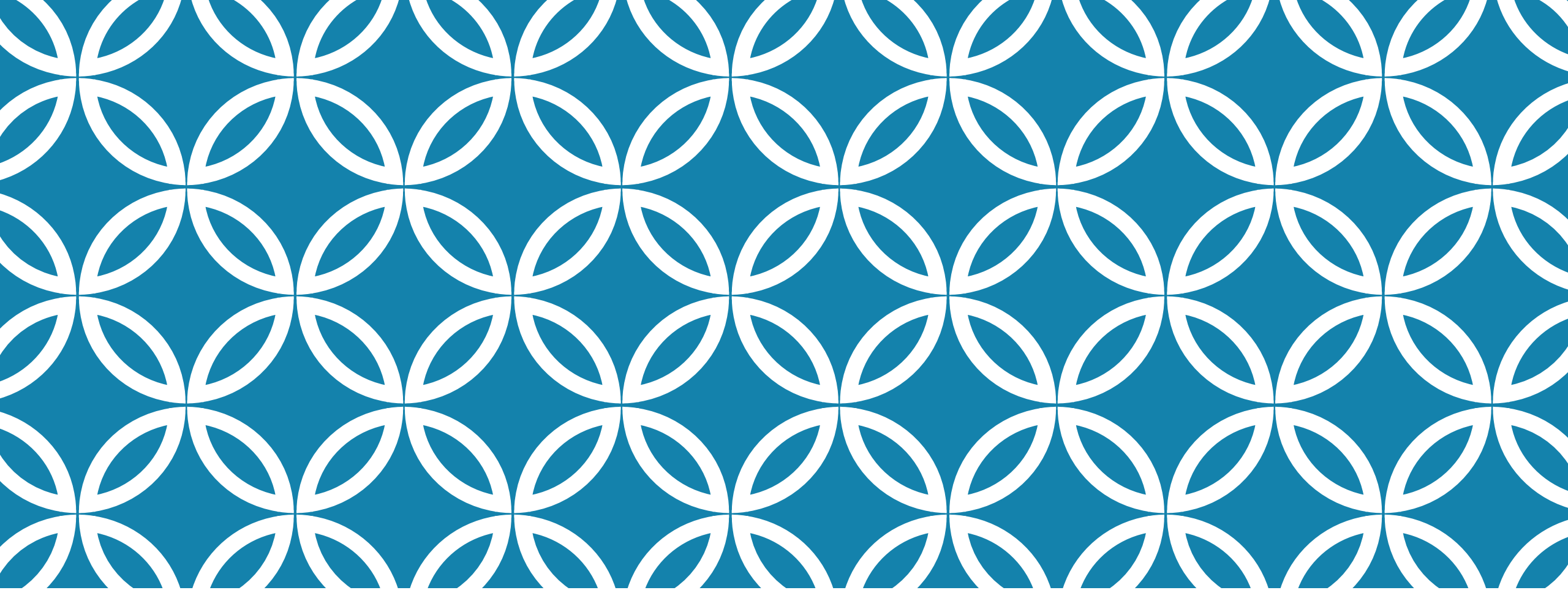


MEMORY MANAGEMENT

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

How procedure calls work in C++ and similar languages

- Procedures
  - Mechanisms
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data

Mechanisms in Procedures

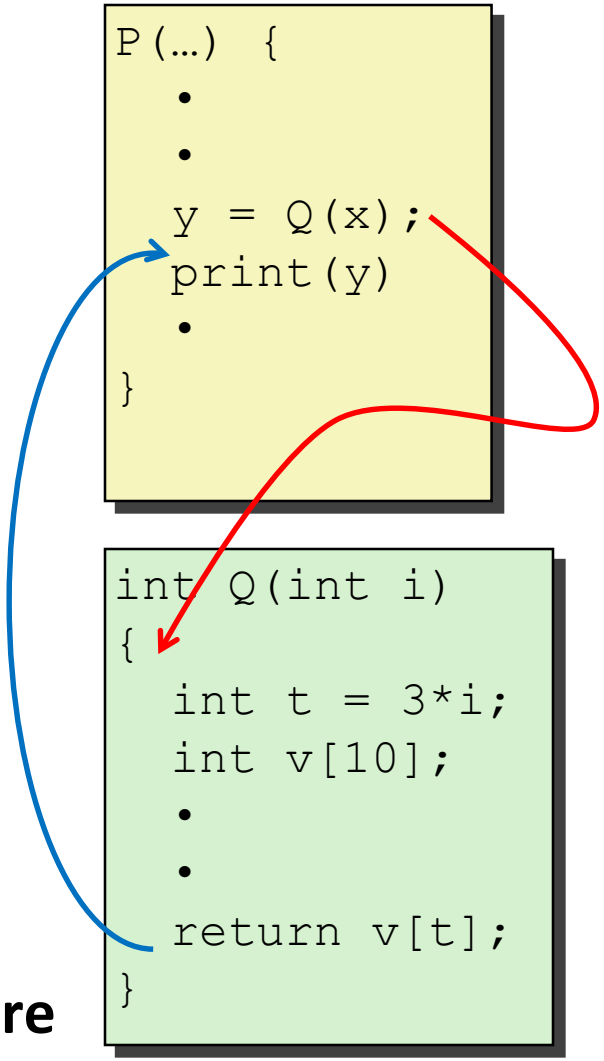
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 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];
}
```

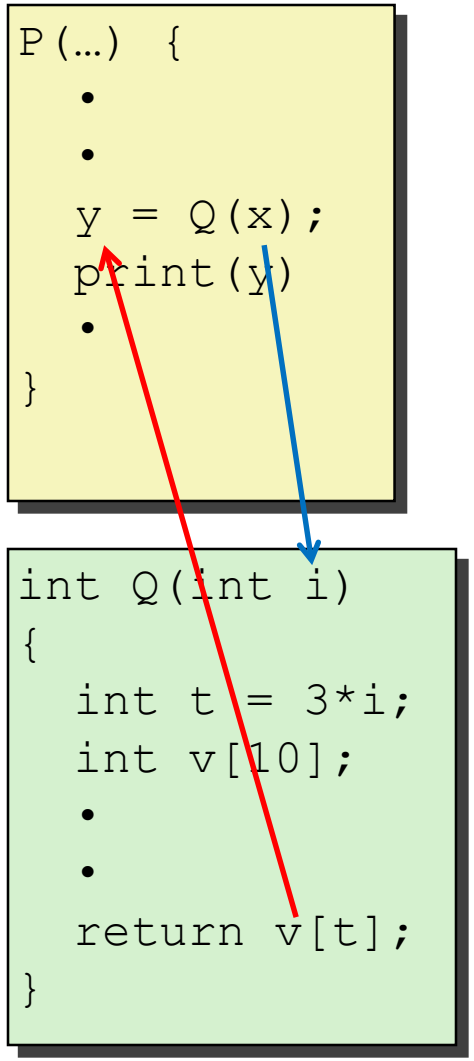
Mechanisms in Procedures

- **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**



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**



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)
   •
}
```

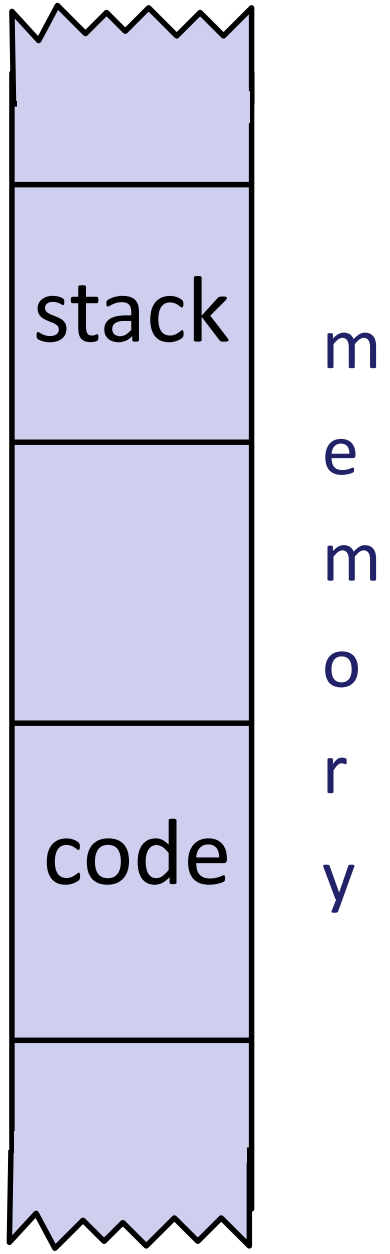
```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

Today

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

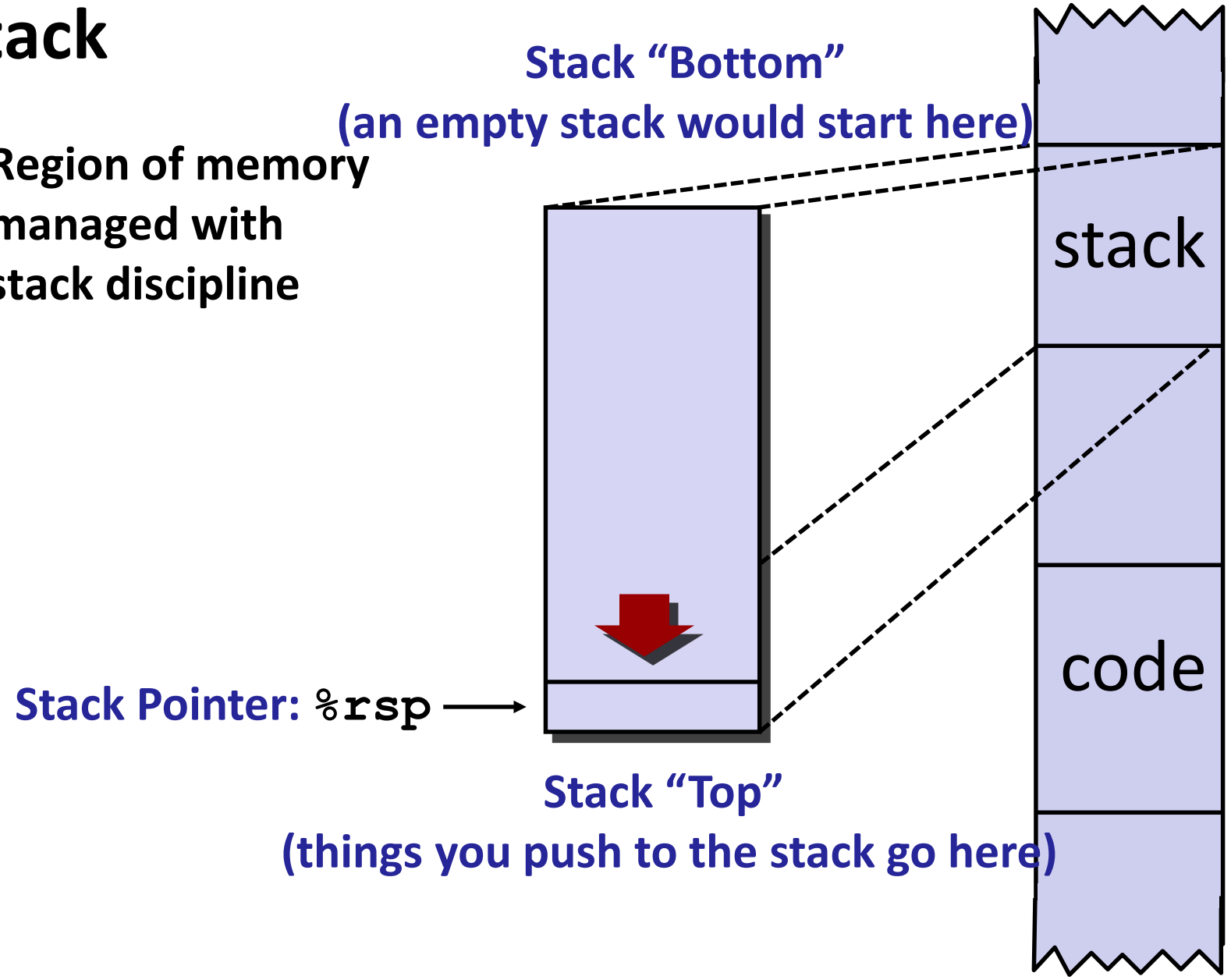
x86-64 Stack

- **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

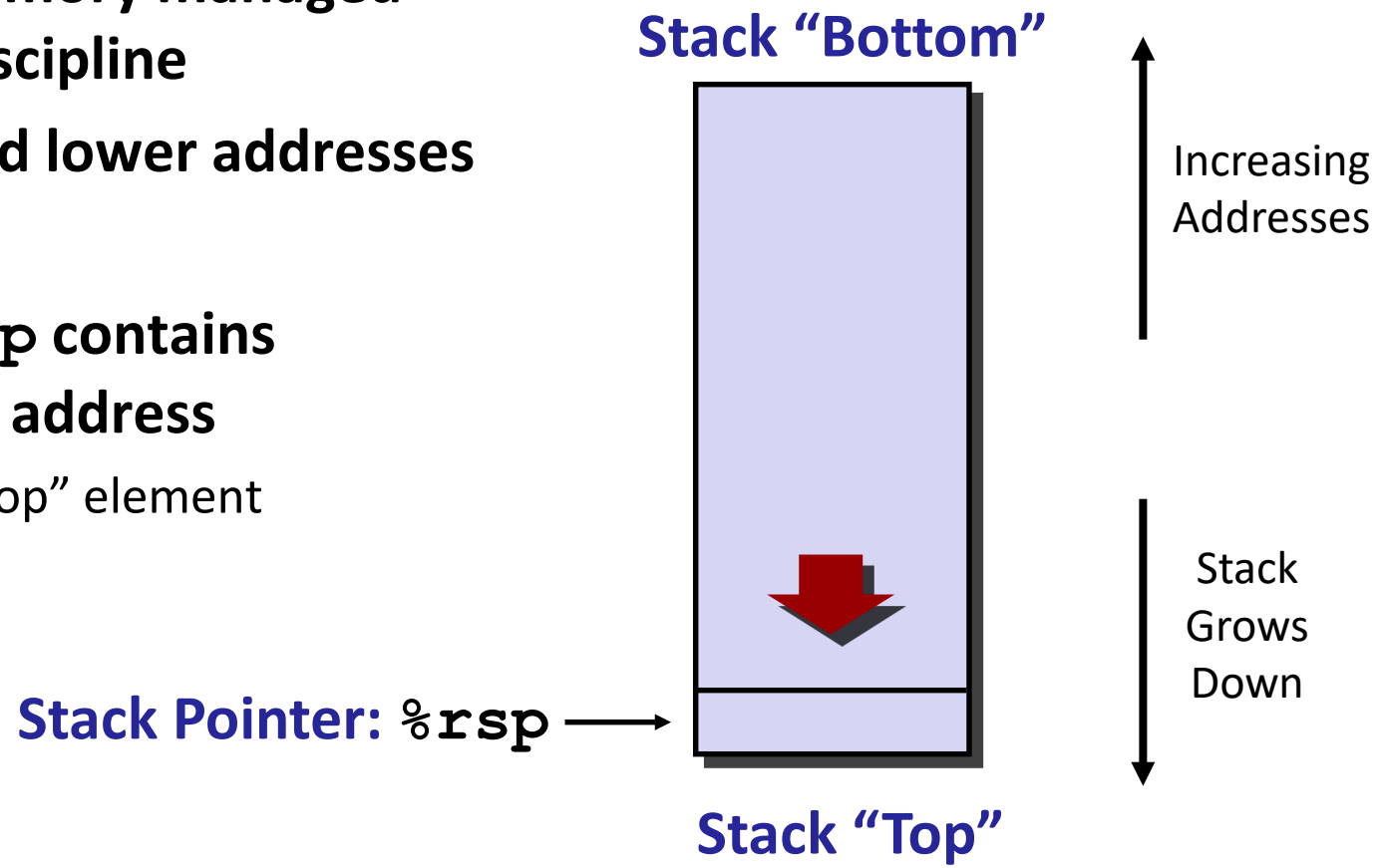
- Region of memory managed with stack discipline



x86-64 Stack

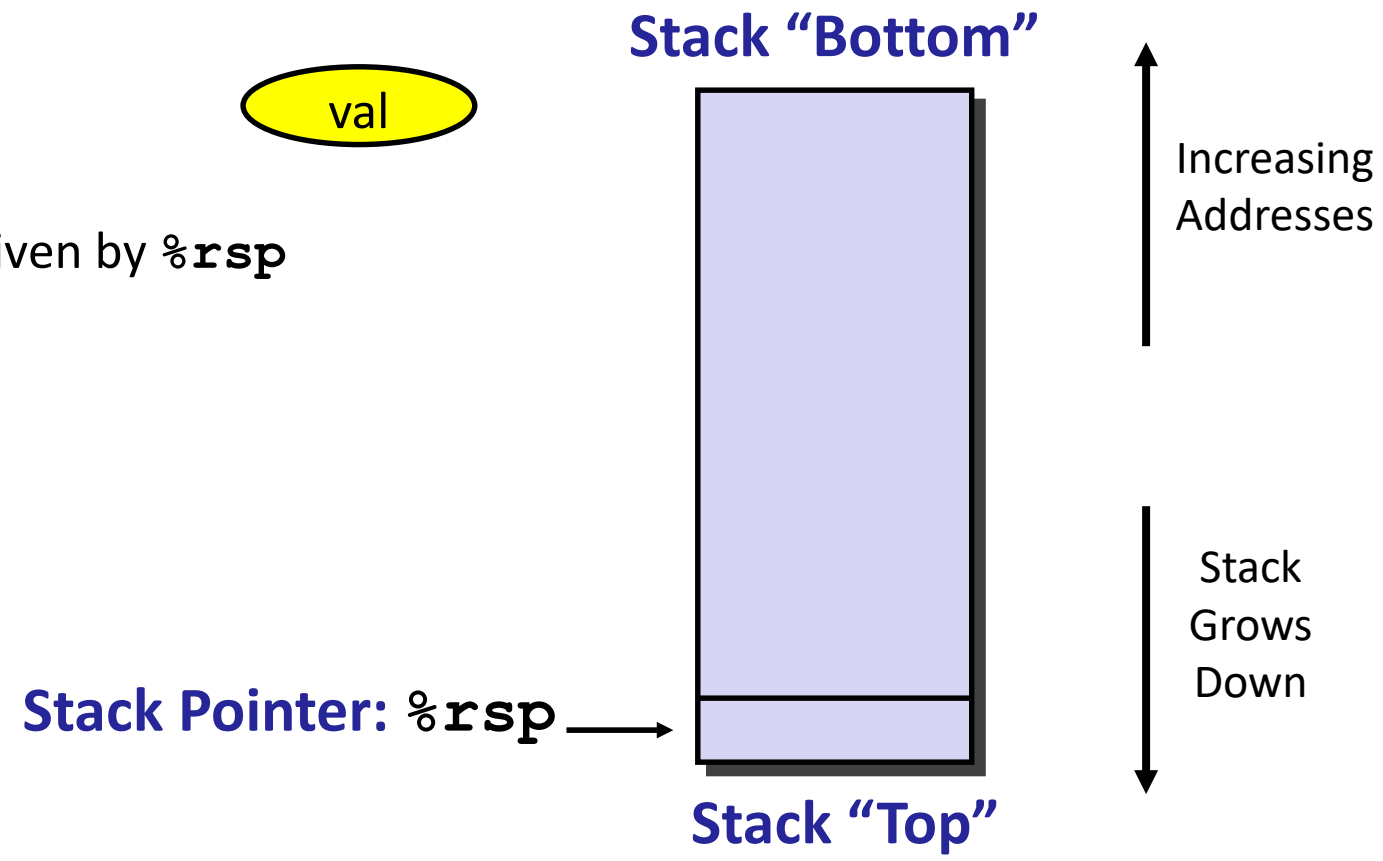
- Region of memory managed with stack discipline
- Grows toward lower addresses

- Register `%rsp` contains lowest stack address
  - address of "top" element



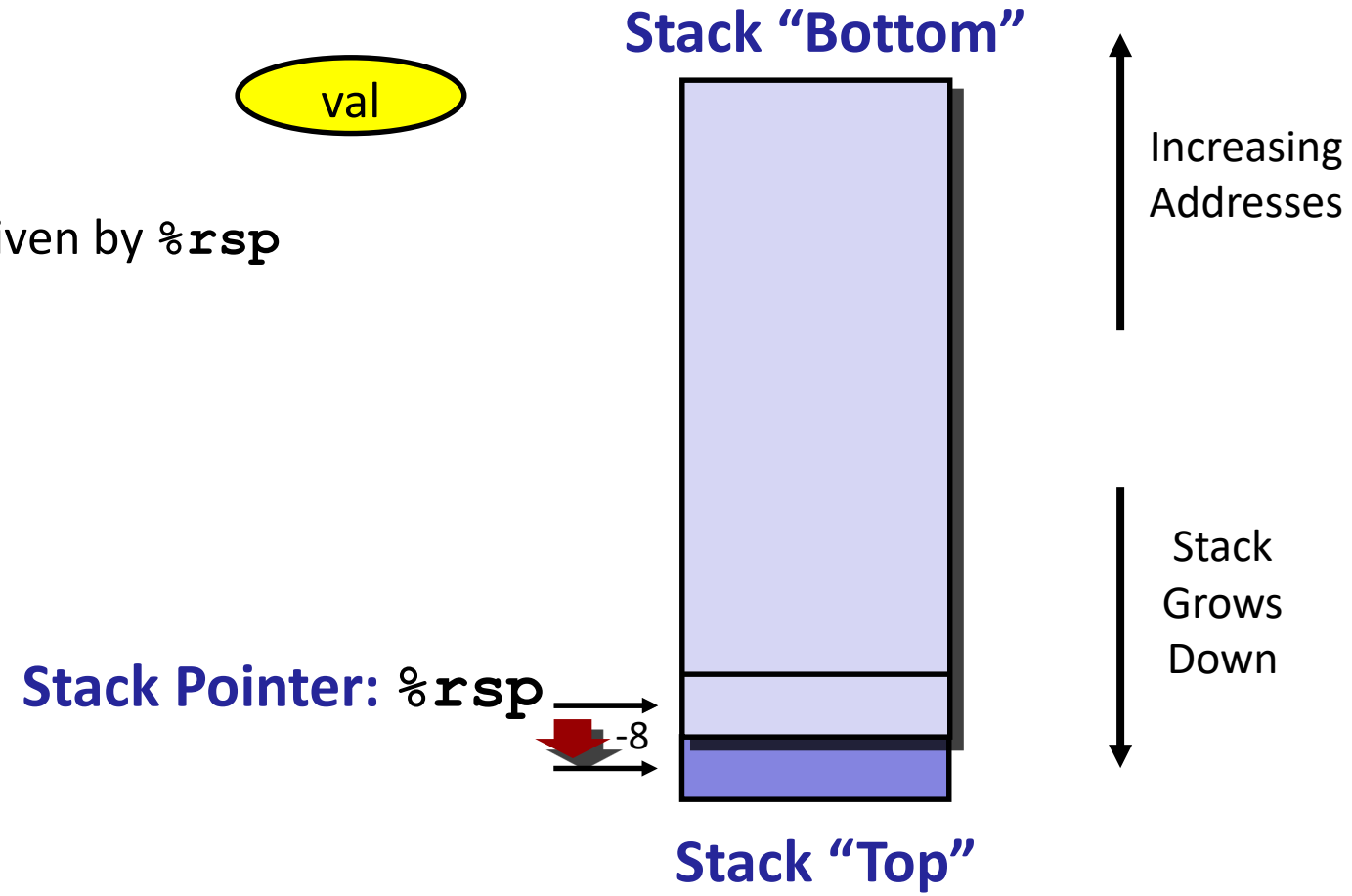
x86-64 Stack: Push

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



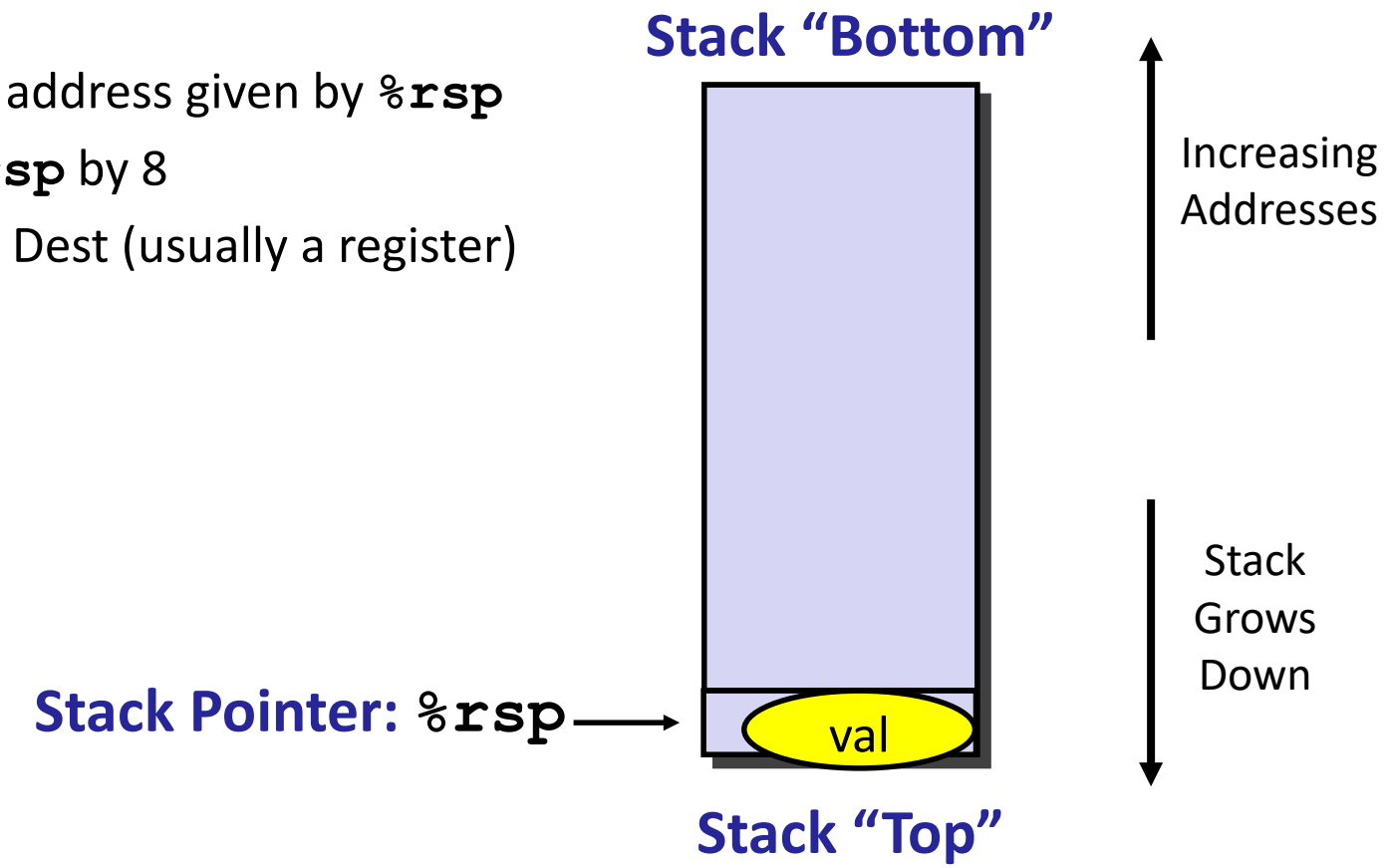
x86-64 Stack: Push

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



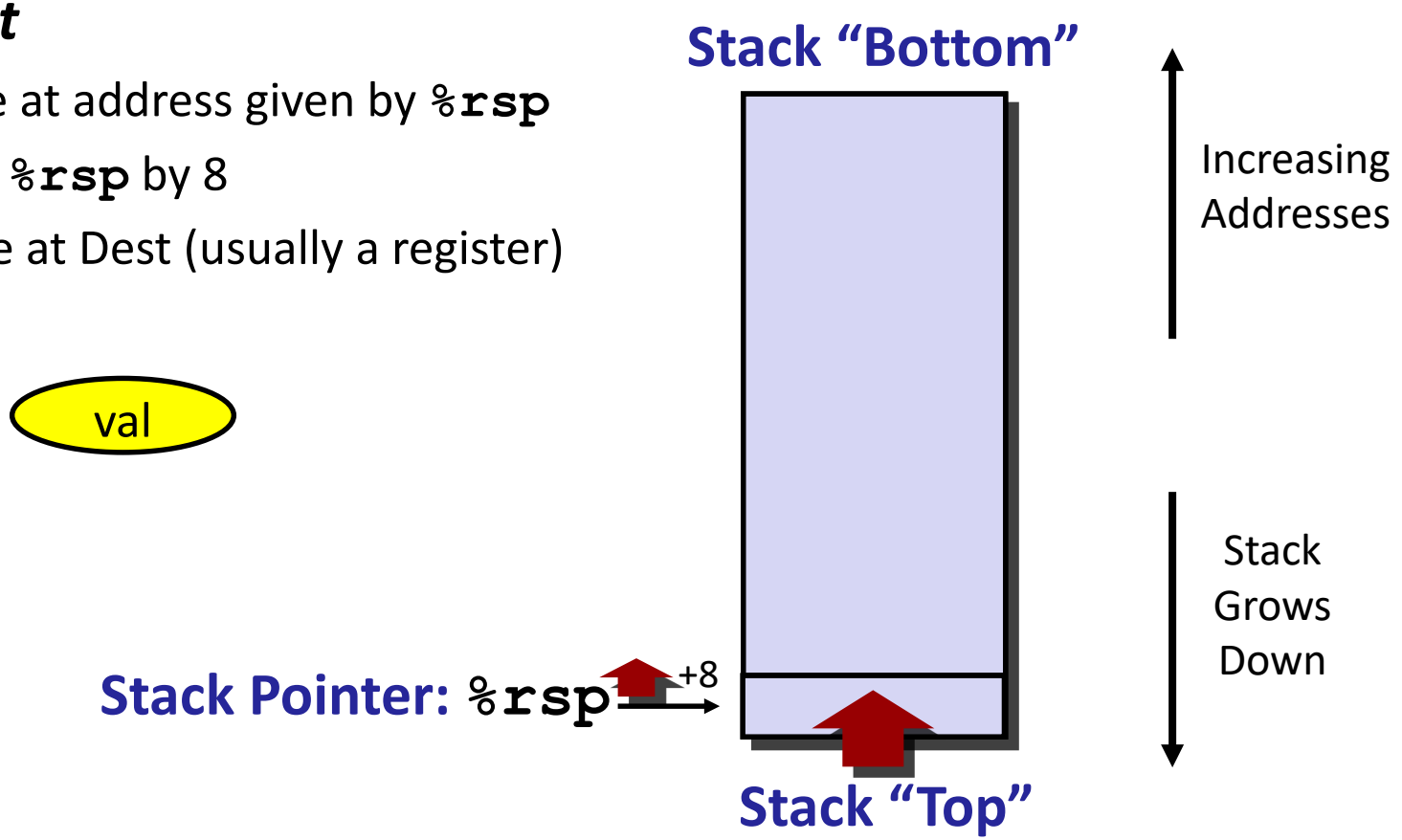
x86-64 Stack: Pop

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



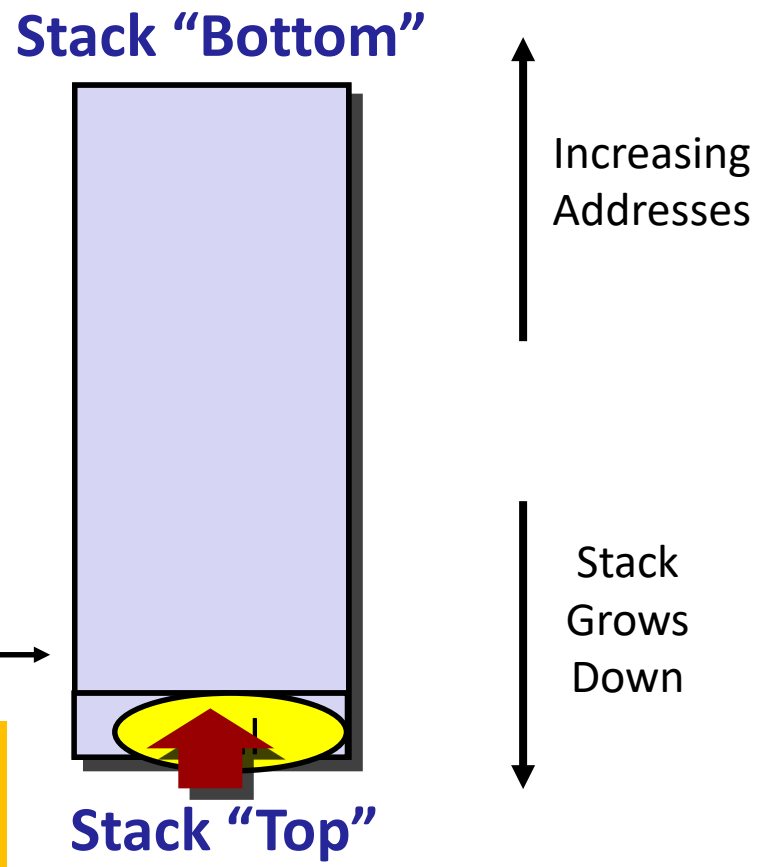
x86-64 Stack: Pop

- `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
```

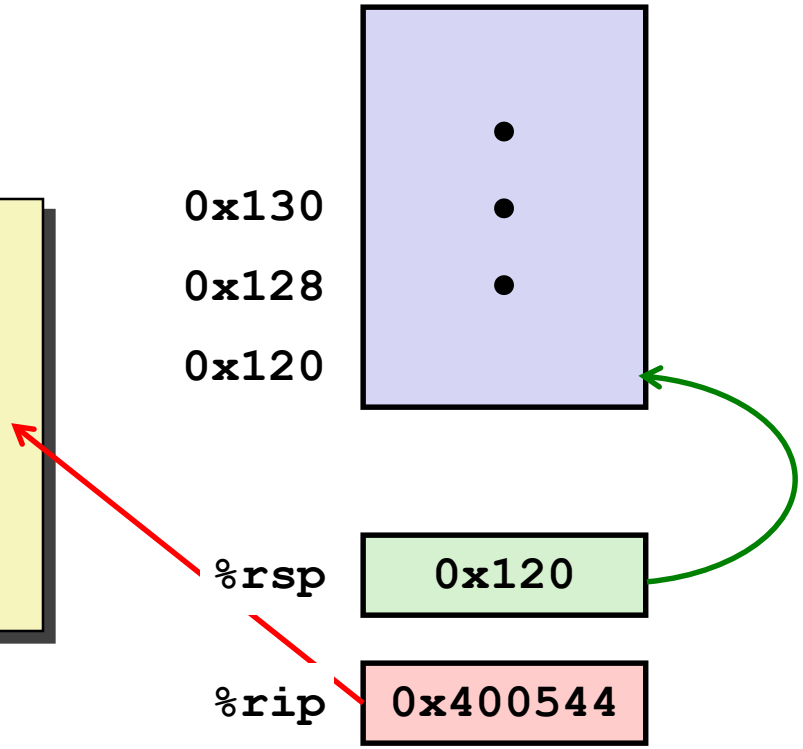
Procedure Control Flow

- **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)
  •
  •
```

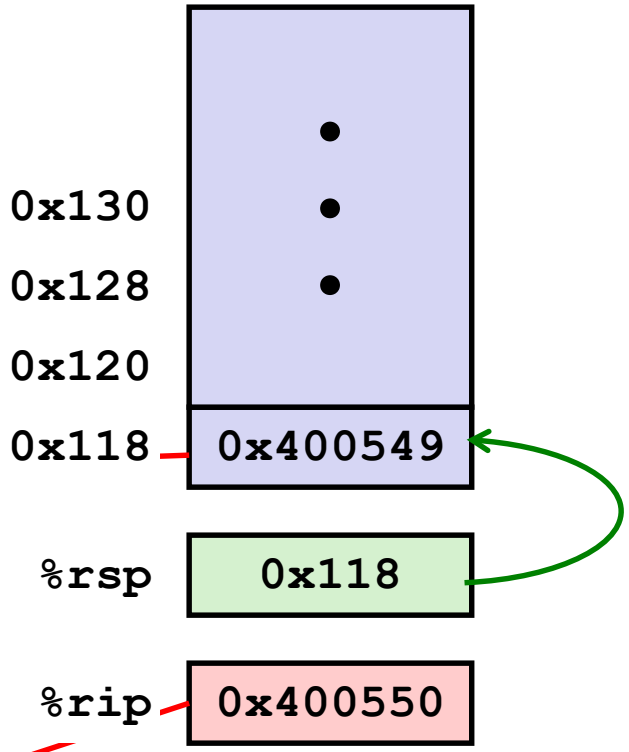
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #2

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

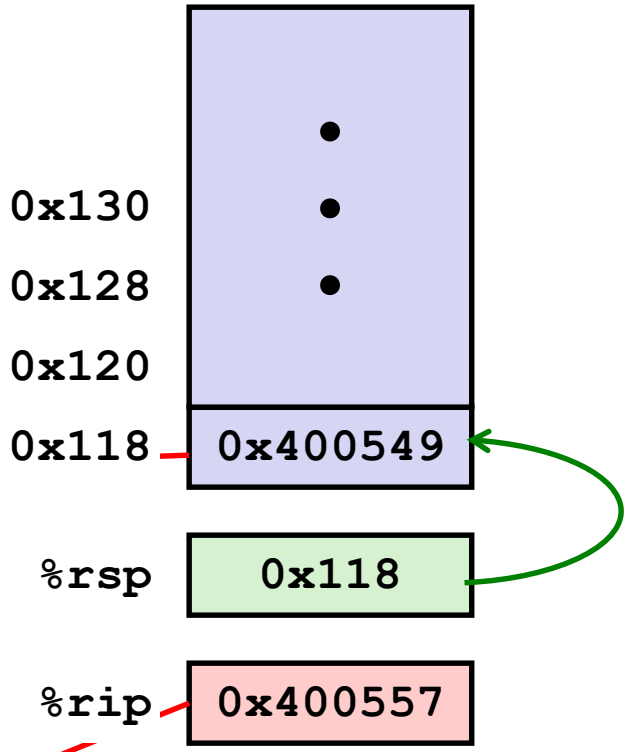
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #3

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

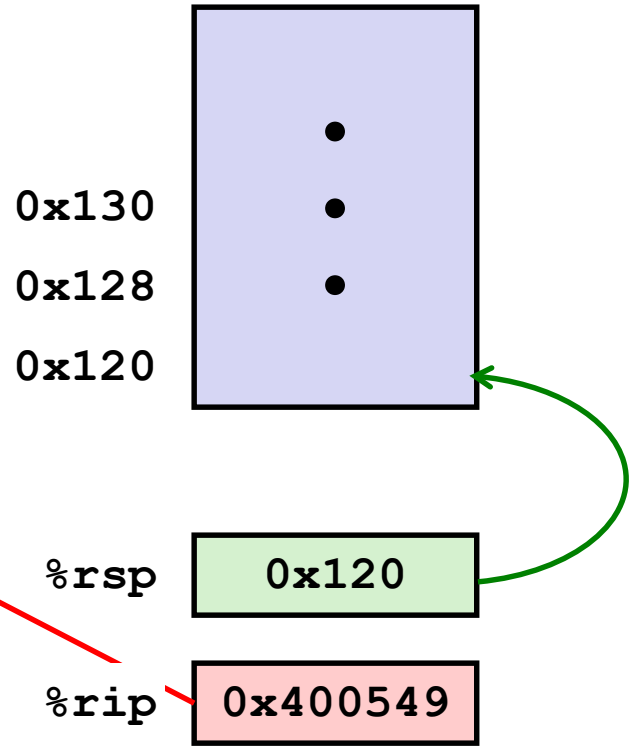
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #4

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

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



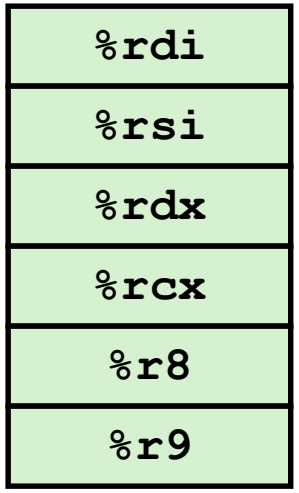
Today

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

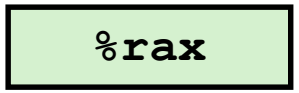
Procedure Data Flow

Registers

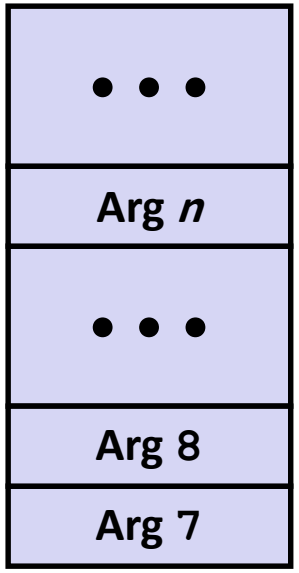
- First 6 arguments



- 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
```

LET'S SEE THIS IN A LARGER CONTEXT

Process: An active instance of some program

Segment: A region of memory that Linux has made available to some process or processes.

Permissions: Linux can control access to segments: read, write, execute. And there can be "gaps" too – holes in the address space with no permission at all.

HOW DO PROGRAMS USE MEMORY?

Your code is compiled to machine instructions

You have global variables, initialized to 0 or perhaps some other value.

As your process executes it consumes, then releases, stack and memory resources.

STANDARD SEGMENTS IN LINUX

Any given process can only see memory that Linux maps into its address space. These segments include

- Code segments that have executable machine instructions
- Data segments that hold various forms of data (globals)
- Mapped files (for speedy access without using open/read)
- Flexible sized "heap" segments managed by malloc/free. A heap grows at the "top" (towards bigger addresses)
- Stack segments: these grow too, but at the bottom.

WHERE DO NEW OBJECTS LIVE?

As a process executes, it will often need to allocate (or free) objects

C++ examples:

- char* ptr = (char*)malloc(size); ….. free(ptr); // Old "C" style
- std::map<std::string,count> myMap; // Modern C++
- Semaphore pcounter(5); // If executed "inline"

WHERE DO NEW OBJECTS LIVE?

Three cases:

- Global variables live in a data segment.
- Things declared “inline” consume space on the stack of the procedure (or code block) where they were declared, and are allocated when that declaration is “executed”
- Other objects are created in the heap and we work with pointers to them: instead of “x” being an object, we have an object xp that holds the address of the object

WHERE DO NEW OBJECTS "LIVE"?

- **Stack:** objects declared "in line" in some scope. As the thread leaves that scope, these are automatically deleted

- **Others are in dynamically allocated memory** as in this example:

```
auto mymap = new std::map<std::string, int>(constructor args);
...
delete mymap;
```

- … **new** automatically uses "malloc." **delete** calls "free"

C++ NOTATIONS FOR ACCESSING THINGS

To access something in the std namespace: std::xxx.

- std::string
- std::map<std::string, int>
- If you don't specify the namespace, C++ looks in the "default" one, and also in any that were imported via the "using" statement.

To access a field or a method in object x: x.field, x.method(args)

- In C++ we often overload an operator: x[k] might call x.get(k)

If xp is a pointer to object x, x->field, x->method(args)

- Remember how std::map overloaded []? If xp points to a map, write (*xp)[k]

POINTERS VERSUS REFERENCES

When you are looking at an instance of an object, we say that you have a reference to the object. In C++ you would use the "dot" notation to access fields in this case.

Example:

```
std::vector<int> my_vec;
```

Here, my_vec is a name for a vector instance. my_vec.size() is the current length of the vector. my_vec[k] is the k'th element.

POINTERS VERSUS REFERENCES

If you have a pointer to the object, you need to indirect through the pointer, using the → operator.

Example:

```
auto vp = new std::vector<int>;
```

Here, vp points to a vector instance. vp→size() is the current length of the vector. (*vp)[k] is the k'th element.

POINTERS VERSUS REFERENCES

Convert a reference to a pointer using &

```
auto vp = &my_vec;  // vp will point to my_vec
```

If you convert a pointer to a reference, C++ makes a copy:

```
auto my_vec = *vp;   // creates a copy of the vector
                     that vp was pointing to.
```

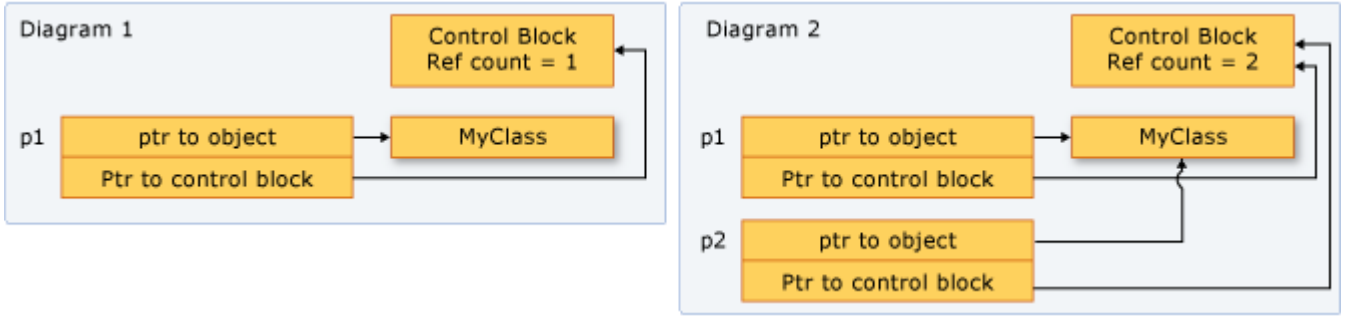
SHARED_PTR

When working with pointers, people often forget to call malloc.

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

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

SHARED_PTR



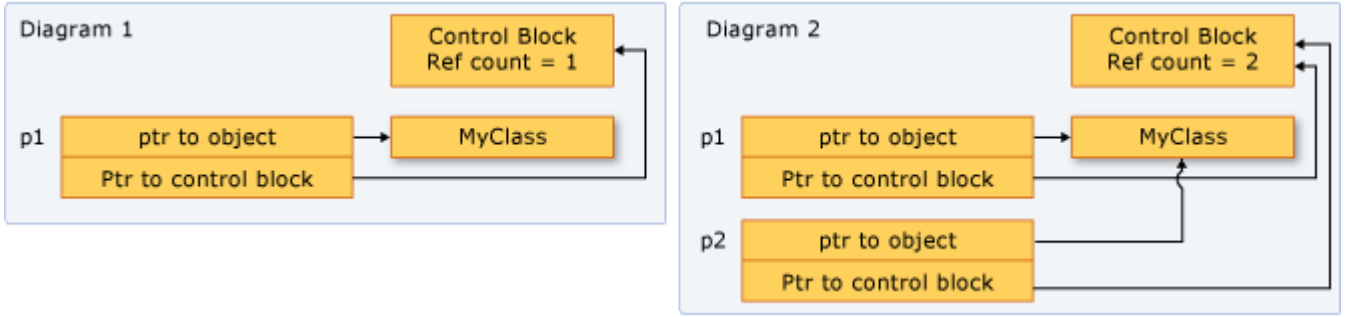
When working with pointers, people often forget to call malloc.

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

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

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* pointer pt, you write pt→name;

With a shared_ptr<foo> you use the identical notation!

MEMORY LEAK

Definition: A program that allocates objects using “new” or “malloc” but neglects to free them.

The memory is consumed, but never released, so the heap gets larger and larger.

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.

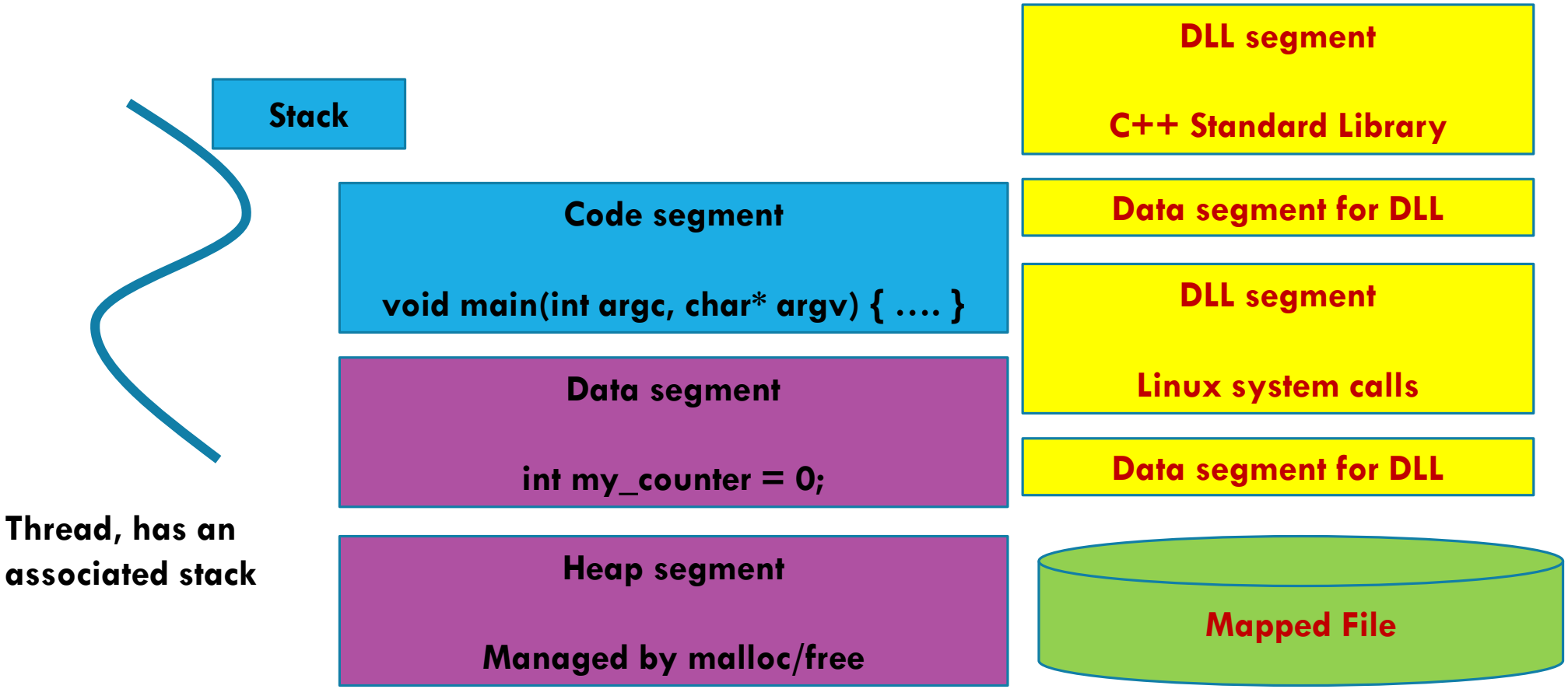
WHICH SEGMENTS HOLD WHICH KINDS OF MEMORY?

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



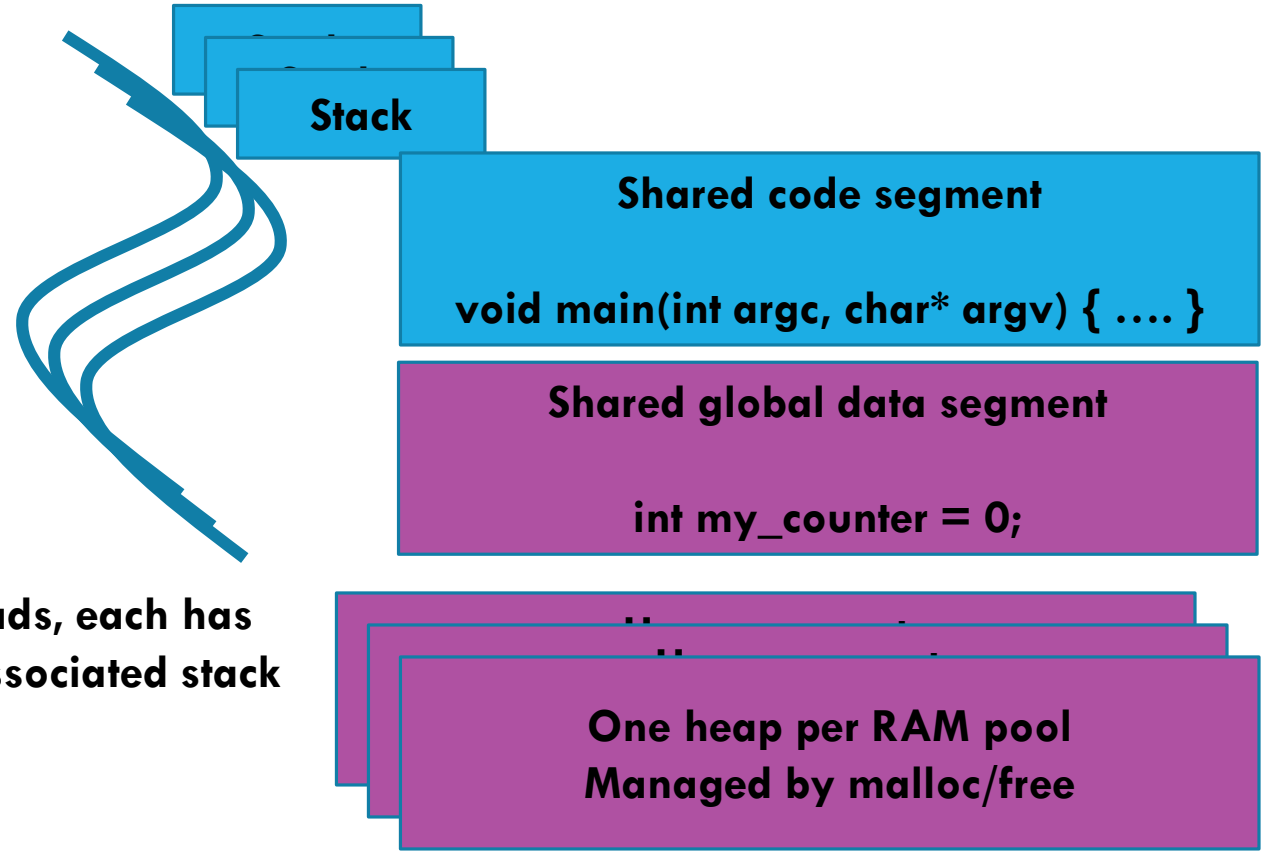
DIFFERENT THREADS IN ONE PROCESS SHARE THE SAME ADDRESS SPACE

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.

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!)