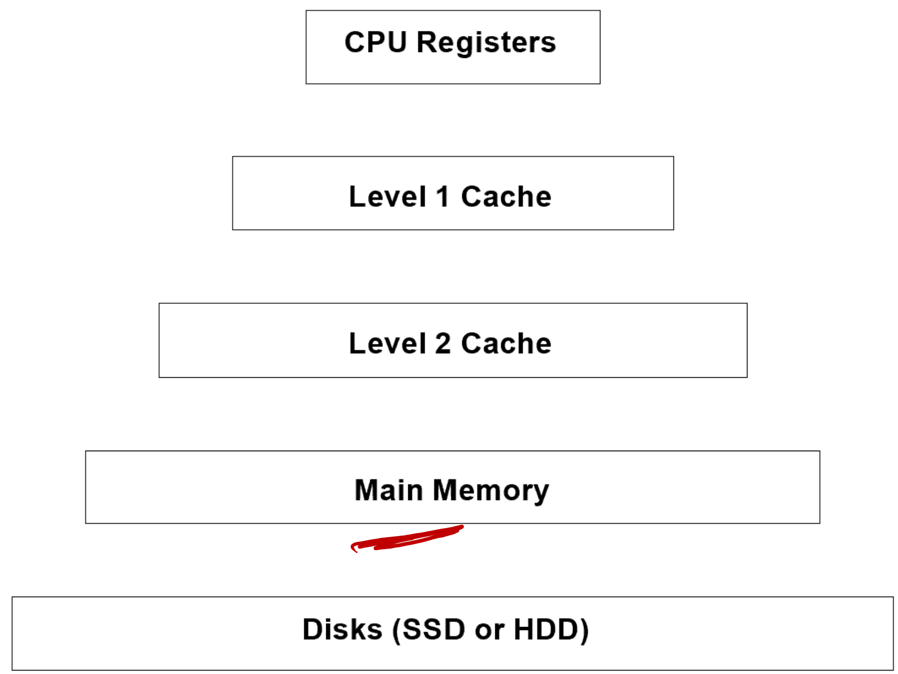
process

pg1

Main memory

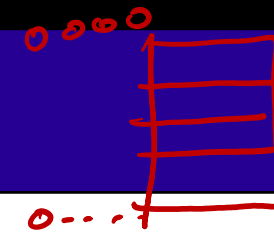
- Programmers want a memory that is
  - large
  - fast
  - Non-volatile

The memory manager handles the memory hierarchy



Background

R|W



- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

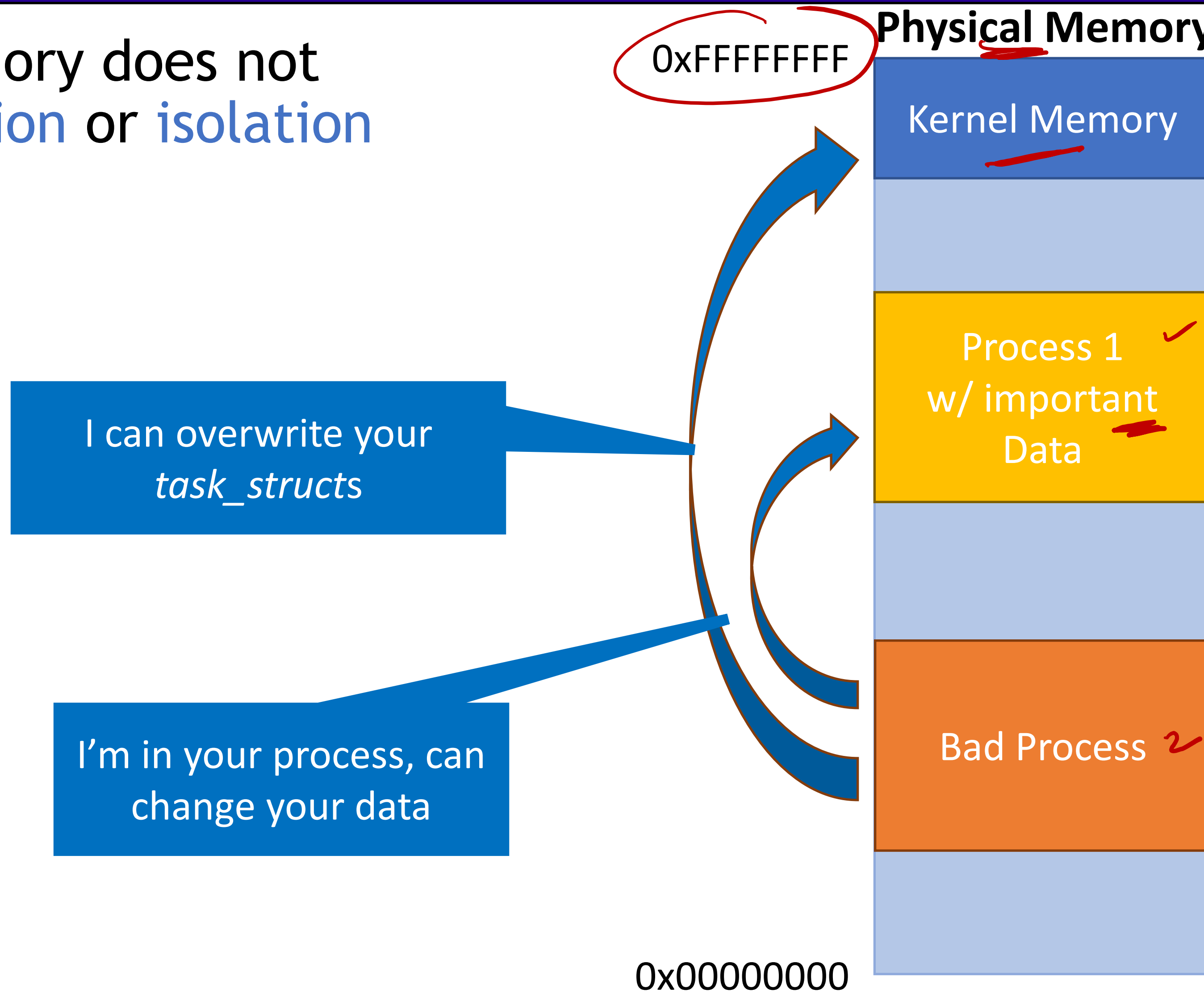
Main Memory

- Main memory is conceptually very simple
    - Code sits in memory
    - Data is either on a stack or a heap
    - Everything gets accessed via pointers

- Memory is a simple and obvious device
    - So why is memory management one of the most complex features in modern OSes?

Protection and Isolation

- Physical memory does not offer protection or isolation



Compilation and Program Loading

- Compiled programs include fixed pointer addresses

- Example:

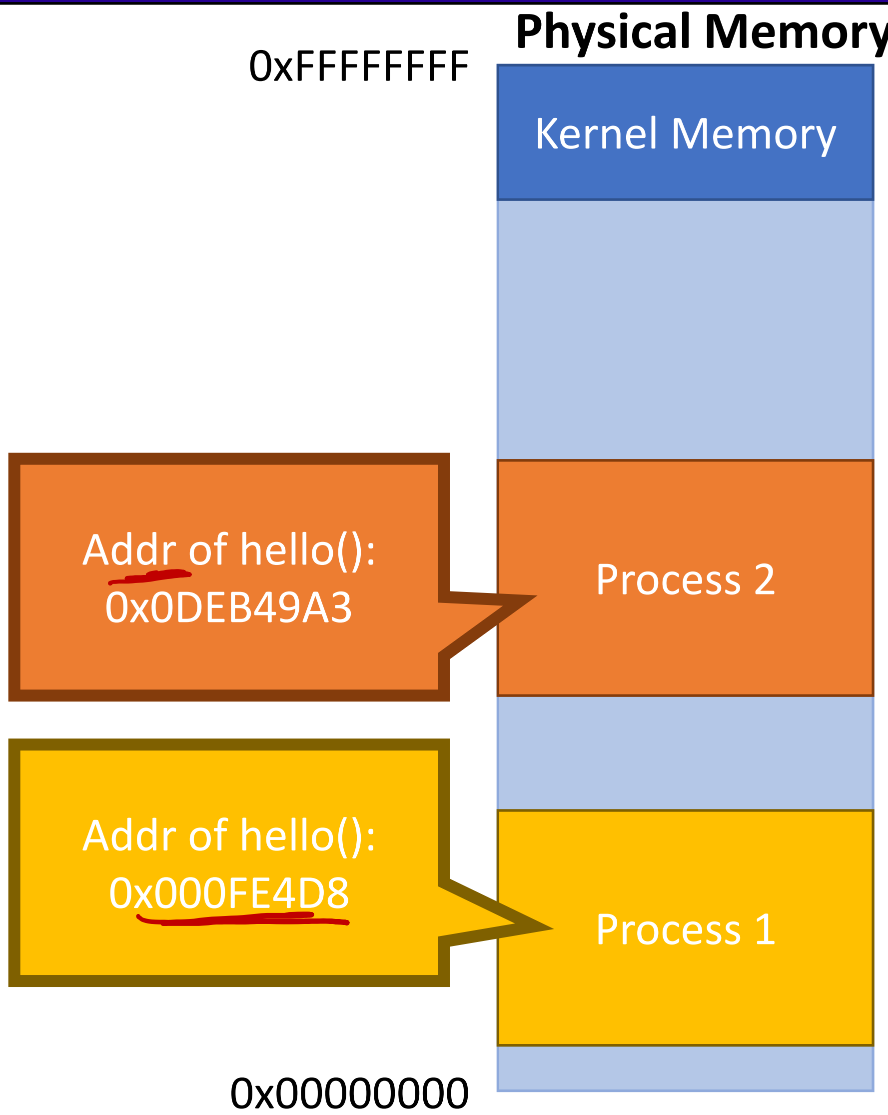
000FE4D8 <hello>:

…

000FE21A: push eax

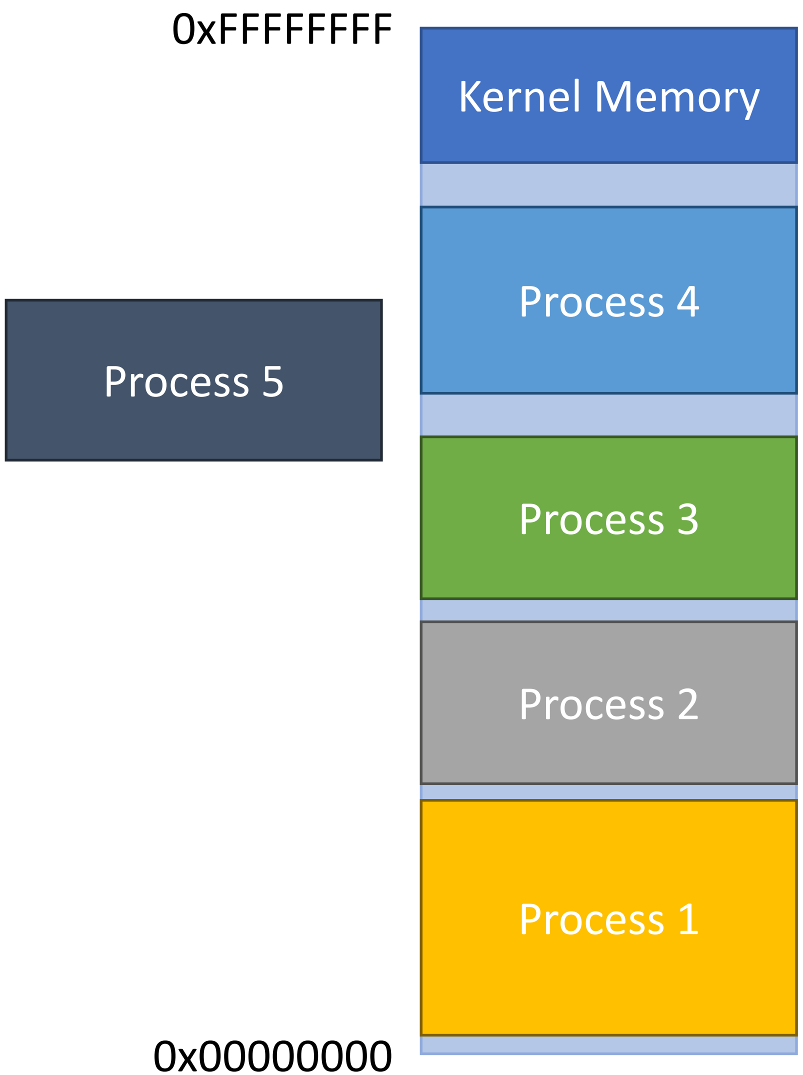
000FE21D: push ebx

000FE21F: call 0x000FE4D8



Physical Memory has Limited Size

- What happens when you run out of RAM?



Physical vs. Virtual Memory

- Clearly, physical memory has limitations
  - No protection or isolation
  - Fixed pointer addresses
  - Limited size
  - Etc.

- Virtualization can solve these problems!
  - As well as enable additional, cool features

Clicker

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
  int x;
  if (fork() == 0)
    x = 90;
  else
    x = 10;
printf(" The value %d is stored in memory location %p\n ", x, &x);
return 0;
}
```

Can you predict the output of this code?

A) Yes
B) No
C) I do not make predictions on Mondays

Clicker

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
    int x;
    if (fork() == 0)
        x = 90;
    else
        x = 10;
printf(" The value %d is stored in memory location %p\n ", x, &x);
return 0;
}
```

```
 The value 10 is stored in memory location 0x7ffffe
d381f8
```

```
 The value 10 is stored in memory location 0x7fffe9
510cd8
 The value 90 is stored in memory location 0x7fffe9
510cd8
```

```
 The value 10 is stored in memory location 0x7fff9f
e02ee8
```

Lets see…

- By now we know that every value (variable) stored by a process is stored at some address in memory

- What we just saw is that different processes can see different values at the same address

- One of the great advents of the 1960s in operating systems was virtual memory

- The location a process sees a value in is not the same as where it actually is.

The story of memory access

x

- x = 10;
- We then compile and execute that program
- How does the variable x get the value 10?

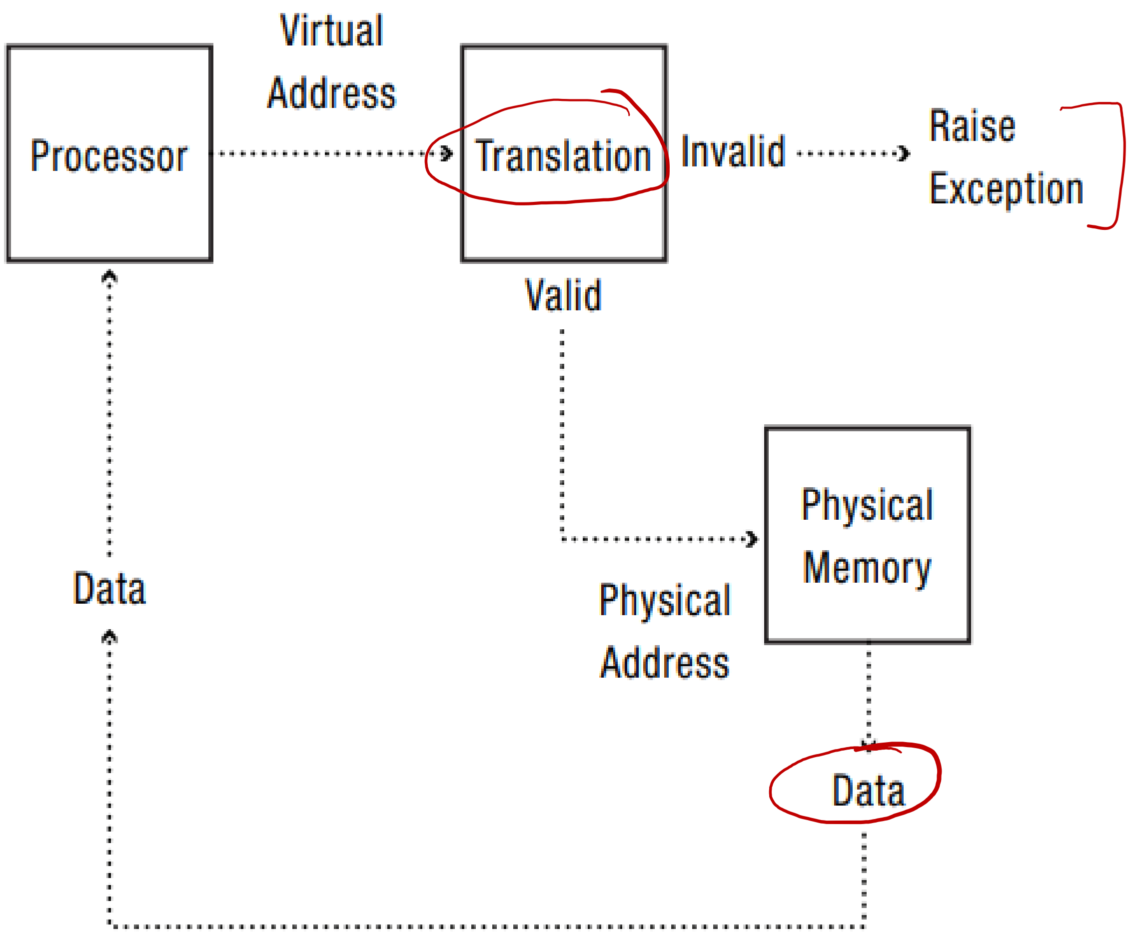
Step 1

- We compile our C code into executable code.
- The compiler will compute the memory location of x on the stack
- It will send out a store instruction.

Step 2

- We execute our C code
- The store instruction will get executed by the CPU
- The CPU has, stored inside of it, a page table which translates **virtual addresses** into **physical addresses**
- The page table will tell the CPU which (physical) address to store the data in RAM

Memory

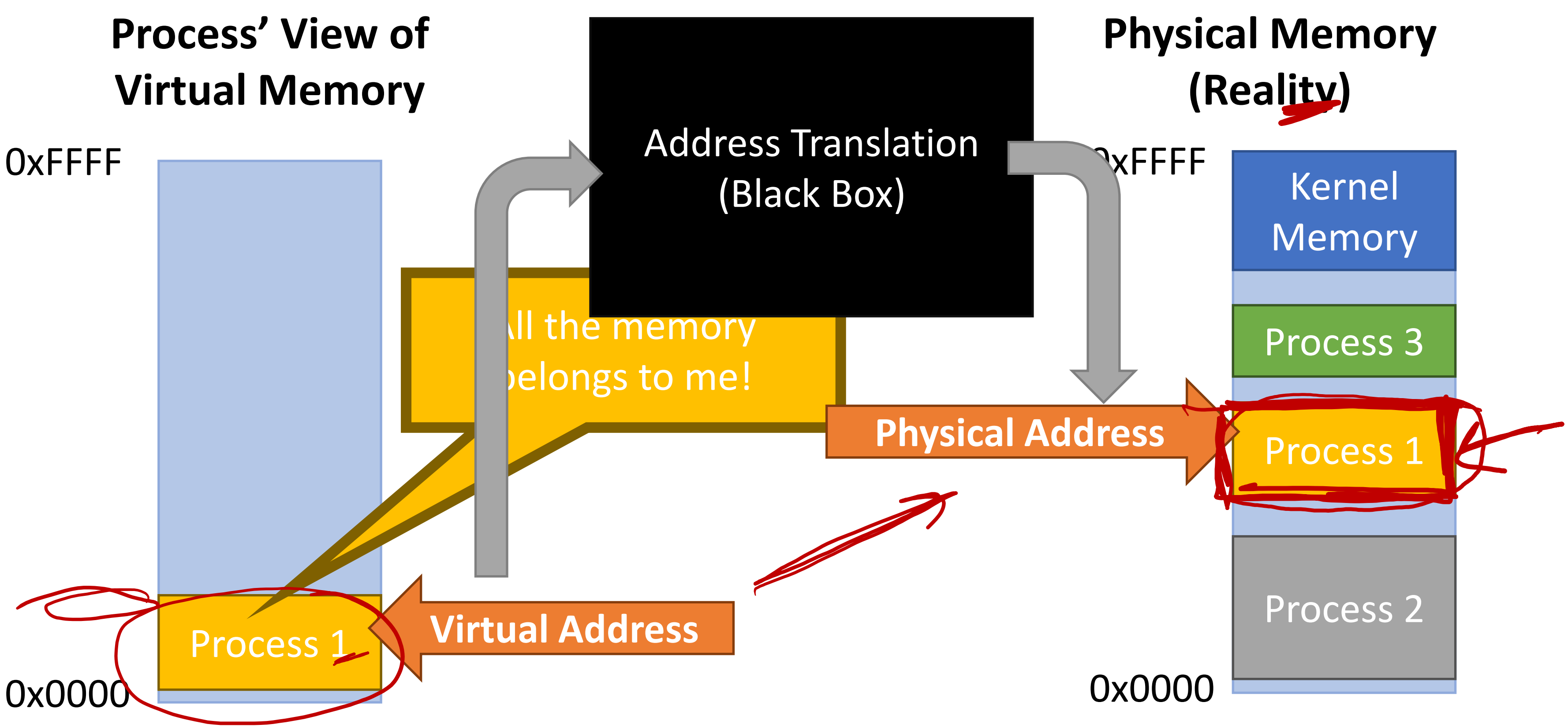

Virtual
Address
Processor
Translation
Invalid
Raise
Exception
Valid
Physical
Memory
Data
Physical
Address
Data

Logical vs physical memory

- A computer's ***physical memory*** (***RAM***) is a hardware structure consisting of a linear sequence of words that hold a program during execution.
  - A ***word*** is a fixed-size unit of data. A typical word size is 1, 2, or 4 bytes.
- A ***physical address*** is an integer in the range [0 : N-1] that identifies a word in physical memory of size n. The address comprises a fixed number of bits.
- During program development, the starting address of a program in physical memory is unknown. To facilitate program development and the sharing of physical memory, the concept of logical address space is employed.
- A ***logical address space*** is an abstraction of physical memory, consisting of a sequence of imaginary memory locations in a range [0: M-1], where m is the size of the logical address space.
- A ***logical address*** is an integer in the range [0: M-1] that identifies a word in a logical address space.
  - Prior to execution, a logical address space is mapped to a portion of physical memory and the program is copied into the corresponding locations.

A Toy Example

- What do we mean by virtual memory?
  - Processes use virtual (or logical) addresses
  - Virtual addresses are translated to physical addresses.



Implementing Address Translation

- In a system with virtual memory, each memory access must be translated

- Modern systems have hardware support that facilitates address translation
  - Implemented in the Memory Management Unit (MMU) of the CPU
  - Cooperates with the OS to translate virtual addresses into physical addresses

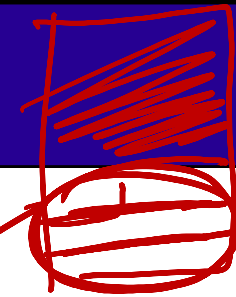
Virtual Memory Implementations

- There are many ways to implement an MMU
  - Base and bound registers
  - Segmentation
  - Page tables
  - Multi-level page tables

Old, simple, limited functionality

Modern, complex, lots of functionality

Goals of Virtual Memory



hole

compaction

- Transparency
  - Processes are unaware of virtualization
- Protection and isolation
- Shared memory and memory-mapped files
  - Efficient interprocess communication
  - Shared code segments, i.e. dynamic libraries
- Dynamic memory allocation
  - Grow heaps and stacks on demand, no need to pre-allocate large blocks of empty memory
- Demand-based paging
  - Create the illusion of near-infinite memory

1) sparse address spaces

flexible Memory placement

Clicker

If the size of logical address space is much larger than the average program size, then a lot of space is wasted ______.

- In Physical memory

(A) True

(B) False

Clicker

If the size of logical address space is much larger than the average program size, then a lot of space is wasted ______.

- On Disk

(A) True

(B) False

Outline

- Introduction
- **Virtualization**
- Registers
- Segmentation
- Swapping
- Fragmentation
- Free Space Management
- Paging

Fork () and Virtual addresses

1. When our C program first started execution, it had its own page table
2. The **int x;** statement allocated space on the stack for a new variable
3. When we make a fork() system call, the operating system splits us into two process
4. Each process has its own page table
5. When we perform the store instruction to set the value of x, each process is storing it in a different location in memory
6. Each process, at this point, has its own separate value of x

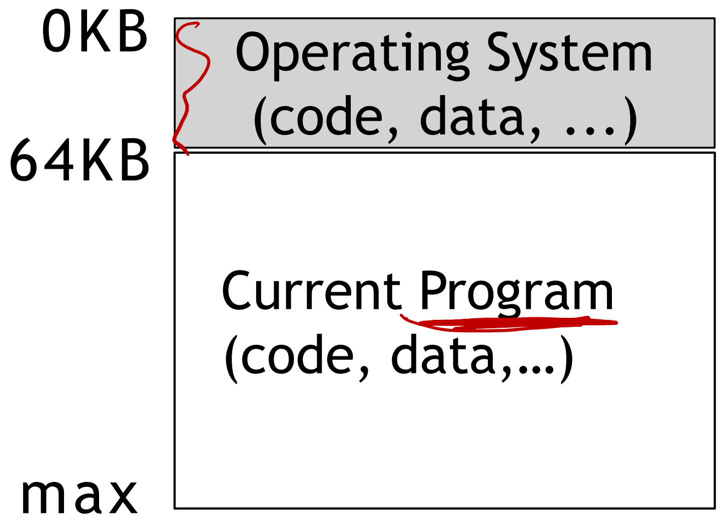
empty

```
int x;
if (fork() == 0)
  x = 90;
else
    x = 10;
printf("...
```

mapping

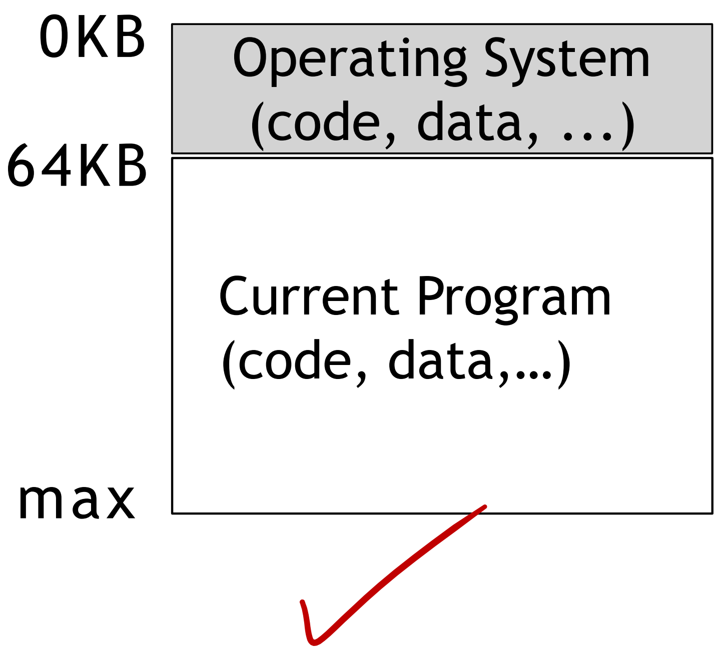
Memory abstraction

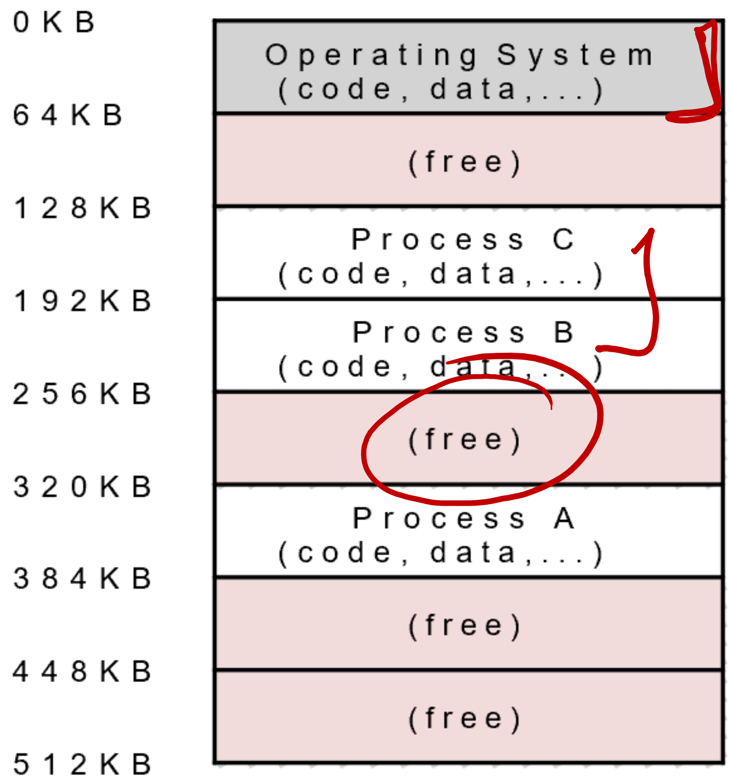
- The OS was a set of routines that sat in memory
- There would be one running program (a process) that currently sat in physical memory and used the rest of memory.



Not efficient!
Low resource utilization

Multiprogramming and Time Sharing





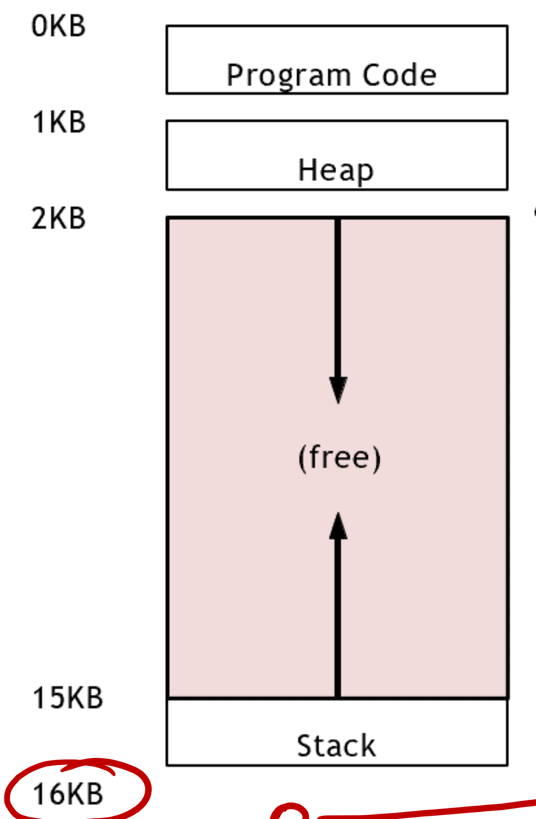
Protection/isolation between processes needed
Solution: address space

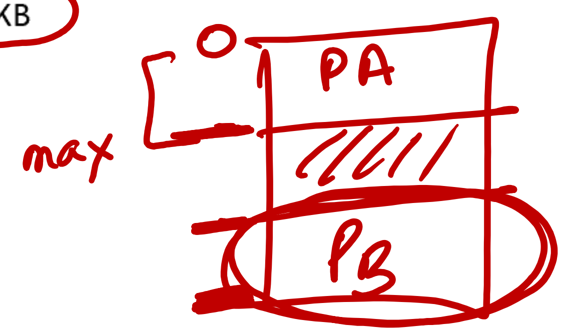
Address Space

- Easy-to-use abstraction of physical memory

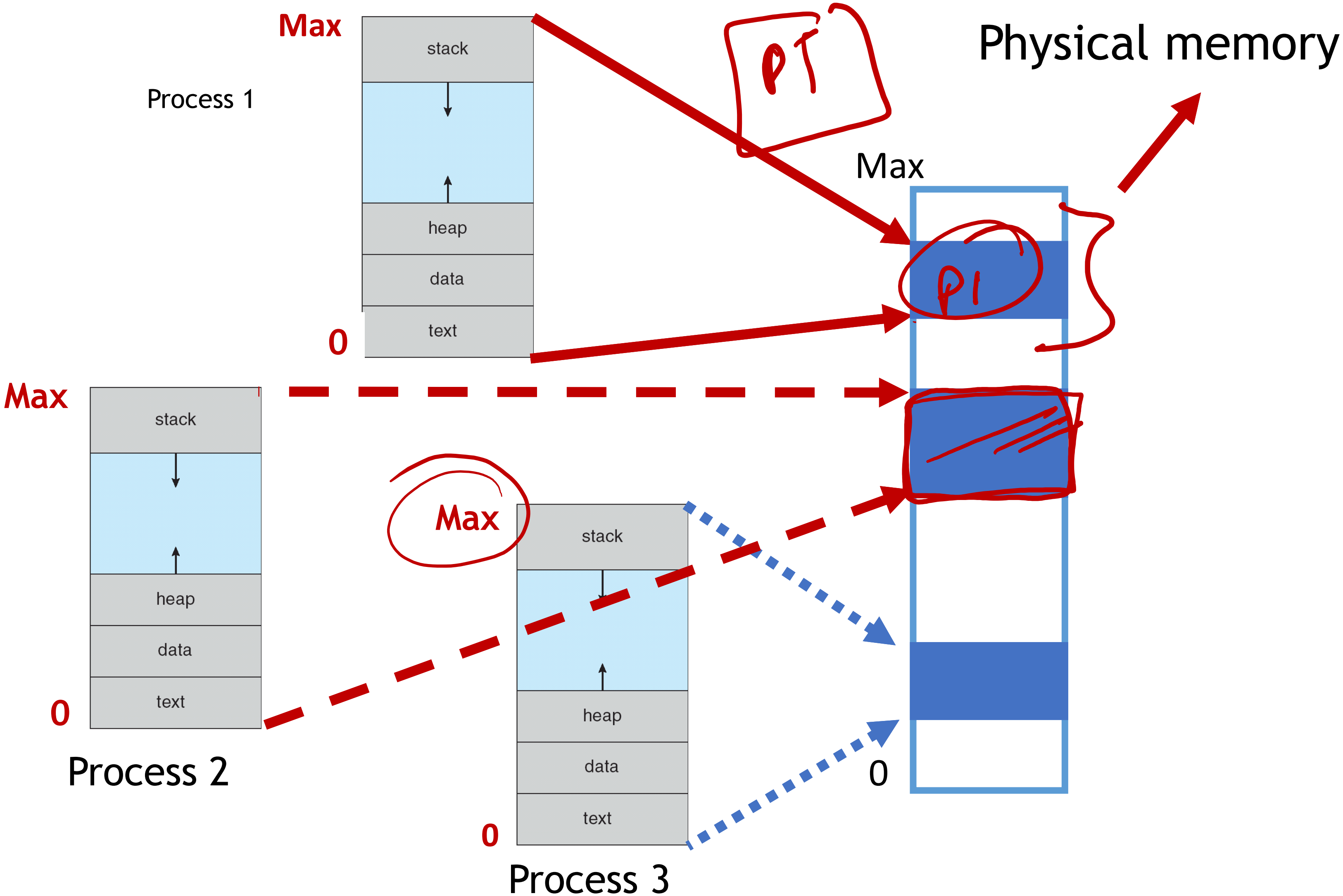
- Every process assumes it has access to a large space of memory from address 0 to a MAX (illusion).
  - Provide isolation between processes
  - Efficiently use memory space

- Each process has its own address space
  - Starts from 0
  - Ends at max memory possible
  - E.g., 2^{32} for 32-bit machines, 2^{64} for 64-bit machines
  - "**Virtual address**"



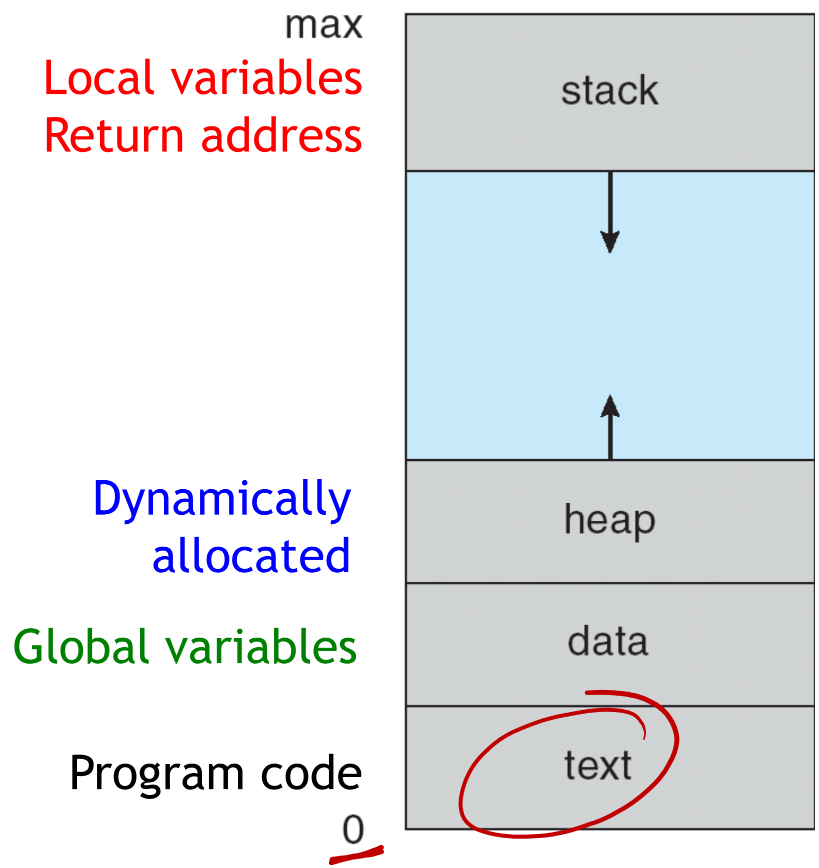


Address Translation



Address Space

```
int global = 0;
int main (void)
{
float  local;
char *ptr;
ptr = malloc(100);
local = 0;
local += 10*5;
printf("%x \n", ptr);
printf("%x at \n", &ptr);
printf("%x at \n", &local);
printf("%x at \n", &global);
return 0;
}
```



All addresses used here are **virtual** (logical) addresses
Applications have **no** direct access to memory using physical addresses

Memory Virtualization

- From the process' point of view, the address space starts at 0

- All addresses in the process address space are expressed as an **offset relative** to the **base** value
- The process address space has been **virtualized**
  - An address in the **physical memory** is called a **physical address**
  - The address manipulated by the CPU is a **logical address** or a **virtual address**

- A program references a **logical address space**, which corresponds to a **physical address space** in the memory

- However "something" needs to tell the CPU how to **translate** from virtual to physical addresses, i.e., some **address translation** mechanism

Outline

- Introduction
- Virtualization
- **Registers**
- Segmentation
- Swapping
- Fragmentation
- Free Space Management
- Paging