MEMORY MANAGEMENT

Professor Ken Birman
CS4414 Lecture 5

IDEA MAP FOR TODAY

Understanding where an object resides is very important in modern systems. In C++, you can't write correct code unless you master this topic

Global objects live in data segments

Inline objects live on the stack

Dynamically created objects live in the heap

Address space for a Linux process: many kinds of segments

If time permits: How malloc manages the heap

WHAT HAPPENS WHEN YOU RUN A COMMAND?

Linux really has three concepts of what a command can be

1. The bash shell has some built-in commands
2. You can take any sequence of commands and put them in a file, change its mode to "executable", and then the file name behaves like a Linux command. You can even pass it arguments, include loops, etc!
3. A compiled program, perhaps starting with C++ source

WHAT HAPPENS WHEN YOU RUN A PROGRAM?

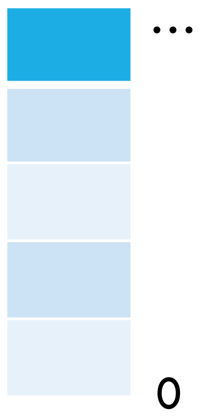
Bash sees that you are trying to execute a program – it finds the file and checks, and learns that it is an executable (and remembers this, for quicker future responses)

Bash uses Linux to prepare an address space and then load and execute the program in the new address space. This is done with the **fork** and **exec** systems calls. (Both have several variants).

We'll discuss later… details not important today

ADDRESS SPACE?

A page is a 4K
block of memory



Every program runs in an isolated address context.

Within it, the addresses your program sees are "virtual". They don't match directly to addresses in physical memory.

A "page table" managed by Linux maps virtual to physical, at a granularity that would usually be 4k (4096) bytes per page.

… PAGES ARE GROUPED INTO SEGMENTS

Rather than just treating memory as one range of pages from address 0 to whatever the current size needed might be, Linux is segmented. There are often **gaps** between them.

Definition: A segment is just a range of virtual memory with some base address, and some total size, and access rules.

One segment might be as small as a single page, or could be huge with many pages. We don't normally worry about page boundaries

A FEW SEGMENT TYPES LINUX SUPPORTS

Code: This kind of segment holds compiled machine instructions

Data: Uses for constants and initialized global objects

Stack: A stack segment is used for memory needed to call methods, or for inline variables (I'll show you an example).

Heap: A heap segment is used for dynamically allocated memory that will be used for longer periods (again, I'll show you)

Mapped files: The file can be accessed as a byte[] vector!

GAPS

The address space will often have “holes” in it.

These are ranges of memory that don’t correspond to any allocated page.

If you try and access those regions, you’ll get a segmentation fault and your process will crash.

STACKS, HEAPS

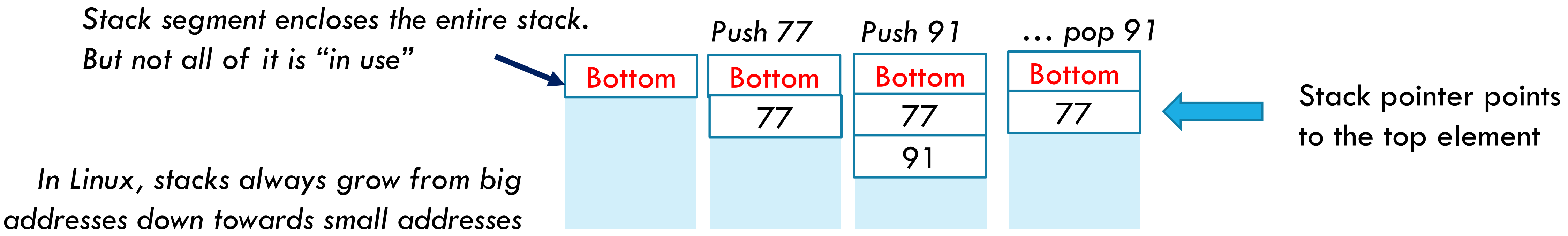
Our programs often need to dynamically allocate memory to hold new objects. Later they might free that memory.

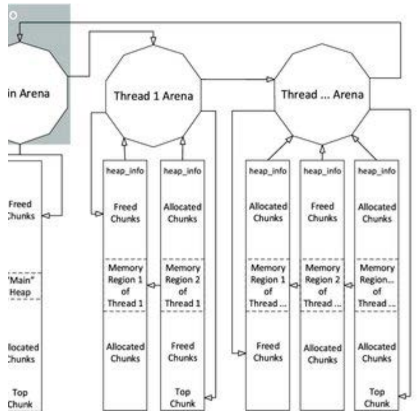
The stack and the heap are two resources for doing this.

STACKS VERSUS HEAPS

A **stack** is a managed region of memory that has a concept of a stack pointer. You "push" objects on the stack, and the stack pointer changes (the value gets *smaller*) by the size of the object

… later you "pop" the object and the stack pointer gets larger.


Stack segment encloses the entire stack.
But not all of it is "in use"
Push 77
Push 91
… pop 91
Bottom
Bottom
77
Bottom
77
91
Bottom
77
Stack pointer points to the top element
In Linux, stacks always grow from big addresses down towards small addresses


in Arena
Thread 1 Arena
Thread ... Arena
heap_info
Freed Chunks
Allocated Chunks
"Main" Heap
Memory Region 1 of Thread 1
Memory Region 2 of Thread 1
Memory Region 1 of Thread ...
Memory Region 2 of Thread ...
Memory Region... of Thread ...
Top Chunk

… HEAPS

A heap is a memory segment accesed via **malloc/free**

NUMA heaps: Linux maintains one heap region (pool) per DRAM module, tries put your new memory “close” to your thread.

INITIALIZATION

In C++ we normally use objects with constructors that initialize the fields to desired values.

For this reason, new memory won’t be automatically zeroed: you’ll put the values you want into that memory. Data segment is initially zero, but that is really a special case.

Of course you can always zero a memory region “by hand”

YOUR C++ PROGRAM “CONTROLS” WHERE THE VARIABLES IT USES WILL LIVE

```
int     cat_count = 0;                    // Global: Lives in a data segment

class Cat {                               //  Definition: Used only by g++
 public:
      int cat_id;                         //  The object will have space for a 32-bit int
      std::string name;                   //  … and for a “string object”
      Cat(std::string given_name) {       // Constructor to initialize a new Cat instance
            name = given_name;
            cat_id = cat_count++;
       }
}
```

YOUR C++ PROGRAM "CONTROLS" WHERE THE VARIABLES IT USES WILL LIVE

```
external Cat   fluffy;    // Global, initialized "elsewhere"

Cat   irma("Irma");        // Global, initialized here.  Object will live in data segment

int main(int argc, char** argv) {

        Cat  streetcat("Grizabella");                    // Stack, created now, actually on the stack.
                                                         // The object will be deallocated when scope exits

        Cat  *catptr = new Cat("Mistophelles");   // Heap!  Remains allocated until you call delete

}
```

Scope: The execution block in which the variable is accessible

PUZZLE: WHERE IS THE BYTE[] FOR STRINGS?

We used std::string to hold the cat names. Where is the memory?

Internally, a std::string includes a **pointer** to a character array: a byte vector, terminated with a null byte (‘\0’).

But where is the string itself, in memory?

In the heap! std::string makes a copy using malloc and memcpy

If you copy a std::string, a heap char string is made by the copy constructor

MALLOC AND FREE

You don't see it, but internally, C++ implements **new** using **malloc**, and **delete** using **free**. These are library methods built in C that manage pools of memory: one per DRAM module on your NUMA computer.

Notice that **new** and **delete** are not needed for global or stack-allocated objects. Question: *why not?*

GLOBAL AND STACK ALLOCATION

A global object will be assigned space in the data segment. The compiler handles this, and runs the constructor either at compile time, or (if the constructor uses things that aren't constants), when the program starts execution.

A stack allocated object will be assigned a chunk of space on the stack when the line of code executes to create the object. The constructor runs when this occurs.

THE STACK IS ALSO USED FOR METHOD CALLS

Roles of the stack:

- Hold return PC
- Hold stack-allocated data
- Hold values of registers that will temporarily be used but then restored to whatever was previously in them
- Hold method arguments that don’t fit into registers
- Hold results from a method, if the result is “large”

CALLING A METHOD…

C++ generates code to put arguments into registers, or onto the stack. It has its own rules to decide which case applies.

The Intel hardware automatically pushes the caller’s PC to the stack. Later it uses this to return to where the call was done.

On return, Intel pops the PC from the stack. C++ pops anything it pushed, and we are back to the state from before the call.

C++ NOTATIONS FOR ACCESSING THINGS

C++ has a concept of a "namespace" used to understand the variable you are referencing.

For example, std::queue is a reference to a class called queue that was defined in the standard library (std:: means "standard")

Pronunciation hint: Ken just says "st-it-id" for std::

VARIABLES VERSUS POINTERS

Suppose some variable cat is in the current scope, and we access it. Some examples:

```
auto cat2 = cat;        //  Constructs a copy
cid = cat.cat_id;       //  References a member
auto cptr = &cat;       // Creates a pointer to cat
```

POINTER: A VARIABLE HOLDING AN ADDRESS

A pointer variable has a 64-bit number in it: a memory location.

You need to make sure it points to a sensible place!

But then can access members, like cptr → cat_id. (*cptr).cat_id is equivalent: (*cptr) “is” the cat object that cptr points to.

ACCESS BY REFERENCE

Often you see methods with types like this:

```
int sum(const int& a, const int& b) { return a+b; }
```

The & “a will be a reference to the argument” Thus, a acts like a second name – an *alias* – for the argument supplied by the caller.

The by reference notation, &, can only be used if the passed argument is a variable – it could appear on the left side of an “=“

C++ ALLOWS REFERENCE RETURN VALUES

For example, you can write a method that returns a reference to some object that is in an array, or even one it just created!

But beware…. A reference or pointer to an object on the stack will be "unsafe" if that stack scope terminates!

And a reference or pointer into the heap is only valid as long as you haven't deleted the object in the heap that it points to!

ARRAYS USE A FORM OF REFERENCES

C++ has two concepts of array indexing.

If myvec is of type int[10], then myvec[k] is the k’th element. Note that C++ doesn’t check for illegal index values!

Overloaded operators become method calls, to methods defined by the class

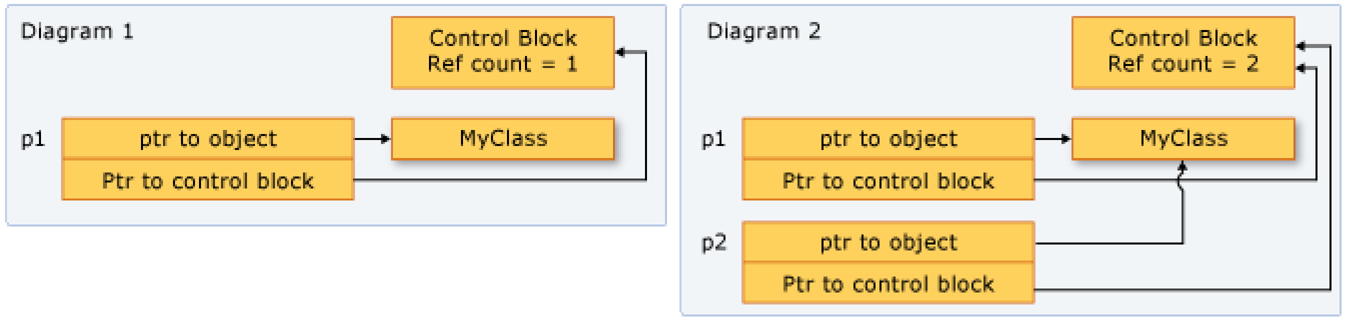
But you can also “overload” the [ ] operator for classes of your own. For example, cat.litter[k] could be the k’th kitten in a list.

SHARED_PTR

When working with pointers, people often call malloc, but then forget to call free. C++ isn't garbage collected, so the malloc'ed objects will linger for as long as the program runs.

This is called a memory leak. The heap segment grows and grows. Eventually a process can run out of space and crash.

SHARED_PTR



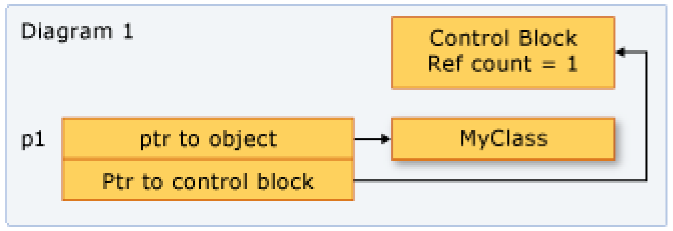
Professional C++ developers prefer not to use pointers directly. We “wrap” them in a shared_ptr template.

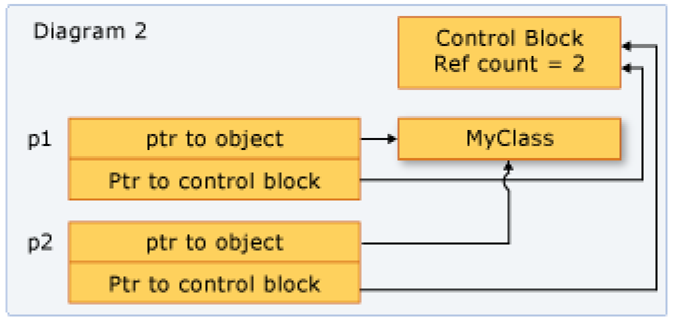
With a shared_ptr, when the object has no more references to it, the **delete** method is called automatically.

This adds garbage collection to C++, in a controlled form!

https://docs.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-shared-ptr-instances?view=vs-2019

SHARED_PTR





Example:

```
auto my_ptr = new shared_ptr<foo>(constructor args);

auto ptr_2 = my_ptr;   // Auto-increments reference count!
```

When a shared_ptr goes out of scope, the reference count is decremented automatically. **Delete** is called if it reaches 0.

USE A SHARED_PTR LIKE ANY POINTER

Suppose foo has a field “name”.

With a **foo* pt**, you write pt→name; pt holds an address.

With a **shared_ptr<foo> pt**, you use the identical notation! The shared pointer object holds the address of the foo object. By overloading the → operator, the shared_ptr mimics a pointer!

MEMORY LEAK

Suppose that your program includes code that might be causing a memory leak.

The memory is consumed, but never released, so the heap gets larger and larger. You'll see this in "top" and your program will slow down when the memory region gets really large.

Best tool for finding leaks: **valgrind**

MALLOC IS "INEXPENSIVE" BUT NOT FREE

It maintains a big pool of memory and uses various techniques to try and keep memory compact.

- **Fragmentation.** Refers to an accumulation of tiny chunks of memory that can't be reused because they are too small for most purposes.
- **Compaction.** Free looks for chances to combine small chunks into larger ones, which are more likely to be useful in future mallocs.

This is different from **garbage collection**, which refers to mechanisms that automatically free an object that no longer has any references to it.

MALLOC/FREE IMPLEMENT DYNAMIC MEMORY MANAGEMENT FOR C++

One worry: malloc is not infinitely fast and can be a bottleneck.

Many performance-intensive applications maintain freelists:

- Only use malloc if the free list is empty.
- This reduces the pressure on the malloc/free subsystem.

HOW A FREELIST WORKS

When you create your class Foo, you also maintain a list of pointers to freed Foo objects: std::list<Foo*> freelist;

Suppose fptr points to a Foo (allocated using **new**):

- When finished with fptr, put it on the freelist (and don't **delete** it). The destructor won't run: fptr is still in use.
- When you need another Foo, check to see if there is a free one on the list. If so, reuse it instead of creating a new object.

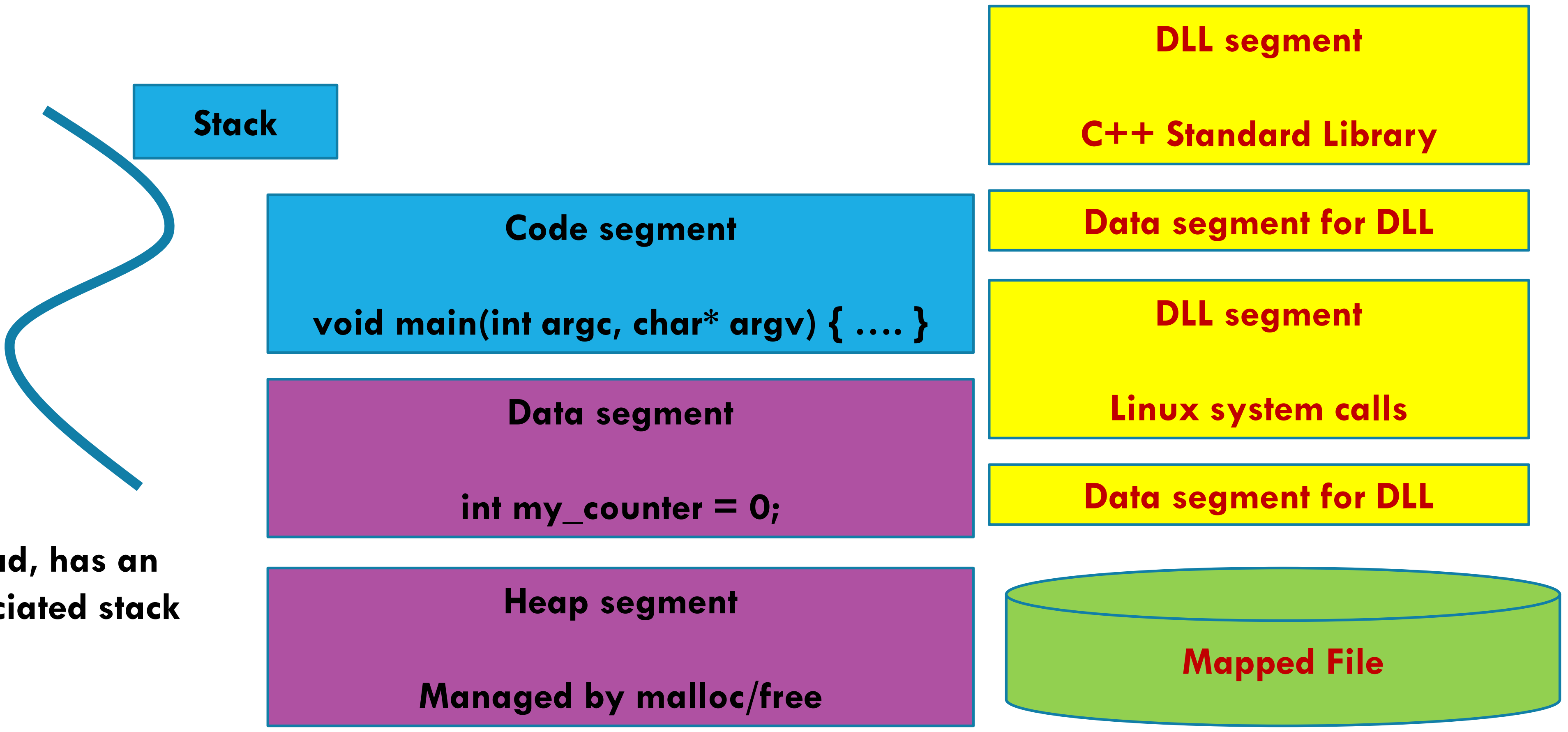
WHICH SEGMENTS HOLD WHICH KINDS OF MEMORY?

Let’s tour the computer from the hardware “up”.

The NUMA computer has a big memory region that encompasses all memory on the machine. Any thread with permission can access any part of this memory (local memory is cheapest).

There may also be memory regions associated with devices such as computer displays, cameras, etc.

VISUALIZING AN ACTIVE PROCESS



Thread, has an associated stack

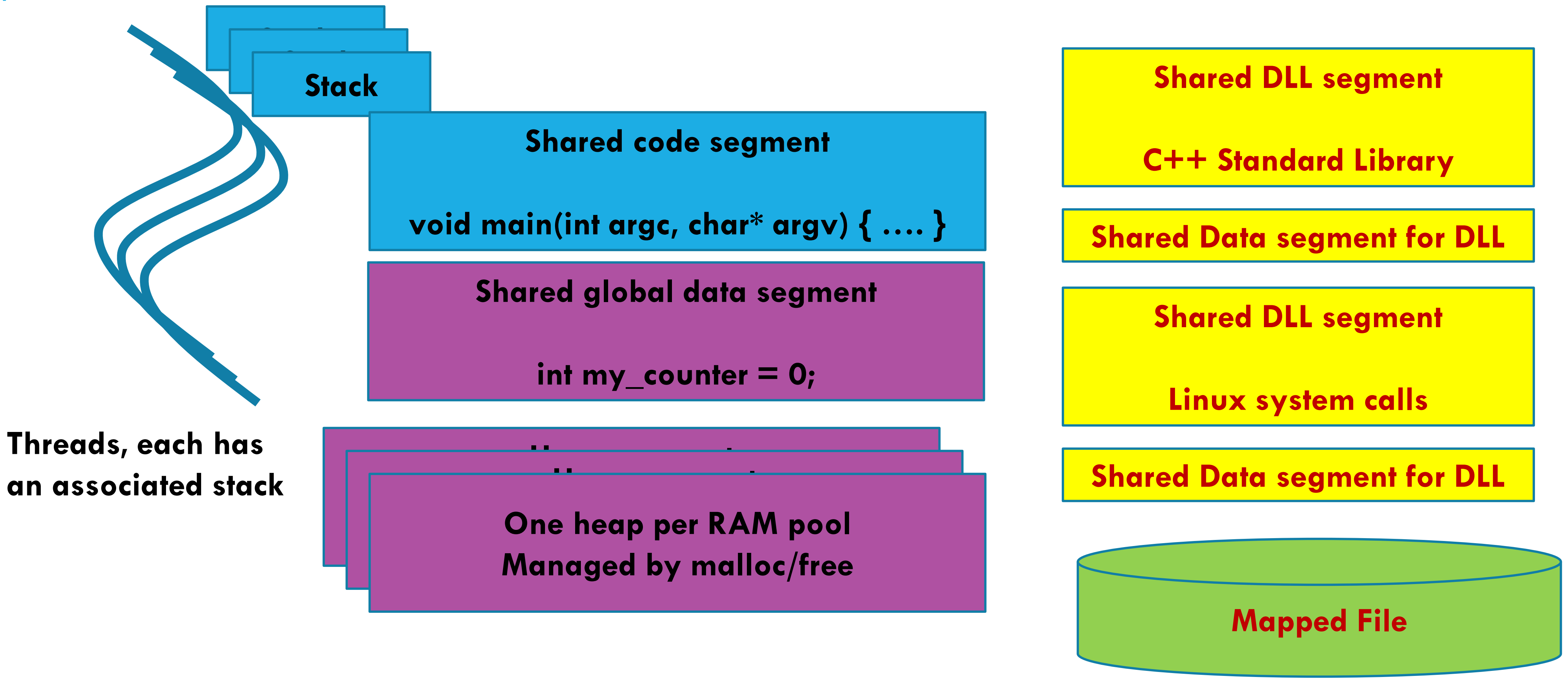
DIFFERENT THREADS IN ONE PROCESS SHARE THE SAME ADDRESS SPACE

The memory of a computer is actually linear, although with gaps used in various ways by the hardware and operating system.

We “think” of the address space as if each thread was next to the other threads, but if you look at the addresses each has its own memory segment.

Linux manages a “mapping” from the addresses each process sees to the actual physical memory. Called a “page table”.

VISUALIZING AN ACTIVE PROCESS



DIFFERENT PROCESSES HAVE DISTINCT ADDRESS SPACES

Each distinct process has its own address space mapping.

Thus an address can mean different things: my 0x10000 might contain code for fast-wc, but your 0x10000 could be part of a data segment.

The hardware knows which process is running, so it can use the proper page table mapping to know which memory it wants.

MAPPED FILES

We will discuss more in a future lecture.

But Linux has a system call that will map a file into memory so that the bytes are directly accessible without doing read/write

For sharing between processes (particularly helpful across programming languages!). *Shared file are limited to one writer.*

VIRTUAL AND PHYSICAL MEMORY

The hardware allows us to “page out” chunks of memory to a disk. If the process touches such a page, a “page fault” occurs.

Then the kernel loads the missing page and lets the process resume execution.

When low on space, this can help… but it also can be costly!

SOME SEGMENTS ARE SHARED BY MULTIPLE PROCESSES

A mapped file appears in memory, like char* array. You can access the bytes directly.

Linux picks the "base address" (hence the same file can easily show up at different places in different processes!)

Changes are automatically rewritten back to the disk. Only one process can do updates; others are "read only"

SOME SEGMENTS ARE SHARED BY MULTIPLE PROCESSES

Consider the standard C++ library. Lots of programs use it!

This segment is read-only, so more than one program can share a single copy. We call it a "dynamically linked library" or DLL

We'll learn how Linux implements DLLs later in the course.

HOW SEGMENTS GROW

Heaps and stacks are the two kinds of segments that can grow as needed, or shrink.

A stack has a limited maximum size, but Linux initially makes it small. As methods call each other and stack space is needed, Linux finds out and quietly grows the "top" of the stack.

This is a case of a "handled" segmentation fault. If you use up the limit, then you get a "stack overflow" error, and a crash.

HOW SEGMENTS GROW

The heap has an initial size, but can be expanded by calling the "sbrk" Linux system call.

Malloc uses this to request extra space. The heap grows at the bottom, towards larger addresses.

With NUMA, there is one heap per RAM, and memory is allocated on a RAM close to the thread that called malloc.

WHAT IF YOU ACCESS A SEGMENT ILLEGALLY?

The most notorious way for a process to crash in Linux is a "segmentation fault"

This means it tried to read from an address that isn't mapped into its address space, or from an "unreadable" region (or write, or execute).

Linux terminates the whole process and might also save a "core" file for you to study using gdb to understand what crashed.

DEEPER DIVE

From the textbook, if we have time

Mechanisms in Procedures

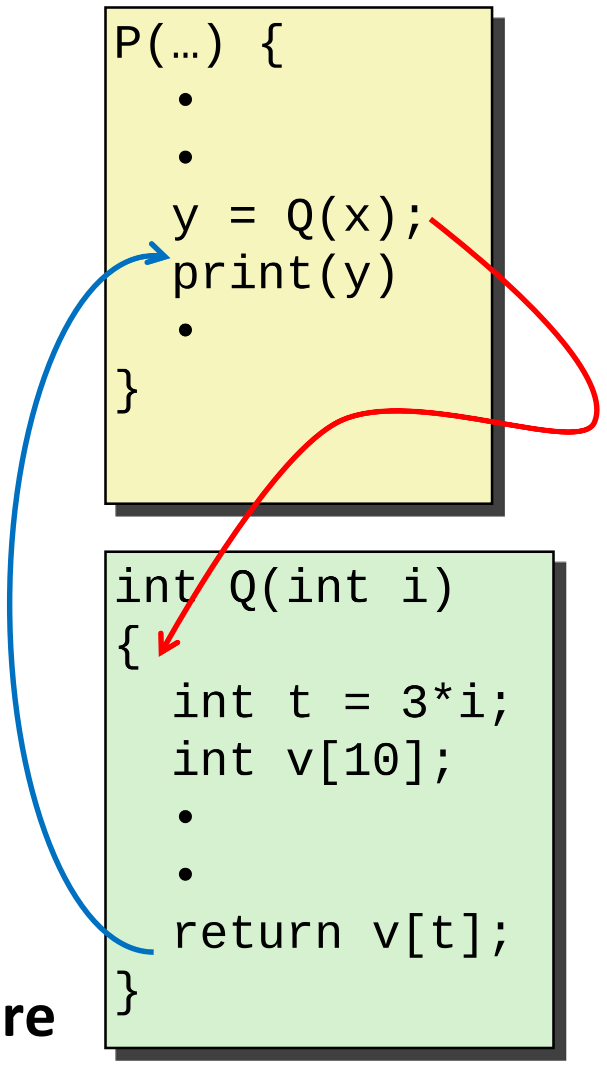
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**

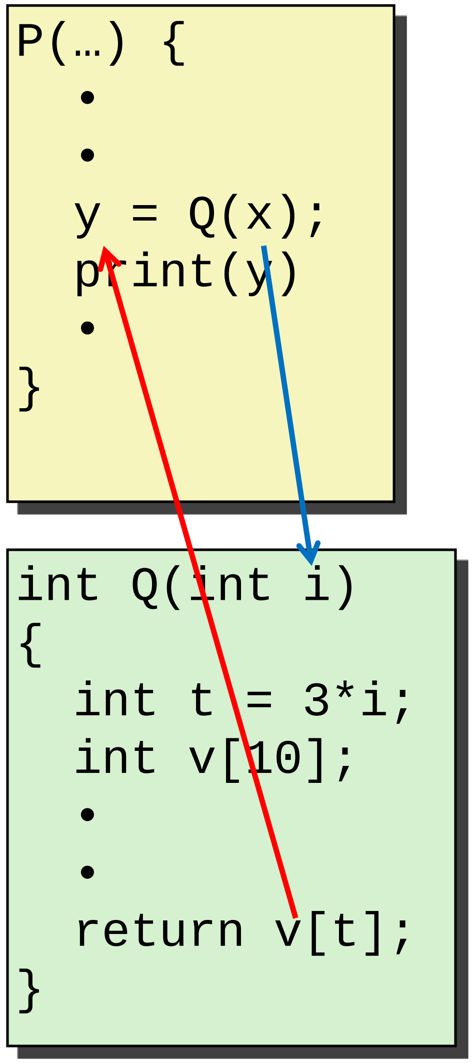


```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**



```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

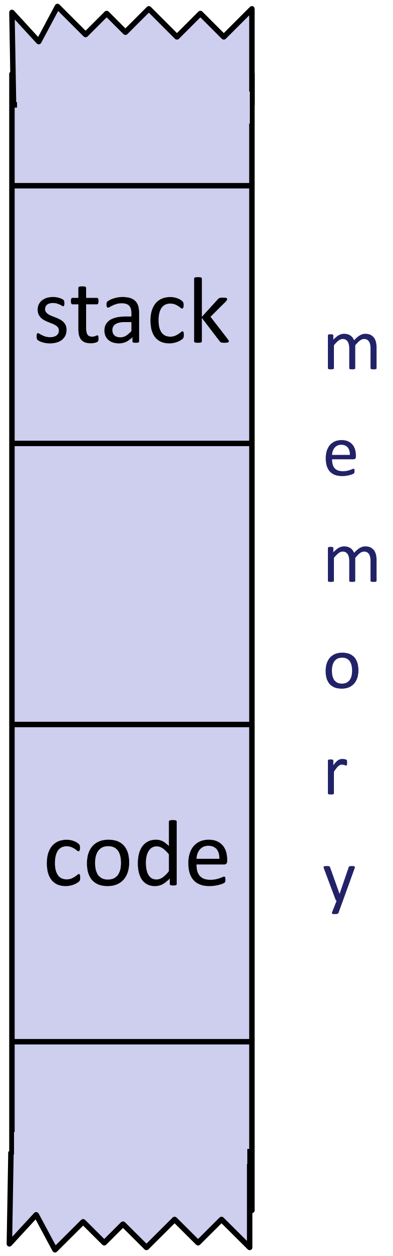
```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Today

- **Procedures**
  - Mechanisms
  - **Stack Structure**
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion

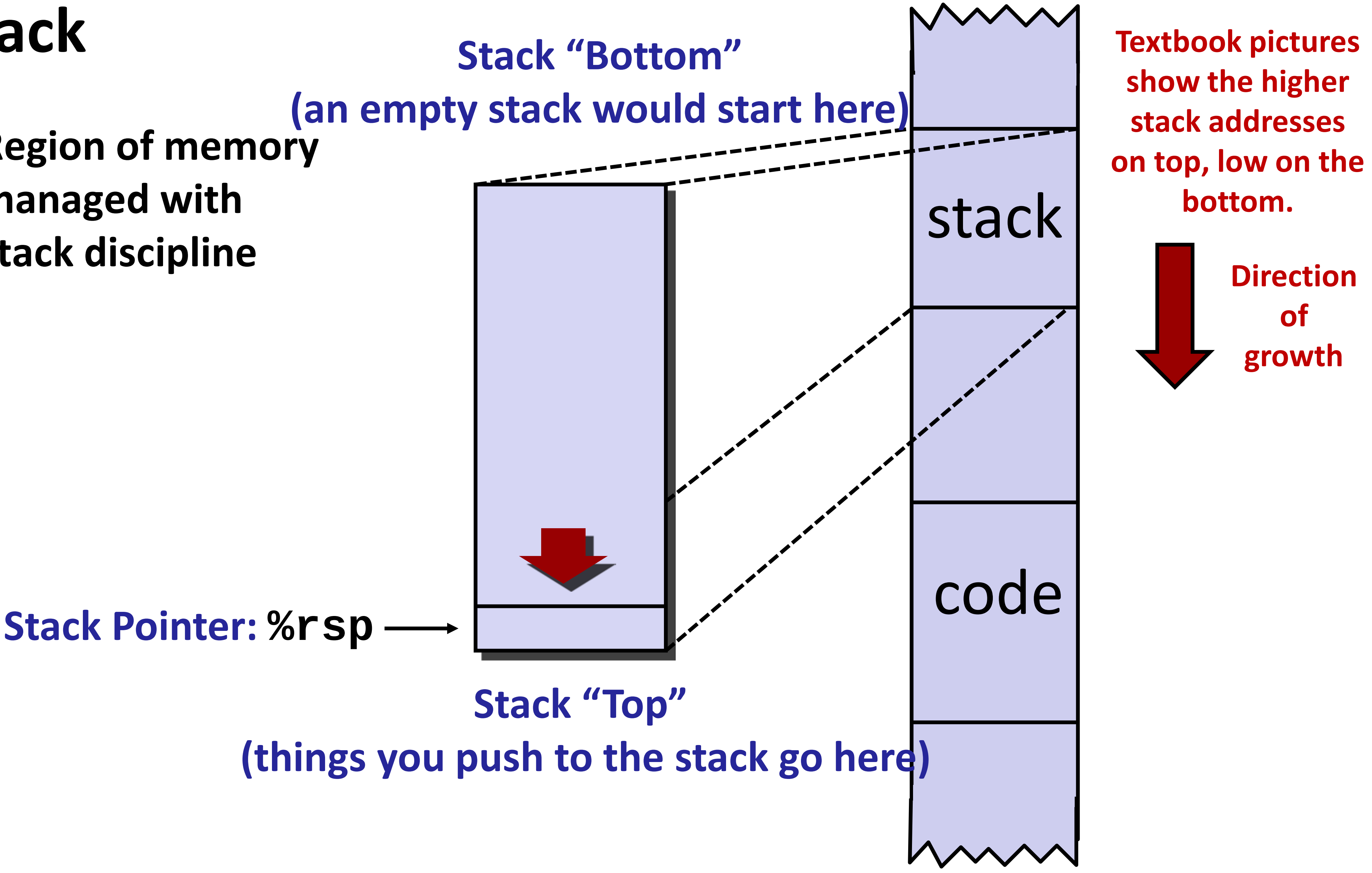
x86-64 Stack

- **Region of memory managed with stack discipline**
  - Memory viewed as array of bytes.
  - Different regions have different purposes.
  - (Like the format of Linux executable files, a policy decision)



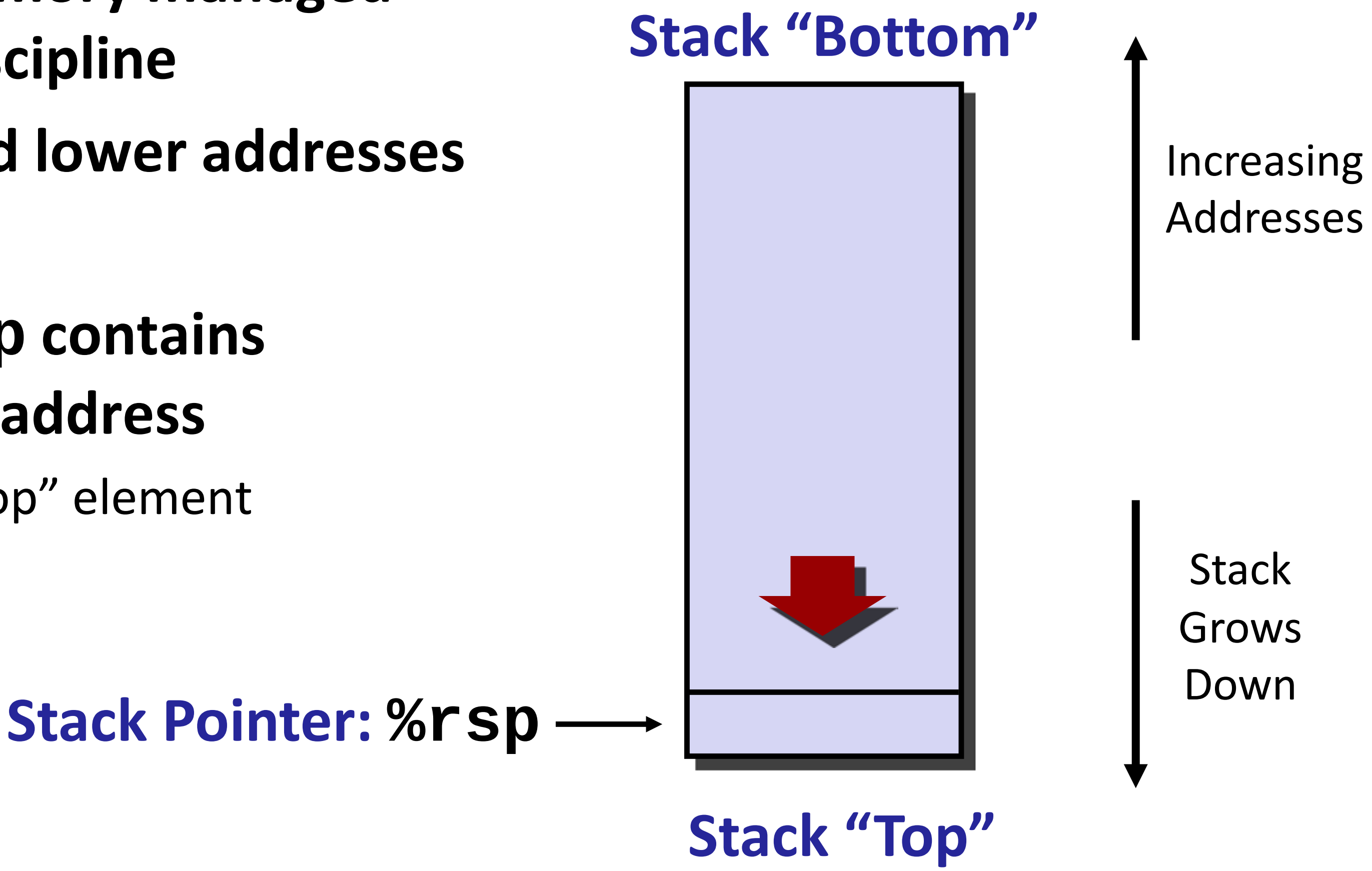
x86-64 Stack

- Region of memory managed with stack discipline


Stack "Bottom"
(an empty stack would start here)
Stack Pointer: %rsp
Stack "Top"
(things you push to the stack go here)
stack
code
Textbook pictures show the higher stack addresses on top, low on the bottom.
Direction of growth

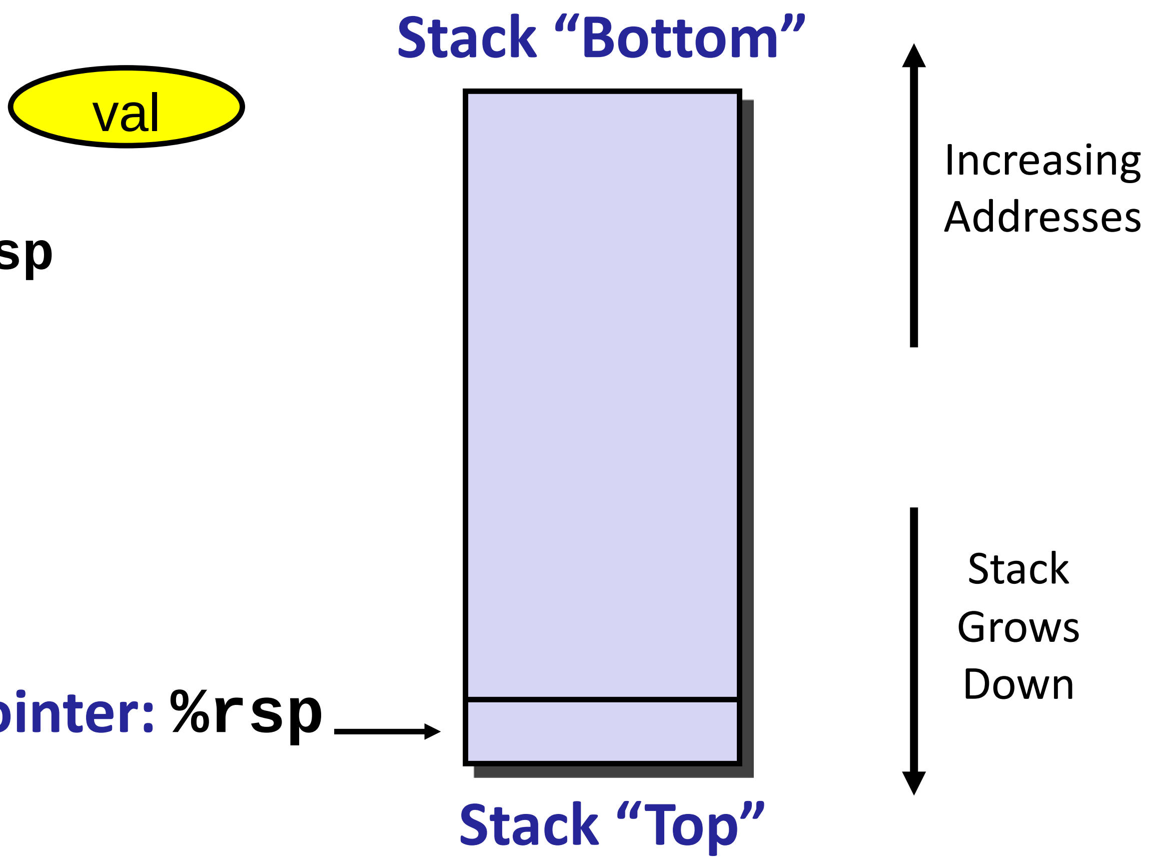
x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
  - address of "top" element



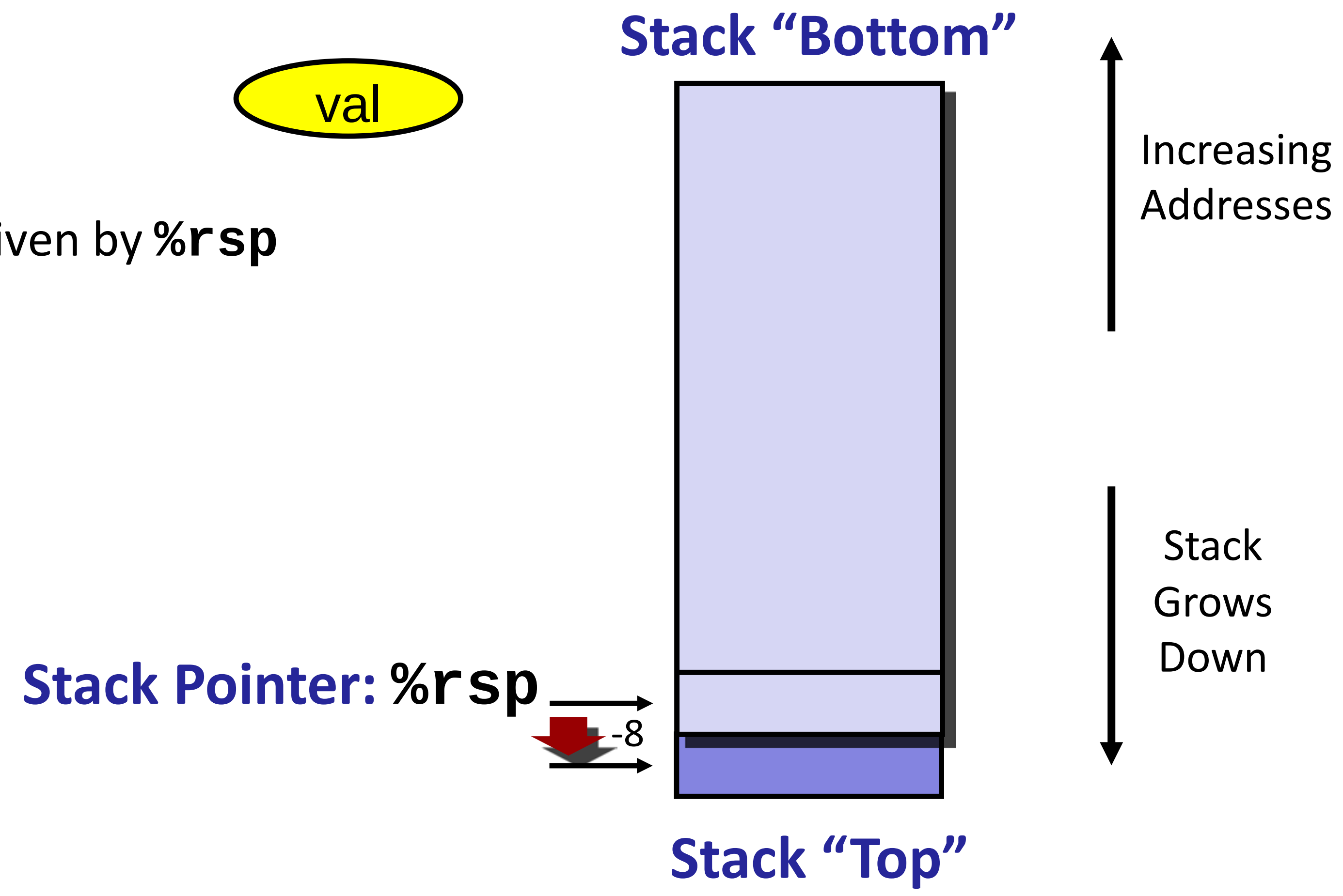
x86-64 Stack: Push

- **pushq** ***Src***
  - Fetch operand at *Src*
  - Decrement **`%rsp`** by 8
  - Write operand at address given by **`%rsp`**



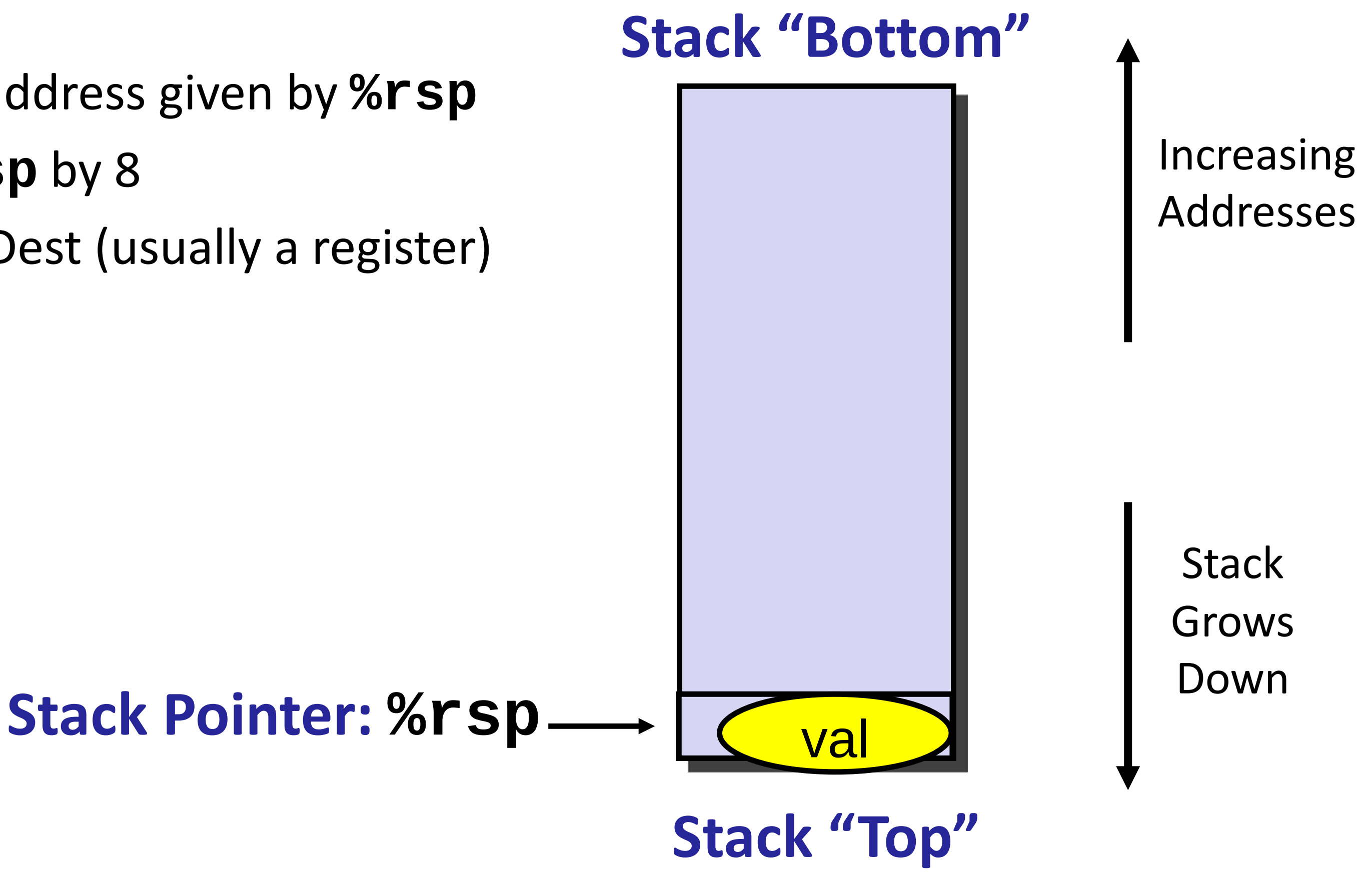
x86-64 Stack: Push

- **pushq** ***Src***
  - Fetch operand at *Src*
  - Decrement **`%rsp`** by 8
  - Write operand at address given by **`%rsp`**



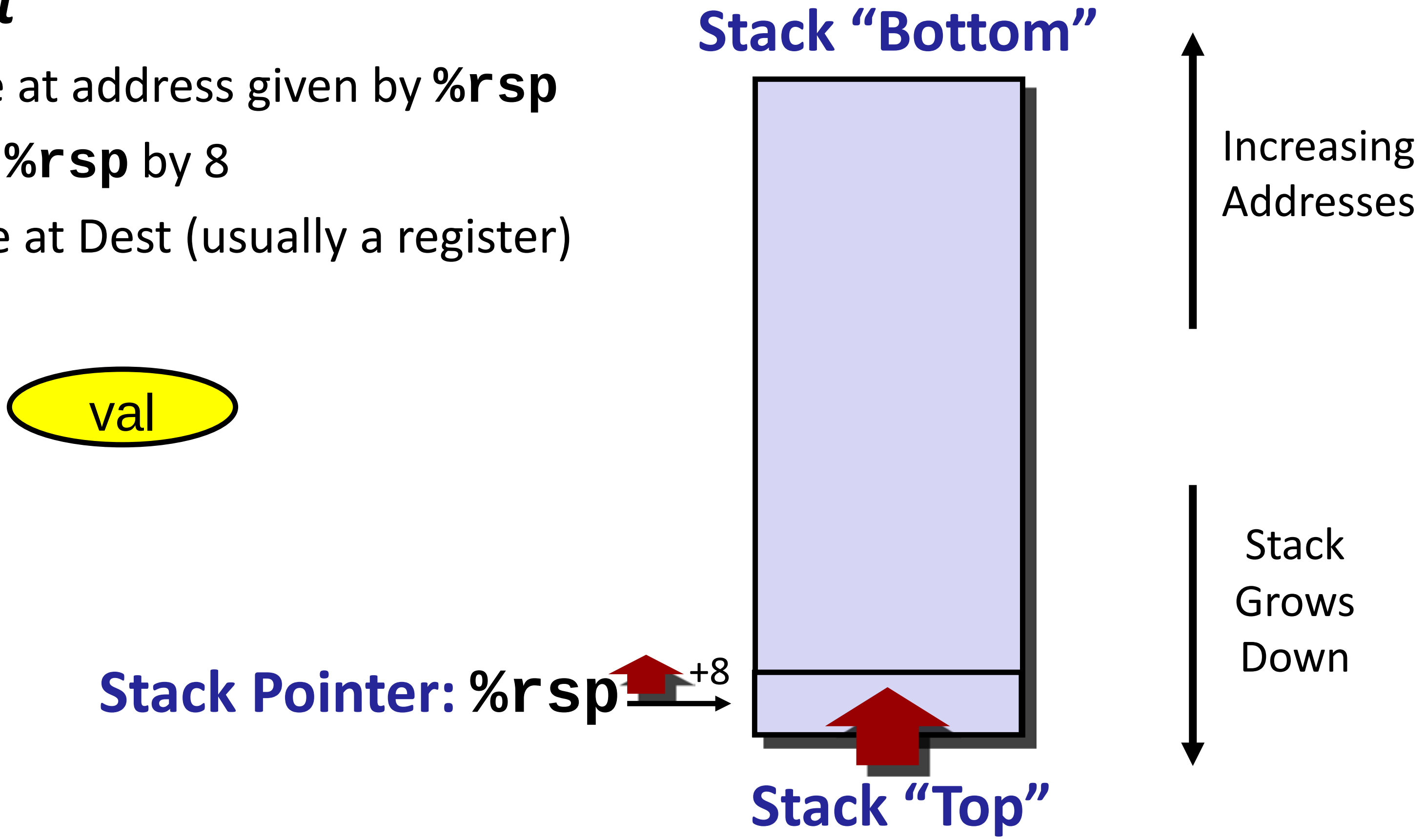
x86-64 Stack: Pop

- **popq** ***Dest***
  - Read value at address given by **`%rsp`**
  - Increment **`%rsp`** by 8
  - Store value at Dest (usually a register)



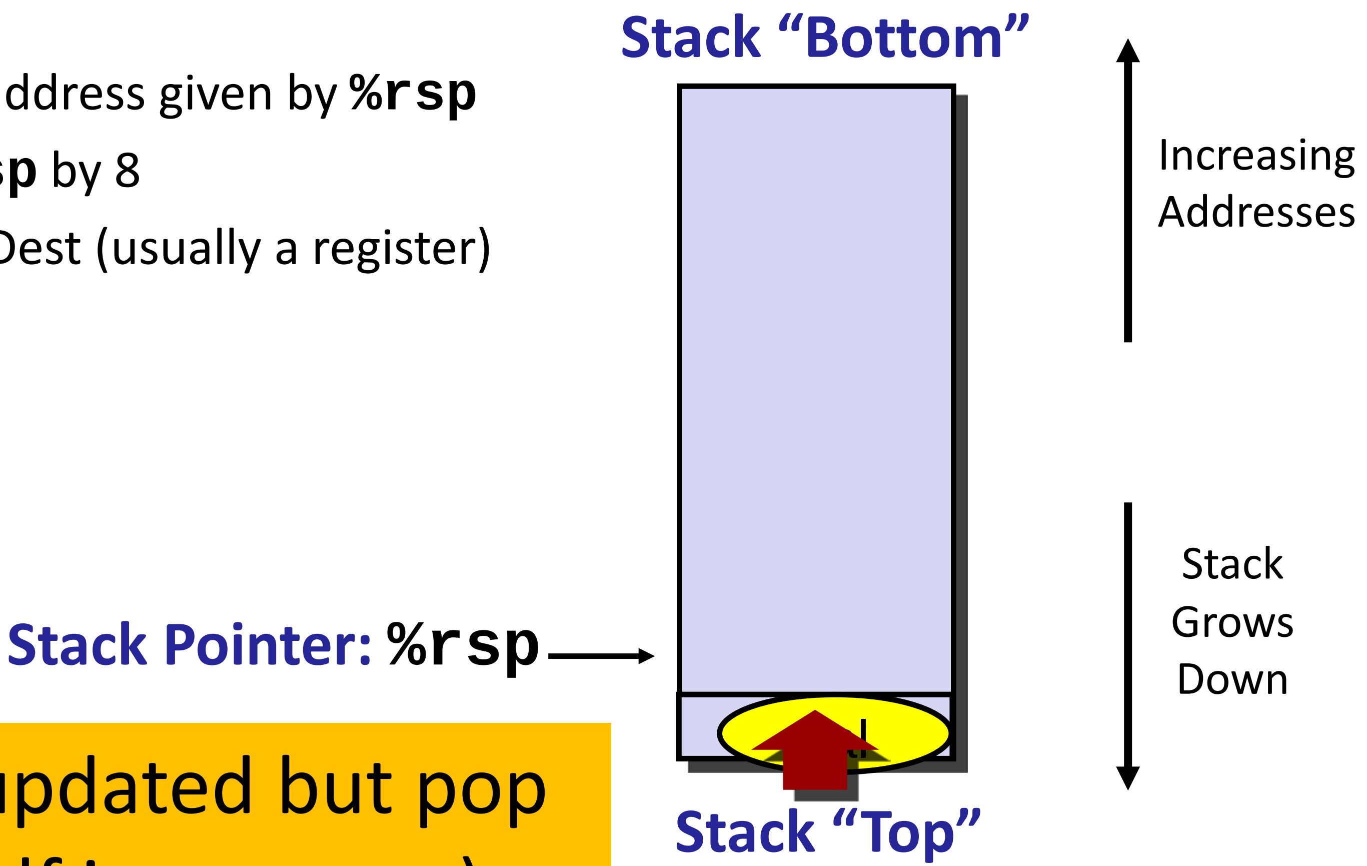
x86-64 Stack: Pop

- **popq** ***Dest***
  - Read value at address given by **`%rsp`**
  - Increment **`%rsp`** by 8
  - Store value at Dest (usually a register)



x86-64 Stack: Pop

- **popq** ***Dest***
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)



(The stack pointer is updated but pop leaves the value itself in memory)

Thought question

- **Why would we care that the value was not somehow “removed” or erased?**

Thought question

- Why would we care that the value was not somehow "removed" or erased?
- … if some other method allocates space on the stack but doesn't initialize the variables, their initial value will be taken from whatever was already there.
- In an application that has internal security rules about which methods can access which data, this could conceivably allow some method to get at data, or a pointer, it should not have been allowed to see!
- Some Linux hacks have taken advantage of this property.

Today

- **Procedures**
  - Mechanisms
  - Stack Structure
  - Calling Conventions
    - **Passing control**
    - Passing data
    - Managing local data
  - Illustration of Recursion

Code Examples

```
void multstore(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

```
long mult2(long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax        # a
  400553:  imul   %rsi,%rax        # a * b
  400557:  retq                    # Return
```

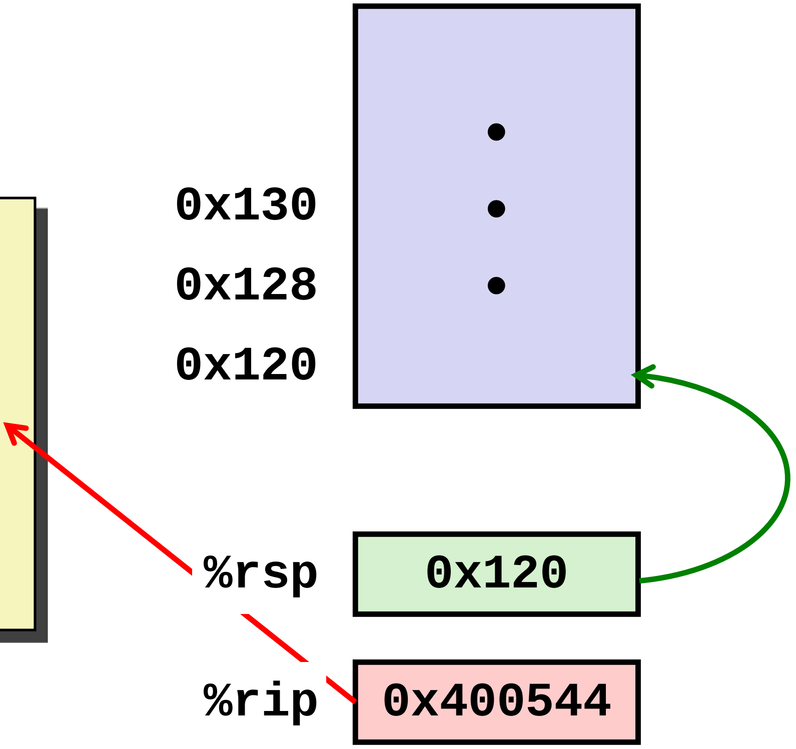
Procedure Control Flow

- **Use stack to support procedure call and return**
- **Procedure call: `call label`**
  - Push return address on stack
  - Jump to ***label***
- **Return address:**
  - Address of the next instruction right after call
  - Example from disassembly
- **Procedure return: `ret`**
  - Pop address from stack
  - Jump to address

Control Flow Example #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

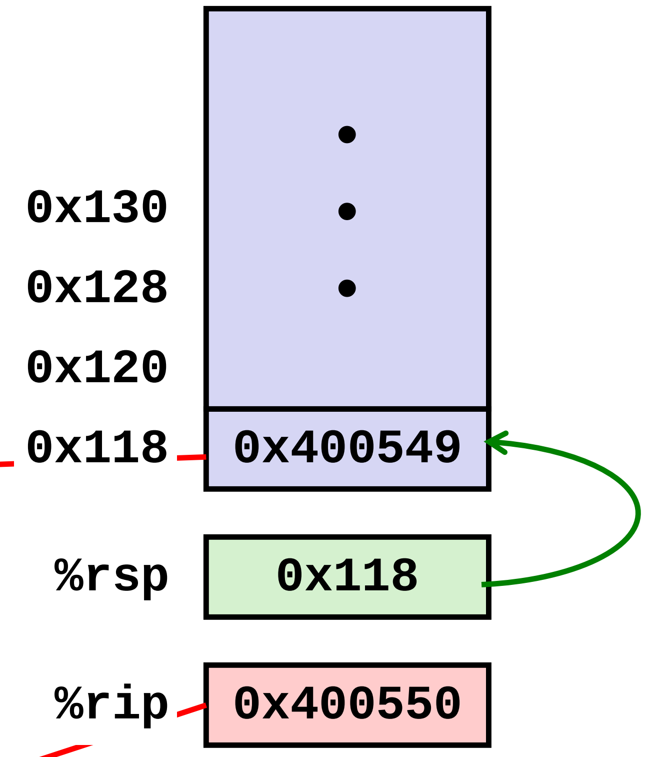
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

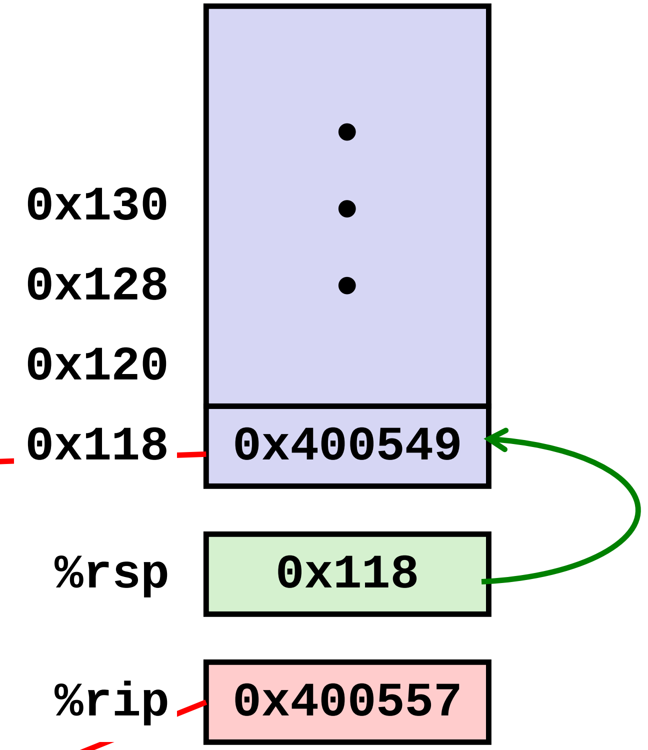
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

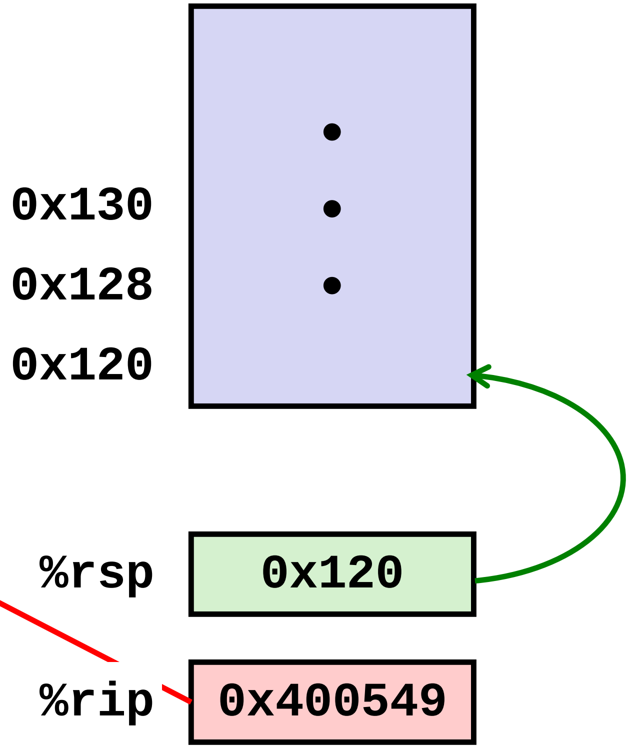
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



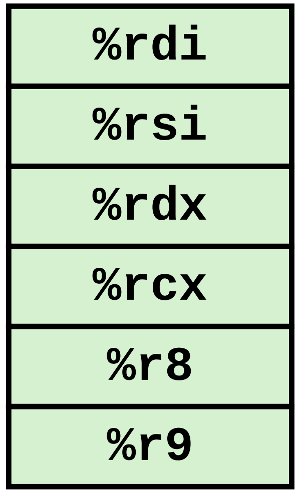
Today

- **Procedures**
  - Mechanisms
  - tack Structure
  - Calling Conventions
    - Passing control
    - **Passing data**
    - Managing local data
  - Illustrations of Recursion & Pointers

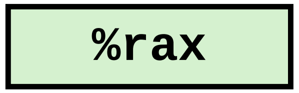
Procedure Data Flow

Registers

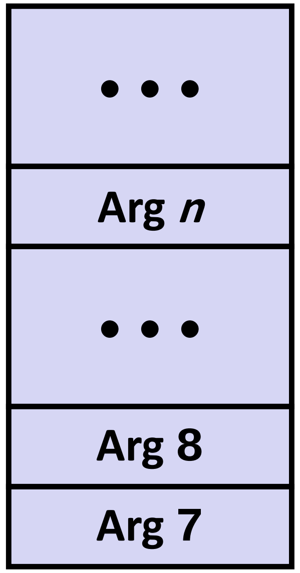
- First 6 arguments



- Return value



Stack



- Only allocate stack space when needed

Data Flow Examples

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: mov    %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)      # Save at dest
  • • •
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax       # a
  400553:  imul   %rsi,%rax       # a * b
  # s in %rax
  400557:  retq                   # Return
```

SUMMARY AND TAKE-AWAYS

Visualize your application as a collection of memory segments.

Some are restricted in various ways: read only, can or cannot grow (and if so, from which end), executable.

Mapped files are a form of segment that allow distinct processes to share memory (even if coded in different languages!)