ITP 30002 Operating System

Address Space and Dynamic Relocation

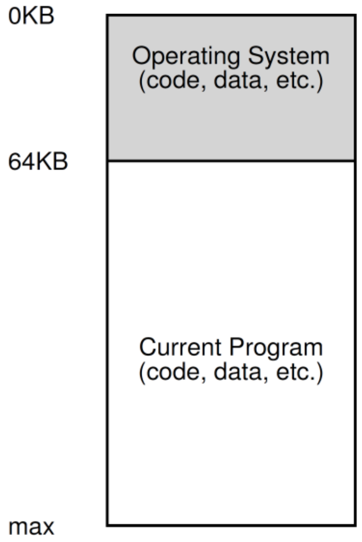
OSTEP Chapters 13 & 15

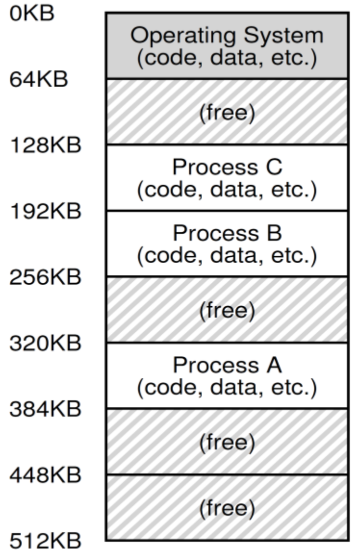
Shin Hong

Motivation

- Early computer systems did not need memory abstraction since there was no issue for a program to occupy whole memory

- Memory abstraction is required with time-sharing
  - approach 1. like CPU context switching, store the entire memory state to a storage device at a context switching
    - heavy context switching cost
  - approach 2. let a process use only a region of memory, and keep multiple processes in the memory at the same time
    - low utilization of memory
    - data protection issue

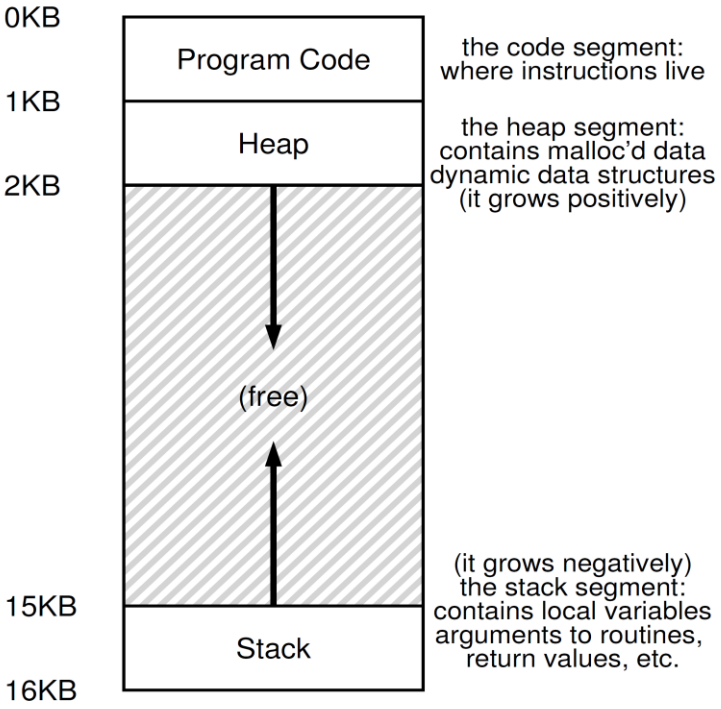

0KB
Operating System (code, data, etc.)
64KB
Current Program (code, data, etc.)
max


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

Abstraction: The Address Space

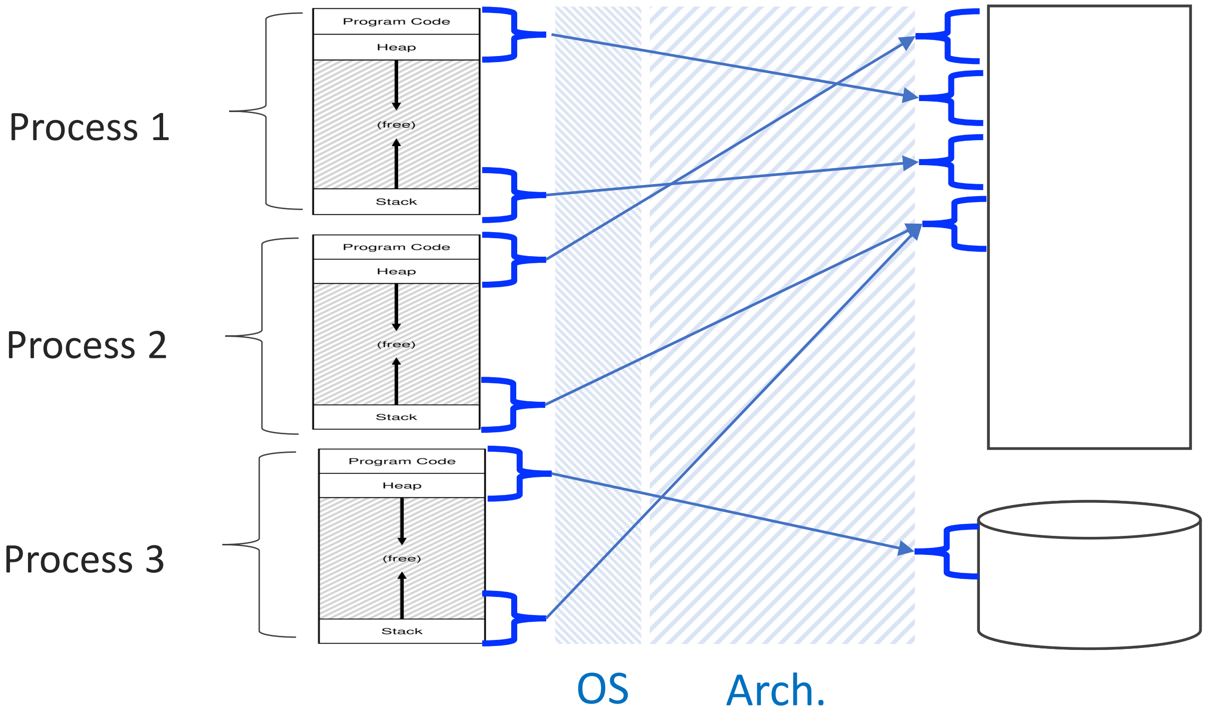
- Address space is the running program's view of memory
  - interface between a process and memory devices

- The address space of a process has a continuous region of addresses which contains the code, the stack, the heap and all memory state


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.

Virtual Memory

- The OS **virtualizes** memory in cooperation with computer architecture
  - the goals of memory virtualization
    - transparency (seamless-ness)
    - time-efficiency and space-efficiency
    - isolation



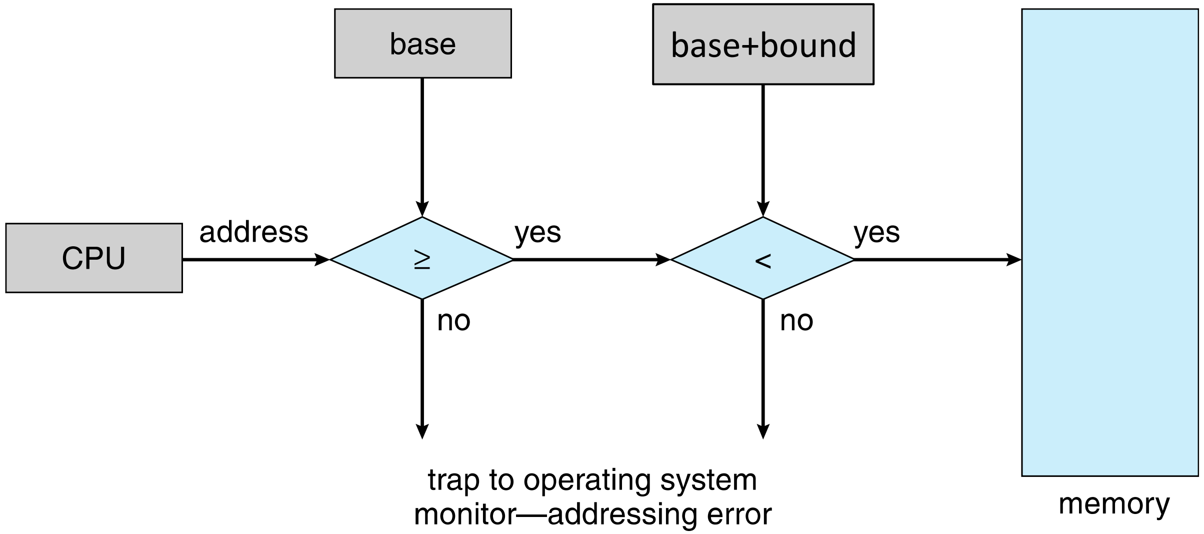
Hardware-based Address Translation

- Let a computer architecture transform each memory access by converting a virtual address to a physical address
    - like a computer architecture translates relative addresses to absolute addresses

- The OS manages a mapping from virtual addresses to physical addresses
    - the OS interposes between an application program and hardware operation at critical points to maintains control over the hardware
    - the critical points includes:
        - process creation/termination,
        - context switching,
        - when a process attends to access forbidden memory regions

Approach 1: Dynamic Relocation

- Assumption
  - The size of the address space for a process is much smaller than the total amount of available memory in the main memory device
  - Every process is given the same amount of address space
  - The MMU of the computer architecture supports the **base** register and the **bound** (limit) register
    - always translate a memory address if it's user mode
    - the base and the bound registers can be accessed only if it's in previlaged mode

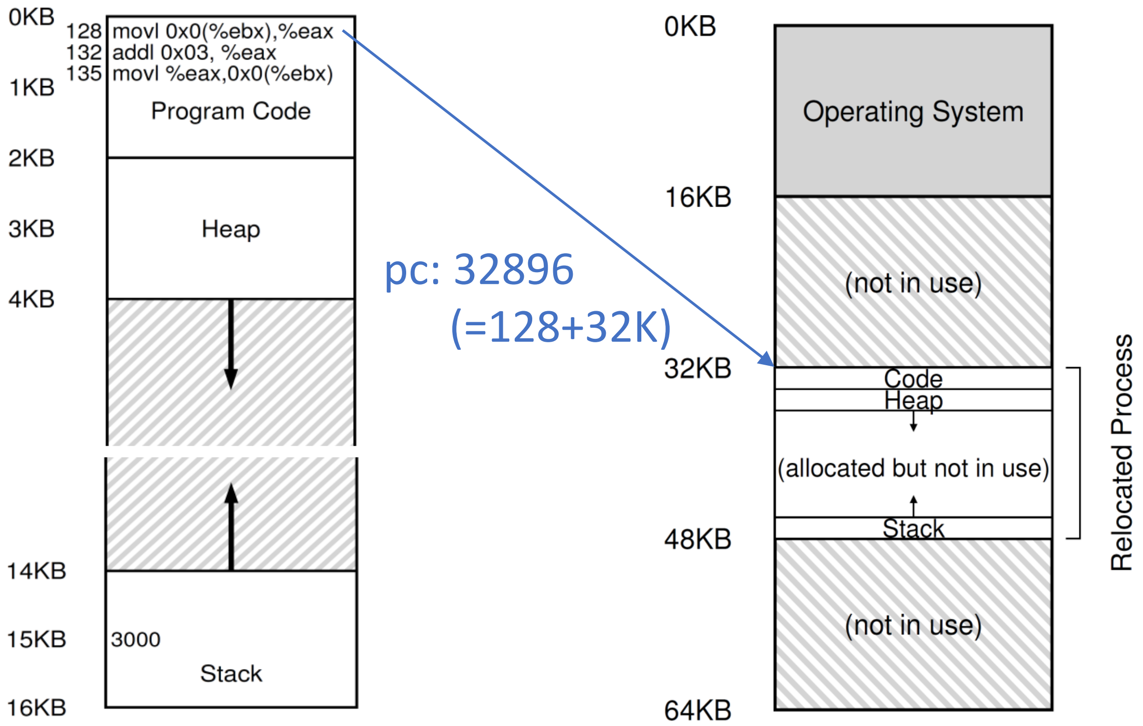


Approach 1: Dynamic Relocation

- Approach
  - Allocate a continuous region of physical memory to a process
  - Store the beginning of the allocated memory region to the base register *b*
  - Always translate a memory address of a program *m* into *m* + *b*
  - Set the bound register to raise a trap if the process tries to access an address beyond its given capacity

```
void func() {
    int x = 3000;
    x = x + 3;
    ...
```

```
128: movl 0x0(%ebx), %eax
132: addl $0x03, %eax
135: movl %eax, 0x0(%ebx)
```



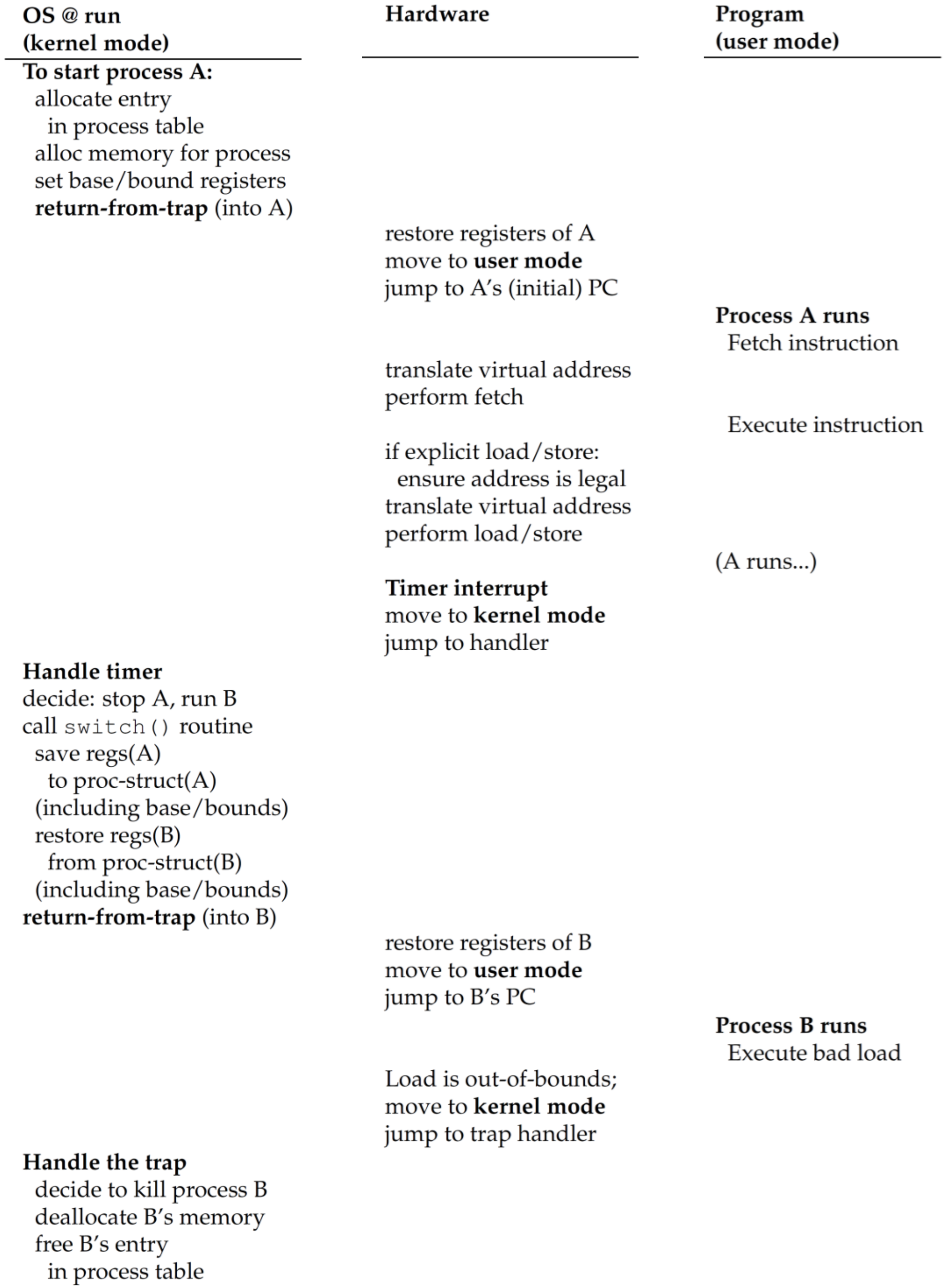
pc: 32896
(=128+32K)

Cooperation of CA and OS

- Computer Architecture
  - enforce address translation and bound check under user mode
  - raise a trap at a bound violation
  - disallow updating the base and the bound register under user mode

- OS
  - split available physical memory into multiple memory slots
    - maintain a process table and a list of free memory slots
  - allocate a free slot to a new process
  - reclaim the used slot at a process termination
  - update base at context switching
  - handle a trap (exception) raised by bound check

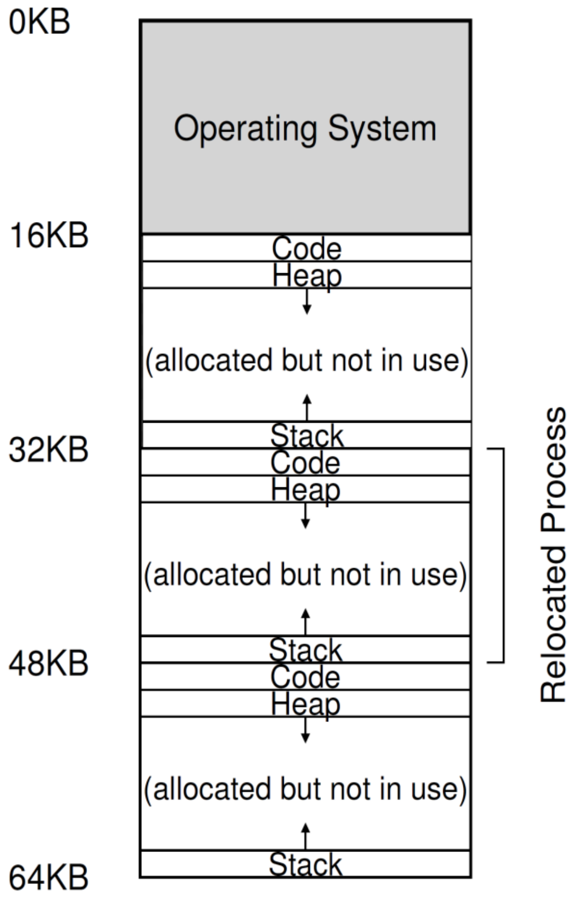
Example. Limited Direct Execution & Dynamic Relocation



OS @ run **(kernel mode)**	**Hardware**	**Program** **(user mode)**
To start process A: allocate entry in process table alloc memory for process set base/bound registers **return-from-trap** (into A)		
	restore registers of A move to **user mode** jump to A's (initial) PC	
		Process A runs Fetch instruction
	translate virtual address perform fetch	
		Execute instruction
	if explicit load/store: ensure address is legal translate virtual address perform load/store	
		(A runs...)
	Timer interrupt move to **kernel mode** jump to handler	
Handle timer decide: stop A, run B call `switch()` routine save regs(A) to proc-struct(A) (including base/bounds) restore regs(B) from proc-struct(B) (including base/bounds) **return-from-trap** (into B)		
	restore registers of B move to **user mode** jump to B's PC	
		Process B runs Execute bad load
	Load is out-of-bounds; move to **kernel mode** jump to trap handler	
Handle the trap decide to kill process B deallocate B's memory free B's entry in process table		

Limitations

- internal fragmentation
- the number of processes aforted in physical memory space
  - runtime cost of write-back at context-switching


0KB
16KB
32KB
48KB
64KB
Operating System
Code
Heap
(allocated but not in use)
Stack
Code
Heap
(allocated but not in use)
Stack
Code
Heap
(allocated but not in use)
Stack
Relocated Process