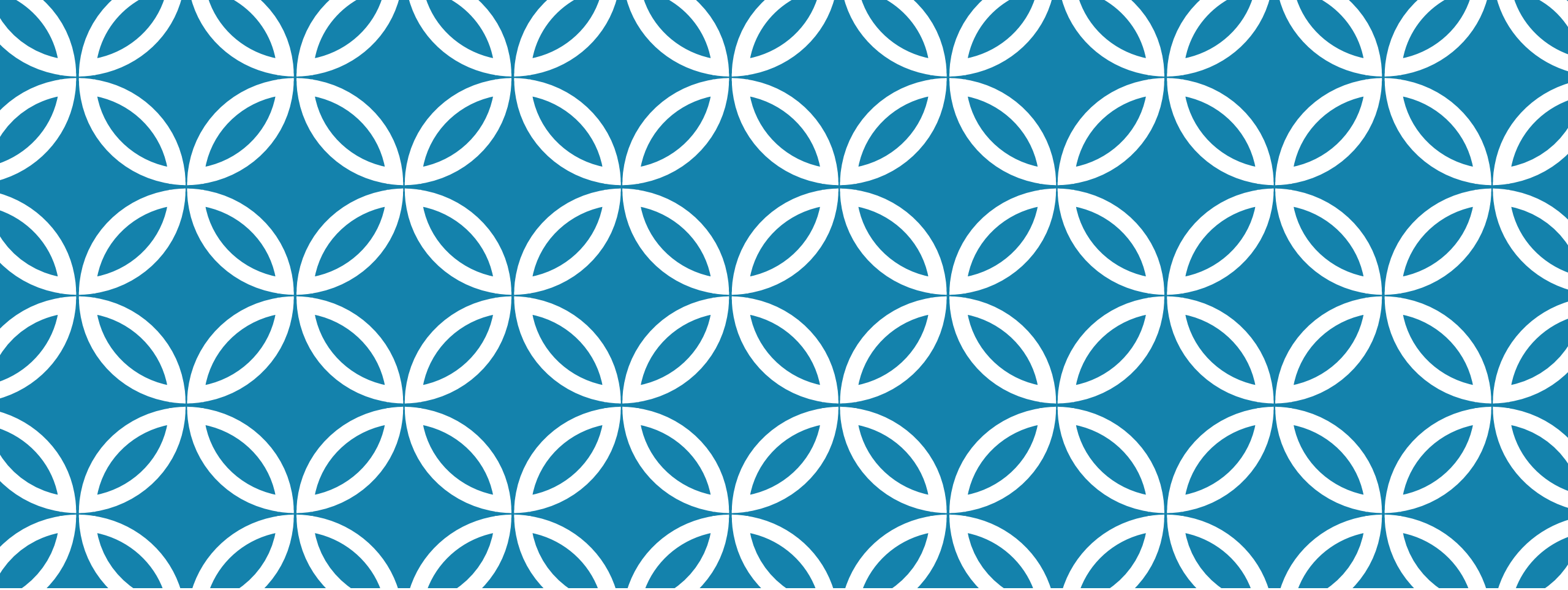


MEMORY MANAGEMENT

Professor Ken Birman

CS4414/5416 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

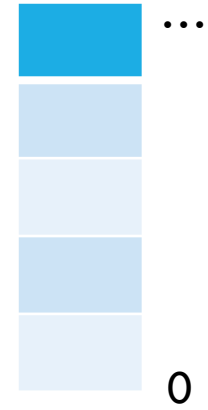
A LINUX PROCESS IS CREATED USING FORK AND EXECL (FROM BASH, THOSE ARE AUTOMATIC).

A parent process (which is often the bash shell) calls **fork**

- This clones the address space of the parent (+ file descriptors, user-id, current folder, etc.)
- Clone adjusts std::cin and std::cout as needed.
- Then via **execl**, the address space is "replaced" with a new one that has a code & data segment at address 0x400000 and new heap and stack segments
- Linux loads the binary image of the new process, then jumps to address **0x400000.**
- The new process initially doesn't have any dynamically loaded libraries (DLLs). Calls to DLLs jump to a loader method via an indirection table. The first call loads the DLL

Note that bash can also run shell scripts "as if" they were processes.

ADDRESS SPACE?



Every program runs in an isolated virtual address context.

All 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. The same physical page can be mapped into multiple page tables.

… 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.

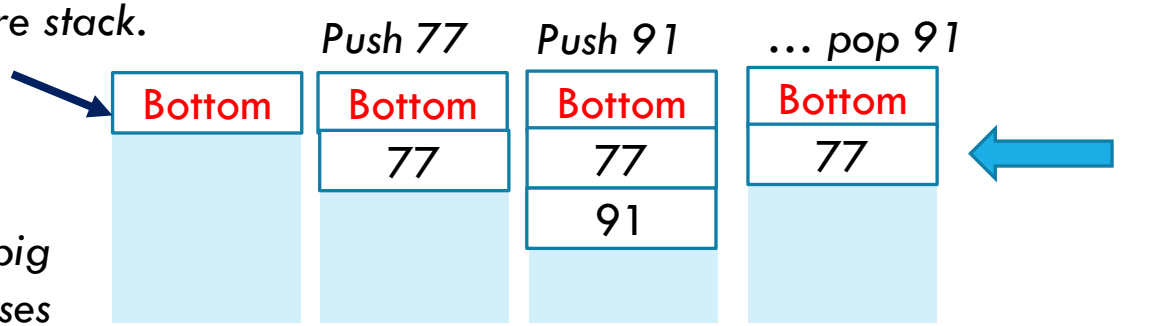
The stack and the heap are two resources for doing this.

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.

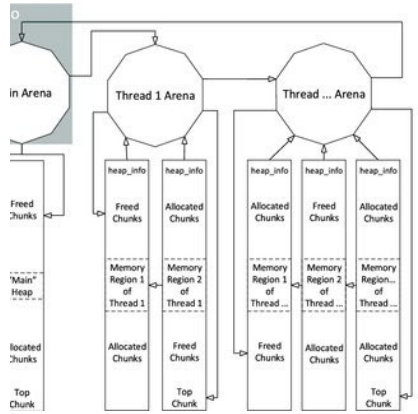
Stack segment encloses the entire stack. But not all of it is "in use"



Stack pointer points to the top element

In Linux, stacks always grow from big addresses down towards small addresses

… HEAPS



A heap is a memory region allocated via **malloc(size)/free.** Access the memory via *pointers*. Use a "static cast" to tell C++ what type of data your region will hold!

The C++ operators include the * operator, as in *ptr = 17, and the → operator (typed as –>). * is used for vectors or arrays. → is used if your memory region holds a data structure of some kind.

WHAT ABOUT OBJECTS IN THE HEAP?

In C++ all objects need to live somewhere, and the heap is a very common choice.

The C++ keyword **new** will automatically do the malloc and then call the object constructor.

When an object goes out of scope, its destructor will run.

INITIALIZATION IS VERY IMPORTANT!

Malloc doesn't zero or initialize the region. In C++ we normally use objects with constructors that initialize the fields to desired values: **RAII** (resource allocation is initialization).

For this reason, new memory won't be automatically zeroed: that would be wasted work. A program's data segment is initially zero, but that is really a special case.

Of course you can always zero a memory region "by hand". Use calloc tor bzero.

YOUR C++ PROGRAM IMPLICITLY CONTROLS WHERE VARIABLES RESIDE

```
// This is typical of what you might find in a .hpp file
// The example describes a class called Cat, with some fields and methods
int     cat_count;                          // 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
};
```

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

```
//  In the .cpp file, we give these values

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

Cat::Cat(std::string& given_name) {  // Constructor to initialize a new Cat instance
            Cat::name = given_name;
            Cat::cat_id = cat_count++;
      }
}
```

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

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

external int     cat_count;

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. But when the Cat object was created the string length was not yet known! In fact a Cat has a std::string object within it. And this std::string counts bytes in the name and then calls malloc to create space for a copy of it.

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

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

Where is the string itself, in memory?
You can access it as obj.c_str()…

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.

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. Const says that **sum** won't modify this argument.

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!

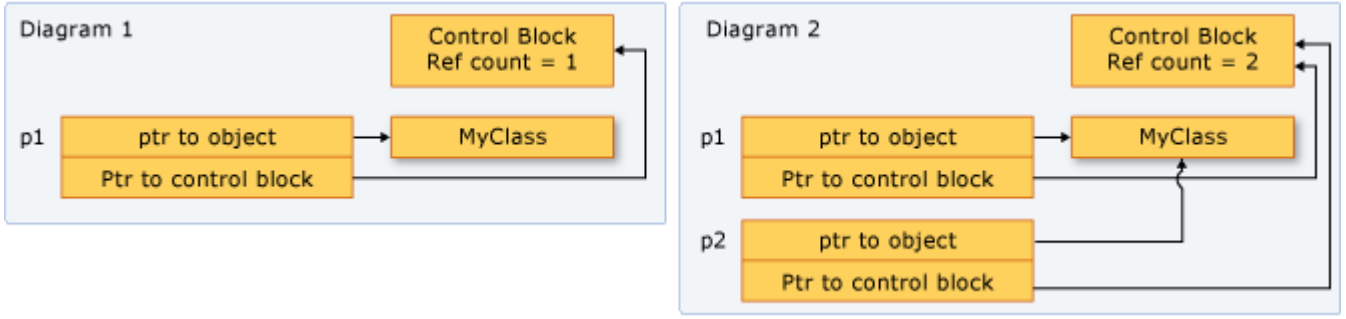
And a reference or pointer into the heap is only valid as long as you haven’t deleted the object in the heap that it points to!

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



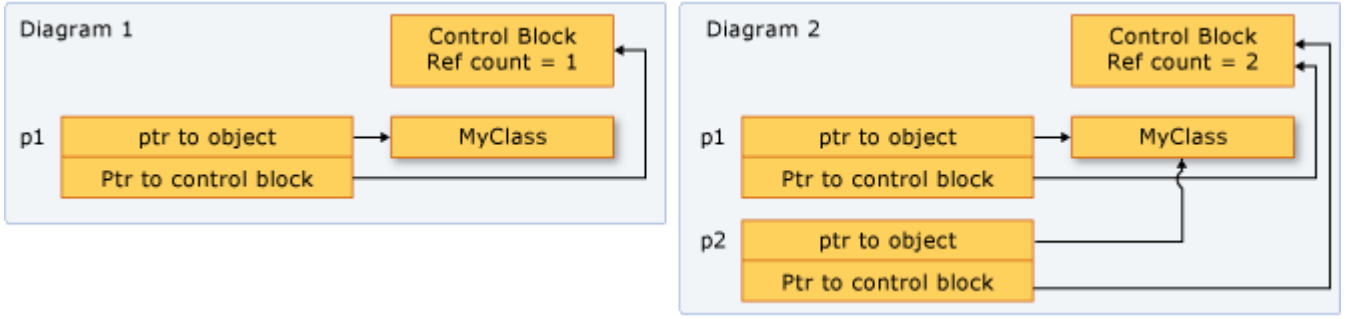
Professional C++ developers prefer not to use pointers directly. We "wrap" them in a shared_ptr template.

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.

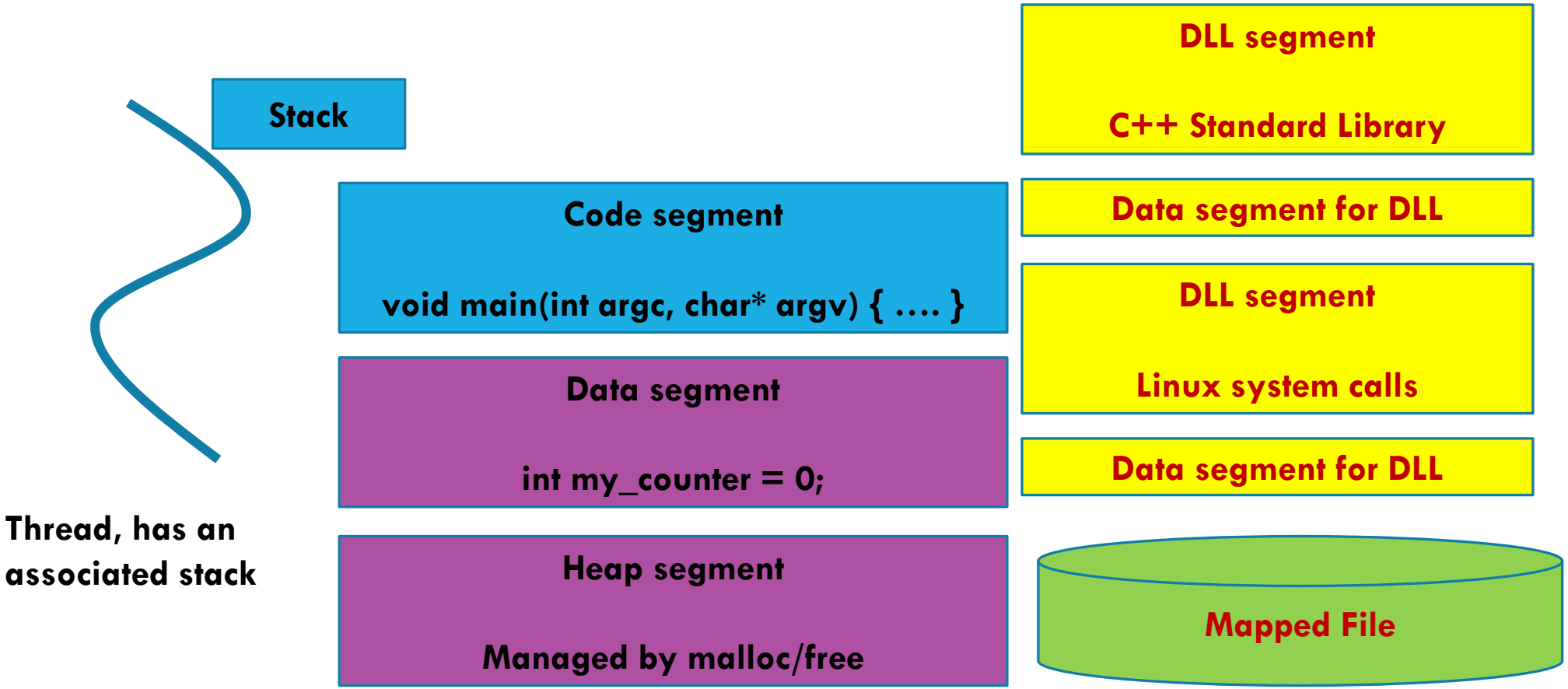
WHICH SEGMENTS HOLD WHICH KINDS OF MEMORY?

Let's tour the computer from the hardware "up".

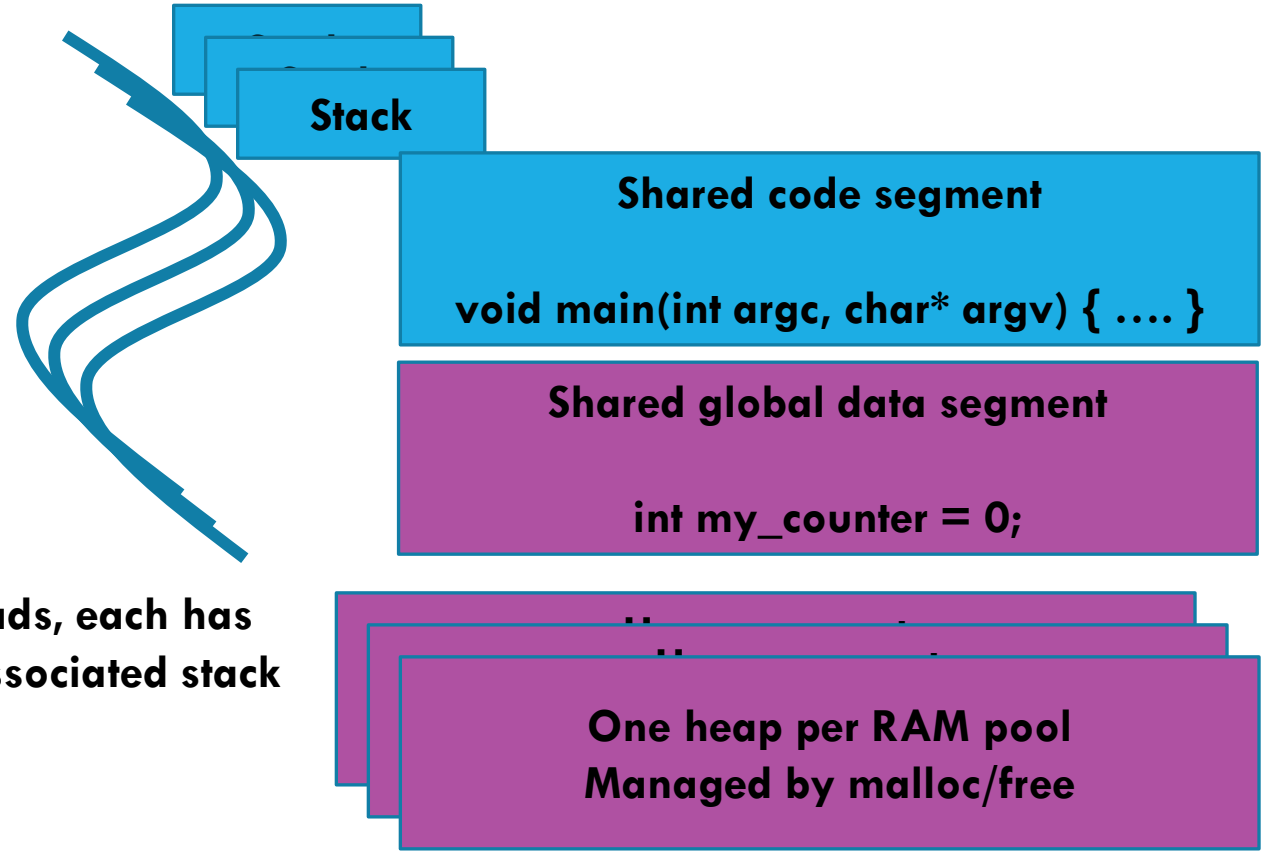
The NUMA computer has a big memory region that encompasses all memory on the machine. Any thread with permission can access any part of this memory (local memory is cheapest).

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

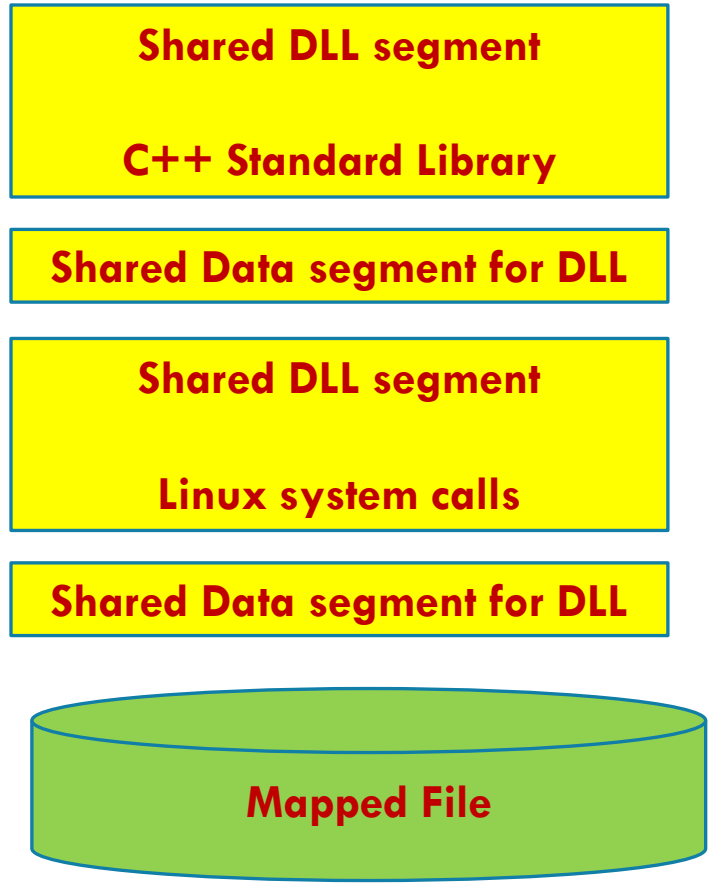
VISUALIZING AN ACTIVE PROCESS



VISUALIZING AN ACTIVE PROCESS



Threads, each has an associated stack



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.

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"

MAPPED FILES

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

Mapped files can be used for sharing between processes (particularly helpful across programming languages!).

- Updates are written to disk… *limited to one writer.*
- For pure sharing, use **shmget.** Same idea, but no file I/O.

… USED TO IMPLEMENT DLLS

Either when the program is launched (DLL methods "needed immediately"), or on the first time a method in a DLL is called,

- The DLL file is mapped via **mmap**
- The data segment used by the DLL is cloned
- Calls into the DLL are "relocated"
  - Note that DLL base addresses can vary! Each process mmaps the DLL independently
  - For this reason, we compile DDLs as "position independent code" and then "relocate" the method calls by literally editing the calling instructions, or by calling indirectly through a table of pointers that we build after mmapping the DLL into memory.

SOME SEGMENTS GROW DYNAMICALLY

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.

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!)

SELF TEST

What are all the segment types used by Linux?

Why might a program map a file rather than just reading it?

Why might two programs want to use shrm_get? Describe a concrete reason and list as many benefits as you can.

shrm_get won't necessarily map a segment at the same place in all processes. Why could this matter? What would you do about it?

Suppose it was your job to implement the DLL concept. Which memory features would you use?

Some of these will require a bit of research – "Claude" from Anthropic is helpful.

SELF-TEST

There is a famous worst case for virtual memory thrashing: it arises when every page reference requires that one page be paged out, and one other page to be loaded!

Design a program that will trigger this behavior. Hint: read about the **limit** command. You will need one number from it!

SELF-TEST

Imagine some DLL shared by many processes.

We remarked that Linux makes a personal copy of the data segment of the DLL for each. **Why is it doing this?**

The methods are automatically called "indirectly". Why can't we just assume the DLL is always loaded at the same virtual address?

SELF-TEST

You've developed the world's fasted LLM! Your boss is thrilled.

… but when the solution is handed off to the operational people who deploy and monitor running solutions, it slows down dramatically.

What would be your first hypothesis for the issue? How could the deployment team investigate to see if your guess is right?

SELF-TEST

Your program uses a number of libraries, but the executable is small, because it links to them as DLLs.

Now the program will be deployed on a new server. The deployment team copies the binary to the server, but it instantly crashes. Then they recompile from source code and now it works!

Why might this happen with DLLs?

SELF-TEST

An important program has been working perfectly.

One day the deployment team reports that a recent update replaced a few files with newer versions, and it stopped working. It loads and runs but hangs. Recompiling didn't help. Now the main company web site is unavailable. This is an emergency and needs to be fixed **now**!

What would be a possible cause, and what quick solutions can you suggest? There isn't enough time to modify the program source code.

SELF-TEST

Several processes in a gaming application running on the same server share a data structure through a mapped file. One of them modifies it, and the others do purely readonly access.

In the past, this file was needed for offline debugging, but now the program is finished and your boss wants it to be as fast as possible.

What overheads could be arising and how can they be eliminated?

C++ BY-REFERENCE SELF-TEST…

Fluffy had a kitten! Assume that a Cat object has a field std::vector<Kitten> litter; Tonight the family wants to put an unnamed kitten object into Fluffy's litter. But they will also keep the kitten object in scope to use it later (for example, k.name = "Fuzzy", issued in the method that called AddKitten).	

void Cat::AddKitten(Kitten k) { // As a method in Cat
  litter.pushback(k);
}
	void Cat::AddKitten(Kitten& k) { // As a method in Cat
	  litter.pushback(k);
	}

void AddKitten(Cat c, Kitten k) { // As a global method
  c.litter.pushback(k);
}
	void AddKitten(Cat& c, Kitten& k) {
	  c.litter.pushback(k);
	}

void AddKitten(Cat *c, Kitten *k) {
  c->litter.pushback(*k);
}
	void AddKitten(Cat& c, const Kitten*& k) {
	  c.litter.pushback(*k);
	}

void AddKitten(Cat& c, Kitten& k) {
  auto kprime = k;
  c.litter.pushback(&kprime);
}
	void AddKitten(const Cat& c, const Kitten& k) {
	  c.litter.pushback(new Kitten(k));
	}

Which of these won't compile? Which doesn't do as expected? Which is "best"?

Note: the two yellow highlighted ones are instance methods of the Cat class. The others are just globally scoped methods that could show up anywhere.