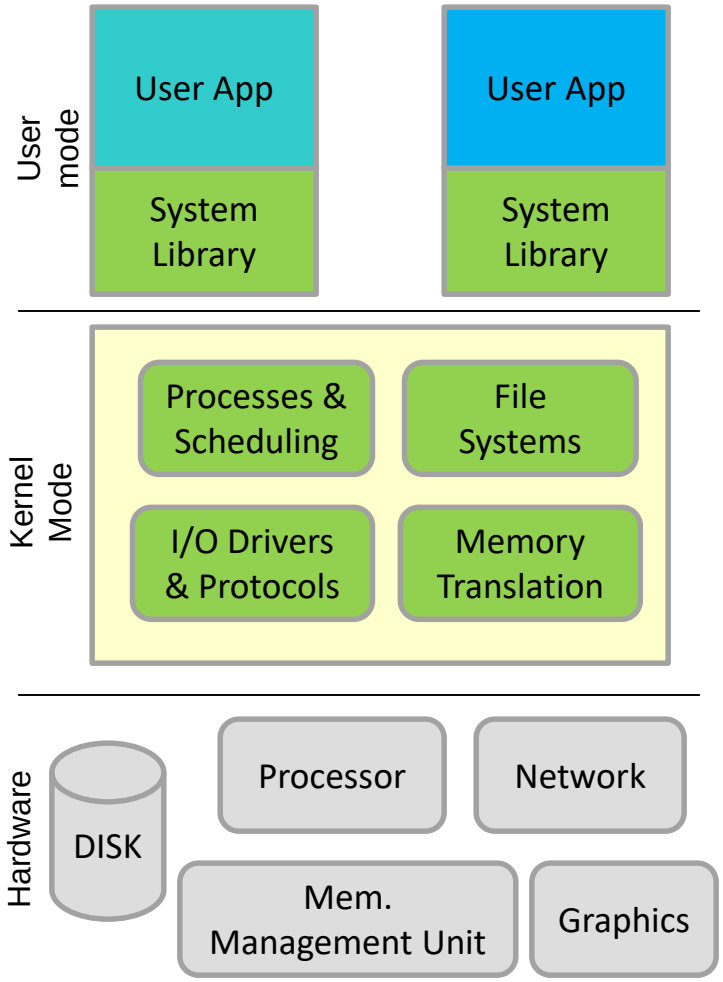
CSCI 350
Ch. 1 Interlude – Review of Architecture, Threading, and the C Language

Mark Redekopp
Michael Shindler

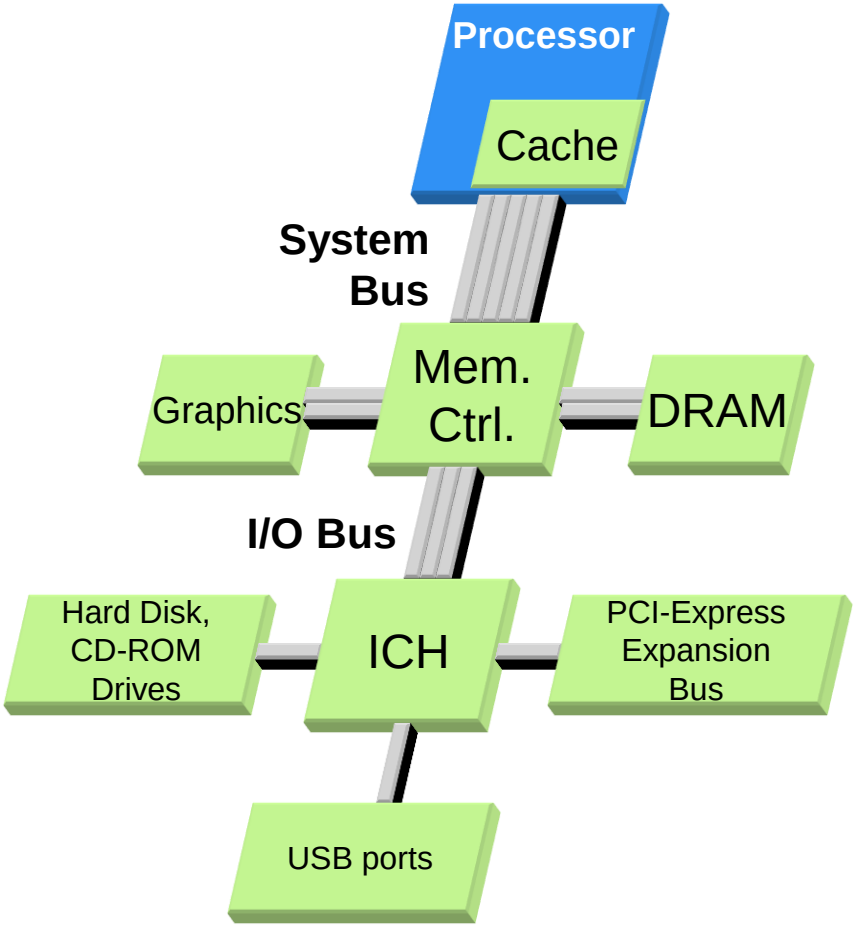
Recall

- An OS is a piece of software that manages a computer's resources
- Resources needing management:
  - CPU (threads and processes)
  - Memory (Virtual memory, protection)
  - I/O (Abstraction, interrupts, protection)
- Let's look a bit deeper at some of the hardware



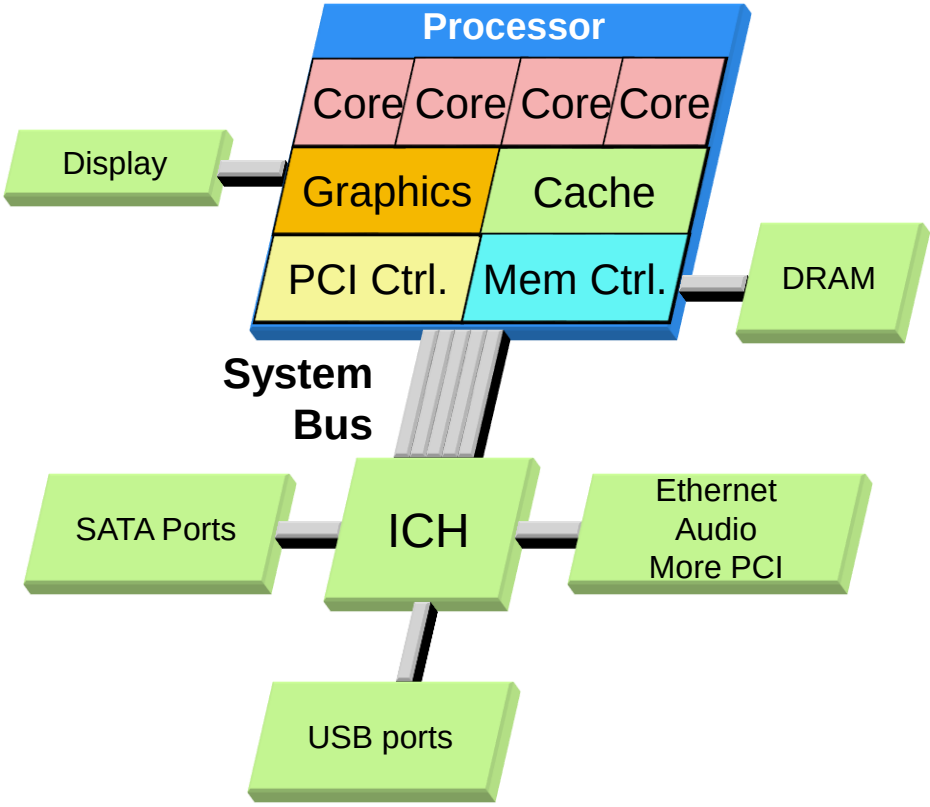
PC Architecture (~10 years ago)

- Place component with the highest data needs closest to the processor
- Memory controller
  - DRAM requires dedicated controller to interpret system bus transactions to memory control signals
  - High bandwidth connection via system bus
- ICH (I/O Controller Hub)
  - Implements USB controller, Hard Drive, CD-ROM and PCI expansion interfaces along with other I/O interfaces
  - Provides lower bandwidth capabilities over I/O bus



Modern PC Architecture

- Moore's Law has allowed greater integration levels
  - Multiple cores and greater levels of cache
  - Memory controller, graphics, and high-speed I/O are integrated onto the processor die

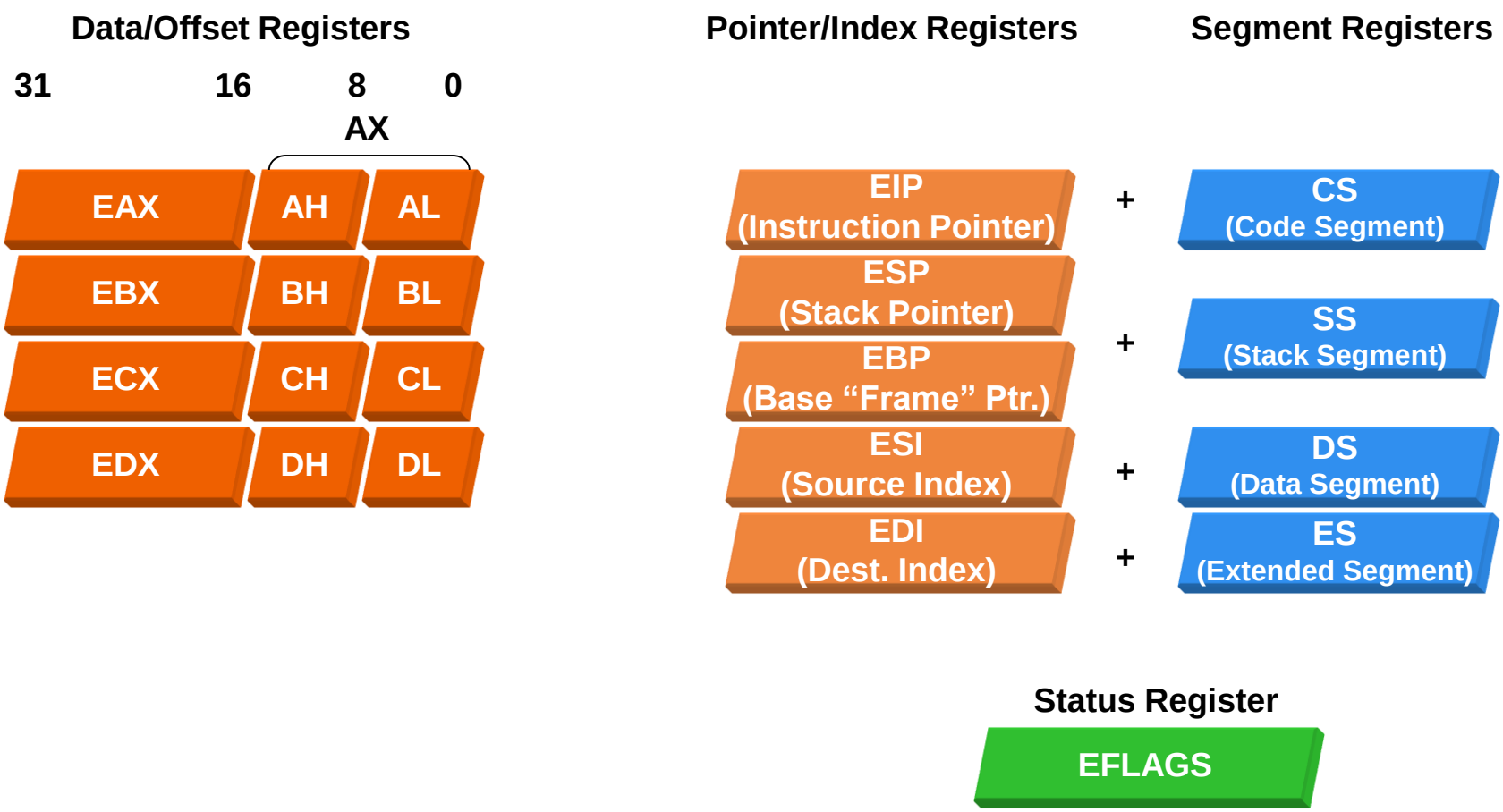


CS 201/356 REVIEW

Assembly

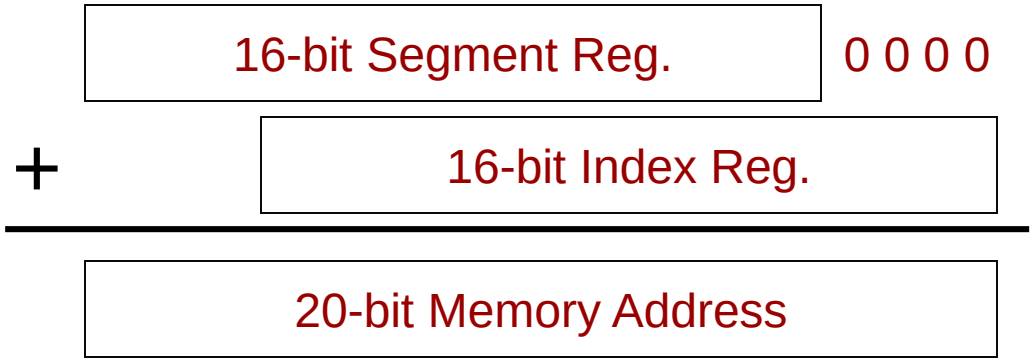
- High-level code is broken into basic instructions
- Programmer's model of a machine (CPU and memory) is everything the programmer needs to know to write and translate software code to 1's and 0's
  - Registers
  - Program Counter (Instruction Pointer)
  - Status/mode flags
  - Data sizes
  - I/O addressing (memory-mapped I/O)
  - Various cache/TLB instructions/settings

Intel (IA-32) Architectures



Real Mode Addressing

- How to make 20-bit addr. w/ 16-bit reg's???
  - Use two 16-bit registers
  - (Segment register * 16) + Index Reg.
- Format:
  - Seg Reg:Index Reg (e.g. CS:IP)



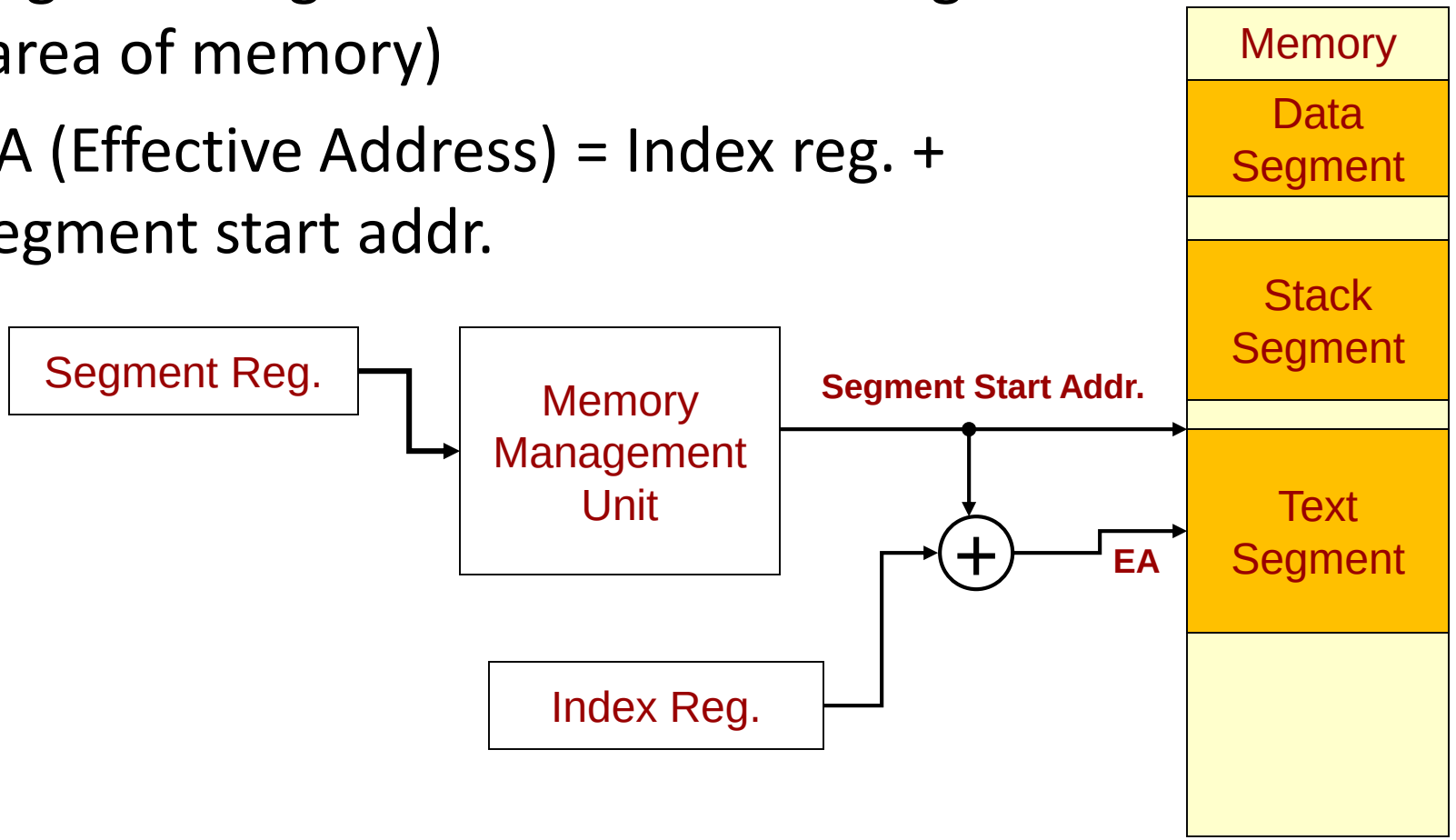
Examples:

0x1A5C:0x405E → 0x1A5C0 + 0x405E = 0x1E61E

0x74AB:0xE892 → 0x74AB0 + 0xE892 = 0x83345

Protected Mode Addressing

- Segment Register value selects a segment (area of memory)
- EA (Effective Address) = Index reg. + segment start addr.



IA-32 Addressing Modes

Name	Example	Effective Address
Immediate	MOV EAX,5	R[d] = immed.
Direct	MOV EAX,[100]	R[d] = M[100]
Register	MOV EAX,EDX	R[d] = R[s]
Register indirect	MOV EAX,[EBX]	R[d] = M[R[s]]
Base w/ Disp.	MOV EAX,[EBP+60]	R[d] = M[R[s]+d]
Base w/ Index	MOV EAX,[EBP + ESI*4]	R[d] = M[(Reg1)+(Reg2)*S]
Base w/ Index & Disp.	MOV EAX,[EBP+ESI*4+100]	R[d] = M[(Reg1)+(Reg2)*S + d]

IA-32 Instructions

- Stack aware instructions
  - 'call' / 'ret' automatically push/pop return address onto stack
  - 'push' / 'pop' instructions for performing these operations
- Memory / Register architecture
  - One operand can be located in memory
    - `add eax, [ebp]` # adds reg. EAX to value pointed at by EBP
- Specialized Instructions
  - xchg src1, src2 # exchanges/swaps operands
  - string copy and compare instructions

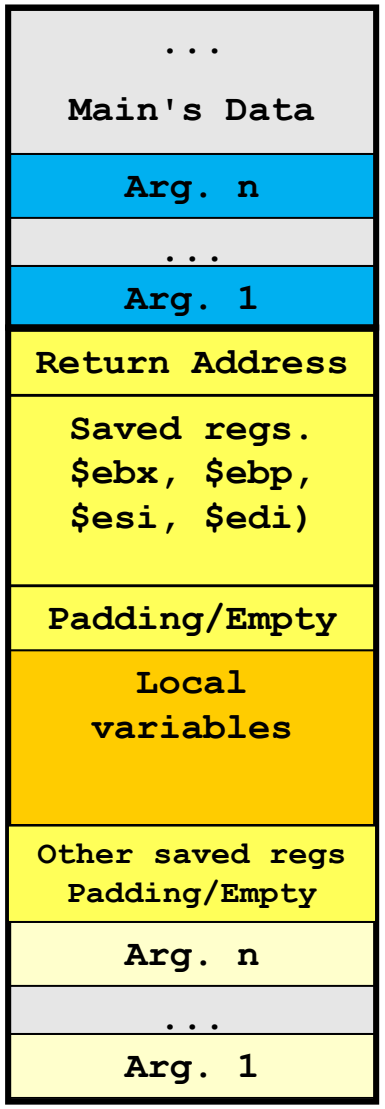
STACK FRAMES & FUNCTION CALLS

Stack Frame Organization

Main's Stack Frame

SUB1's Stack Frame

Stack Growth



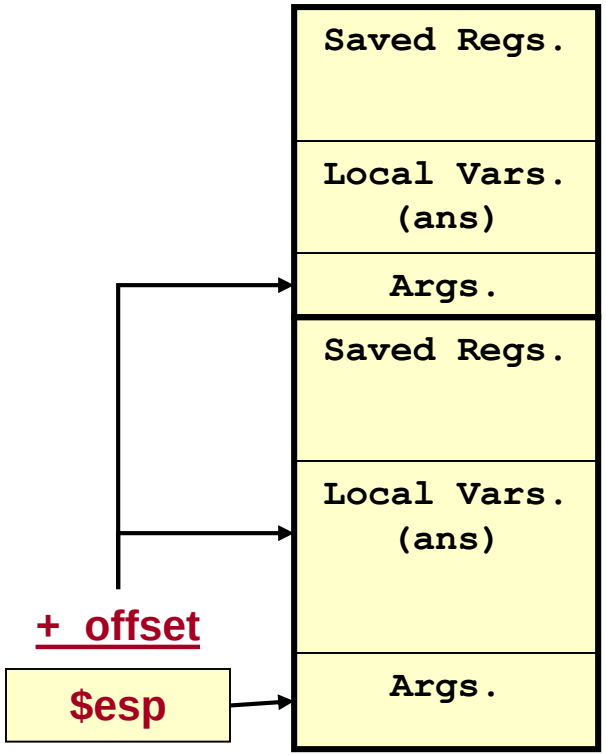
- Args. for SUB1 (or any other routine called by main)
- Return address pushed by 'call' instruc.
- ABI requires specific registers be saved (those shown to the left). Others are allowed to be overwritten (i.e. caller needed to save them if he/she wanted them)
- Empty slot if needed because local data space must be double-word aligned (multiple of 8)
- Space for any local/automatic declared variables in a routine
- Space for any "non-preserved" registers that need to be saved when SUB1 calls another routine
- Args. for any other routine called by SUB1

Building a Stack Frame

- Caller routine…
  - Save/push caller's "unpreserved" registers (use 'push' instruction)
  - Push arguments onto stack
  - Execute 'call' (Saves return address onto stack [i.e. CS:IP])
- Callee routine…
  - Save/push "preserved" registers ($ebx, $ebp, $edi, $esi)
  - Allocate space for local variables by moving $esp down
  - Save/push "non-preserved registers" (e.g. $ecx, if needed)
  - Execute Code
  - Place return value in $eax
  - Restore/pop "non-preserved registers"
  - Deallocate local variables by moving $esp up
  - Restore/pop "preserved registers"
  - Return using 'ret' (Pops the return address off the stack)
- Caller routine…
  - Pop arguments
  - Restore/pop "non-preserved registers"

Accessing Values on the Stack

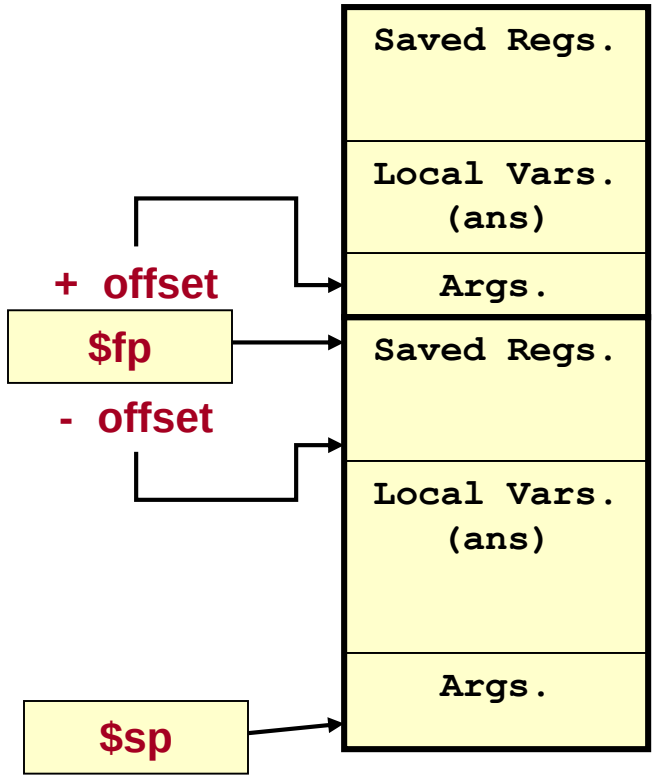
- Stack pointer ($esp) is usually used to access only the top value on the stack
- To access arguments and local variables, we need to access values buried in the stack
- We can use an offset from $esp
- However if $esp moves (due to additional push/pops) then we have to use a different offset to get the same data from the stack
  - This can make readying the code harder
- Possible improvement: Use a pointer that won't change during function execution (aka the base/frame pointer, $ebp)



To access parameters we could try to use some displacement [i.e. ($esp,8)]

Base/Frame Pointer

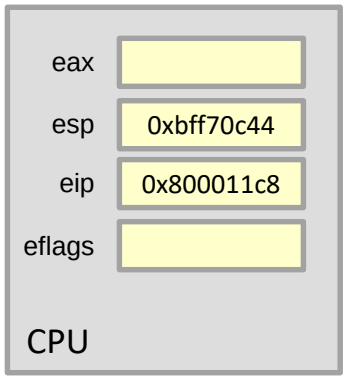
- Use a new pointer called Base/Frame Pointer ($ebp) to point to the base of the current routines frame (i.e. the ***first*** word of the stack frame)
- $ebp will not change during the course of subroutine execution
- Can use constant offsets from $ebp to access parameters or local variables
  - Key 1: $ebp doesn't change during subroutine execution
  - Key 2: Number of arguments, local variables, and saved registers is a known value

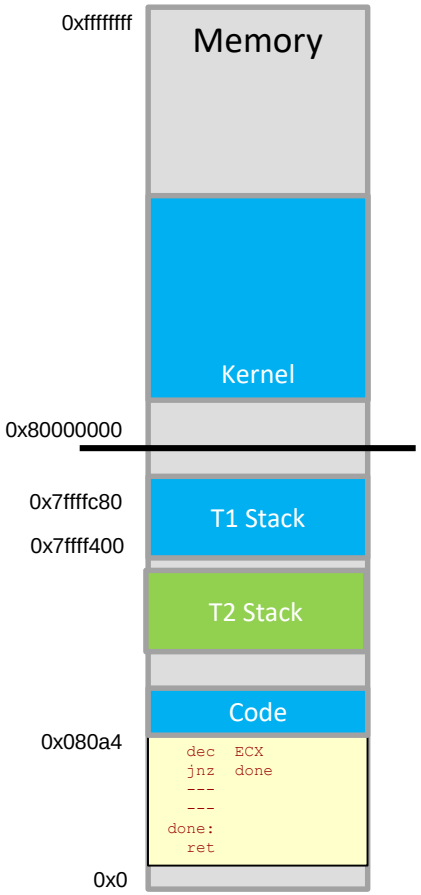


THREAD BASICS

What is a Thread?

- Registers + PC + Stack representing a single execution sequence
- Independently scheduled unit of code





Atomic Operations

- Operation appears to be indivisible
- Given `int x=0;` and multiple threads
- Is `x++;` atomic?
- No…it translates to:
  - load/move
  - add
  - store/move

Code

```
Thread 1:
x++;
  load x
  add 1,x
  store x
```

```
Thread 2:
x++;
  load x
  add 1,x
  store x
```

Ordering Option 1

```
Thread 1:
  load x
  add 1,x
  store x
```

```
Thread 2:
  load x
  add 1,x
  store x
```

Result 2

Ordering Option 2

```
Thread 1:
  load x
  add 1,x
  store x
```

```
Thread 2:
  load x
  add 1,x
  store x
```

Result 1

Synchronization Primitives

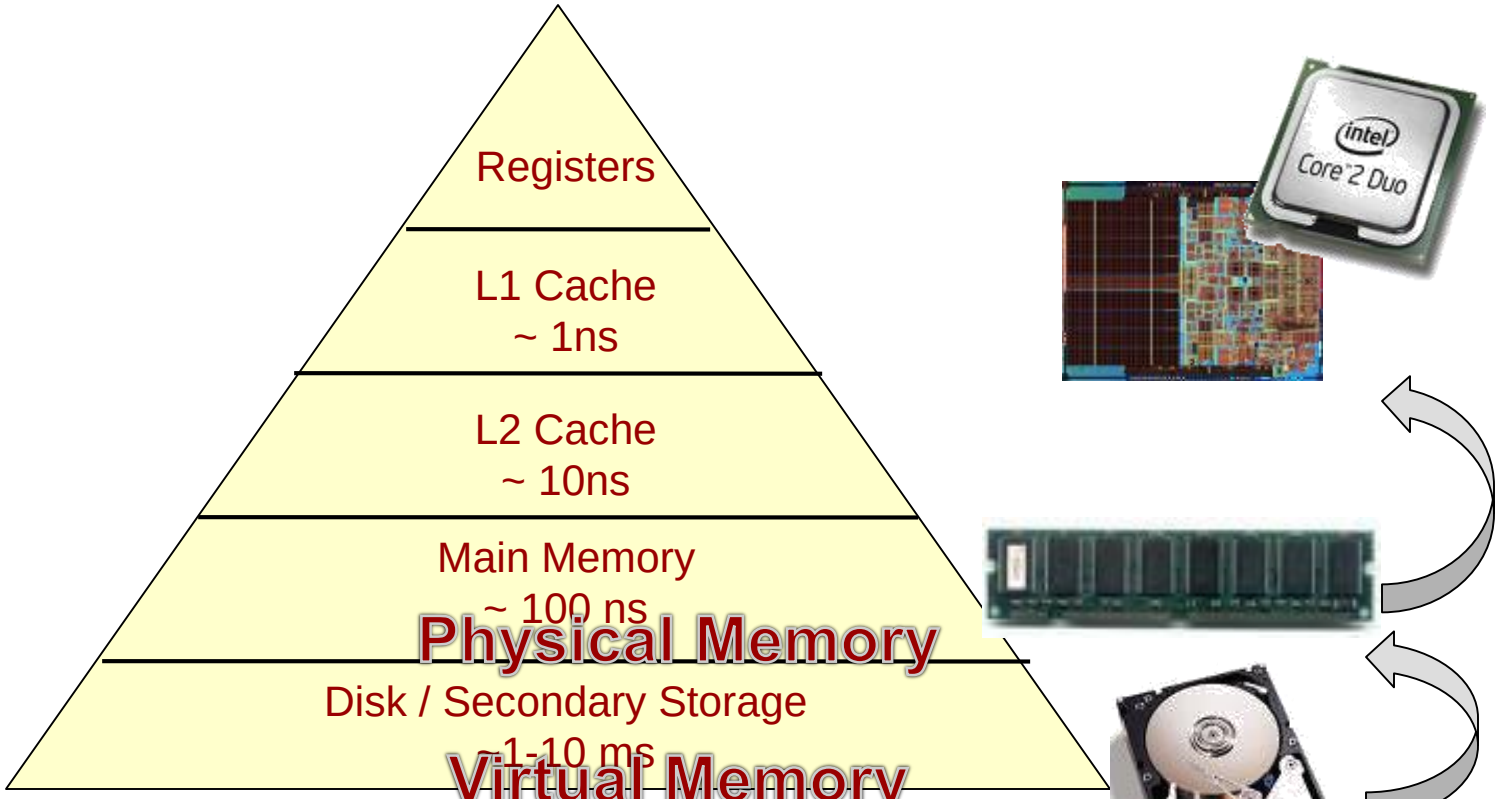
- What are the key operations for these primitives?
- Locks
- Condition Variables
- Semaphores

Caching & Virtual Memory

- What do you remember about caching and virtual memory?

Memory Hierarchy & Caching

- Lower levels act as a cache for upper levels

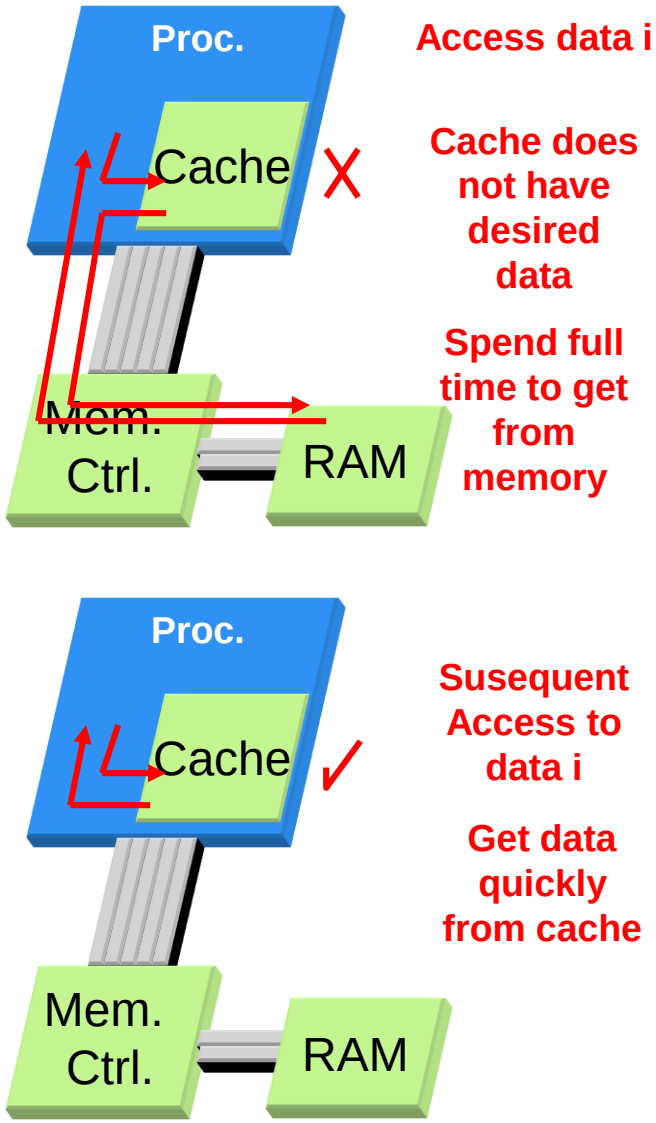


L1/L2 is a "cache" for main memory

Virtual memory provides each process its own address space in secondary storage and uses main memory as a cache

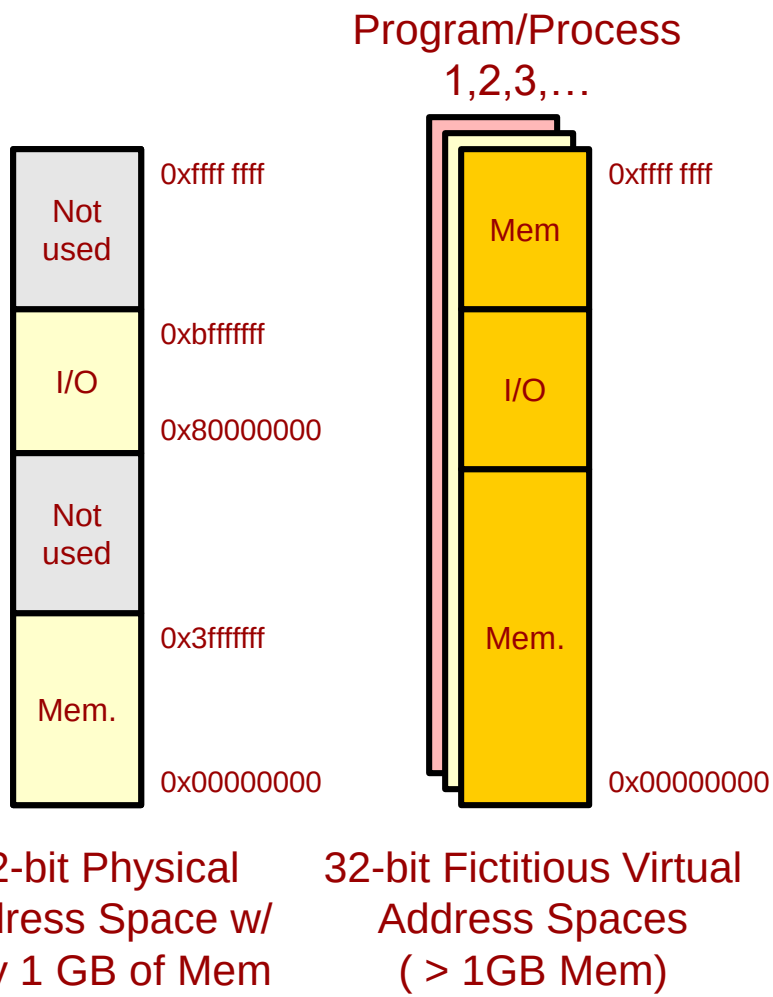
Cache Example

- When processor attempts to access data it will first check the cache
  - If the cache does not have the data, it must go to the main memory (RAM) to access it
  - If the cache has the desired data, it can supply it quickly



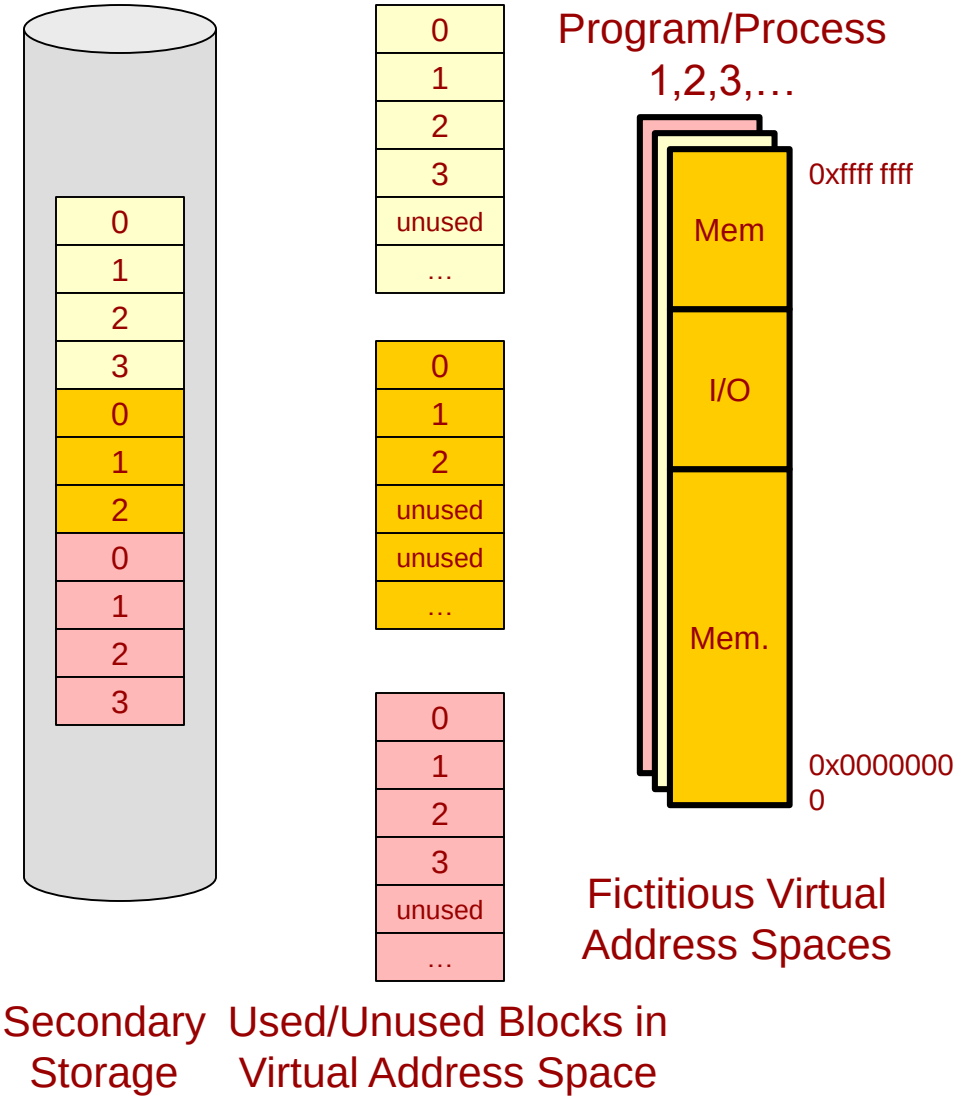
Address Spaces

- Physical address spaces corresponds to the actual system address bus width and range (i.e. main memory and I/O)
- Each process/program runs in its own private "virtual" address space
  - Virtual address space can be larger (or smaller) than physical memory
  - Virtual address spaces are protected from each other
- **Process (def.): Address Space + Threads**



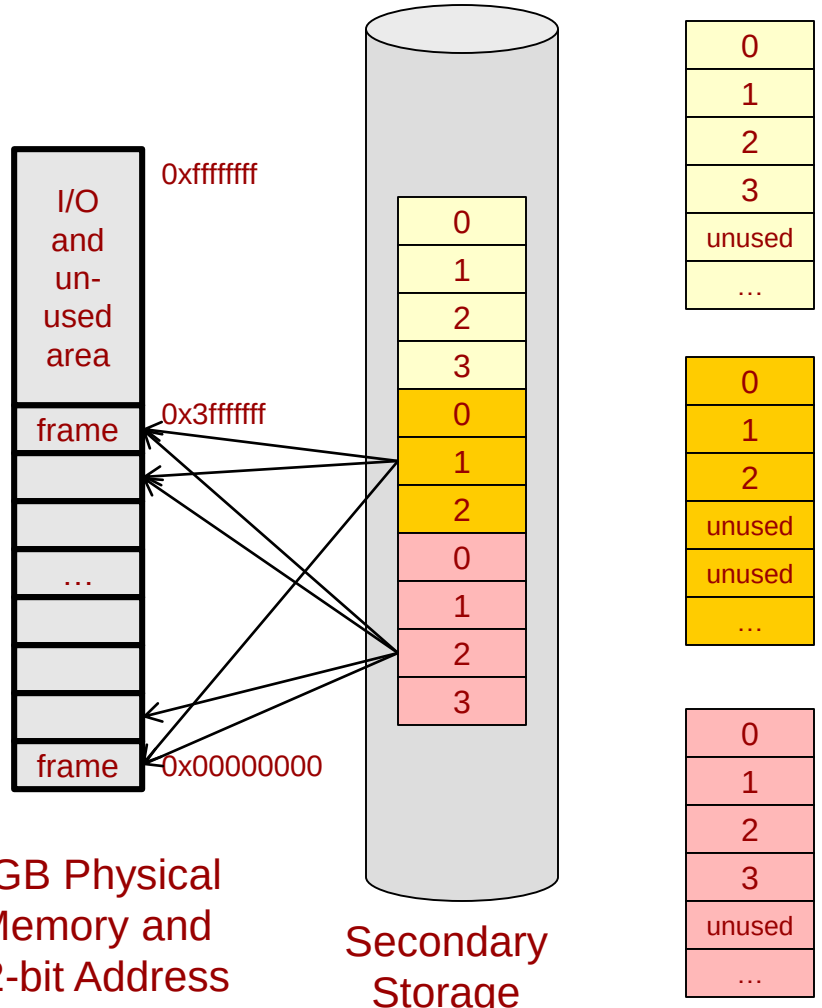
Virtual Address Spaces

- Virtual address spaces are broken into blocks called “pages”
- Depending on the program, much of the virtual address space will not be used
- All used pages are “housed” in secondary storage (hard drive)



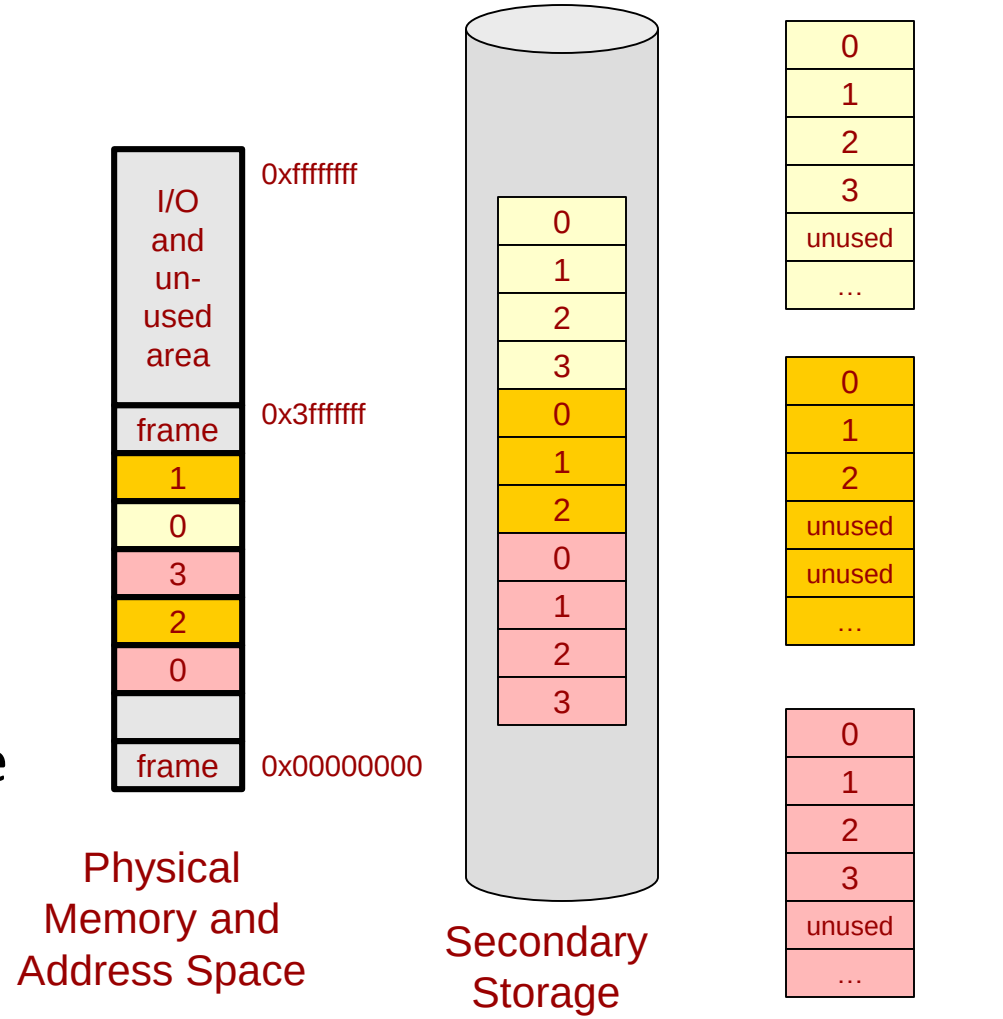
Physical Address Space

- Physical memory is broken into page-size blocks called “page frames”
- Multiple programs are run concurrently and their pages (code & data) need to reside in physical memory
- Physical memory acts as a cache for pages from the secondary storage as each program executes



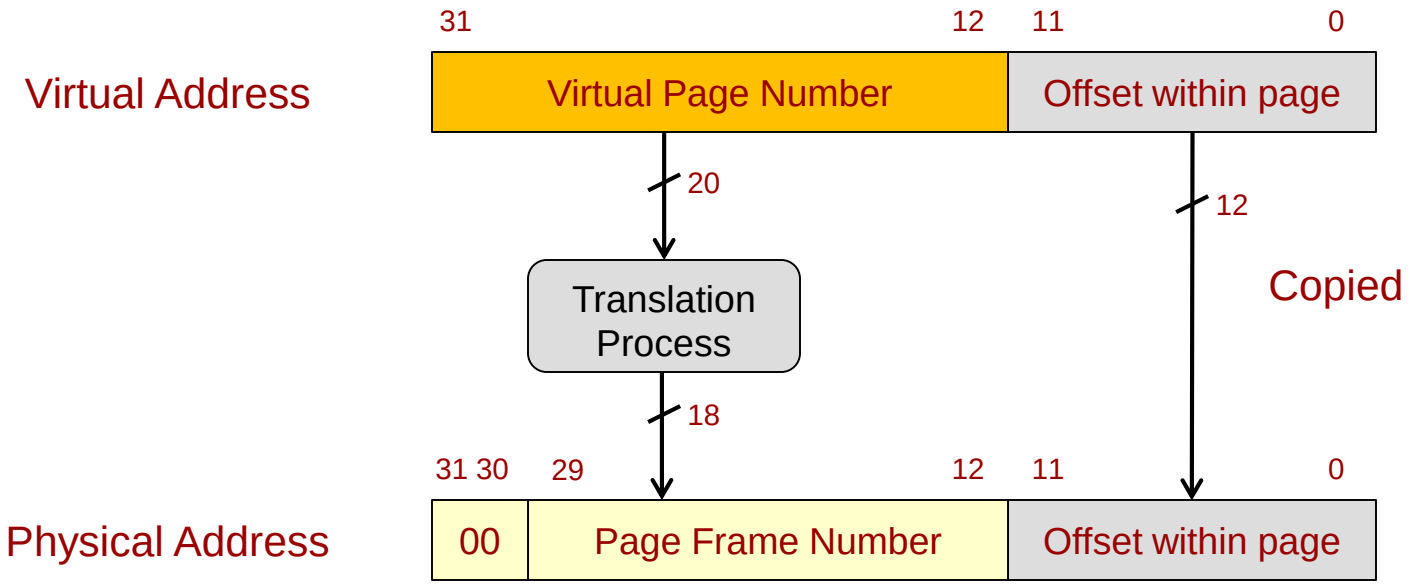
Physical Memory Usage

- HW & the OS will translate the virtual addresses used by the program to the physical address where that page resides
- If an attempt is made to access a page that is not in physical memory, a “page fault exception” is declared and the OS brings in the page to physical memory (possibly evicting another page)



Page Size and Address Translation

- Usually pages are several KB in size to amortize the large access time
- Example: 32-bit virtual & physical address, 1 GB physical memory, 4 KB pages
- Virtual page number to physical page frame translation performed by HW unit = MMU (Mem. Management Unit)



C REVIEW

Main Differences for CS350

- No reference types, only pointers
- No classes, only structs
  - And structs in C cannot have member functions
- Different I/O routines
  - printf() and scanf() replace cout and cin
- Different memory allocation functions
  - malloc() and free() replace new and delete

No Reference Types

- Reference types (i.e. int&) do not exist in C, only pointers (i.e. int*)

```
int main()
{
  int x=5,y=7;
  swapit(x,y);
 cout <<"x,y="<< x<<",”<< y;
  cout << endl;
}

void swapit(int &x, int &y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

DOES NOT COMPILE IN C

```
int main()
{
  int x=5,y=7;
  swapit(&x,&y);
  cout <<"x,y="<< x<<",”<< y;
  cout << endl;
}

void swapit(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
```

MUST PASS BY POINTER

C Structs

- A way to group values that are related, but have different data types
- Similar to a class in C++
- Capabilities of structs changed in C++!
  - C
    - Only data members (no functions)
    - Must create an instance by including keyword 'struct' in front of struct typename
  - C++
    - Like a class (data + member functions)
    - Default access is public

```
struct Person{
  char name[20];
  int age;
};
int main()
{
  // C++ decl.
  Person p1;

  // C decl.
  struct Person p1;
  struct Person *ptr = &p1;

  p1->age = 19;
  return 0;
}
```

Mimicking OOP

- So if we can't have member functions for a struct what should we do?
- Just write globally scoped functions that take in a pointer to a struct instance as the first argument

C++ Class - deck.h

```
class Deck {
 public:
   Deck();    // Constructor
   ~Deck();   // Destructor
   void shuffle();
   void cut();
   int get_top_card();
 private:
   int cards[52];
   int top_index;
};
```

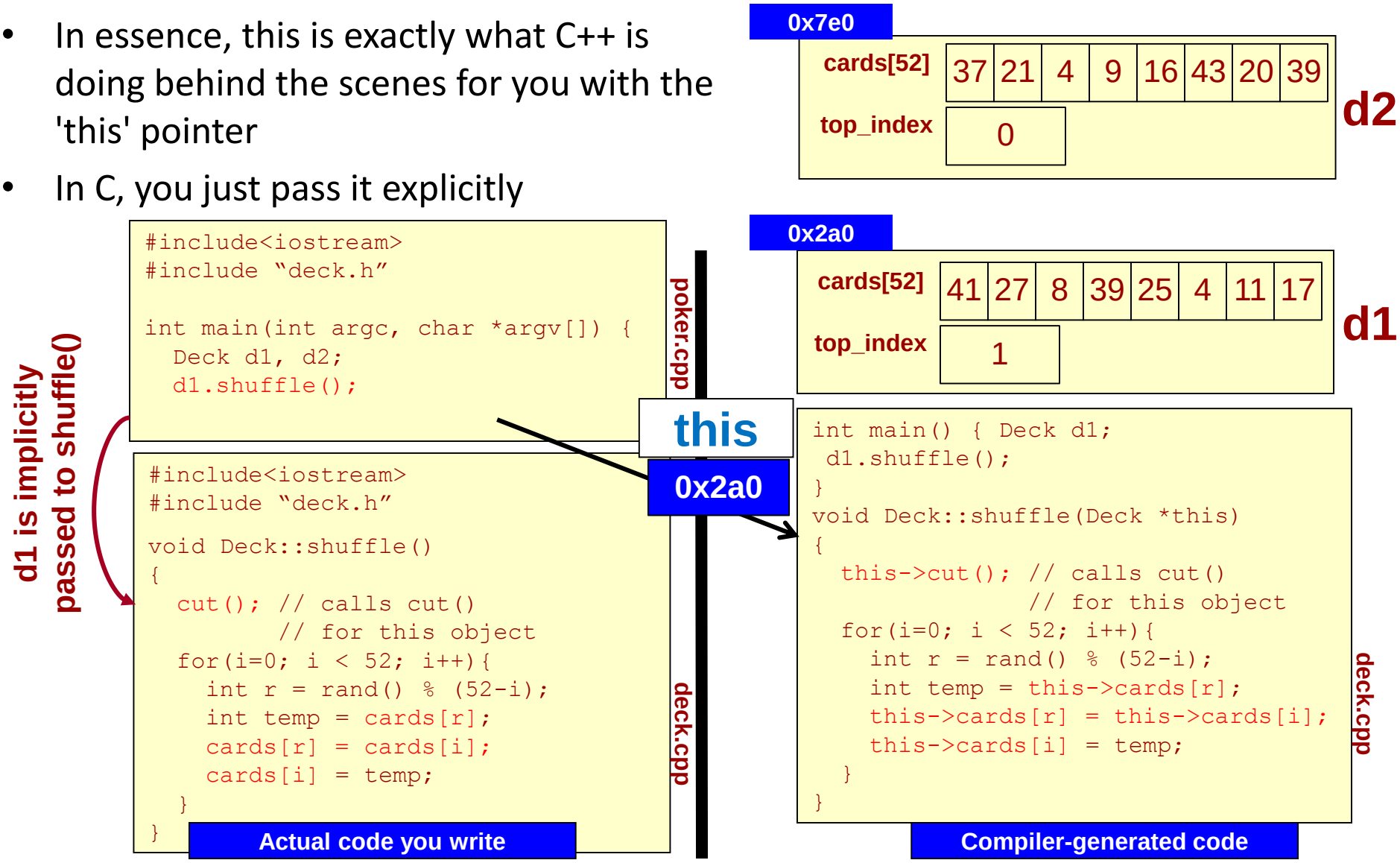
C Equivalent - deck.h

```
struct Deck {
   int cards[52];
   int top_index;
};

// Prototype global functions
// Each func. takes a ptr. to a Deck
void deck_init(struct Deck* d);
void deck_destroy(struct Deck* d);
void deck_shuffle(struct Deck* d);
int deck_get_top(struct Deck* d);
```

this Pointer

- In essence, this is exactly what C++ is doing behind the scenes for you with the 'this' pointer
- In C, you just pass it explicitly



static Keyword

- In the context of C, the keyword 'static' in front of a global variable or function prototype indicates the variable is only visible within the current file and should not be visible (accessed) by other source code files
- Can be used as a sort of 'private' helper function declaration

```
// Globals
int thread_count = 0;

static struct thread* the_thread;

// Functions
void thread_init(struct thread* t);

static void thread_init_helper(
   struct thread* t);
```

thread.c

```
void thread_init(struct thread*);
void thread_init_helper(
   struct thread*);
int f1()
{  // Will compile
   thread_count++;
   // Will NOT compile
   the_thread = NULL;

   struct thread t;
   // Will compile
   thread_init(&t);
   // Will NOT compile
   thread_init_helper(&t);
}
```

other.c

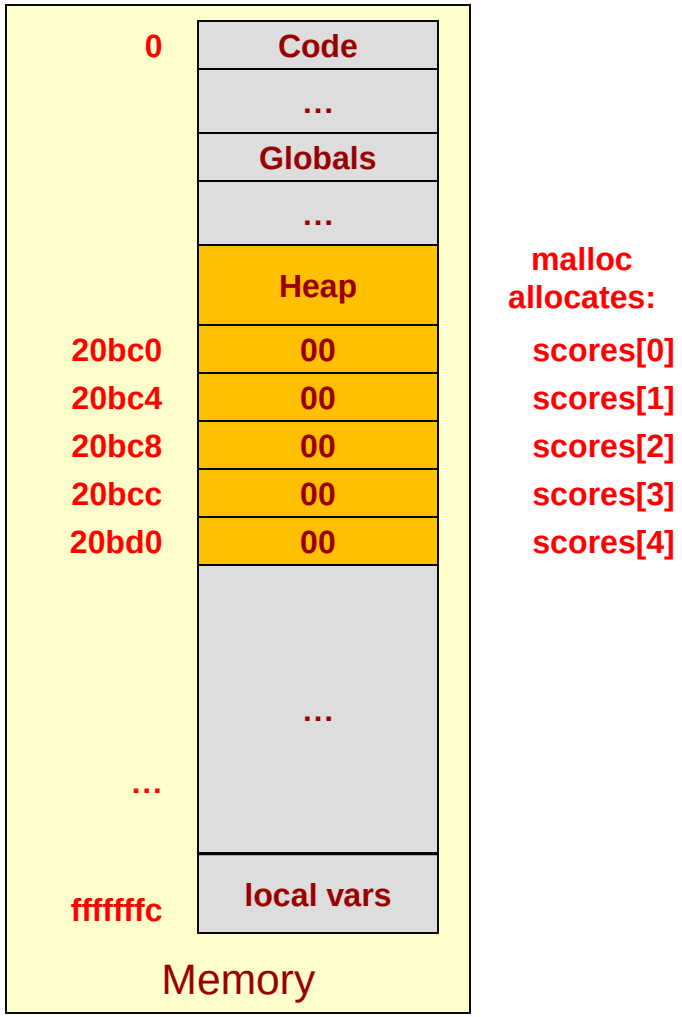
C Dynamic Memory Allocation

- malloc(*int num_bytes*) function in stdlib.h
  - Allocates the number of bytes requested and returns a pointer to the block of memory
- free*(void * ptr)* function
  - Given the pointer to the (starting location of the) block of memory, free returns it to the system for re-use by subsequent malloc calls

```
int main(int argc, char *argv[])
{
  int num;
  printf("How many students?\n");
  scanf("%d", &num);

  int *scores = (int*) malloc(num * sizeof(int));
  // can now access scores[0] .. scores[num-1];


  free(scores); // deallocate
  return 0;
}
```



Printf/Scanf vs. cout/cin

C I/O TECHNIQUES

printf

- C standard library function to display text to the screen
- printf(*format_string*, *var1*, *var2*, …);
- Format string:
  - Just like a normal string (enclosed by double-quotes => " ")
  - Includes normal text and place holders starting with '%' where you want to display a variable's value ("hi", "x=%d")
  - %[num1][.num2][d,u,x,lf,c,s,p]

num1	Minimum chars/field width
.num2	Maximum chars/field width (or, for floating point, number of characters to print after decimal point
d,x,lf,c,s,p	d=Decimal (int), u=unsigned Decimal (int), x=Hex, lf = Floating Point (double), c = ASCII character, s = string of characters, p = address/pointer

scanf

- C standard library function to get keyboard input from user
- scanf(*format_string*, &*var1*, &*var2*, …);
- Waits for user to enter info and hit 'Enter'
- Format string:
  - Just like a normal string (enclosed by double-quotes => " ")
  - Includes place holders starting with '%' indicating what kind of data you want to get from the user ("%d", "%lf")
  - Multiple inputs are fine ("%d%d") and will look for any whitespace as the separator between the inputs
  - %[d,x,lf,c,s,u]

d,u,x,lf,c,s	d=signed Decimal (int), u=unsigned Decimal (int), x=Hex, lf = Floating Point (double), c = ASCII character, s = string of characters

C I/O Examples

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int x;
   char c;
   double f = 7.5,g;
   printf("Enter an integer followed by a single character then a floating point number:\n");
   scanf("%d%c%lf",&x,&c,&g);
   printf("x * 2 is %04d and the character after c is %c\n",2*x,c+1)
   printf("result of 7.5/%lf = %4.2lf\n",g,f/g);
   return 0;
}
```

```
Enter an integer followed by a single character then a floating point number: 5 a 2.0
x * 2 is 0010 and the character after a is b
result of 7.5/2.000000 = 3.75
```

Do an Internet search for "printf scanf format strings" for more information and examples.

C FILE I/O

FILE * variables

- To access files, C (with the help of the OS) has a data type called 'FILE' which tracks all information and is used to access a single file from your program
- You declare a pointer to this FILE type (FILE *)
- You "open" a file for access using fopen()
  - Pass it a filename string (char *) and a string indicating read vs. write, text vs. binary
  - Returns an initialized file pointer or NULL if there was an error opening file
- You "close" a file when finished with fclose()
  - Pass the file pointer
- Both of these functions are defined in stdio.h

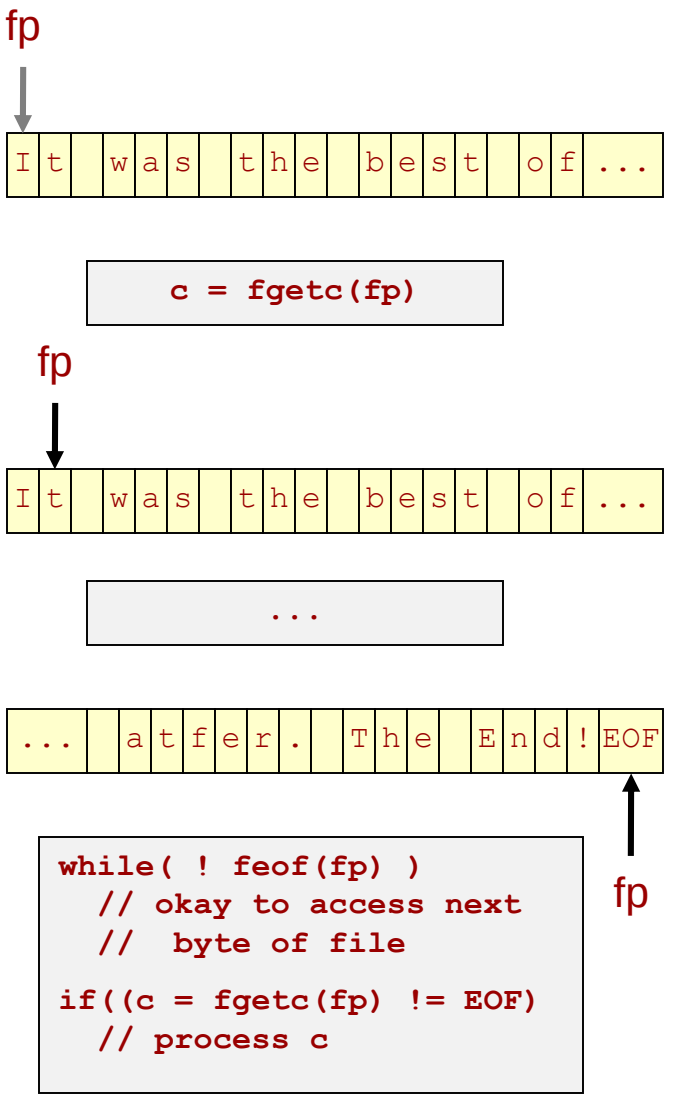
```
int main(int argc, char *argv[])
{
  char first_char, myline[80];
  int x; double y;
  FILE *fp;
  fp = fopen("stuff.txt","r");
  if (fp == NULL){
    printf("File doesn't exist\n");
    return 1;
  }
  // read an int and a double
  fscanf(fp, "%d%lf",&x,&y);
  // read next raw char. of file
  first_char = fgetc(fp);
  // read thru first '\n' of file
  fgets(myline, 80 ,fp);
  fclose(fp);
  return 0;
}
```

Second arg. to fopen()

"r" / "rb" = read mode, text/bin file
"w" / "wb" = write mode, text/bin file
"a" / "ab" = append to end of text/bin file
"r+" / "r+b" = read/write text/bin file
others…

File Access

- Many file I/O functions
  - Text file access:
    - fprintf(), fscanf()
    - fputc(), fgetc(), fputs(), fgets()
  - Binary file access:
    - fread(), fwrite()
- Your file pointer (FILE * var) implicitly keeps track of where you are in the file
- `EOF` constant is returned when you hit the end of the file or you can use feof() which will return true or false.



Text File Input

- fgetc(FILE*)
  - Reads a single ASCII character
- fgets(char* buf, int num, FILE* fp)
  - Reads up to 'num'-1 characters or until a newline ('\n') or EOF character are reached (whichever happens first) placing the results into buf (and adding a null character).
  - If a '\n' is encountered it is stored in buf (it is not discarded).
  - Stops at EOF…If EOF is first char read then the function returns NULL
  - Will append the NULL char at the end of the characters read
- fscanf(FILE* fp, char* format_str, …)
  - Reads the values indicated by the format string into the variables pointed to by the remaining arguments
  - Useful to convert ASCII chars to another variable type or parse at whitespace
  - Returns number of successful items read or ‘EOF’ if that is the first character read

Text File Output

- fputc(int character, FILE* fp,)
  - Write a single ASCII character to the file
  - Even though character is of type 'int' you would usually pass a 'char'
- fputs(const char* string, FILE* fp)
  - Write a null-terminated string to the file (null character is not written to the file)
- fprintf(FILE* fp, const char* format_string, …)
  - Writes the values indicated by the format_string to the file indicated by fp.

Binary File I/O

- fread()
  - Pass a pointer to where you want the data read from the file to be placed in memory (e.g. `&x` if x is an int or `data` if data is an array)
  - Pass the number of 'elements' to read then pass the size of each 'element'
  - # of bytes read = number_of_elements * size_of_element
  - Pass the file pointer
- fwrite
  - Same argument scheme as fread()

```
int main(int argc, char *argv[])
{
  int x;
  double data[10];
  FILE *fp;

  fp = fopen("stuff.txt","r");
  if (fp == NULL){
    printf("File doesn't exist\n");
    exit(1)
  }
  fread(&x, 1, sizeof(int), fp);
  fread(data, 10, sizeof(double),fp);

  fclose(fp);
  return 0;
}
```

Changing File Pointer Location

- Rather than read/writing sequentially in a file we often need to jump around to particular byte locations
- fseek()
  - Go to a particular byte location
  - Can be specified relative from current position or absolute byte from start or end of file
- ftell()
  - Return the current location's byte-offset from the beginning of the file

```
int main(int argc, char *argv[])
{
  int size;
  FILE *fp;

  fp = fopen("stuff.txt","r");
  if (fp == NULL){
    printf("File doesn't exist\n");
    exit(1)
  }
  fseek(fp,0,SEEK_END);
  size = ftell(fp);

  printf("File is %d bytes\n", size);


  fclose(fp);
  return 0;
}
```

Third arg. to fseek()

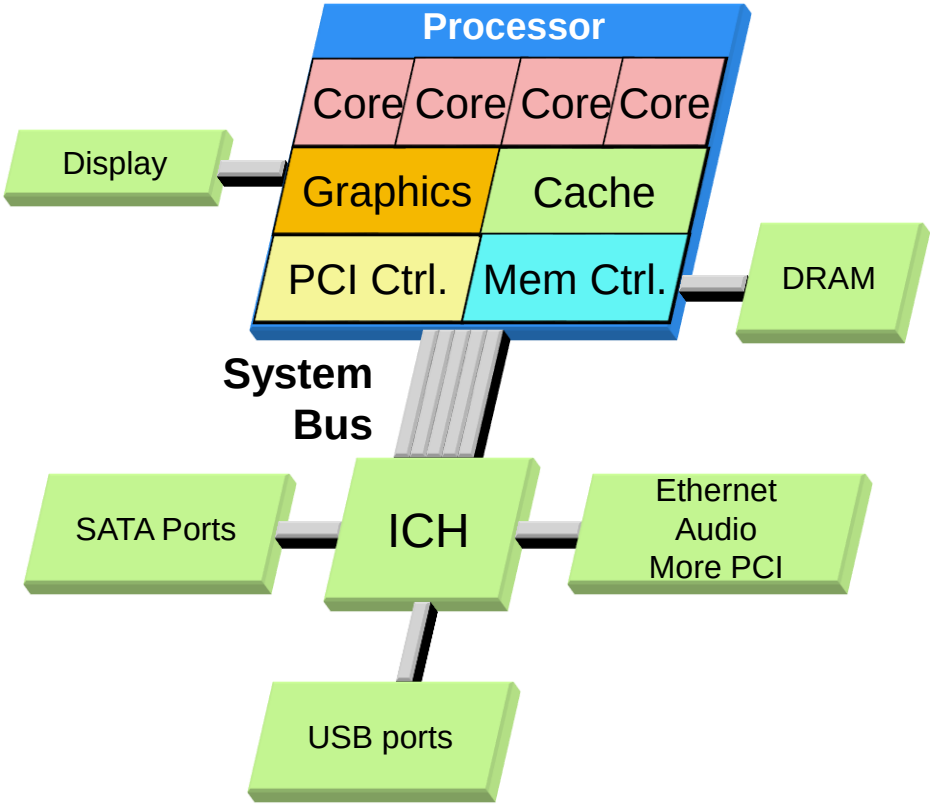
SEEK_SET = pos. from beginning of file
SEEK_CUR = pos. relative to current location
SEEK_END = pos. relative from end of file (i.e. negative number)

Let's have fun by understanding how a modern system even boots to an OS…

PC ARCHITECTURE AND BOOT PROCESS

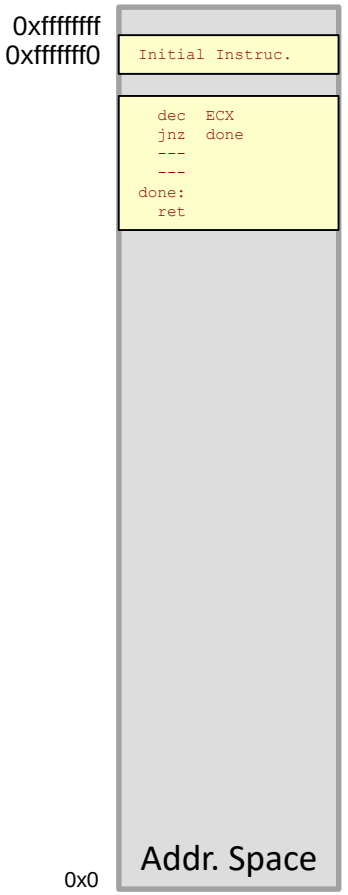
Modern PC Architecture

- Moore's Law has allowed greater integration levels
  - Multiple cores and greater levels of cache
  - Memory controller, graphics, and high-speed I/O are integrated onto the processor die



Intel Boot Process

- On power on, each core races for a lock bit
  - First one executes the boot sequence
  - Others wait for Start-Up Inter-Processor Interrupt
- On power on, fetch instruction at 0xFFFFFFF0
  - Address corresponds to ROM/Flash
  - Jump to initialization code (BIOS) elsewhere
- Bootstrap
  - Choose mode: Real, Flat protected, Segmented protected



https://www.cs.cmu.edu/~410/doc/minimal_boot.pdf
http://www.drdobbs.com/parallel/booting-an-intel-architecture-system-par/232300699

Intel Boot Process

- Determine basic memory map and DRAM configuration (but DRAM still not functional)
- Enable Interrupt Controllers for Protected Mode operation
  - Setup data-structures and base address registers (BARs) needed to support interrupts (i.e. descriptor tables [IDT, GDT], and Task-State Segment [TSS])
- Questions
  - Where is code right now?
  - Can this code be written using functions?
- Configure cache to be used as RAM for stack and place it NEM (No Eviction Mode)

https://www.cs.cmu.edu/~410/doc/minimal_boot.pdf
http://www.drdobbs.com/parallel/booting-an-intel-architecture-system-par/232300699

Intel Boot Process

- Initialize DRAM
  - Write to all memory and ensure ECC flags match data (either via BIOS or HW mechanism)
- Copy BIOS code into DRAM and take processor out of special cache mode (and flush cache)
- Initialize other cores (sending them an initial EIP/PC)
- Discover and initialize various I/O devices
  - Timers, Cache, PCI bus, SATA (hard drive access)
  - Determine address ranges (memory map)
- Load Master Boot Record from first sector of boot device
  - Points to where OS is located and how to load code into memory
  - Transfer is now transferred to the OS

https://www.cs.cmu.edu/~410/doc/minimal_boot.pdf
http://www.drdobbs.com/parallel/booting-an-intel-architecture-system-par/232300699

Windows Boot Process

- OS Loader
  - Loads system drivers and kernel code
  - Reads initial system registry info
- OS Initialization
  - Kernel Init
    - Init kernel data structures & PnP manager
  - Session SMSSInit (SMSS = Session Manager)
    - Initializes registry, starts other devices/drivers, and start other processes
  - Winlogon Init
    - User logon screen, service control manager, and policy scripts
  - Explorer Init
    - DWM (Desktop Window Manager), shell
- Post Boot phase
  - Other services are started (tray icons, etc.)
- Other good reference:
  - http://www.cs.fsu.edu/~zwang/files/cop4610/Fall2016/windows.pdf

https://social.technet.microsoft.com/wiki/contents/articles/11341.the-windows-7-boot-process-sbsl.aspx