Process Address Space and Address Translation

2023/24 COMP3230B

Contents

- Process address space
  - What is the relation between physical memory and process address space?

- With certain assumptions
  - How can we build an efficient virtualized memory system?
  - How can OS ensure that all memory accesses issued by the application are properly restricted/guarded?

Learning Outcomes

- ILO 2b - describe the principles and techniques used by OS in effectively virtualizing memory resources.

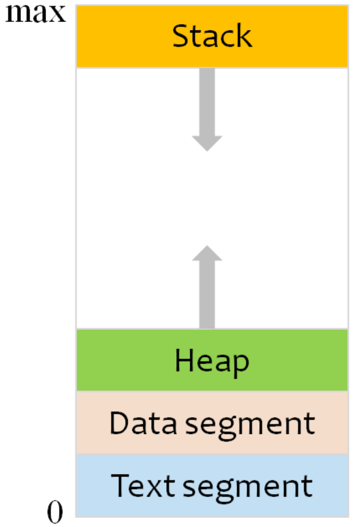
Readings & Reference

- Required Readings
  - Chapter 13 - The Abstraction: Address Spaces
    - http://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf
  - Chapter 15 – Mechanism: Address Translation
    - http://pages.cs.wisc.edu/~remzi/OSTEP/vm-mechanism.pdf

Program in execution

- Each process has its own address space consisting of:
  - Text, Data, Heap, and Stack

- Address space is the **process's own view** of the memory in the system
  - A process (programmer) **thinks** it is being loaded into "memory" starting at address 0 and has a potentially large address space (to max)
    - e.g., for 32 bits, Max is 2^{32} = 4 GiB

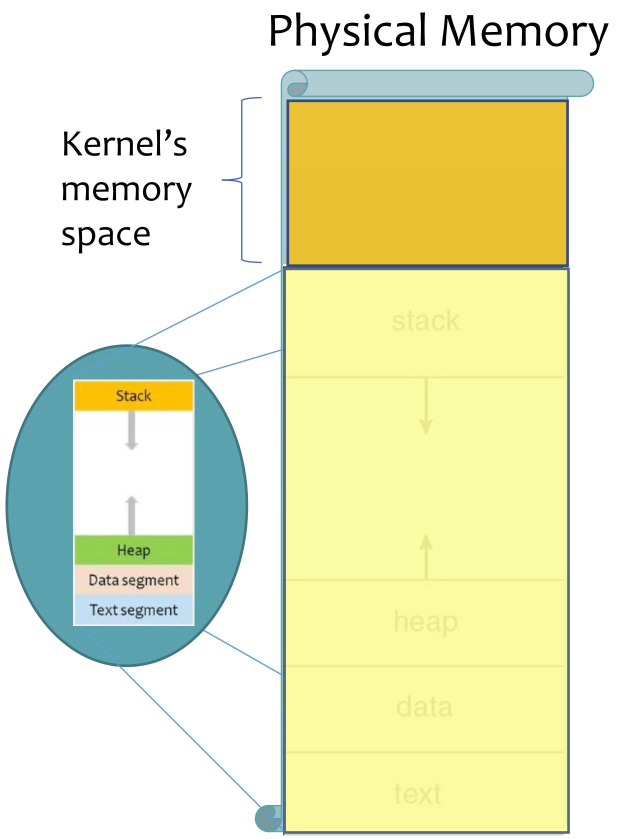


Notation

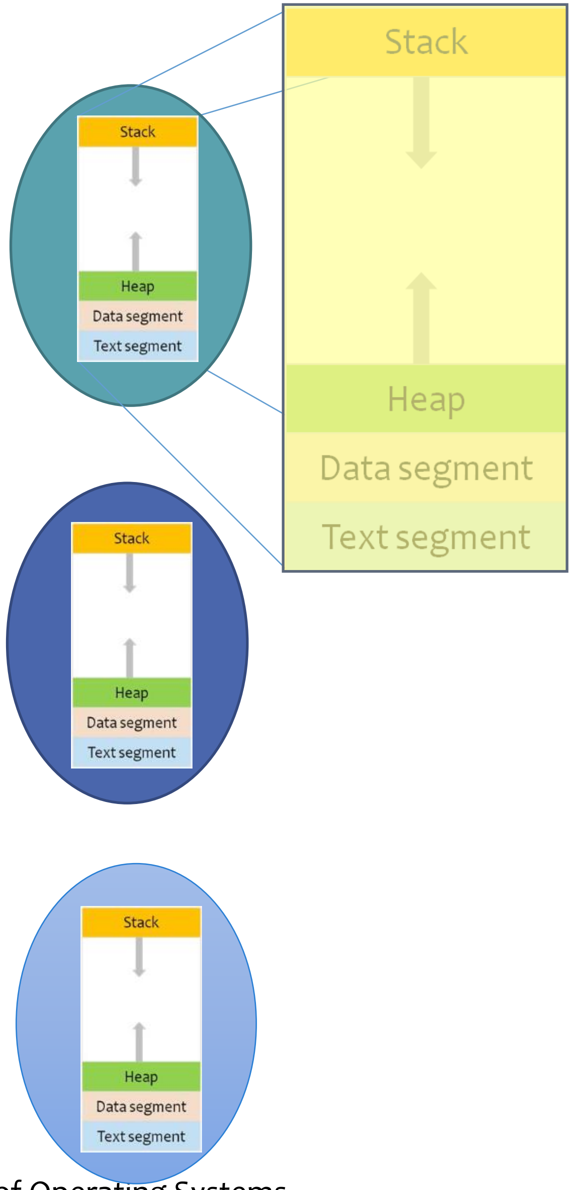
- About memory capacity/size
  - KiBytes = KiB = 2^{10} bytes = 1024 bytes
  - MiBytes = MiB = 2^{20} bytes = 1048576 bytes
  - GiBytes = GiB = 2^{30} bytes

Where to place the process’s memory?

- The simplest case – Imagine that we are using a system just supports “single user” “single program” at a time
  - Can a process use all available physical memory?
  - Where to place the OS?
    - Allocate a “partition” in the “memory” to house the OS
  - Remaining can be used by the program



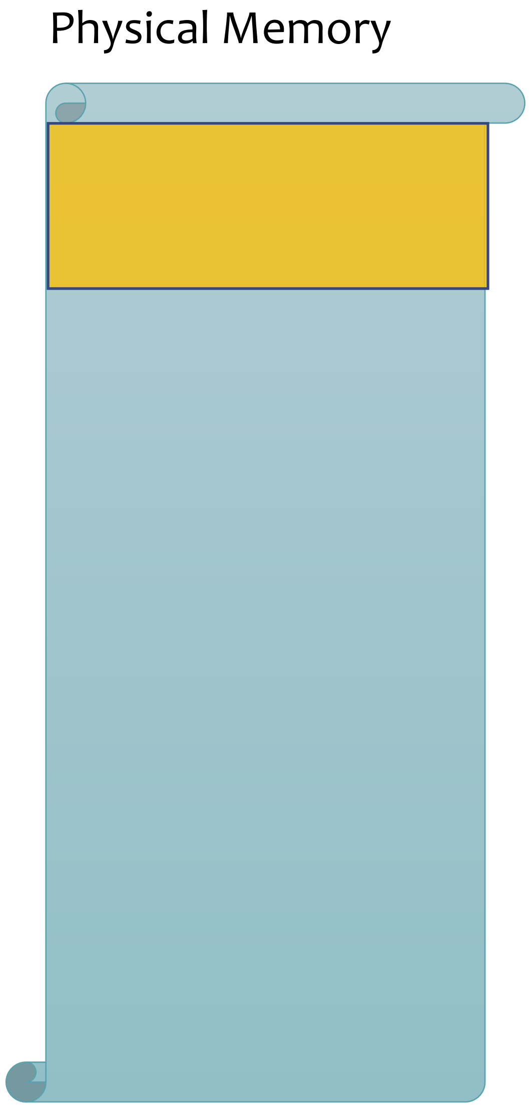
Where to place the processes' address spaces in the main memory?



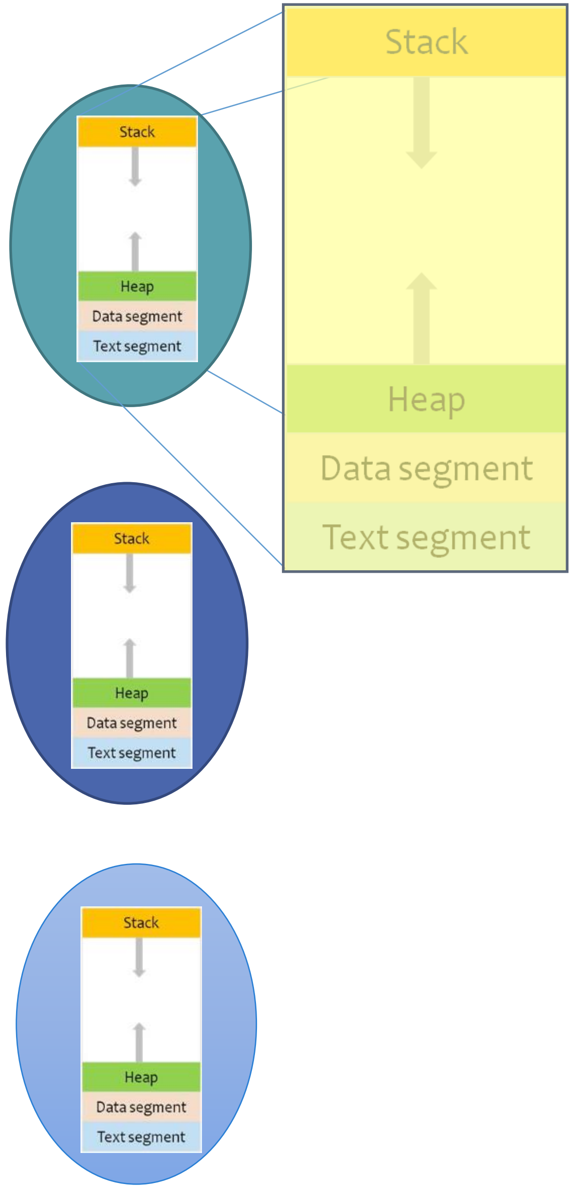
To support multiple programs, how to share?

- Maybe using time sharing
  - Let one process running for a short while, getting full access to all memory (except OS partition), then stop it and store process's state and all memory contents to the disk, load in another process's state and memory, and run it, and so forth

- Very high overhead; very poor performance



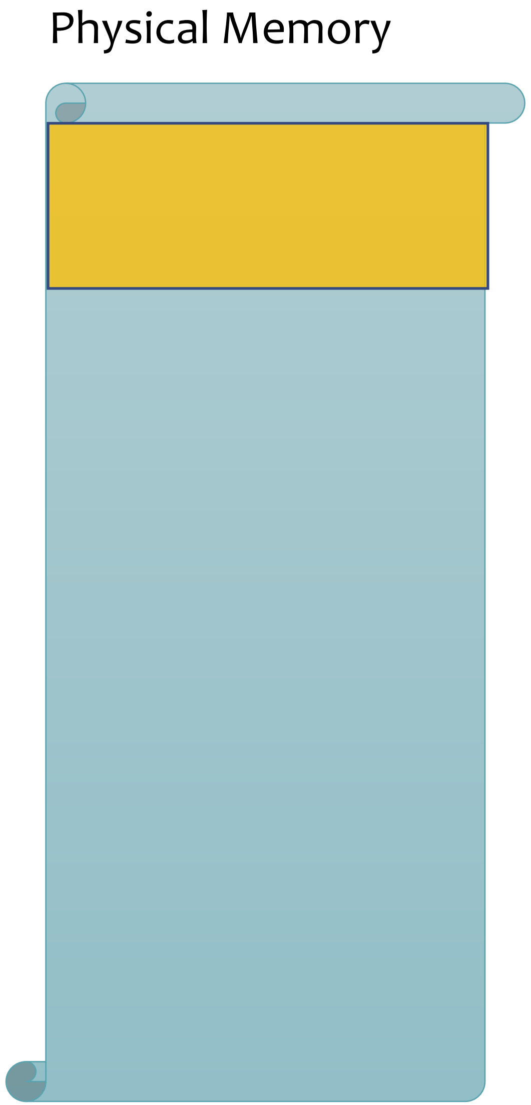
Where to place the processes' address spaces in the main memory?



Can we have multiple processes that reside in the physical memory at the same time?

Assumption 1

Assume the address space of a process is relatively small, i.e., size of physical memory is **much larger** than a process's address space.



Where to place the processes’ address spaces in the main memory?

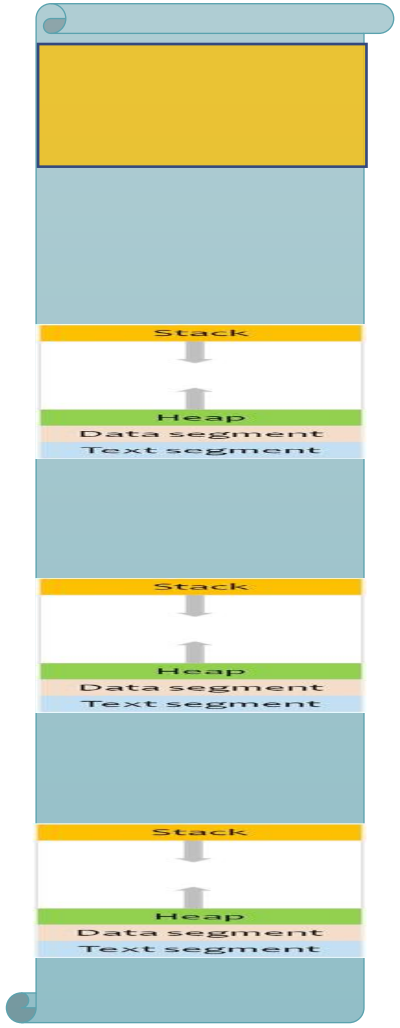
- How to share?
  - **Divide** the physical memory into **different partitions**; each partition is allocated to one process

? At fix location ?

? At any available location ?

- The Cruxes
  - **Which partition is assigned to which process?**
  - **How to find the actual physical address of “a memory location within a process”** (we shall call the memory location in the process as a **virtual address**)?
    - Example: physical location of virtual address 100 of process A should be different from physical location of virtual address 100 of process B
  - If there are more processes than the available partitions, **how to handle this situation?**
  - Process A should not touch (access) Process B’s memory partition. **How can we secure the system?**

Physical Memory



Virtual Memory

- OS provides a **transparent** mechanism in virtualizing the memory
  - **Any address** you see in a program is a virtual address

```
int abc;

printf("The memory location of \"abc\" is at %p\n", (void *) &abc);
```

```
The memory location of "abc" is at 0x7ffeb3514ea4
```

  - It is **the OS and the hardware** working together to **translate** virtual addresses to physical addresses

Virtual Memory

- OS should strive to make the virtualization as **efficient** as possible
  - The translation action is being done for **each instruction** and **each memory access** !!!
  - Must be very fast as being invoked for every instruction executed

- OS should make sure a process cannot access memory locations outside its address space
  - This helps to protect processes from one another

- Memory management unit (MMU)
  - Special-purpose hardware **in processor** for quickly translating virtual addresses to physical address

Responsibility of OS

- Set up the hardware (MMU) for virtual-to-physical address translations (for each process)

- Keep track of the utilization of physical memory
  - Where are the free memory blocks?
  - Which memory blocks are in use?

- Protect against inappropriate memory accesses by applications

Make it simple

- Assumption 1
  - Assume the address space of a process is **relatively small**, i.e., size of physical memory is much larger than a process's address space.
- Assumption 2
  - Assume the process's address space must be **placed contiguously** in physical memory.
- Assumption 3
  - Assume that the address space of every process is exactly the **same size**.

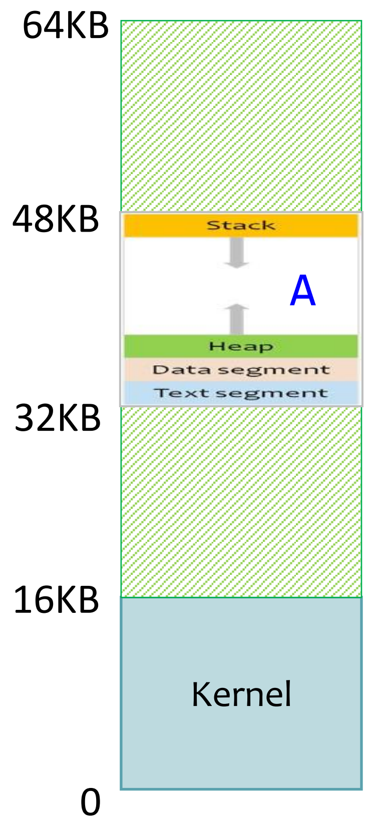
Where to place a process?

- To be ***flexibile***, OS wants to place a process in **anywhere** (of suitable size) **in physical memory**,
  - **not** necessary starting at physical address zero; OR
  - **not** necessary fix at the same physical location for each execution of that process

- The Crux
  - How can we place a process somewhere else in physical memory in a way **that is transparent** to the process?
  - In particularly, the physical location of a variable may change

Mechanism – Dynamic Relocation

- Hardware in MMU
  - We need two hardware registers – **base** and **bounds**
    - base register – stores the starting physical address of a process's address space in the physical memory
    - bounds (limit) register – delimits the size of the address space of a process
  - Example
    - When OS loads process A to the memory block between 32KB-48KB in the physical memory
    - It sets
      - base register to 32KB
      - bounds register to 16KB



Mechanism – Dynamic Relocation

- Address Translation
  - For any memory reference generated by the process,
  - CPU (MMU) calculates

    physical address = virtual address + base
  - and gets the memory content stored at this physical location

- Dynamic Relocation
  - The physical location of the process may change during runtime
    - e.g., a suspended process is being resumed; its memory contents have to be moved from disk back to physical memory
  - OS just needs to **update the base register** when the process is loaded to a new physical location

Mechanism – Dynamic Relocation

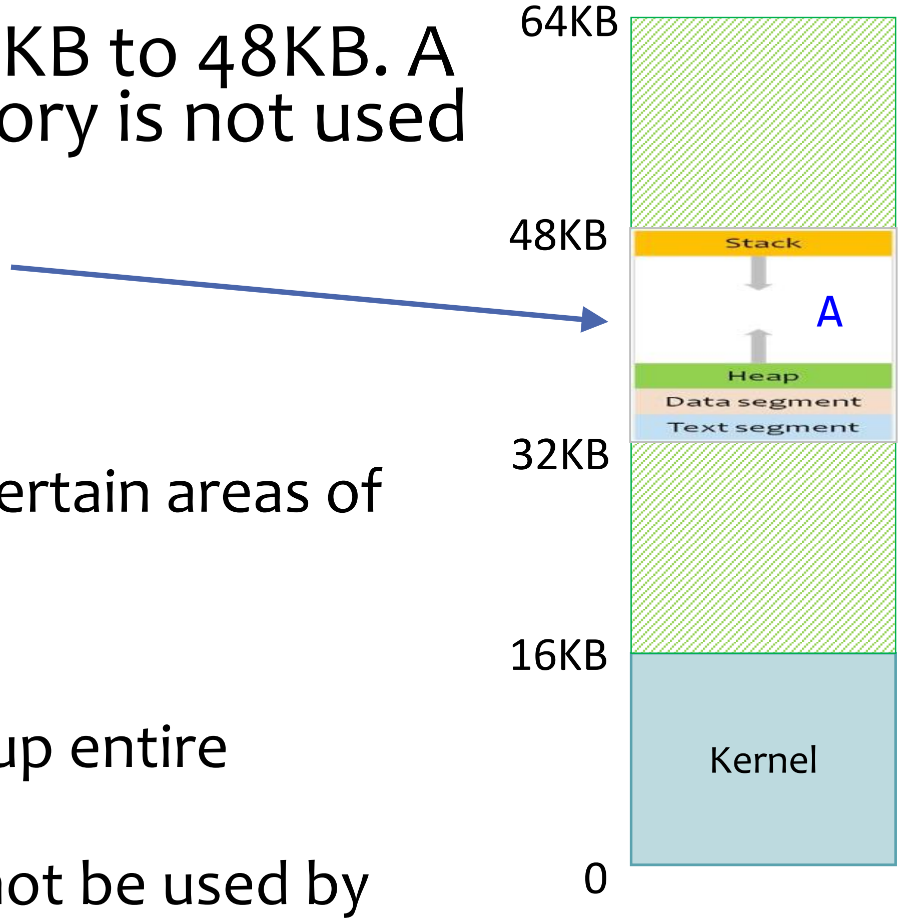
- Protection
  - The processor, before translating a virtual address, will check that the memory reference is **within bounds**
  - If is out of bounds, the processor will **raise an exception** (an interrupt) to alert OS to take appropriate action

OS Issues

- With assumptions 1, 2, & 3, OS divides the physical memory into memory blocks and use a data structure (e.g., **free list**) to indicate which blocks are free and which are in used
  - OS needs to **consult and update the free list** when starting a new process
  - OS needs to **update** the free list upon termination of a process

A Flaw:
Internal Fragmentation

- Example: Process A is in the memory block 32KB to 48KB. A lot of space within this block in physical memory is not used
  - Space between the stack and heap is wasted



- **Fragmentation**
  - The phenomenon wherein **the system cannot use** certain areas of "available" memory
- **Internal** Fragmentation
  - Process is allocated a partition, but it may not take up entire allocated partition
  - Memory **internal** to a partition is not used, but cannot be used by other processes
  - **Waste of memory**

The Crux

- How do we **effectively** support a process address space with (potentially) a lot of free space between the stack and the heap?

Summary

- Only one pair of base-and-bounds registers on each CPU, **OS must save and restore** the two registers when it **switches between processes**

- To virtualize memory,
  - OS needs the help from hardware (base-and-bound) to efficiently translate virtual address to physical address
    - This makes the translation process transparent to the applications

  - OS and hardware combine to ensure no process can generate memory references outside its own address space