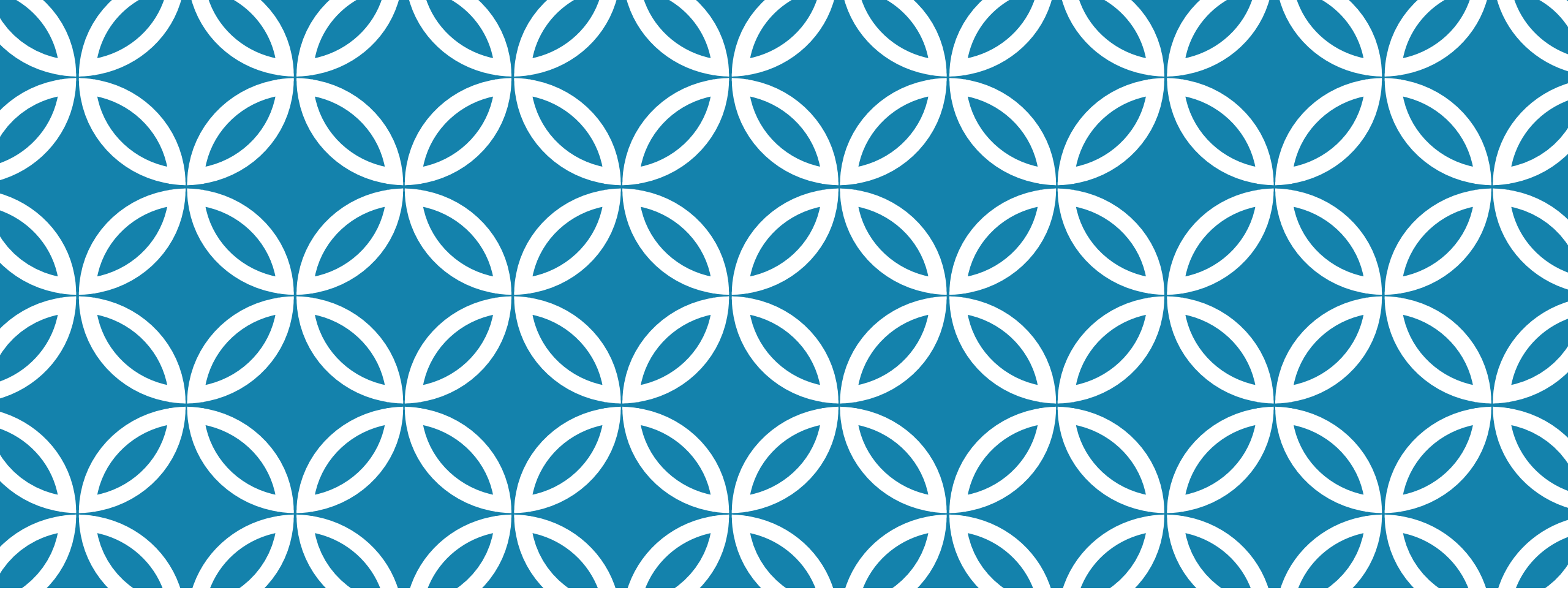


PERFORMANCE DEBUGGING

Professor Ken Birman

CS4414/5416 Lecture 15

IDEA MAP FOR TODAY

If I didn't build some application, how can I understand it's performance without spending months?

Aren't there millions of things that can shape performance?

Today will be a "big picture" lecture about the visualizing all those different elements with a purpose: not "why it does this" but rather "what shapes its speed?"

YOUR JOB? BE A DETECTIVE!

You suspect that some program isn't remotely close to peak potential execution speed… but much of the code isn't yours.

You need to be the sleuth and track down the bottleneck!

- This centers on developing a mental image of this code as it executes
- You'll need to have a theory of how fast it "could be", than search for evidence that something is slowing it down

WHAT DO WE KNOW ABOUT PERFORMANCE?

A program needs to use –O3 and "map" cleanly to the hardware

It should be wary of mismatch-related costs, like large objects being in a slow part of memory relative to code that needs to intensively access more than can fit in cache

Threads can perform poorly due to lock conflicts, "convoy" issues

Asynchronous pipelined flows via circular buffers are very effective. If a producer waits for the consumer to reply, it stalls

WHAT IS A "CONVOY EFFECT"?

In systems with some form of locking or waiting, this occurs if some component is **even a tiny bit slower than the others.**

Requests back up, and the backlog can cascade to earlier stages.

Many systems are **meta-stable**: even after the original cause is gone, the delay lingers! We've experienced this on highways: with the same traffic there can be steady fast movement or bunched-up slow motion.

PUZZLE?

In effect, you need to hunt for the “root cause” yet it may be gone by the time you can really snapshot the system.

This forces us to use a mixture of guesswork and experimentation to understand when the system is at risk of a problem, what might trigger it, and how to reduce the risks

THIS IS THE OPPOSITE OF ALGORITHMIC WORK! MORE LIKE ENGINEERING…

All the elements of a complex system are continuously active

The work they are doing is highly efficient (they aren't suffering memory access delays, or compiled poorly, or wasting time on things that could have been precomputed at compile time…)

The solution performs well at a range of scales and over a wide range of deployment settings

A GOOD DETECTIVE HAS AN OPEN MIND

You do start with a belief ("this is way too slow!"), and perhaps even a mental image of why ("I bet the program is stalling due to memory fetch delays")… but your theory could be wrong.

Sometimes the most obvious "issue" isn't the root cause – it may be a symptom of the real cause, but "upstream" from it.

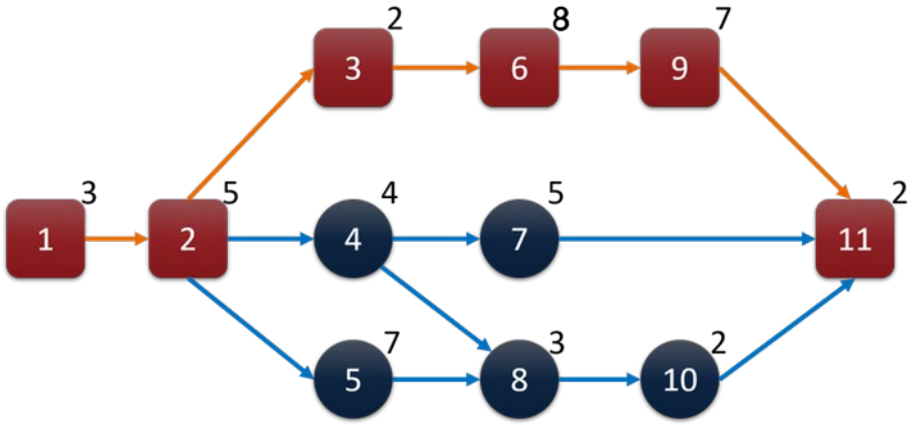
- On a highway, when you see slow cars ahead of you, *they* are probably not the cause. The cause is further up the road!

DEFINITION: CRITICAL PATH

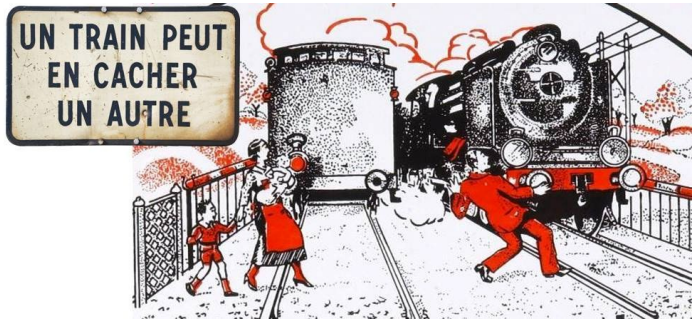
A critical path is the longest end-to-end sequence of sequentially dependent activities in an application.

This example shows 11 subtasks in some program (node numbers) each annotated by the expected delay.

The application has parallelism, yet the steps shown in red determine the critical path



FOCUS ON THIS PATH FIRST



If you can't improve it, you may be stuck…

But removing one bottleneck can expose another. Don't be dismayed if performance isn't magically solved by your first improvement

If often takes two or three interventions to get a big benefit

GOOD PERFORMANCE VERSUS BUSY WORK

One huge challenge for performance tuning is that a busy machine often isn't an optimized machine!

We can be busy for a good reason, like training a machine-learning model

But often a busy computer is "thrashing" – doing work pointlessly

IDEAL VERSUS REALITY…

Ideally, we want all the "moving parts" seamlessly interacting to provide a smooth, efficient workflow

In practice we often find that most parts of the system are bottlenecked behind some very busy but ineffective component

JUMPING TO CONCLUSIONS

It can be very tempting to rush to optimize some part of your program where you've just come up with an idea to speed it up

When you have a theory, design a validation/invalidation test to see if you are right

- For example, "add a bug" to your code that causes it to skip part of a step you think is the issue, but still keeps data consistent
- Does the application as a whole speed up as much as expected?

BIG PICTURE PROCESS

It is important to approach a systems programming challenge by really visualizing the whole task – all aspects of the solution

This includes the tasks that the operating system or network will be responsible for, and perhaps even things that other services are providing (in larger settings your programs often talk to services that run on other machines or in the cloud)

DOMAIN CROSSINGS CAN BE COSTLY

A domain crossing occurs when one element of a system talks to some totally different element of the system

This can occur in a single process, or between processes over a network, or even when fetching from a NUMA remote memory

Domain crossings often introduce waiting and costly overheads

MODERN SYSTEMS HIDE THESE COSTS

For example, via caching, prefetching, asynchronous pipelines.

These are all **great** things to do. You want all of them!

But this means that the same logic might be faster or slower depending on factors you aren't controlling. But sometimes once you understand those, you *can* control them.

TOOLS OF THE TRADE

When you approach a performance question, pause and think about this big picture, and try to visualize *all aspects*

- Where does the program really spend its time?
- Why is it spending time there? Is this what you would expect?
- Are there delaying factors that might be causing that bottleneck to actually be far less efficient than ideal?
- Can you intervene at that one spot and eliminate the issue?

ISOLATION TESTING

Used to study some component of your application. You create a dedicated specialized test to measure its speed or hunt for bugs.

You often can do this by "breaking" your application– with some special argument, main just calls the test logic, then exits.

This allows you to understand the speed of that element and to tune it, without worrying about the rest of your program.

INTERPOSITION TESTING

The Linux linker has a feature that will “reroute” calls to a method through a version you can provide. And your version can still call the original version if you like.

This allows you to do fine-grained timing measurements of just a specific method, perhaps in just a specific situation.

Slides on interposition technique details are at the end of the slide deck.

HOW FAST "SHOULD" YOUR CODE BE?

With a whiteboarding process you can often arrive at very crude estimates – rough but still very useful!

- Time needed to do the file I/O
- Computational time per "data item", and "how many items"?
- Will there be a great deal of copying needed?
- What aspects look very sequential to you?

HIERARCHY OF DELAY

Think of each part of your application in terms of

- *Bandwidth:* How fast data can be moved through it.
- *Latency:* How long it takes.

Keep in mind that the disk and Linux and the network are all parts of your application even if you didn't code those

A BUSY THING CAUSES DELAY. BUT SO DOES AN IDLE THING!

We tend to think that delay is always caused by heavy loads

This is sometimes true. If you put a storage device under heavy load, it bogs down. But this might not consume CPU time.

But often, being "overloaded" shows up as "100% idle". That component is spending all its time *waiting*, not *computing*.

CPU IS NOT ALWAYS THE ISSUE!

Sometimes we see components that are waiting for other things.

Each type of device has a minimal delay. This can grow if a backlog occurs due to overload.

- Reading from a storage device? Normally < 1 ms
- Reading over a network? Similar, but also depends on
  - …. Where the service resides
  - …. *How* you talk to it

NETWORK TYPES

The fastest networks are used in high speed clusters or data centers. Some use hardware accelerators called RDMA (remote DMA transfer over the network – ultra high bandwidth)

TCP/IP is fast in a cluster or inside a data center, but can be much slower with a wide-area link.

Terms: LAN means "local area network". WAN: "wide area".

NETWORKED SYSTEMS

A big topic in upcoming lectures… the issue is often triggered by moving data from program to program, through pipelines.

Even waiting for the receiver to acknowledge success, so that the sender can reclaim storage, can trigger a system-wide slowdown!

ML systems are especially at risk: they work with large data objects like ML models, aggregated collections of queries, etc.

ASYNCHRONOUS PIPELINING

A *huge* tool is the idea of creating a steady flow via a pipeline

We say that we have a pipeline if there is some "sender" and some "receiver", and they can both run simultaneously

Like a bucket brigade, a pipeline buffers some data (a cost), freeing sender and receiver to run in parallel

PIPELINES HIDE DELAY!

They let us request something "long before" we need it. A producer task can run faster than the consumer task.

If the data becomes available when the receiver isn't ready to process it, that data just waits in the pipeline.

And because the sender can reclaim memory as soon as the data is in the pipeline, we don't see a backlog effect.

BUT DON'T LET PIPELINES GET "TOO DEEP"

If a pipeline is holding huge amounts of data, or huge amounts of some other resources, costs accumulate

- That memory could have been useful elsewhere
- Linux limits how many files can be open all at once
- Data might even become stale, if the underlying files change

Use your analysis to select a smart pipeline size – "depth"

OFTEN WE HAVE ADEQUATE MEMORY AND PROCESSORS TO SHIFT LOGIC THIS WAY

A pipeline is just one way to use memory to speed things up. A machine has many resources… how do we keep things busy?

This is especially worrying in big ML applications because they often run on incredibly big, powerful servers with expensive GPUs

Danger? That the server sits totally idle when waiting for GPU computing to finish, wasting an incredibly valuable resource.

DON'T SWEAT THE SMALL STUFF

Start by trying to understand whether something is 10x slower than it should be.

Finding the major bottlenecks, or the very inefficient pieces of a solution, can pay off: fixing those first gives dramatic improvements… After that, you can focus on smaller things

ALGORITHMS (SOMETIMES) MATTER…

As a student you've learned a lot about algorithms

If the complexity genuinely reflects the costly resource, and we are in a situation where asymptotic costs are the bottleneck, picking the right algorithm is key.

But those two "ifs" are not minor points!

EFFICIENT ALGORITHMS DON'T ALWAYS FOCUS ON THE COSTLY RESOURCE

Many algorithms were created using standard metrics like compute time for one thread, or space consumed

In a parallel setting with a lot of memory, we might be fine with spending memory to save time – we saw examples earlier today.

And computing may actually be "cheap" too!

WHAT WOULD BE A COSTLY RESOURCE OTHER THAN MEMORY OR CPU TIME?

Idle resources

Network delays

Distributed convoy effects

Overloaded networked servers

… ALGORITHMS ARE IMPORTANT, BUT ARE ULTIMATELY TOOLS, LIKE OTHER TOOLS

When we work with algorithms we are working in a very conceptual way, highly abstracted from concrete resources. An algorithm is a **design pattern**

Our challenge as systems builders – engineers – is to map our understanding of the application into "relevant" algorithmic questions where the metrics we optimize are the costly aspects of the overall application pipeline.

MIDPOINT STRETCH: BIG PICTURE

Having the big picture is central to performance-oriented systems programming. You develop it by running experiments and using profiling tools to study the running system.

Once you identify critical paths with apparent problems, you can gain control in many ways – sometimes with our direct C++ code, but sometimes by rearranging our program in clever ways.

MIDPOINT STRETCH: BOTTLENECKS

Start by understanding bottlenecks and the critical path, and visualizing the desired flow of your computation.

You won't be able to improve performance unless you understand goals, and understand where you started.

Random changes just make code messy, add bugs, and might not help – we want to only make *the right changes*.

MIDPOINT STRETCH: BOTTLENECKS

They really come in two forms

- Unavoidable work being done as efficiently as possible
- Accidental work (or delays, perhaps even idle time) arising from some form of mismatch between our code and the system

Once you identify a bottleneck, you can often intervene to improve exactly the slow step

OUR PERFORMANCE-TUNING TOOLKIT

WHAT SORTS OF RESOURCES EXIST?

Linux has many kinds of performance-debugging tools!

- Tools that can tell us how busy various things are, like **gprof**
- Ways to watch network or file I/O, O/S system calls, like **perf**
- Fancier visualization tools to graph things over time
- Tricks to get a program to "reveal" what it is doing when it seems very slow, like inserted timing measurements

… a challenge is that the slow thing might not be the actual cause!

WE CAN'T COVER TOOL APIs IN LECTURE!

In recitation we do show you quite a few

But each different performance puzzle tends to need its own specialized way of proving or disproving your guesses

For many purposes, tracking time **by hand** is the best tool!

TIME IN C++/LINUX SYSTEMS

std::chrono::high_resolution_clock::now();

Just measure the time before and after doing something:

$$\Delta = T_{after} - T_{before}$$

For accuracy, measure 100 or 1000 iterations. Save the measurements and print data afterwards – **never print anything inside the timing test.**

But be wary of time in distributed systems. With NTP running, clocks are synchronized to within a few milliseconds – but a millisecond is a huge number, so it isn't safe to compare numbers from different clocks.

WHAT IF THE THING WE WANT TO PROFILE IS NOT "ACCESSIBLE" TO US?

Gprof sometimes isn't very useful. Yet if code is in libraries or involves complicated distributed actions, there may not be an obvious way to measure before and after times!

For this, a trick called *interposition* is very useful

It involves help from the C++ compiler and linker

WHAT IS LINKING?

Linking is a technique that allows programs to be constructed from multiple object files

Linking can happen at different times in a program's lifetime:

- Compile time (when a program is compiled): **static linking ("archives")**
- Load time (when a program is loaded into memory): **uncommon**
- Run time (while a program is executing): **dynamic linking (DLLs)**

Understanding linking can help you avoid nasty errors and make you a better programmer

ARCHIVES, DLLS

In Linux, an **archive** is a collection of files concatenated into a larger file, for convenience and speed

A **static library** is normally an archive of compiled "object files" for methods called by user code, plus a symbol table for the linker/loader to use to quickly find what it needs

A **dynamically linked library** (DLL) is similar, but the entire library is mapped into the application address space at runtime, as a segment

GETTING VERY FANCY: LIBRARY INTERPOSITIONING (FOR SERIOUS HACKERS!)

Library interpositioning: powerful linking technique that allows programmers to intercept calls to arbitrary functions.

Interpositioning can occur at:

- Compile time: When the source code is compiled
- Link time: When the relocatable object files are statically linked to form an executable object file
- Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

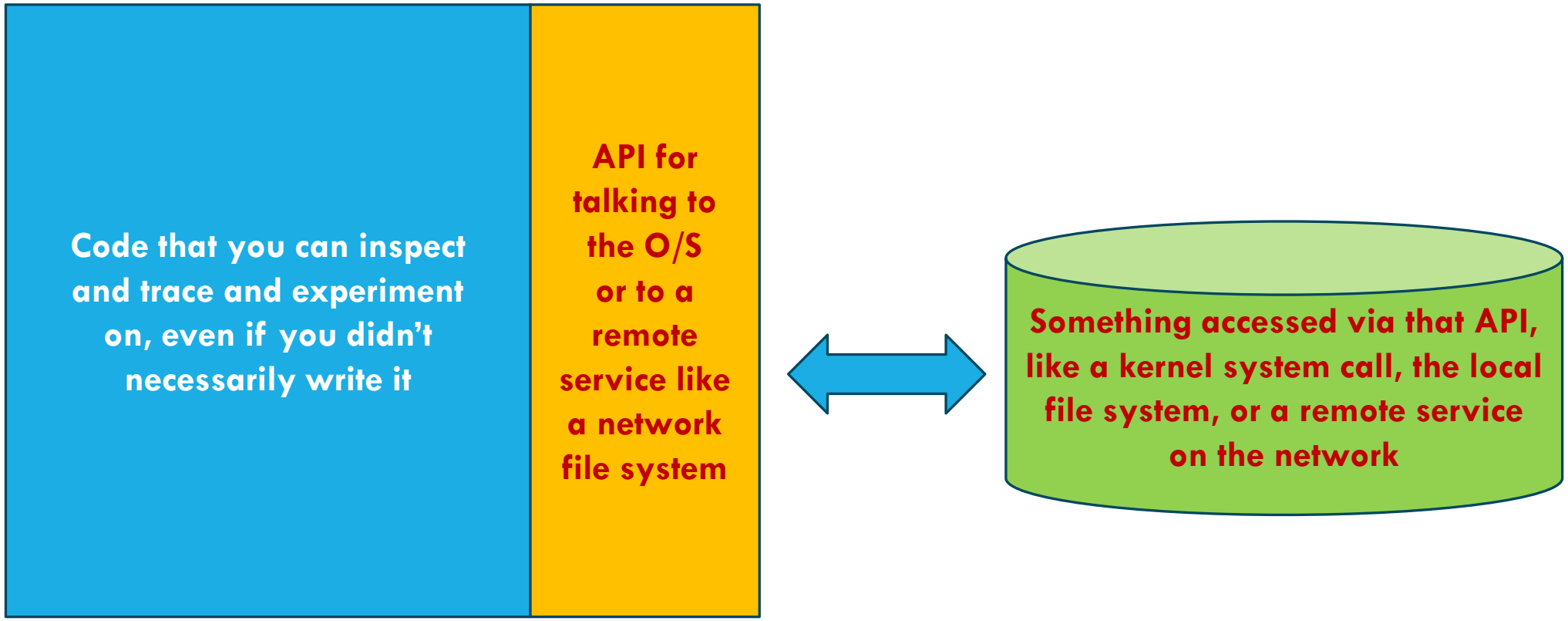
1-2-3 RECIPE FOR INTERPOSITIONING

Given an executable that obtains **something** from a library.

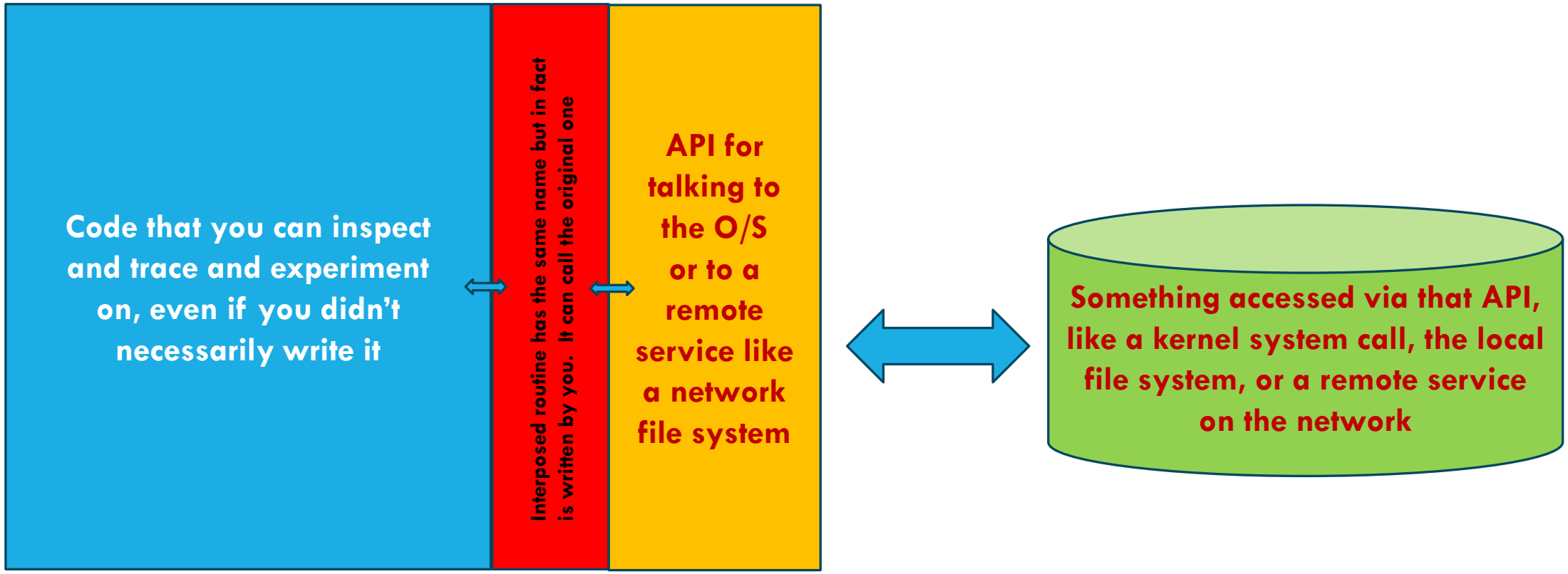
Create a .o file that defines **something**, using the same API the executable expected. Relink the executable against your .o file.

Now your implementation of **something** will be called

BEFORE AND AFTER INTERPOSITION



BEFORE AND AFTER INTERPOSITION



1-2-3 RECIPE FOR INTERPOSITIONING

… notice you can still call the original version of **something** from inside your replacement!

But you use a slightly different name.

… So, have it call **_something.** This will be undefined… claim that it is in a library

1-2-3 RECIPE FOR INTERPOSITIONING

So now we have the original executable, and it calls your version of **something,** which calls **_something.**

Create a new DLL library that defines **_something.** It calls the original **something, from the original DLL.**

Now we have "wrapped" **something**!

… SHORTCUT

This is such a valuable approach that in Linux, the C++ linker automates several steps to make it easy to do.

Eliminates the need to create the extra helper DLL.

Time permitting, I'll show you an example that wraps malloc

SOME INTERPOSITIONING APPLICATIONS

Security

- Confinement (sandboxing)
- Behind the scenes encryption

Debugging

- In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
- Code in the SPDY networking stack was writing to the wrong location
- Solved by intercepting calls to Posix write functions (write, writev, pwrite)

- Source: Facebook engineering blog post at:
- https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/

SOME INTERPOSITIONING APPLICATIONS

Monitoring and Profiling

- Count number of calls to functions
- Characterize call sites and arguments to functions
- Malloc tracing
  - Detecting memory leaks
  - Generating address traces

Changing a local resource into one accessed over a network

EXAMPLE PROGRAM

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

int main(int argc,
         char *argv[])
{
  int i;
  for (i = 1; i < argc; i++) {
    void *p =
          malloc(atoi(argv[i]));
    free(p);
  }
  return(0);
}
```

int.c

Goal: trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code.

Three solutions: interpose on the library `malloc` and `free` functions at compile time, link time, and load/run time.

We won't cover this example if we are short on time; it is not required and you won't see questions about these slides on a quiz

COMPILE-TIME INTERPOSITIONING

```c
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Time permitting

COMPILE-TIME INTERPOSITIONING

```
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
```

malloc.h

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
linux> make runc
./intc 10 100 1000
malloc(10)=0x1ba7010
free(0x1ba7010)
malloc(100)=0x1ba7030
free(0x1ba7030)
malloc(1000)=0x1ba70a0
free(0x1ba70a0)
linux>
```

Search for <malloc.h> leads to /usr/include/malloc.h

Search for <malloc.h> leads to

Time permitting

LINK-TIME INTERPOSITIONING

```c
#ifdef LINKTIME
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Time permitting

LINK-TIME INTERPOSITIONING

```
linux> make intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl \
    int.o mymalloc.o
linux> make runl
./intl 10 100 1000
malloc(10) = 0x91a010
free(0x91a010)
. . .
```

Search for <malloc.h> leads to /usr/include/malloc.h

The "-Wl" flag passes argument to linker, replacing each comma with a space.

The "--wrap,malloc" arg instructs linker to resolve references in a special way:

- Refs to malloc should be resolved as __wrap_malloc
- Refs to __real_malloc should be resolved as malloc

Time permitting

LOAD/RUN-TIME INTERPOSITIONING

```c
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get addr of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    return ptr;
}
```

Observe that we DON'T have `#include <malloc.h>`

mymalloc.c

Time permitting

LOAD/RUN-TIME INTERPOSITIONING

```
/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */}
#endif
```

mymalloc.c

Time permitting

NOTE: DON'T CALL PRINTF IN MALLOC/FREE

We going overloading malloc…

… but this means that debugging our code using a printf wouldn't work: calling anything that does a malloc can cause a recursion that wouldn't terminate!

… printf("something") turns into std::cout << "something", and this in turn creates a std::string("something"). **Program crashes.**

LOAD/RUN-TIME INTERPOSITIONING

```
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr 10 100 1000)
malloc(10) = 0x91a010
free(0x91a010)
. . .
linux>
```

Search for <malloc.h> leads to /usr/include/malloc.h

The LD_PRELOAD environment variable tells the dynamic linker to resolve unresolved refs (e.g., to malloc) by looking in mymalloc.so first.

Type into (some) shells as:

```
env LD_PRELOAD=./mymalloc.so ./intr 10 100 1000)
```

Time permitting

INTERPOSITIONING RECAP

Compile Time

- Apparent calls to **`malloc`**/**`free`** get macro-expanded into calls to **`mymalloc`**/**`myfree`**
- Simple approach. Must have access to source & recompile

Link Time

- Use linker trick to have special name resolutions
  - **`malloc`** → **`__wrap_malloc`**
  - **`__real_malloc`** → **`malloc`**

Load/Run Time

- Implement custom version of **`malloc`**/**`free`** that use dynamic linking to load library **`malloc`**/**`free`** under different names
- Can use with ANY dynamically linked binary

```
env LD_PRELOAD=./mymalloc.so gcc -c int.c)
```

Time permitting

HOW IS INTERPOSITIONING USED?

Very helpful for adding timing code to apparent bottleneck paths without really modifying the original program

You interpose your timing check at some method you know the original code calls (a method coming from a DLL!), and then your code remembers timing but calls the original method.

At the end of the run, some other interposed method prints the timing

YOU CAN EVEN USE INTERPOSITIONING TO IMPROVE A SLUGGISH APPLICATION

First, profile it to understand where bottlenecks arise

Next, identify ways to speed up the slow steps, such as “cache frequently reused objects”.

Then use interposition to replace some existing calls to things like file read and write with improved logic that leverages your idea

SUMMARY

Performance tuning is an art and can be incredibly satisfying

The trick is to build a mental picture of what is going on, but also what "should" be happening and how the actual system departs from expectations

Performance tuning rewards "hacking" – taking the code into a sandbox and running it in isolation to test portions of it.

SUMMARY

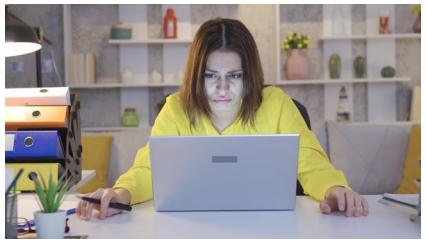
Performance tuning is an art and can be incredibly satisfying

The trick is to build a mental picture of what is going on, but also what "should" be happening and where it departs from expectations

Alicia and Jamal and Shouxu prefer the term "microbenchmarking". It seems more scientific!

Performance tuning rewards "hacking" – taking the code into a sandbox and running it in isolation to test portions of it.

SELF-TEST QUESTIONS



All of us have experienced frustrating delays on web pages.

Put on your deerstalker hat and see how many possible factors you can think of that might cause delay when interacting with a web site like Amazon.com

Now… same question, but what if instead of Amazon.com, you are asking an ML like Claude or ChatGPT some sort of question?

SELF-TEST QUESTIONS

Thinking about programs you have worked on, identify some examples of performance issues that:

- Caused the program to be very slow yet the problem could be solved quickly
- Involved a lot of delays that could be eliminated using some form of pipelining
- Involved a program/hardware mismatch: with the same code but on some other platform, it wouldn't be at all slow

SELF-TEST QUESTIONS

Why might a profiling tool like gprof give confusing results for timing of methods that read files or do network I/O?

How would you use interpositioning to take an existing program and modify it (without changing any of the existing source code) to time (instrument) those kinds of methods?

SELF-TEST QUESTIONS

Suppose that you wanted to create your own file or object caching layer.

How could you use interpositioning to get some existing legacy application to use your layer, but without changing the existing code base?

Why is this often preferable to changing the code base, if the legacy application is open source?