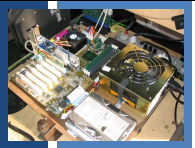
TCSS 422: OPERATING SYSTEMS



**Memory API,
Address Translation,
Memory Segmentation,
Free Space Management**

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

OBJECTIVES

- Optional Ungraded Quiz – Synchronized Array
- Homework 2 Questions
- Homework 3
- Ch. 14
  - Memory API
- Ch. 15
  - Address Translation
- Ch. 16
  - (Memory) Segmentation
- Ch. 17
  - Free Space Management

FEEDBACK FROM 2/14

- What is a bounded buffer, and when is it used?
- What is piping in operating systems?
- What is a condition variable?

FEEDBACK - 2

- Homework #2: I'm using "for_each_process(task)" but Ubuntu can't find this function… HELP!?
- In your kernel module make file, note the required files:

```
All target:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

- Check to be sure you have this kernel sources:
- sudo apt-get install build-essential linux-headers-`uname –r`
- Helpful command - list kernel modules:
  `$lsmod`

FEEDBACK - 3

- "TCSS 422 has overlap with TCSS 333 for memory maps, memory/address translation"…
- Initial chapters of memory virtualization may be review
- However, memory virtualization spans chapters 13, 14, 15, 16, 17, 18, 19, 20, 21, 22. . . *(10 chapters)*
  - *Suspect this is not all review…*

CHAPTER 14: THE MEMORY API

MALLOC

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Allocates memory on the heap
- size_t unsigned integer (must be +)
- size size of memory allocation in bytes
- Returns
- SUCCESS: A void * to a memory address
- FAIL: NULL
- sizeof() often used to ask the system how large a given datatype or struct is

SIZEOF()

- Not safe to assume data type sizes using different compilers, systems
- Dynamic array of 10 ints
- Static array of 10 ints

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

```
4
```

```
int x[10];
printf("%d\n", sizeof(x));
```

```
40
```


FREE()

```
#include <stdlib.h>

void free(void* ptr)
```

- Free memory allocated with malloc()
- Provide: (void *) ptr to malloc'd memory
- Returns: nothing

What will this code do?

```
#include<stdio.h>

int * set_magic_number_a()
{
  int a =53247;
  return &a;
}

void set_magic_number_b()
{
  int b = 11111;
}

int main()
{
  int * x = NULL;
  x = set_magic_number_a();
  printf("The magic number is=%d\n",*x);
  set_magic_number_b();
  printf("The magic number is=%d\n",*x);
  return 0;
}
```

10

What will this code do?

```
#include<stdio.h>

int * set_magic_number_a()
{
  int a =53247;
  return &a;
}

void set_magic_number_b()
{
  int b = 11111;
}

int main()
{
  int * x = NULL;
  x = set_magic_number_a();
  printf("The magic number is=%d\n",*x);
  set_magic_number_b();
  printf("The magic number is=%d\n",*x);
  return 0;
}
```

Output:

```
$ ./pointer_error
The magic number is=53247
The magic number is=11111
```

We have not changed *x but the value has changed!!

Why?

11

DANGLING POINTER (1/2)

- Dangling pointers arise when a variable referred (a) goes "out of scope", and it's memory is destroyed/overwritten (by b) without modifying the value of the pointer (*x).
- The pointer still points to the original memory location of the deallocated memory (a), which has now been reclaimed for (b).

DANGLING POINTER (2/2)

- Fortunately in the case, a compiler warning is generated:

```
$ g++ -o pointer_error -std=c++0x pointer_error.cpp

pointer_error.cpp: In function 'int*
set_magic_number_a()':
pointer_error.cpp:6:7: warning: address of local
variable 'a' returned [enabled by default]
```

- This is a common mistake - - - accidentally referring to addresses that have gone "out of scope"

CALLOC()

```
#include <stdlib.h>

void *calloc(size_t num, size_t size)
```

- Allocate "C"lear memory on the heap
- Calloc wipes memory in advance of use…
- `size_t num` : number of blocks to allocate
- `size_t size` : size of each block(in bytes)
- Calloc() prevents…

```
char *dest = malloc(20);
printf("dest string=%s\n", dest);

dest string=►►F
```


REALLOC()

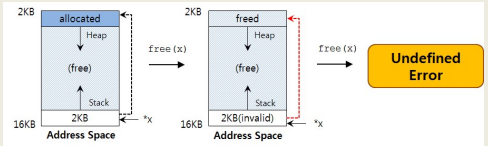
```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Resize an existing memory allocation
- Returned pointer may be same address, or a new address
  - New if memory allocation must move
- `void *ptr`: Pointer to memory block allocated with malloc, calloc, or realloc
- `size_t size`: New size for the memory block(in bytes)
- EXAMPLE: realloc.c
- EXAMPLE: nom.c

DOUBLE FREE

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory
free(x); // free repeatedly
```

- Can't deallocate twice
- Second call core dumps



SYSTEM CALLS

- brk(), sbrk()
- Used to change data segment size (the end of the heap)
- Don't use these
- Mmap(), munmap()
- Can be used to create an extra independent "heap" of memory for a user program
- See man page

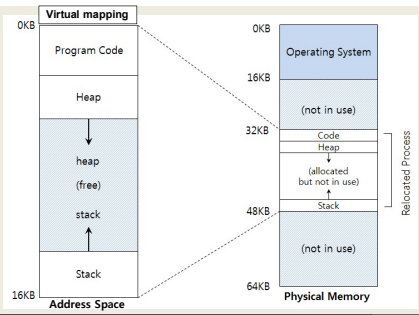
CHAPTER 15: ADDRESS TRANSLATION

OBJECTIVES

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

ADDRESS TRANSLATION

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



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:

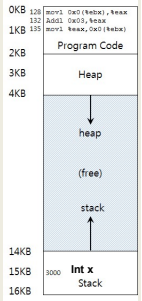
$$0 \leq virtual\ address < bounds$$

INSTRUCTION EXAMPLE

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

- Base = 32768
- Bounds =16384
- Fetch instruction at 128 (virt addr) ↑
  - Phy addr = virt addr + base reg
  - 32896 = 128 + 32768 (base)
- Execute instruction
  - Load from address (var x is @ 15kb=15360)
  - 48128 = 15360 + 32768 (base) -- found x…
- Bounds register: terminate process if
  - ACCESS VIOLATION: Virtual address > bounds reg

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



MEMORY MANAGEMENT UNIT

- MMU
  - Portion of the CPU dedicated to address translation
  - Contains base & bounds registers
- Base & Bounds 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)

DYNAMIC RELOCATION OF PROGRAMS

- Hardware requirements:

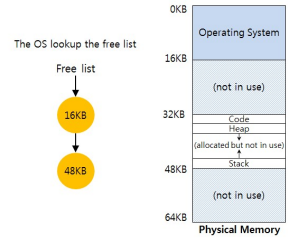
Requirements	HW support
Privileged mode	CPU modes: kernel, user
Base / bounds registers	Registers to support address translation
Translate virtual addr; check if in bounds	Translation circuitry, check limits
Privileged instruction(s) to update base / bounds regs	Instructions for modifying base/bound 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

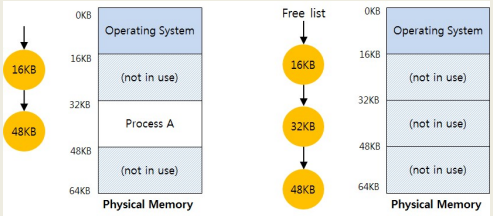
OS: WHEN PROCESS STARTS RUNNING

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



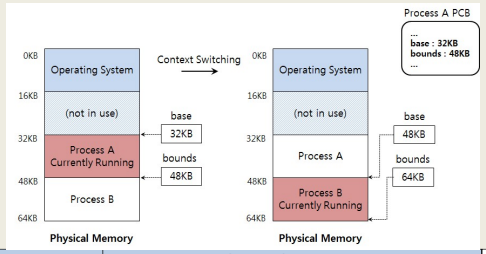
OS: WHEN PROCESS IS TERMINATED

- OS places memory back on the free list



OS: WHEN CONTEXT SWITCH OCCURS

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



DYNAMIC RELOCATION

- OS can move process data when not running

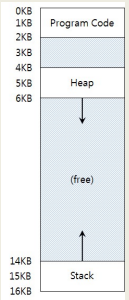
1. OS deschedules process from scheduler
2. OS copies address space from current to new location
3. OS updates PCB (base and bounds registers)
4. OS reschedules process

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

CHAPTER 16: 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
- How can these issues be addressed?

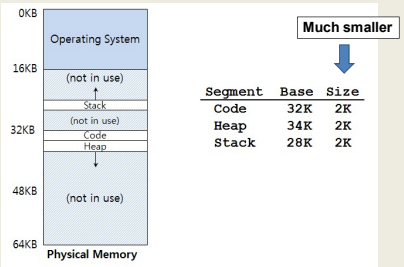


MULTIPLE SEGMENTS

- Memory segmentation
- Address space has (3) segments
  - Contiguous portions of address space
  - Logically separate segments for: code, stack, heap
- Each segment can placed separately
- Track base and bounds for each segment (registers)

SEGMENTS IN MEMORY

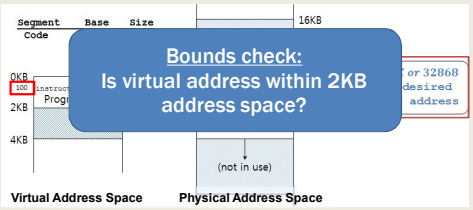
- Consider 3 segments:



ADDRESS TRANSLATION: CODE SEGMENT

$$physical\ address = offset + base$$

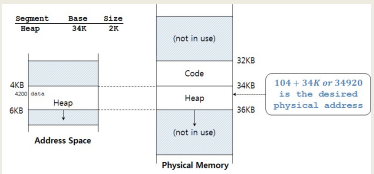
- Code segment - physically starts at 32KB (base)
- Starts at "0" in virtual address space



ADDRESS TRANSLATION: HEAP

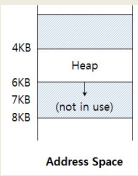
Virtual address + base is not the correct physical address.

- Heap starts at virtual address 4096
- The data is at 4200
- Offset= 4200 – 4096 = 104 (virt addr – virt heap start)
- Physical address = 104 + 34816 (offset + heap base)



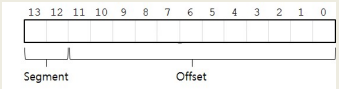
SEGMENTATION FAULT

- Access beyond the address space
- Heap starts at virtual address: 4096
- Data pointer is to 7KB (7168)
- Is data pointer valid?
- Heap starts at 4096 + 2 KB seg size = 6144
- Offset= 7168 > 4096 + 2048 (6144)

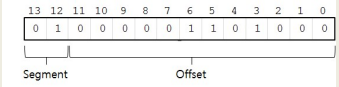


SEGMENT REGISTERS

- 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

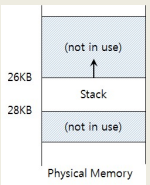
SEGMENTATION DEREFERENCE

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

- VIRTUAL ADDRESS = 01000001101000 (on heap)
- SEG_MASK = 0x3000 (11000000000000)
- SEG_SHIFT = 01 → ***heap*** (mask gives us segment code)
- OFFSET_MASK = 0xFFF (00111111111111)
- OFFSET = 000001101000 = 104 (isolates segment offset)
- OFFSET < BOUNDS : 104 < 2048

STACK SEGMENT

- Stack 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

SHARED CODE SEGMENTS

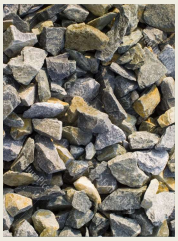
- Code sharing: enabled with HW support
- Supports storing shared libraries in memory only once
- DLL: dynamic linked library
- .so (linux): shraed object in Linux (under /usr/lib)
- Many programs can access them
- Protection bits: track permissions to segment

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 GRANULARITY

- Coarse-grained
- Manage memory as large purpose based segments:
  - Code segment
  - Heap segment
  - Stack segment

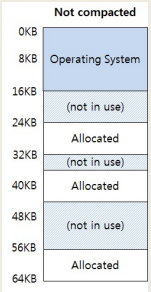


SEGMENTATION GRANULARITY - 2

- Fine-grained
- Manage memory as list of segments
- Code, heap, stack segments composed of multiple smaller segments
- Segment table
  - On early systems
  - Stored in memory
  - Tracked large number of segments

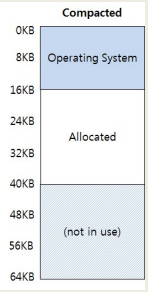
MEMORY FRAGMENTATION

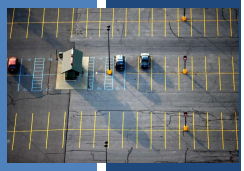
- Consider how much free space?
- We'll say about 24 KB
- Request arrives to allocate a 20 KB heap segment
- Can we fulfil the request for 20 KB of contiguous memory?



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)





CHAPTER 17: FREE SPACE MANAGEMENT

FREE SPACE MANAGEMENT

- Management of memory using
- Only fixed-sized units
  - Easy: keep a list
  - Memory request → return first free entry
    - Simple search
- With variable sized units
  - More challenging
  - Results from variable sized malloc requests
  - Leads to fragmentation

QUESTIONS