

CS 4414: Recitation 12

Sagar Jha



Today: Quiz
review and
general summary
of the course

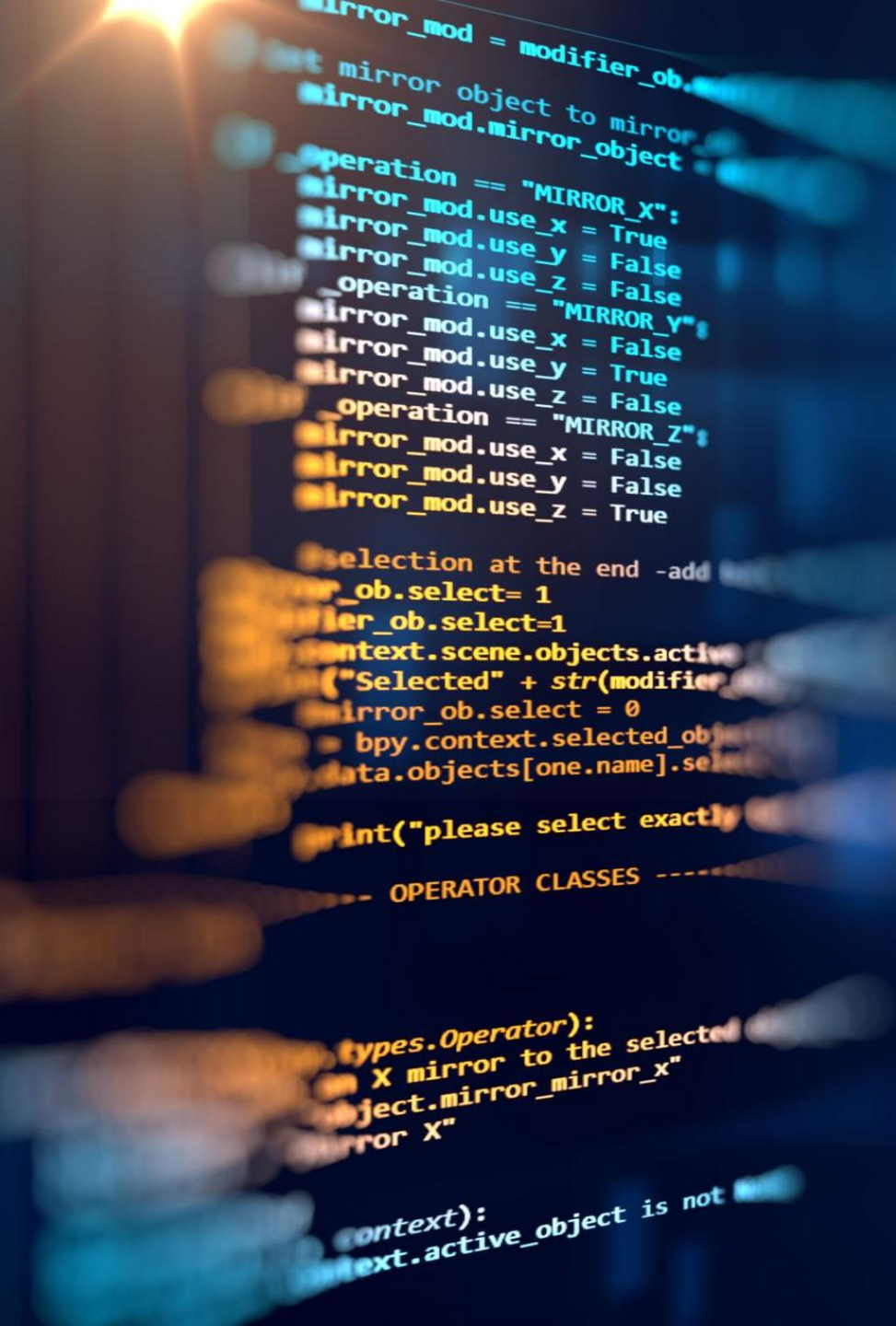
Goal of Systems Programming

- Write code that performs well
- It's not only about algorithm complexity. Why?

```
mirror_mod = modifier_ob.  
#set mirror object to mirror  
mirror_mod.mirror_object =  
#operation == "MIRROR_X":  
mirror_mod.use_x = True  
mirror_mod.use_y = False  
mirror_mod.use_z = False  
#operation == "MIRROR_Y":  
mirror_mod.use_x = False  
mirror_mod.use_y = True  
mirror_mod.use_z = False  
#operation == "MIRROR_Z":  
mirror_mod.use_x = False  
mirror_mod.use_y = False  
mirror_mod.use_z = True  
  
#selection at the end -add  
mirror_ob.select= 1  
modifier_ob.select=1  
context.scene.objects.active  
= bpy.context.selected_objects[0]  
data.objects[one.name].select  
  
print("please select exactly  
one mirror")  
  
--- OPERATOR CLASSES ---  
  
class MirrorOperator(bpy.types.Operator):  
    bl_label = "Mirror to the selected  
object mirror_mirror_x"  
    bl_name = "mirror_x"  
  
    def execute(self, context):  
        if context.active_object is not None:
```

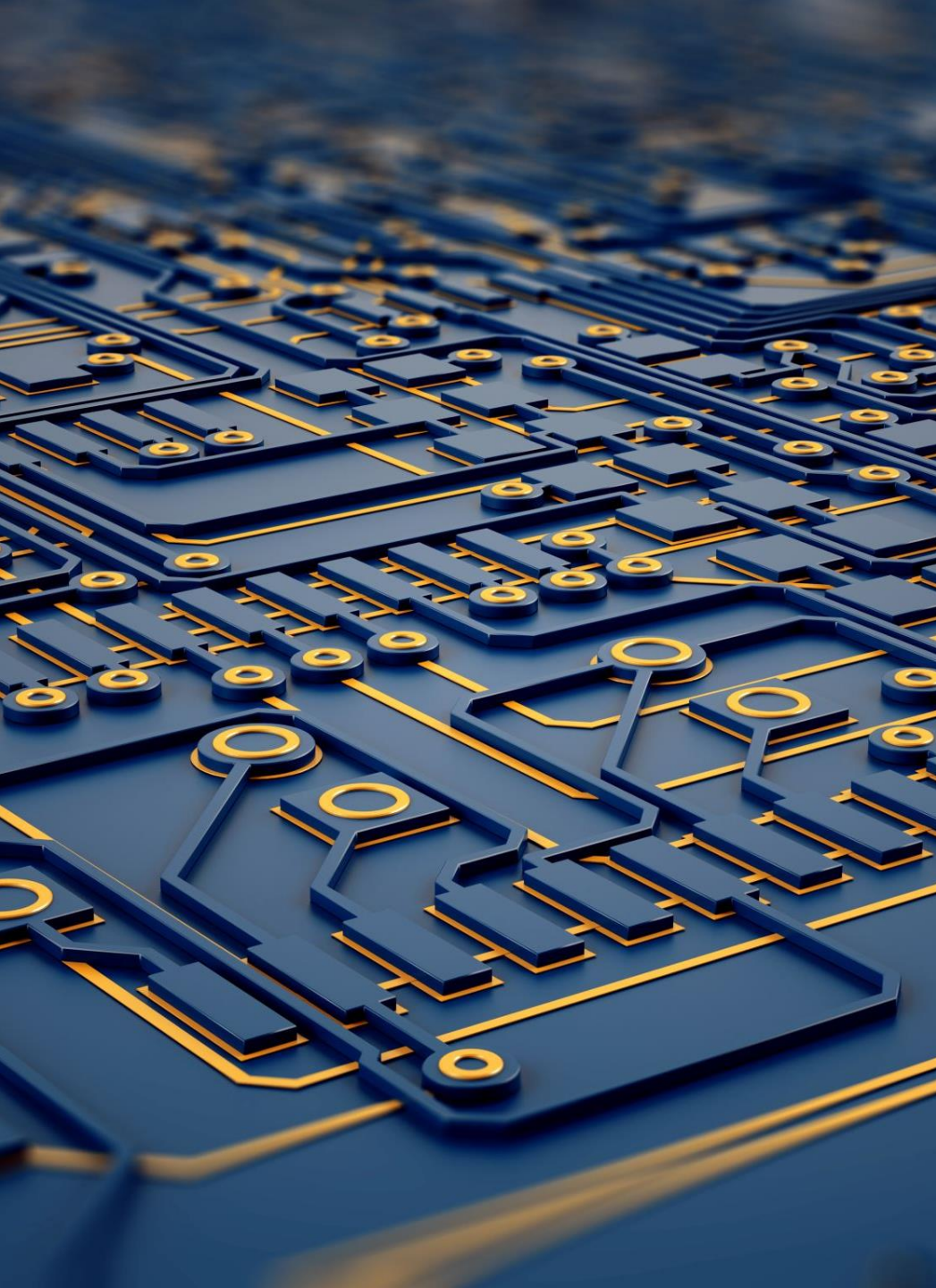
Goal of Systems Programming

- Write code that performs well
- It's not only about algorithm complexity. Why?
 - Theoretical improvements don't always translate to better application runtimes
 - Which algorithm? A system can be very complex with many features
 - What if the code that implements the algorithm is inefficient?
 - Sometimes heuristics work better
- How to measure performance?
 - Latency: Time taken to complete an operation
 - Throughput: number of operations completed per second



What does it mean to write optimal systems code?

- Design for the hardware
 - understand NUMA architecture
- Make use of programming language features
 - understand how to use C++ efficiently
- Understand how to work with the OS
 - learn Linux command line tools and how to use the filesystem
- Profile code to understand which parts to optimize
 - use gprof to understand program time distribution
- Take advantage of hardware parallelism
 - learn how to parallelize your code and coordinate efficiently among threads



I. Design for the hardware

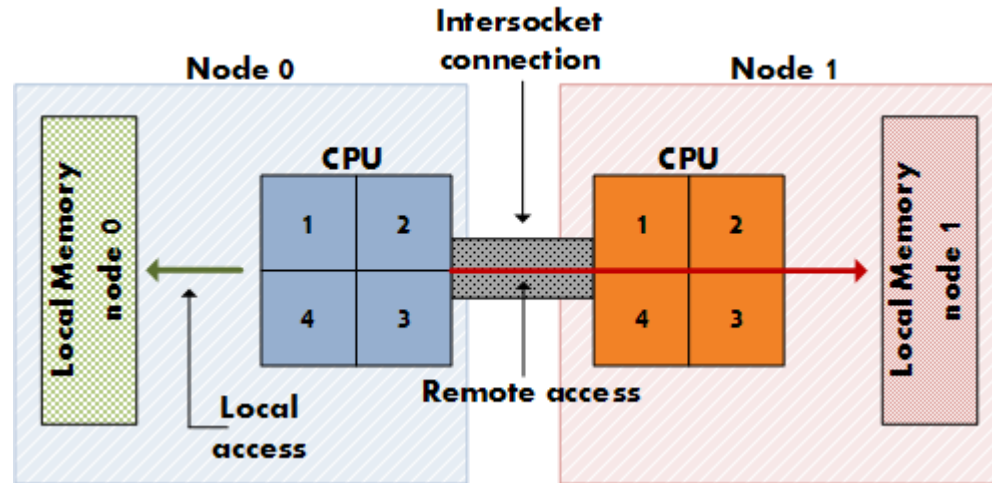
Hardware considerations

- Is it an x86 or an ARM processor?
 - Might change the compiler you use
- Are you designing for a low memory system?
- Are you designing for small number of cores? For large number of cores?

Hyperthreading

- Multiple (typically two) logical cores can run on the same physical core
- How does it work?
 - Some parts of a processor can be duplicated, while the execution unit remains the same
- What is the impact of hyperthreading?
 - Performance can increase by up to 50%, but can also decrease in some cases

Non-uniform memory access (NUMA)



- In multi-core processor architectures, a processor can access local memory faster than non-local memory (shared memory or memory local to other processors)
- In uniform memory architectures, the cost of accessing memory is higher, and increases with the number of processors

How can you find the CPU architecture information on Linux?

- Command `lscpu` gives you the information
- `lscpu` output on `compute16` on Fractus

field	value
Architecture	x86_64
CPU(s), On-line CPU(s) list	32, 0-31
Thread(s) per core	2
Core(s) per socket	8
Socket(s)	2
NUMA node(s)	2
NUMA node0 CPU(s)	0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s)	1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31

How does NUMA affect performance?

```
int counter = 0;
std::function<void()> repeated_increment =
    [&counter]() {
        for(uint32_t i = 0; i < 1000000000; ++i) {
            counter++;
        }
    };

std::thread t1(repeated_increment);
std::thread t2(repeated_increment);
t1.join();
t2.join();
```

Assignment of threads to cores	Total time taken (avg. over 10 runs)
taskset 0x3	9.04s
taskset 0x5	5.63s

Question 1: Why such a big discrepancy in performance?

How does NUMA affect performance?

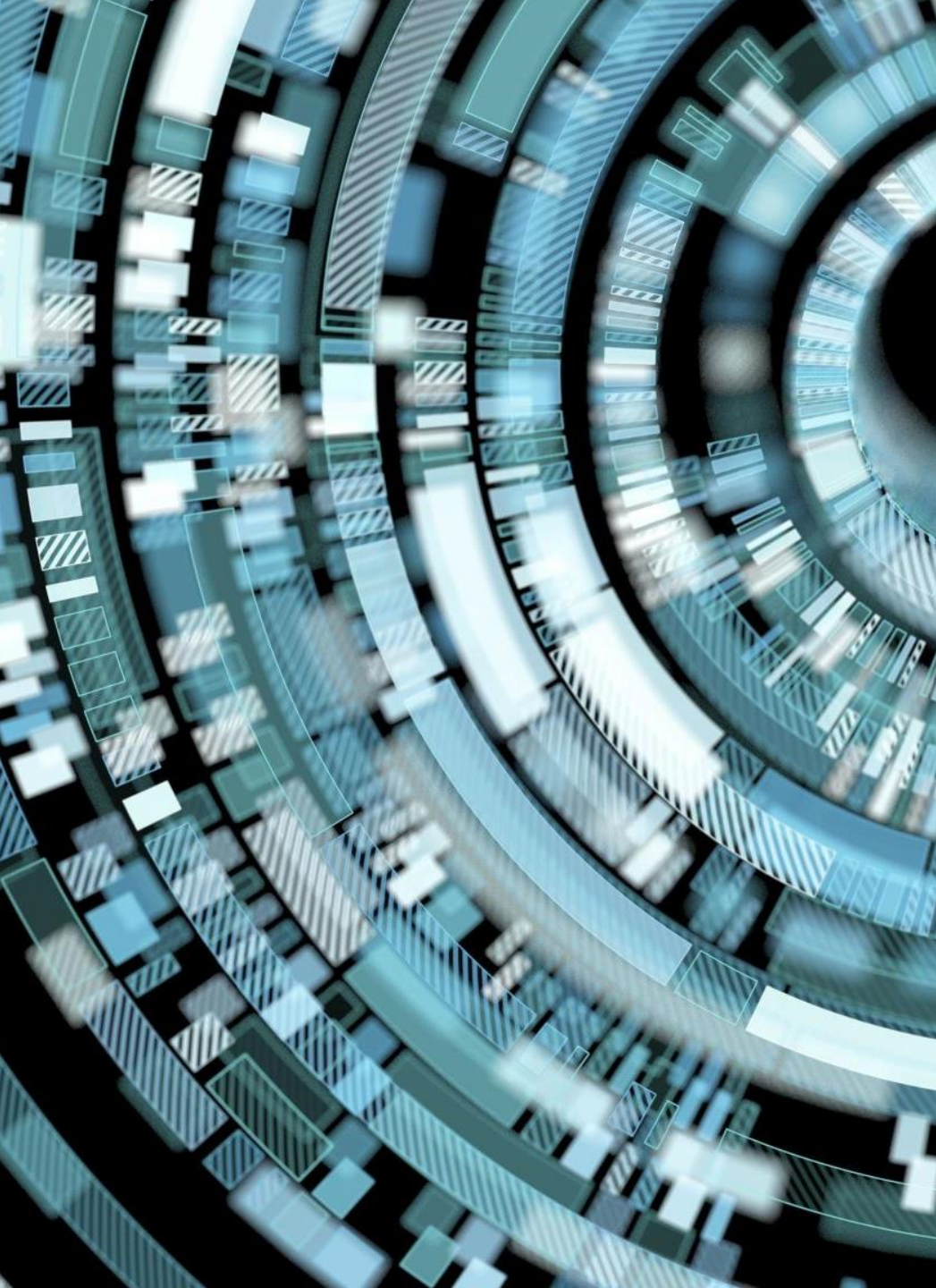
- We know accessing remote memory is slower. In what cases, will our program be forced into doing that?

How does NUMA affect performance?

- We know accessing remote memory is slower. In what cases, will our program be forced into doing that?
- If we run out of memory on the same NUMA node, the OS will allocate more memory on a remote node
- Two threads running on different NUMA nodes sharing data
- Modern systems have caches, but the behavior remains the same
 - need to access remote memory on a cache miss
 - overhead of cache coherence with memory on the remote node
- **NUMA-aware design:** Designing applications to take full advantage of the NUMA architecture

Using the taskset command

- Force the program to run on a specific set of CPU cores
- General format: `taskset <mask> <command>`
- For each CPU core i , the i^{th} bit in the mask is 1 if core i should be used, 0 otherwise
- Typically specified as a hexadecimal number
- 0x3 is 11 in binary – use CPU cores 0 and 1
- 0x5 is 101 in binary – use CPU cores 0 and 2



II. Make use of programming language features

Why use C++ for systems programming?

Why use C++ for systems programming?

- C++ is designed for systems programming
- Static type checking – memory layout of each object is known beforehand
- Code compiles down to the architecture
- C++ compilers spend significant time during compilation to improve performance at runtime. g++ is the best compiler at optimizing code
- Reduced runtime checking for maximum performance
- C++ gives you many options for each programming feature – pick and choose based on the exact need and performance requirements
- Punishes you every time you make a mistake – develops good programming habits in the long run

Learn effective usage of C++ containers

- When to use a vector, when to use a list, when to use a map...
- **std::vector** is the most important C++ data structure
- It's heavily optimized for good performance

Using std::vector effectively

```
std::vector<uint32_t> random_elements;  
uint64_t running_sum = 0;  
for(uint32_t i = 0; i < one_million; ++i) {  
    random_elements.emplace(random_elements.begin(),  
                            (running_sum + get_random_number()) % 1000);  
    running_sum += random_elements.front();  
}
```

Question 2: What's wrong with the code above?

std::vector::emplace is expensive

```
for(uint32_t i = 0; i < one_million; ++i) {  
    random_elements.push_back((running_sum + get_random_number()) % 1000);  
    running_sum += random_elements.back();  
}  
std::reverse(random_elements.begin(), random_elements.end());
```

Two improvements:

- Using `push_back` to insert elements
- Reversing the vector at the end

Question 3: Can we do even better?

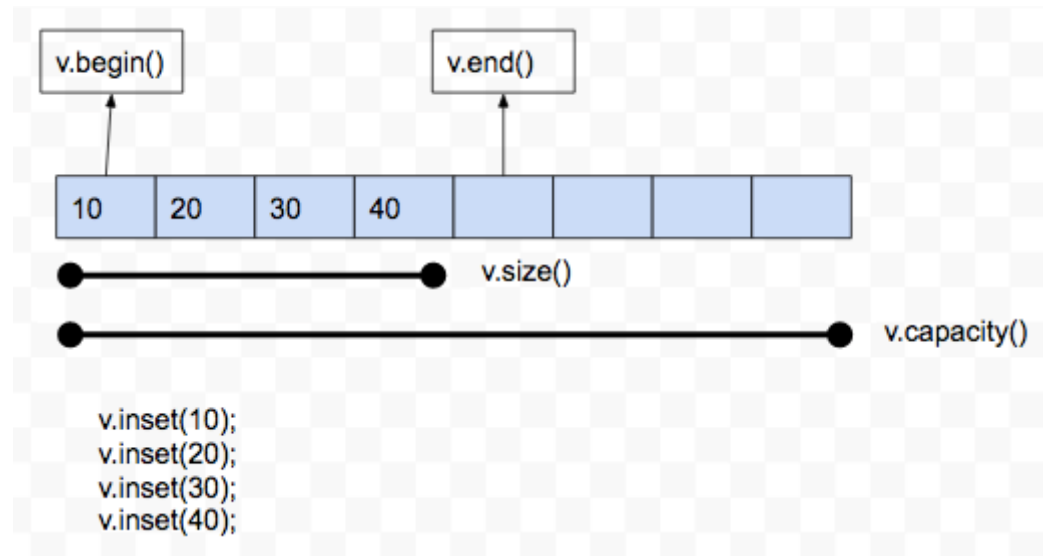
Resize the vector with the known size upfront

```
std::vector<uint32_t> random_elements(one_million);  
for(int32_t i = one_million - 1; i >= 0; --i) {  
    random_elements[i] = (running_sum + get_random_number()) % 1000;  
    running_sum += random_elements[i];  
}
```

Looking at the numbers

Approach	Runtime
emplace	55 seconds
push_back and reverse	60 milliseconds
resize upfront	30 milliseconds

Theory: How does `std::vector` work?



- `std::vector` is a collection of elements stored **contiguously** in memory
- Contiguous storage provides random access in $O(1)$ time
- `emplace` is linear because elements after the inserted position need to be moved right
- `push_back` is amortized $O(1)$
- when size equals capacity and a new element is inserted at the end, the vector needs to be reallocated and moved to a new memory location

When to use other containers?

- Use `std::list` if you need to insert in the middle given an iterator
- Traversing an `std::list` is costlier because non-contiguous storage of elements leads to worse cache behavior
- `std::map` provides deterministic $\log n$ insert and find, while `std::unordered_map` provides amortized $O(1)$
- prefer `std::map` unless certain that `std::unordered_map` will be better

What's wrong with the following code?

```
std::vector<double> recursively_process(std::vector<double> numbers,
                                       uint32_t num_times = hundred_thousand) {
    if(num_times == 0) {
        return numbers;
    }
    // modify the numbers using some math operations... finally
    return recursively_process(numbers, num_times - 1);
}

int main() {
    std::vector<double> numbers(ten_thousand);
    // initialize the elements somehow...
    recursively_process(numbers);
}
```

Question 4: Can you guess which optimization brought the runtime from 5 seconds down to 2?

Pass arguments by reference when possible

- Not applicable for very small data types (int, double, bool etc.)
- Fun fact: In my undergrad, I helped a friend in ECE with this exact problem
- What's the larger idea with passing arguments by reference?

Larger idea: Avoid copying objects

- Pass arguments by reference
- Share an object across multiple entities using `std::shared_ptr`
- Pass ownership of an object using `std::move`
- Disable copying explicitly in the class definition. E.g. My Bignum class:
 - `Bignum(const Bignum&) = delete;`
 - `Bignum& operator=(const Bignum&) = delete;`

Learn to use lambda functions

Question 5: Partition students based on their score – Below 75 and on or above 75

Given

- `template< class ForwardIt, class UnaryPredicate >`
`ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate p);`
 - Reorders elements between first and last such that elements for which **p** is true are to the left of the elements for which **p** is false
 - Returns iterator to the first element of the second partition
- `std::vector<Student> students`
- `int Student::get_score() const;`
 - returns a number between 0 and 100

Solution: encode the partition condition into a lambda function

```
std::partition(students.begin(), students.end(),  
              [] (const Student& s) {  
                  return s.get_score() < 75;  
              });
```

Metaprogramming using templates

- Generic programming: How can we write code that works with different types?
- We write a template for the code. The compiler generates the code based on the template. All types are completely defined at compile-time!
- Variadic templates: Using this, a function or class that can take variable number of template arguments

Variadic templates question

Question 6: Define a class `print_all` that takes any number of arguments of a type and calls `print` on each one of those arguments

- For example,

```
print_all (student1, student2)
```

should work as well as

```
print_all (student1, student2, student3, student4)
```

assuming `void Student::print() const` is defined

Solution: Use recursion!

```
void print_all() {  
}  
  
template <typename T, typename... Ts>  
void print_all(const T& t, const Ts&... ts) {  
    t.print();  
    print_all(ts...);  
}
```


C++'s RAI technique

```
std::mutex m;  
std::function<void()> process =  
    [&m]() {  
        m.lock();  
        somefunction();  
        m.unlock();  
    };
```

Question 7: Function process is supposed to be called from different threads in different parts of the code. It calls `std::mutex::lock` and `std::mutex::unlock` explicitly. In testing, it is found that the code deadlocks. What could possibly be happening in `somefunction`?

C++'s RAII technique

- Tie resource allocation and release to the lifetime of an object
- Works because C++ destroys an object right when it goes out of scope
- Use `std::unique_ptr` instead of raw pointers
- Use `std::unique_lock` (or variants) to lock or unlock mutexes

```
std::scoped_lock<std::mutex> lock(m);  
somefunction();
```

Operating Systems & Co.



Freemspire



III: Understand how to work with the OS

How can knowledge of the OS help write better systems code?

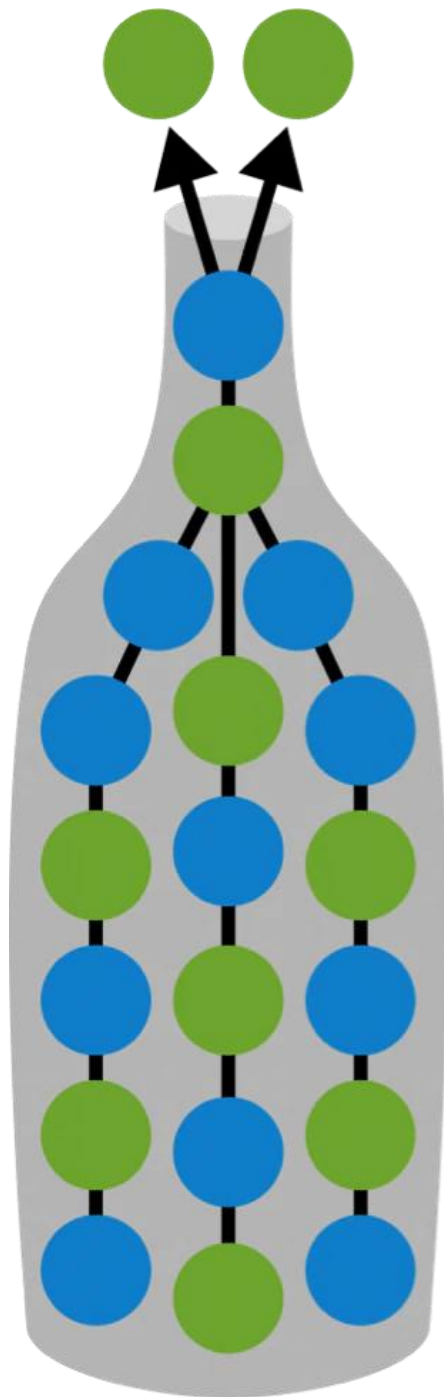
- The OS can reveal information about the hardware
 - How many cores? How many NUMA nodes?
 - Is the disk an SSD or an HDD?
 - Is a GPU available?

How can knowledge of the OS help write better systems code?

- The OS can reveal information about the hardware
 - How many cores? How many NUMA nodes?
 - Is the disk an SSD or an HDD?
 - Is a GPU available?
- You can use powerful existing libraries written for the OS
- The OS can reveal information about the resources used by your program – Task Manager in Windows or Systems Monitor (or commands `htop`, `time`...) in Linux
 - can tell you how many cores or how much memory is used by your program
- You can evaluate the performance of your code using `bash`
- Understanding the filesystem can be vital for program performance. Should you be using some other filesystem for better performance?

Why did we use Linux in the course?

- Linux is overwhelmingly popular in embedded systems (RTLinux), supercomputing clusters, mobile phones (Android), and cloud servers
- Linux is open source – It has amazing support for systems programming libraries
- Linux is generally much faster than other operating systems (Windows for e.g.)
- Linux's centralized package management system makes it seamless to install software libraries



IV: Profile code
to understand
which parts to
optimize

Bottleneck analysis: Targeted algorithmic improvements

- Profiling with gprof or by performance tests can guide the optimization process
- We made several informed decisions about what to optimize:
 - Binary search based long division was key to better performance with base 10000. It wasn't really important with base 10.
 - In my Sokoban solver implementation, BFS worked better than DFS for game-tree search because solutions were rare to find in the tree. Heuristics such as filtering out dead states had massive impact on the runtime.
 - I decided to not worry about pipelining opening files and processing them in my word count program since opening files took an insignificant amount of time.
- Understanding the workload can similarly provide great insight into systems design
 - There are entire systems dedicated to optimizing read throughput since read requests can make up to 95% of application workloads

To optimize, or not to optimize, that is the question

Question 8: A multithreaded program consists of a sequential step S followed by a perfectly parallel step P. Step S takes 5% and P takes 95% of the total runtime when the program is run with a single thread. There is an option to optimize either of the two steps by 25%. Under what conditions (number of parallel threads) would it be more beneficial to optimize S than P?



V. Take
advantage of
hardware
parallelism

Amdahl's law

$$\text{Speedup}(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Serial part of job =
1 (100%) - Parallel part

Parallel part is divided
up by N workers

- Infinite parallelism will not give infinite speedup
- Performance is limited by the cost of the sequential parts
- We saw it firsthand in word count: While the parallel file processing was dominating runtime for smaller #threads, the sequential merge step became a bottleneck with large #threads (~64)

Main takeaway

- **It helps to make your program more parallel (get rid of the sequential steps) than to optimize the parallel steps**
- That's why efficient thread synchronization is a big deal – whenever a thread holds a lock that prevents other threads from progressing, your program loses performance

Race conditions and deadlocks

- Race condition occurs when two threads access a critical section simultaneously.
- A critical section is part of the code that must be run exclusively at a given time by a single thread. It is the part where variables shared across threads are accessed safely. An `std::mutex` can be used to implement mutual exclusion.
- Deadlock occurs when the system cannot make progress.
- What is the most salient feature of a deadlock?
 - Threads (or processes in a distributed system) are waiting on each other

std::atomic vs std::mutex

Question 9: Which of the following is definitely going to be more efficient in the word-count program:

- 1. We maintain an std::set of unprocessed files. Each thread acquires a mutex and removes an element from the set. That is the next file it processes.**
- 2. We maintain a vector of all files to process. Files are processed from left to right, thus we maintain an atomic integer that stores the position of the first unprocessed file. Each thread atomically reads and increments the integer and process the file corresponding to the position read.**

When to use an `std::atomic`?

- `std::mutex` is general purpose. It provides a lot more features than an `std::atomic`. E.g., `std::lock` can lock multiple mutexes simultaneously atomically. When the mutex is already locked, the thread will be blocked instead of hogging the CPU.
- When they both meet the synchronization requirements, `std::atomic` is going to be much more efficient.
- `std::atomic` only wraps a single variable, which is often a primitive type. Must use mutex if atomicity is needed over multiple state variables.

Read/write locking pattern

- Multiple readers can read simultaneously when no one is writing. Only one writer can write at any time.
- Use an `std::shared_mutex` to achieve this pattern. For reading, acquire a shared lock on the mutex. For writing, acquire a unique lock.

Purpose of a condition variable

- When a thread needs to wait for a condition that can only be enabled by another thread, we use an `std::condition_variable`. This is combined with acquiring/releasing the mutex to guarantee mutual exclusion.
- Synchronization is on the condition, not on just waking up from the wait. There are spurious wake-ups, there may be multiple threads waiting when only few of them can proceed.
- A condition variable works only with a mutex. Use an `std::condition_variable_any` object to work with a shared mutex



VI. Miscellaneous topics


The quiz might contain

- One or two questions about security (security attacks, protection mechanisms)
- One basic question about NFS, Ceph, Zookeeper
- One question about TCP (rate control, network speed)

Good software development practices are essential!

- Develop good programming practices including those for writing efficient code
- Be comfortable in your editor, use version control for tracking developments
- Review code from time to time for quality. Refactoring code is essential to maintaining it over long periods of time
- Rewrite completely if the current version is too complicated because of too many features
- Wait, but how does it relate to systems programming?

Good software development practices are essential!

- Develop good programming practices including those for writing efficient code
- Be comfortable in your editor, use version control for tracking developments
- Review code from time to time for quality. Refactoring code is essential to maintaining it over long periods of time
- Rewrite completely if the current version is too complicated because of too many features
- Wait, but how does it relate to systems programming? 

Less time to code = More time to optimize
- Next time: We will discuss my solution to HW4 (last recitation)