Review of Computer Architecture

Glenn Bruns
CSUMB

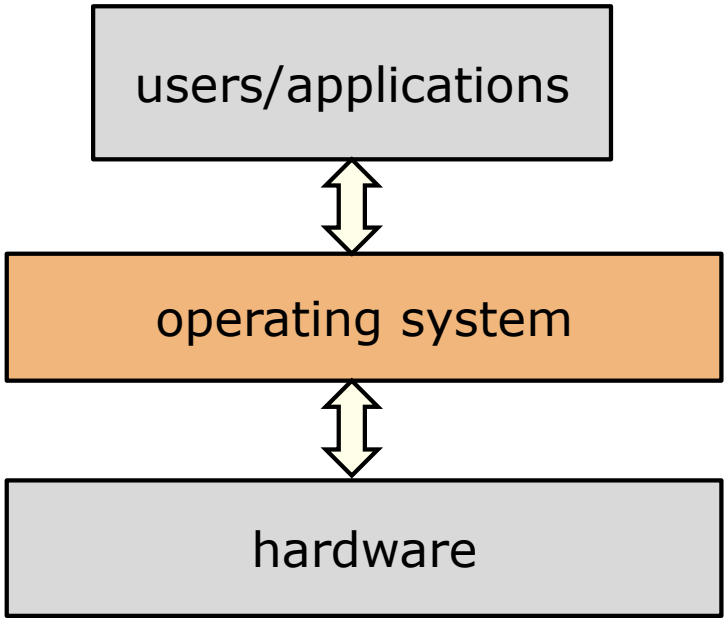
Lecture Objectives

At the end of this lecture, you should be able to:

- List the components of a computer
- Name some of the buses of a computer
- Describe a “memory hierarchy”
- Explain the layout of a program in memory

Motivation

- An OS is a software layer between the computer hardware and users/applications
- To understand what an OS does you need to know a little about the hardware

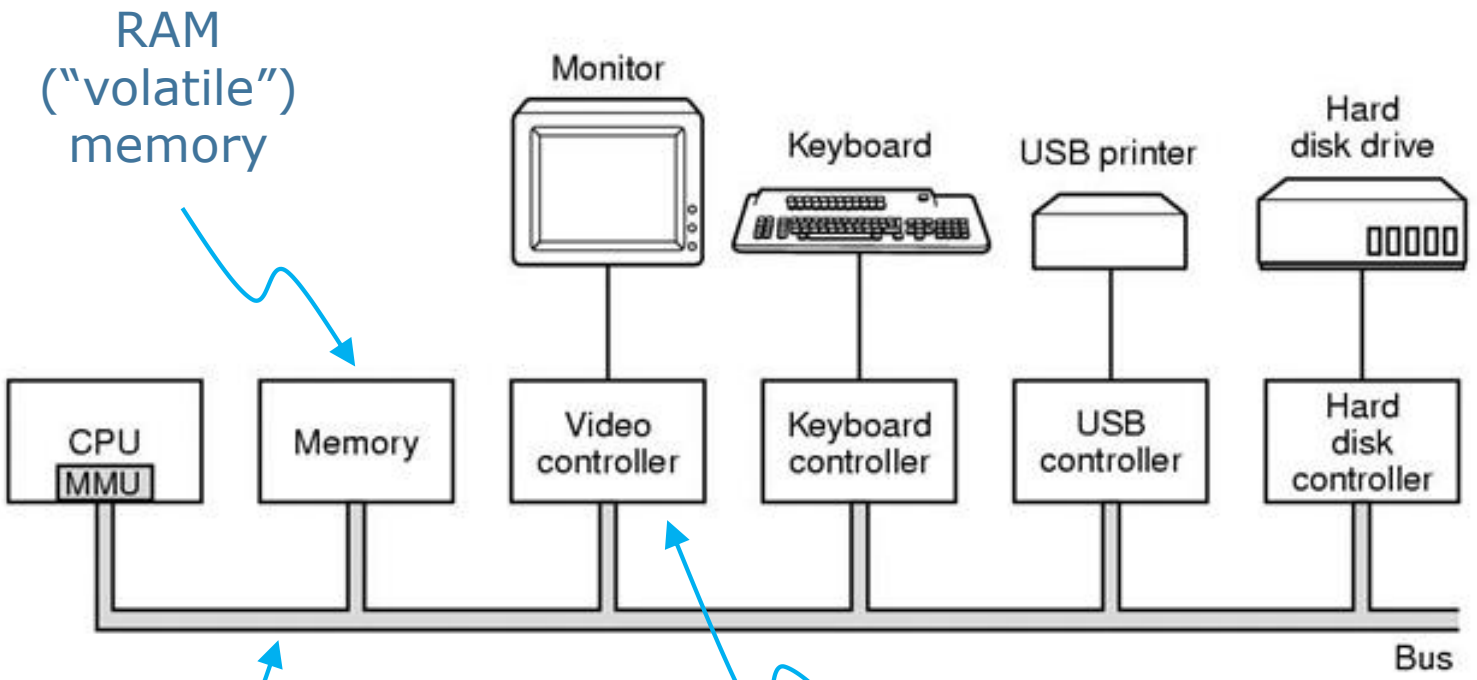


Components of a Computer

Question: what are the three main parts of a computer?

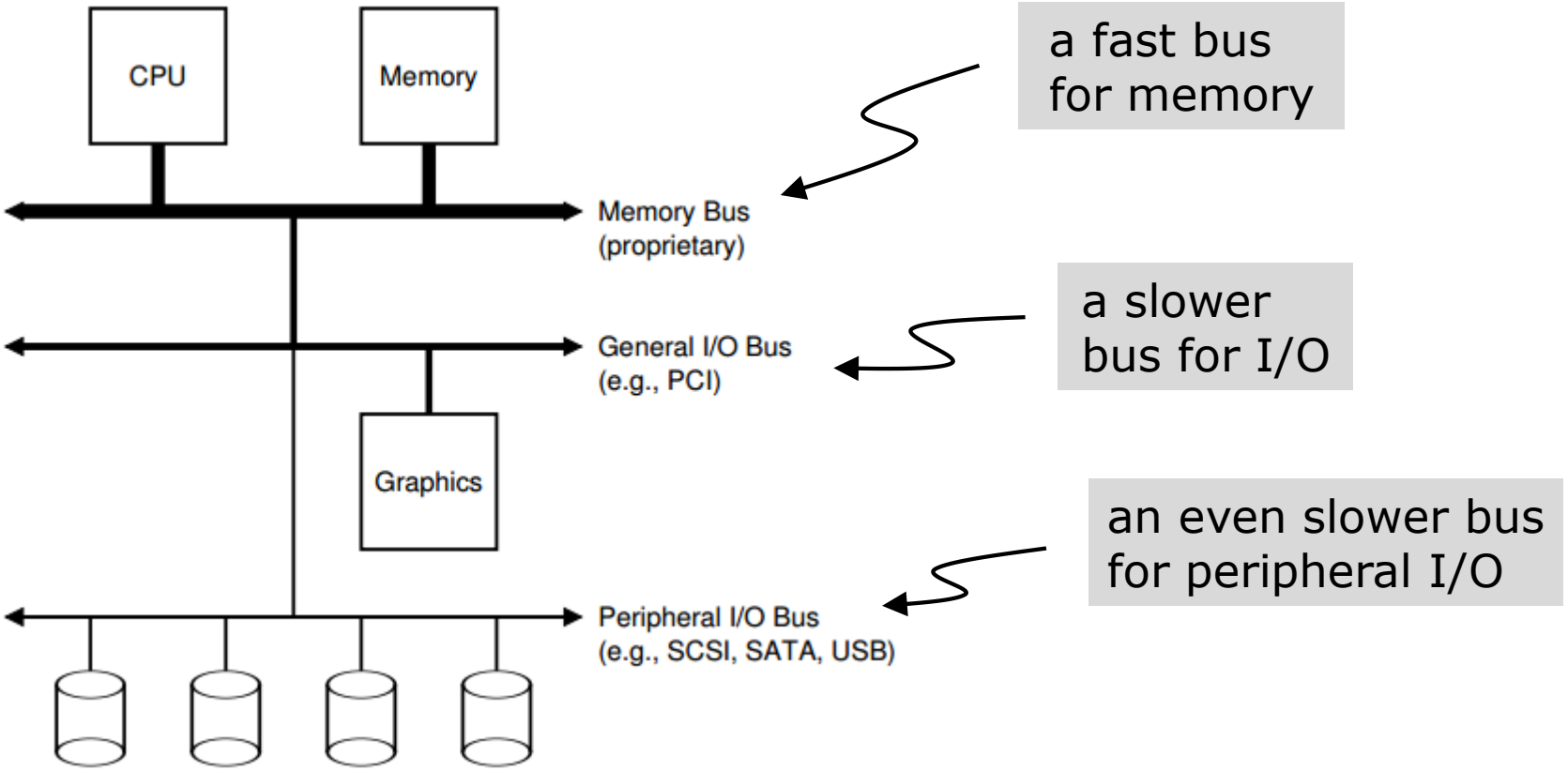
- Processor: performs computations
- Storage: stores data
- I/O devices: mouse, keyboard, printer, …

Computer Architecture



(Figure from Tanenbaum, *Modern Operating Systems*, 3rd edition)

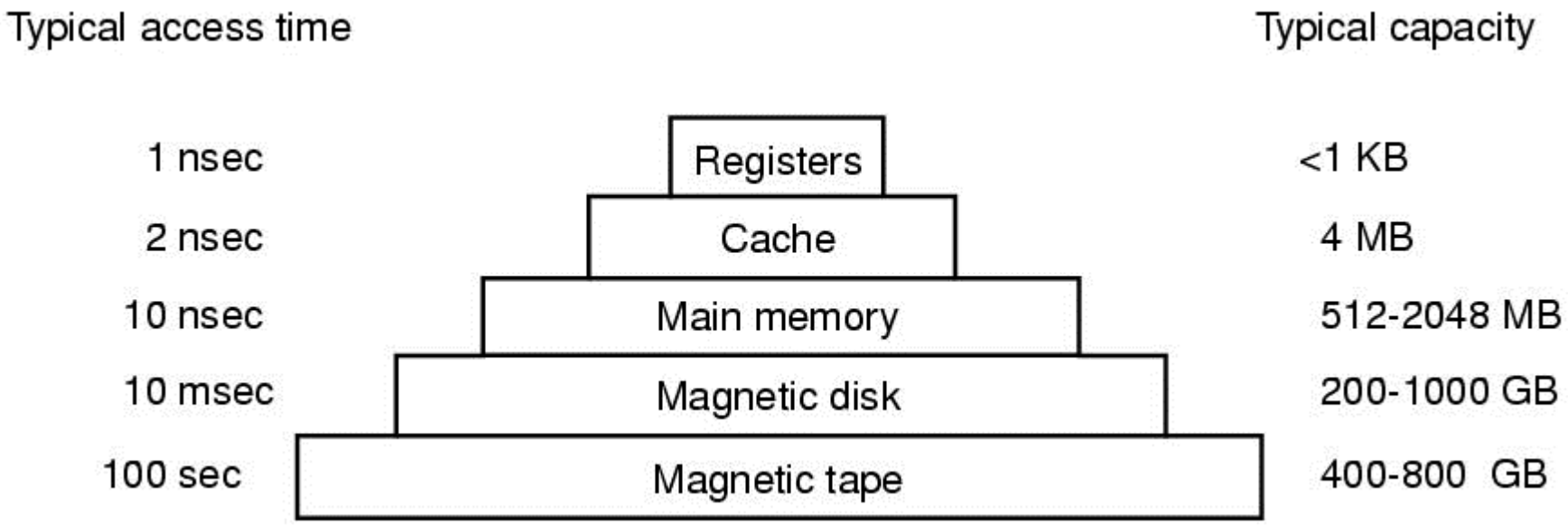
A little detail on buses



High performance components are closest to the CPU

(Figure from Operating Systems: Three Easy Pieces, Arpaci-Dusseau and Arpaci-Dusseau)

Computer storage hierarchy



Higher layers are faster but have higher cost per bit.
Higher layers can be used to *cache* data in lower layers.

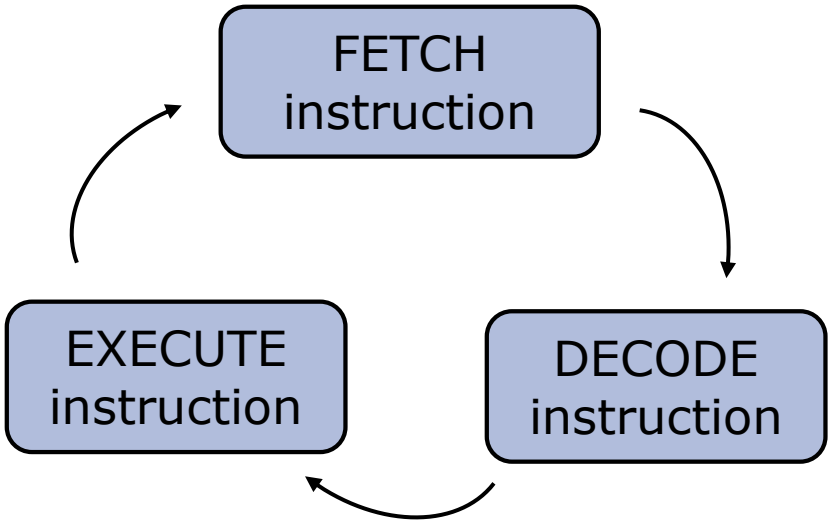
(Figure from Tanenbaum, *Modern Operating Systems*, 3rd edition)

Central Processing Unit (CPU)

Executes instructions in main memory

Contains a little memory (*registers*), including the *program counter* and *stack pointer*

CPU has a simple instruction cycle:



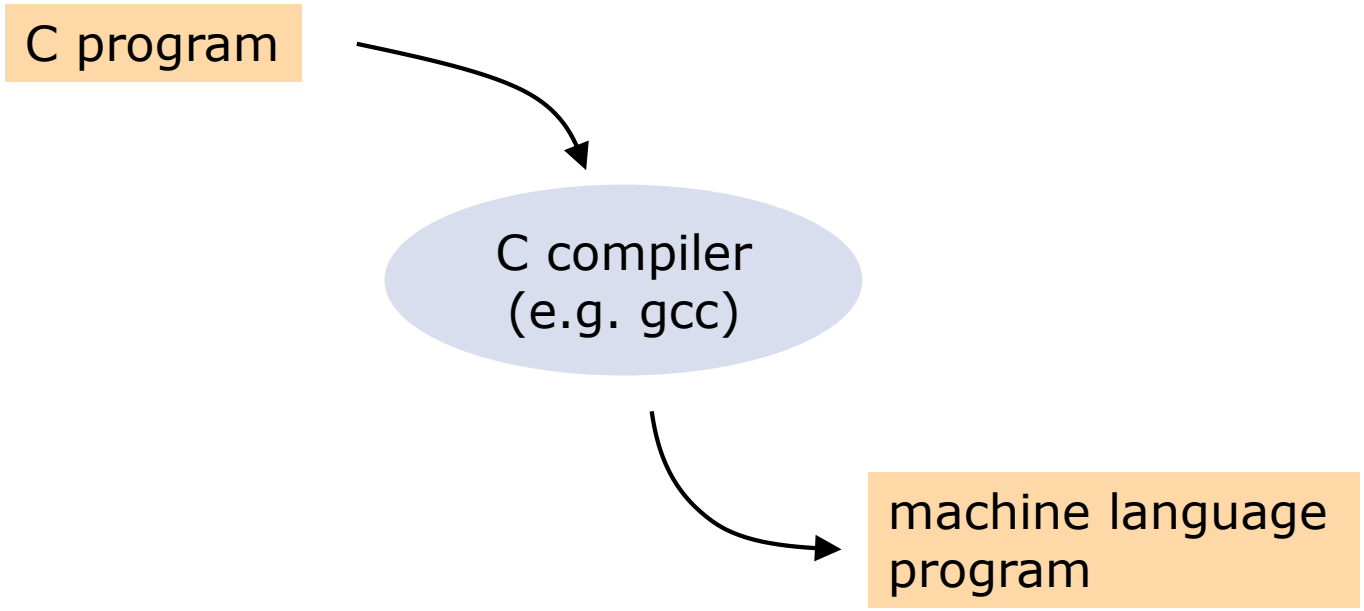
fetch: get instruction from memory location given by program counter

decode: get instruction arguments; break instruction down into parts

C Programming

- The language of Linux; also, the language of OS courses everywhere in the Milky Way galaxy.
- A simple and low-level language
- C is a subset of C++ (almost)
  - http://stackoverflow.com/questions/1201593/c-subset-of-c-where-not-examples

Machine language (aka binary)



- also called "binary programs", or a "program binary"
- super-low level programs – just a sequence of bytes
- the instructions are CPU specific
- Instruction Set Architecture (ISA) just means the set of instructions supported by a CPU

"Disassembling" a binary

```
int add_one_to(int iparam) {
    int ilocal = iparam + 1;
    return ilocal;
}
int main(int argc, char **argv) {
    int iresult = 0;
    iresult = add_one_to(iresult);
    return 0;
}
```


A tiny C program

Assembly code recreated from binary for function add_one_to()

```
gdb -q a.out
Reading symbols from /home/CLASSES/brunsglenn/return-oriented-prog/a.out...(no
debugging symbols found)...done.
(gdb) disas add_one_to
Dump of assembler code for function add_one_to:
   0x08048394 <+0>:     push   %ebp
   0x08048395 <+1>:     mov    %esp,%ebp
   0x08048397 <+3>:     sub    $0x10,%esp
   0x0804839a <+6>:     mov    0x8(%ebp),%eax
   0x0804839d <+9>:     add    $0x1,%eax
   0x080483a0 <+12>:    mov    %eax,-0x4(%ebp)
   0x080483a3 <+15>:    mov    -0x4(%ebp),%eax
   0x080483a6 <+18>:    leave
   0x080483a7 <+19>:    ret
End of assembler dump.
```

ebp, esp, eax are names of registers

'mov b,c' moves b to c

How to translate function calls?

- Where to store local variables of a function?
  - the memory used to store local variables isn't needed once the function returns
  - there can be many nested calls to a function
- Where to store the arguments of a function call?
- How does the function know where to return to?

```
// return true if a node in the
// tree has value key
int search_tree(TREE tree, int key) {
  int val;  // local variable

  if (tree != NULL) {
    val = value(root(tree));
    if (val == key) {
      return true;
    }
    search_tree(left(tree), key);
    search_tree(right(tree), key);
  }
  return false;
}
```

Solution: use a stack

Suppose we have the call

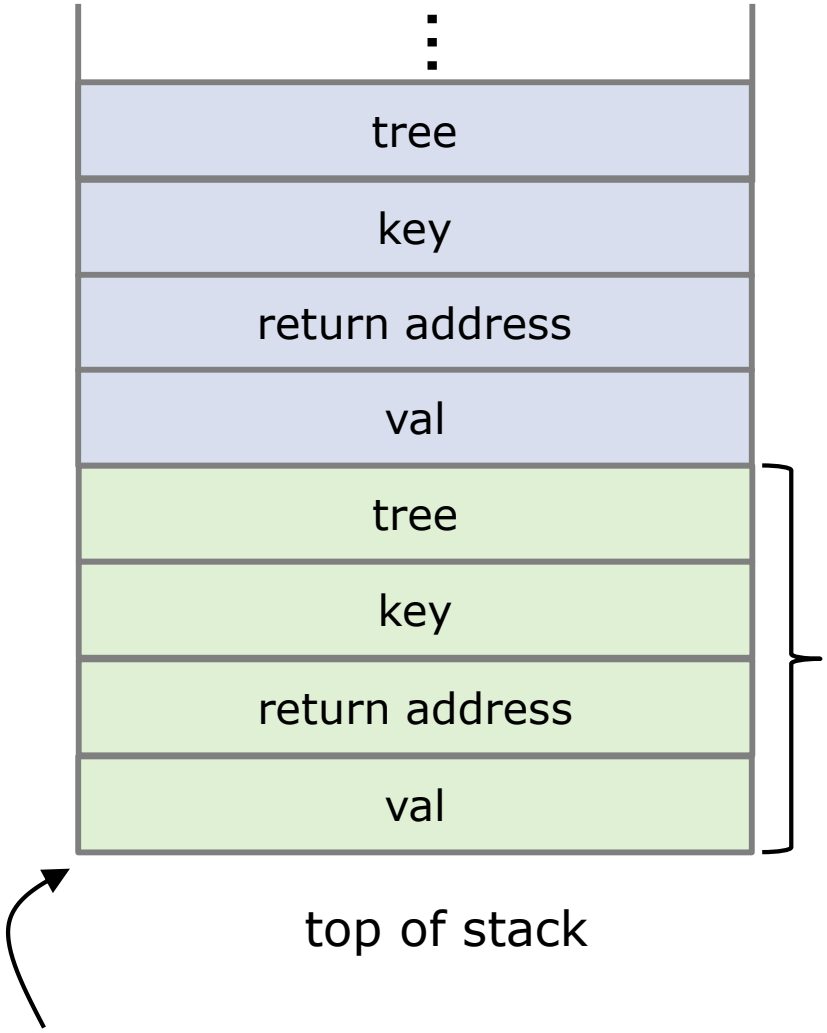
search_tree(t, 5)

- the values of t and key are pushed onto a stack
- the address where search_tree should return to is pushed on the stack
- space is created on the stack for local variable val

```
// return true if a node in the
// tree has value key
int search_tree(TREE *tree, int key) {
  int val;  // local variable

  if (tree != NULL) {
    val = value(root(tree));
    if (val == key) {
      return true;
    }
    search_tree(left(tree), key);
    search_tree(right(tree), key);
  }
  return false;
}
```

Solution: use a stack




```
// return true if a node in the
// tree has value key
int search_tree(TREE *tree, int key) {
  int val;  // local variable

  if (tree != NULL) {
    val = value(root(tree));
    if (val == key) {
      return true;
    }
    search_tree(left(tree), key);
    search_tree(right(tree), key);
  }
  return false;
}
```

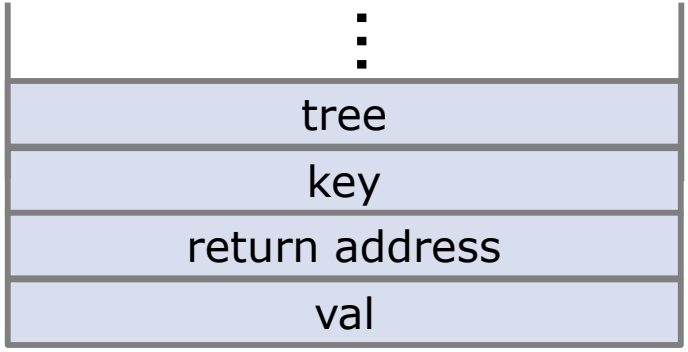
Using the stack

when search_tree() runs:

- it gets the value of val from the stack
- it gets the values of tree and key from the stack

when search_tree() returns:

- the stack frame is popped off the stack




```
// return true if a node in the
// tree has value key
int search_tree(TREE *tree, int key) {
  int val;  // local variable

  if (tree != NULL) {
    val = value(root(tree));
    if (val == key) {
      return true;
    }
    search_tree(left(tree), key);
    search_tree(right(tree), key);
  }
  return false;
}
```

Looking at how the stack is used

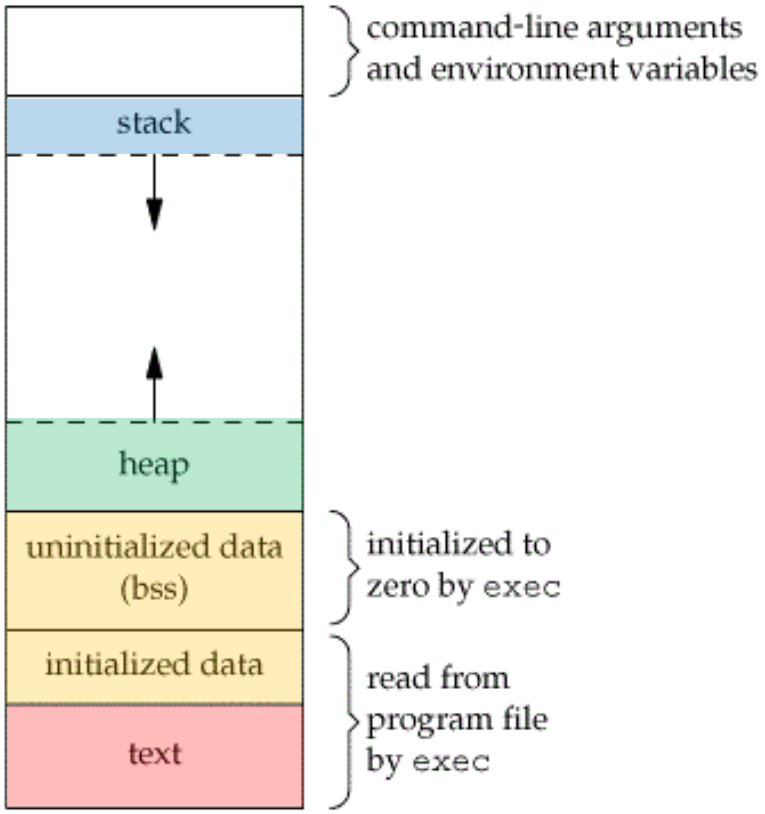
```
int add_one_to(int iparam) {
    int ilocal = iparam + 1;
    return ilocal;
}
int main(int argc, char **argv) {
    int iresult = 0;
    iresult = add_one_to(iresult);
    return 0;
}
```

registers:
eax – general purpose
esp – pointer to top of stack
ebp – pointer to the stack frame

```
gdb -q a.out
Reading symbols from /home/CLASSES/brunsglenn/return-oriented-prog/a.out...(no
debugging symbols found)...done.
(gdb) disas add_one_to
Dump of assembler code for function add_one_to:
   0x08048394 <+0>:     push   %ebp             # push ebp on stack to remember it
   0x08048395 <+1>:     mov    %esp,%ebp        # point ebp to top of stack
   0x08048397 <+3>:     sub    $0x10,%esp       # make stack space for local vars
   0x0804839a <+6>:     mov    0x8(%ebp),%eax   # copy iparam to eax register
   0x0804839d <+9>:     add    $0x1,%eax        # add 1 to the register
   0x080483a0 <+12>:    mov    %eax,-0x4(%ebp)  # copy result to variable ilocal
   0x080483a3 <+15>:    mov    -0x4(%ebp),%eax  # copy ilocal to eax register
   0x080483a6 <+18>:    leave                   # copy ebp to esp, restore ebp
   0x080483a7 <+19>:    ret                     # return control back to caller
End of assembler dump.
```

Memory layout for a running program

- text segment
  - stores the program's code
  - also called 'code segment'
- data segment
  - used to store static variables (global vars, static local vars)
- heap segment
  - used to store dynamically-allocated variables
- stack segment
  - used to store program call stack
  - "stack overflow" if too many nested calls



Summary

- The main components of a computer are the processor, memory, and I/O
- A computer has multiple buses, with different speeds
- Memory also comes in different speeds – faster memory is used to cache slower memory
- The four regions of a program in memory are: text segment, data segment, heap, stack