CSIS 429 Operating Systems

Lecture 4: Memory management – Address Spaces & Memory API

September 16th 2020

Textbook chapters

- Read “Address Spaces” and “Memory API”

Intro	**Virtualization**		**Concurrency**	**Persistence**	**Appendices**
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	Flash-based SSDs
		21 Swapping: Mechanisms	34 *Summary*	44 Data Integrity and Protection	
		22 Swapping: Policies		45 *Summary*	
		23 Case Study: VAX/VMS		46 *Dialogue*	
		24 *Summary*		47 Distributed Systems	
				48 Network File System (NFS)	
				49 Andrew File System (AFS)	
				50 *Summary*	

Review: Operating Systems virtualize CPUs

- Operating system scheduler
  - Makes it look like each process has the CPU to itself

- Hard parts
  - How an OS does this:
    - context switch to save register state
    - monitoring each process and adjusting priorities
  - Being fair to all running processes - MLFQ

Operating Systems virtualize memory

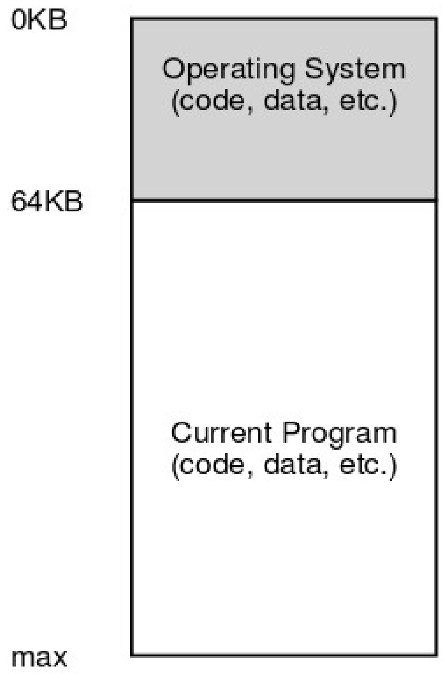
- Remember: CPUs do very simple tasks
  - Fetch instruction from memory
  - Decode instruction
  - Execute instruction, often accessing memory

To virtualize memory, the OS needs to virtualize how memory is accessed

→ give each process its own private “address space” - the illusion that the process alone uses memory.

Early Operating Systems: DOS

- DOS ran one program at a time.
- Each program's view of memory:



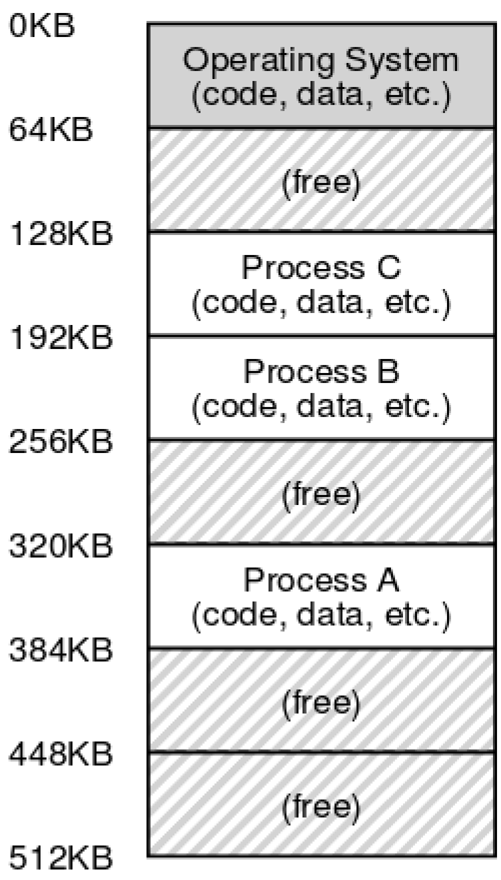
Operating Systems: The Early Days

Early OSes: Multiprogramming

Being able to handle multiple processes "simultaneously" became important.

One solution: buy more memory

Each program's view of memory can be the same

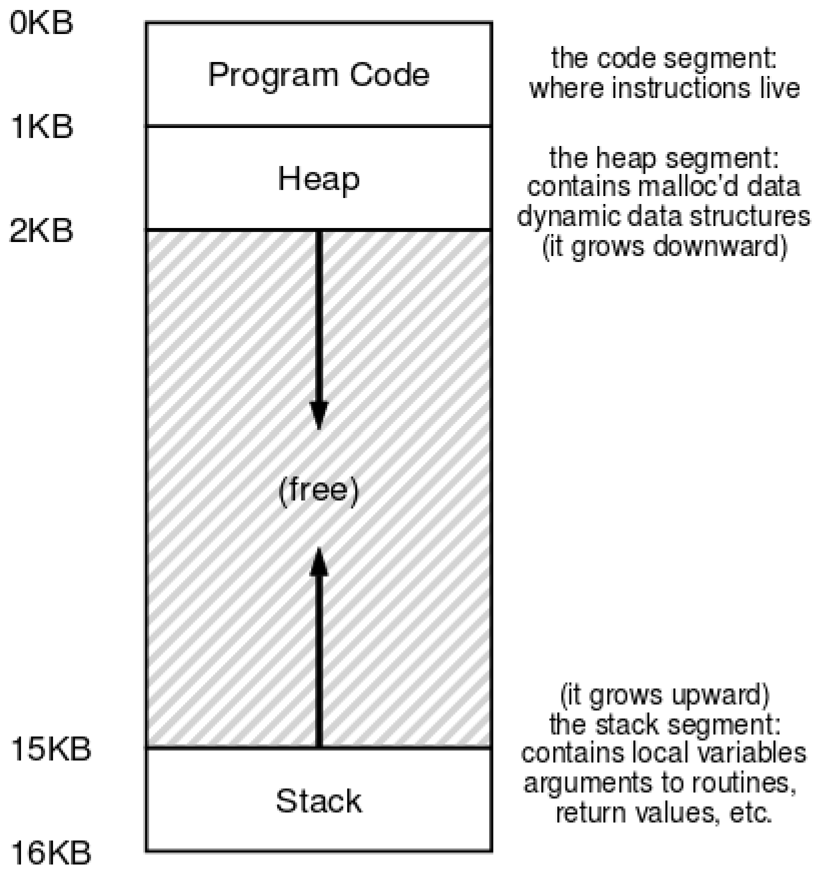


Three Processes: Sharing Memory

Process address space in simple OS

- Each process has its own private "address space"
  a process "thinks" its memory starts at physical memory location 0 and ends at 16KB

  → OS translates every address generated by a process to a DRAM address by adding a "Relocation" register value.



Virtual address space

- To virtualize memory, the OS needs to virtualize how memory is accessed

  each process has its own “address space”

  → OS has to translate every (virtual) address generated by a process to a DRAM address.

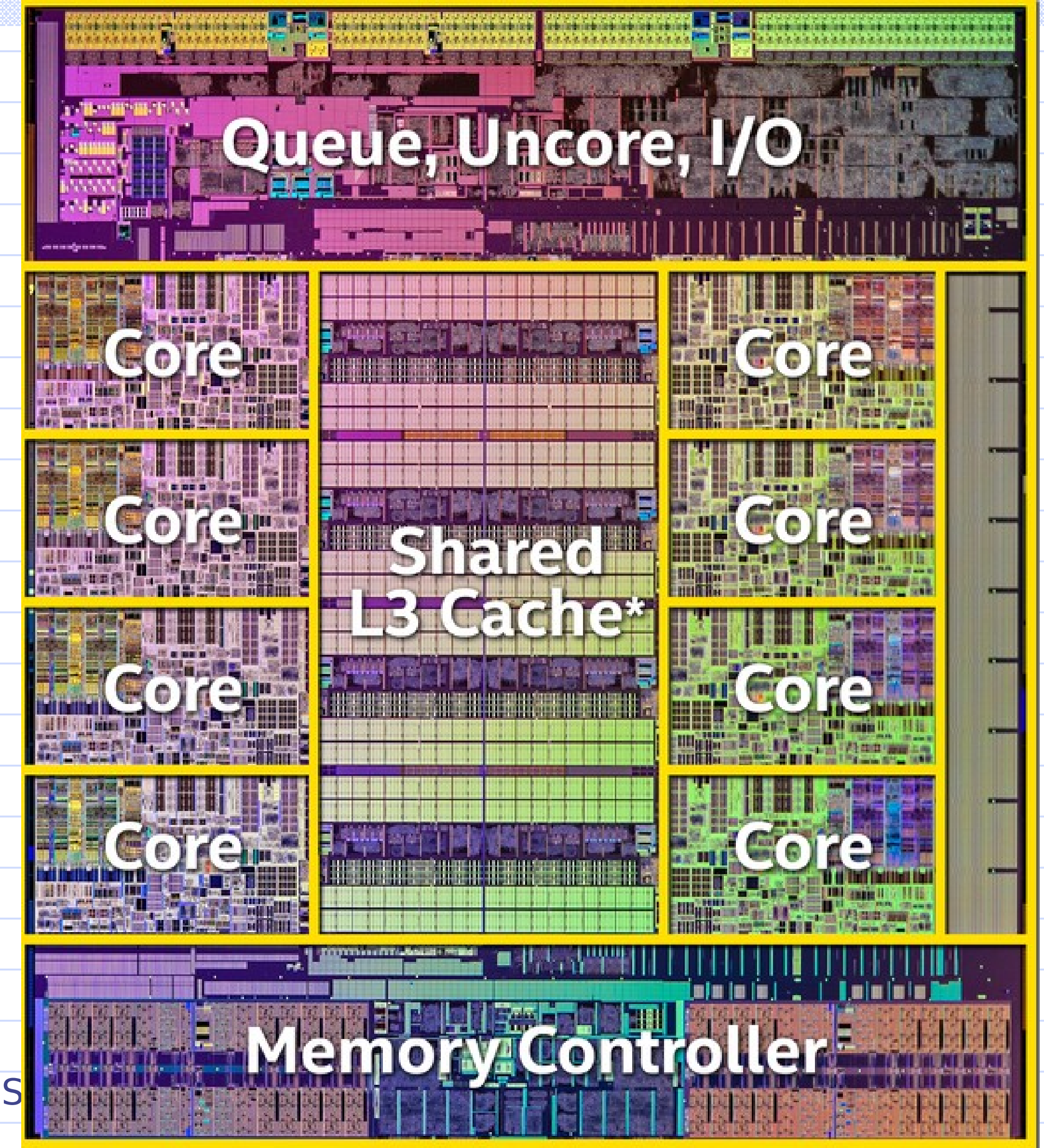
  → requires a lot of help from the hardware

Block diagram of a newer Intel CPU

Cores have ALU, FPU, L1 and L2 caches.

Note large areas for
- L3 cache
- Memory Controller

Uncore is an Intel idea to put non-core functions like interconnections between cores, Cache control, etc.



Address Translation

Every address generated by a process is treated as a "virtual address" by the OS

OS translates each virtual address to a DRAM address.

The "crux" of the problem:

How can the OS build the abstraction of a private address space while actually having many processes use physical memory?

And doing so securely? And efficiently?

Memory API in Unix

The “crux” of the problem:

How does Unix allow programs to allocate and manage memory?

What some pitfalls in allocating memory and how can we avoid them?

Two Types of Memory in a C program: Stack

- Allocation and deallocation are managed implicitly

```
void f(){
    int x; // x lives on the stack
    ...
}
```

→ compiler inserts code to use stack space

→ OS and CPU help

→ code inserted by compiler deallocates upon return

Two Types of Memory in a C program: Heap

- Allocation and deallocation are managed explicitly

```
void f(){
    char *p;           // p lives on the stack
    p = malloc(10); // p points to 10 bytes on heap
    ...
}
```

→ Memory allocated on the heap

→ Stays allocated even after function returns

→ Can lead to “memory leak” if not careful

Heap: malloc

```
p = malloc(10); // p points to 10 bytes on heap
```

The `malloc` function is defined in `stdlib.h`

The parameter that goes into `malloc` is of type `size_t`

- Not a good idea to use a “magic number” like 10
- Better to give some indication of what it’s for, like: sizeof(int)*10

The return value is of type “void*” - untyped pointer.

should return NULL if allocation fails but currently most implementations don’t!

Be careful with sizeof

What will this print?

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

How about this?

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

Deallocating memory: `free`

- It is good practice to match every execution of a `malloc` with a `free`

```
int* x = malloc(10*sizeof(int));
...
free(x);
```

Pitfall: forgetting to allocate memory

This code will likely cause a segmentation fault

```
char* src = "hello";
char* dst;           // oops! unallocated
strcpy(dst, src); // segfault and die
```

Remember: “It compiled” or “It ran” **!=** “It is correct”

Pitfall: forgetting to allocate memory

- Fix:

```
char* src = "hello";
char* dst = (char*) malloc(strlen(src)+1);
strcpy(dst, src); // works!
```

Other pitfalls

- Forgetting to initialize allocated memory. Can use calloc
- Forgetting to free memory → memory leak!
- Freeing memory before you are done → dangling pointer
- Freeing same memory twice → can crash

Next Lectures: How Address Translation works

- Address Translation
- Segmentation
- Managing Free Space
- Paging
- Translation Lookaside Buffer