TCSS 422: OPERATING SYSTEMS

Address Translation and Segmentation

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

HARDWARE BASED ADDRESS TRANSLATION

- For each and every memory reference… address translation is performed
- Hardware transforms
  - Virtual address → physical address
- OS tracks which memory locations are free / in-use

OBJECTIVES

- Address translation
- Base and bounds
- HW and OS Support
- Memory segments
- Memory fragmentation

EXAMPLE: ADDRESS TRANSLATION

```
void func()
        int x=0;
        ...
        x = x + 3; // this is the line of code we are interested in
```

- **Load** value from memory
- **Increment** by three
- **Store** value back in memory
- In assembly…

MEMORY VIRTUALIZATION

- Using hardware support provide virtualization that is:
  - Efficient
  - Flexible
  - Secure and isolated

EXAMPLE: ADDRESS TRANSLATION - 2

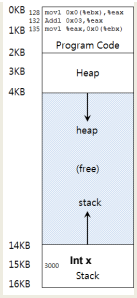
```
128 : movl 0x0(%ebx), %eax      ; load 0+ebx into eax
132 : addl $0x03, %eax          ; add 3 to eax register
135 : movl %eax, 0x0(%ebx)      ; store eax back to mem
```

- Load value at address into register (eax)
- Add (3) to eax register
- Store the value of eax back into memory

EXAMPLE: ADDRESS TRANSLATION - 3

- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

- Program's perspective:
  - Address space starts at 0
- Machine's perspective:
  - Program is located somewhere, not at 0

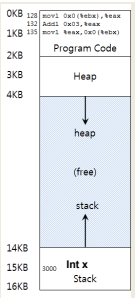


INSTRUCTION EXAMPLE

`128 : movl 0x0(%ebx), %eax`

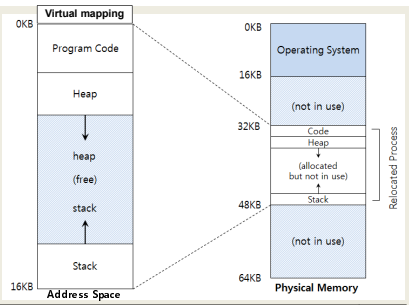
- Base = 32768
- Bounds =16384
- Fetch instruction at 128 ↑
  - Phy addr = virt addr + base reg
  - 32896 = 128 + 32768 (base)
- Execute instruction
  - Load from address
  - 48128 = 15360 + 32768 (base)
- Bounds register – terminates process if
  - Virtual address > bounds reg

$$physical\ address = virtual\ address + base$$



PLACEMENT IN PHYSICAL RAM

- 64KB Address space example
- Translation: mapping virtual to physical



MEMORY MANAGEMENT UNIT

- MMU
  - Portion of the CPU dedicated to address translation
  - Contains base & bounds registers
- Example:
  - Consider address translation
  - 4 KB (4096 bytes) address space, loaded at 16 KB physical location

	Virtual Address	Physical Address
	0	16384
	1024	17408
	3000	19384
FAULT	4400	20784 (out of bounds)

BASE AND BOUNDS

- Dynamic relocation
- Two registers base & bounds: **on the CPU**
- OS places program in memory
- Sets base register

$$physical\ address = virtual\ address + base$$

- Bounds register
  - Stores size of program address space (16KB)
- OS verifies that every address:

$$0 \leq virtual\ address < bounds$$

DYNAMIC RELOCATION OF PROGRAMS

- Hardware requirements:

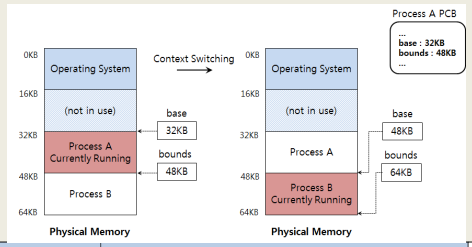
Requirements	HW support
Privileged mode	CPU modes: kernel, user
Base / bounds registers	Registers to support address translation
Translate virtual addr; check if in bounds	Translation circuitry, check limits
Privileged instruction(s) to update base / bounds regs	Instructions for modifying base/bound registers
Privileged instruction(s) to register exception handlers	Set code pointers to OS code to handle faults
Ability to raise exceptions	For out-of-bounds memory access, or attempts to access privileged instr.

OS SUPPORT FOR MEMORY VIRTUALIZATION

- For base and bounds OS support required
  - When process starts running
    - Allocate address space in physical memory
  - When a process is terminated
    - Reclaiming memory for use
  - When context switch occurs
    - Saving and storing the base-bounds pair
  - Exception handlers
    - Function pointers set at OS boot time

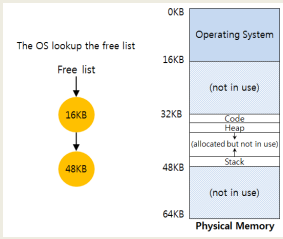
OS: WHEN CONTEXT SWITCH OCCURS

- OS must save base and bounds registers
  - Saved to the Process Control Block PCB (task_struct in Linux)



OS: WHEN PROCESS STARTS RUNNING

- OS searches for free space for new process
  - Free list: data structure that tracks available memory slots



DYNAMIC RELOCATION

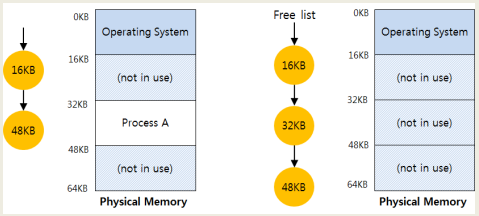
- OS can move process data when not running

1. OS deschedules process from scheduler
2. OS copies address space from current to new location
3. OS updates PCB (base and bounds registers)
4. OS reschedules process

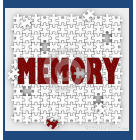
- When process runs new base register is restored to CPU
- Process doesn't know it was even moved!

OS: WHEN PROCESS IS TERMINATED

- OS places memory back on the free list

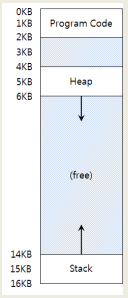


SEGMENTATION



BASE AND BOUNDS INEFFICIENCIES

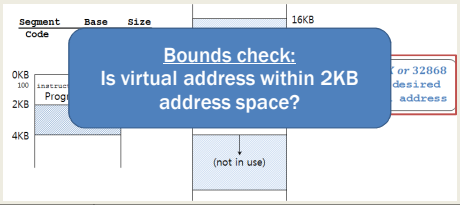
- Address space
  - Contains significant unused memory
  - Is relatively large
  - Preallocates space to handle stack/heap growth
- Large address spaces
  - Hard to fit in memory



ADDRESS TRANSLATION: CODE SEGMENT

$$physical\ address = offset + base$$

- Code segment - physically starts at 32KB (base)
- Starts at "0" in virtual address space



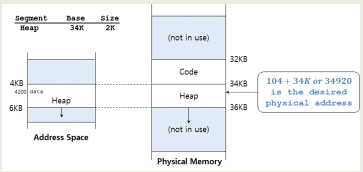
SEGMENTATION

- Address space has (3) segments
  - Contiguous portions of address space
  - Logically different: code, stack, heap
- Each segment can placed separately
- Base and bounds for each segment

ADDRESS TRANSLATION: HEAP

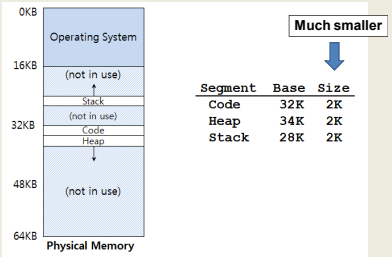
Virtual address + base is not the correct physical address.

- Heap starts at virtual address 4096
- The data is at 4200
- Offset= 4200 – 4096 = 104
- Physical address = 104 + 34816



SEGMENTS IN MEMORY

- Consider 3 segments:

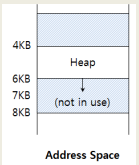


Much smaller

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

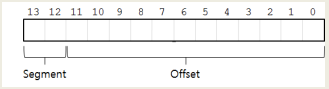
SEGMENTATION FAULT

- Access beyond the address space
- Heap starts at virtual address: 4096
- Data pointer is to 7KB (7168)
- Heap starts at 4096 + 2 KB bounds = 6144
- Offset= 4096 + 7168 > 6144

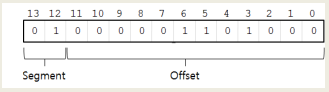


SEGMENT REGISTERS

- Used to dereference memory during translation



- First two bits identify segment type
- Remaining bits identify memory offset
- Example: virtual heap address 4200 (01000001101000)



Segment	bits
Code	00
Heap	01
Stack	10
-	11

SHARED SEGMENTS

- Code sharing: enabled with HW support
- Protection bits: track permissions to segment
- Supports storing shared libraries in memory only once
- Many programs can access them
- DLL: dynamic linked library
- .so (linux): shraed object in Linux (under /usr/lib)

Segment Register Values(with Protection)

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

SEGMENTATION DEREFERENCE

```
1   // get top 2 bits of 14-bit VA
2   Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3   // now get offset
4   Offset = VirtualAddress & OFFSET_MASK
5   if (Offset >= Bounds[Segment])
6        RaiseException(PROTECTION_FAULT)
7   else
8        PhysAddr = Base[Segment] + Offset
9        Register = AccessMemory(PhysAddr)
```

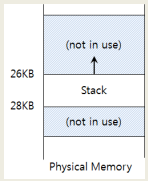
- VIRTUAL ADDRESS = 01000001101000
- SEG_MASK = 0x3000 (11000000000000)
- SEG_SHIFT = 01 → ***heap***
- OFFSET_MASK = 0xFFF (00111111111111)
- OFFSET = 000001101000 = 104
- OFFSET < BOUNDS : 104 < 2048

SEGMENTATION GRANULAITY

- Coarse-grained
  - Code segment
  - Heap segment
  - Stack segment

STACK SEGMENT

- Grows backwards (FILO)
- Requires hardware support:
- Direction bit: tracks direction segment grows



Segment Register(with Negative-Growth Support)

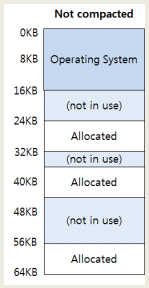
Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

SEGMENTATION GRANULARITY - 2

- Fine-grained
- Individual segments consisting of multiple smaller segments
- Segment table
  - On early systems
  - Stored in memory
  - Tracked large number of segments

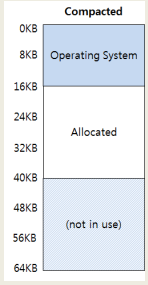
MEMORY FRAGMENTATION

- Consider how much free space?
- We'll say about 24 KB
- Request arrives to allocate a 20 KB heap segment
- Can we fulfil the request for 20 KB of contiguous memory?



COMPACTION

- Supports rearranging memory
- Can we fulfil the request for 20 KB of contiguous memory?
- **Drawback:** Compaction is slow
  - Rearranging memory is time consuming
  - 64KB is fast
  - 4GB+ … slow
- Algorithms:
  - Best fit: keep list of free spaces, allocate the most snug segment for the request
  - Others: worst fit, first fit… (in future chapters)



QUESTIONS