Lecture 5: Independent and Cooperating Threads

Xin Liu

xl24j@fsu.edu

COP 4610 Operating Systems

Outline

- Independent Threads
- Cooperating Threads
- Race Condition
- Loss of Atomicity
- Introduction to Synchronization

Recap: Process

- An address space + at least one thread of execution
  - Address space offers protection among processes
  - Threads offer concurrency
- A fundamental unit of computation

- In Lecture 4, we learned that FIFO, Round Robin, SJF, SRTF, Priority, Multilevel Feedback Queues, Lottery, and others are designed to manage the execution order and resource allocation for multiple processes, **not** threads.

Recap: The Birth of the First Process

CPU Reset → Firmware (BIOS/UEFI) → Boot Loader (MBR, LILO/GRUB)

↓

Kernel_start()

↓

Init (Process 1)

Init is the first process and the first user-space program that the OS runs.

How does the OS locate init? The answer can be found in the Linux kernel source code:
https://elixir.bootlin.com/linux/v6.10.9/source/init/main.c#L1523

You can also type this command in the Linux terminal to check which init system your OS is using:

ls /sbin/init -l

Recap: The Birth of the First Process

CPU Reset → Firmware (BIOS/UEFI) → Boot Loader (MBR, LILO/GRUB)

↓

Kernel_start()

↓

Init (Process 1)

↓

Application Program (state machine)
+
system call

↙ ↓ ↘

Process Management | Memory Management | File Management

You can use the following system call commands to create the world:

- Process Management
    - fork, exec, and exit
- Memory Management
    - mmap –virtual address space
- File management
    - open, close, read, write
    - mkdir, link, unlink

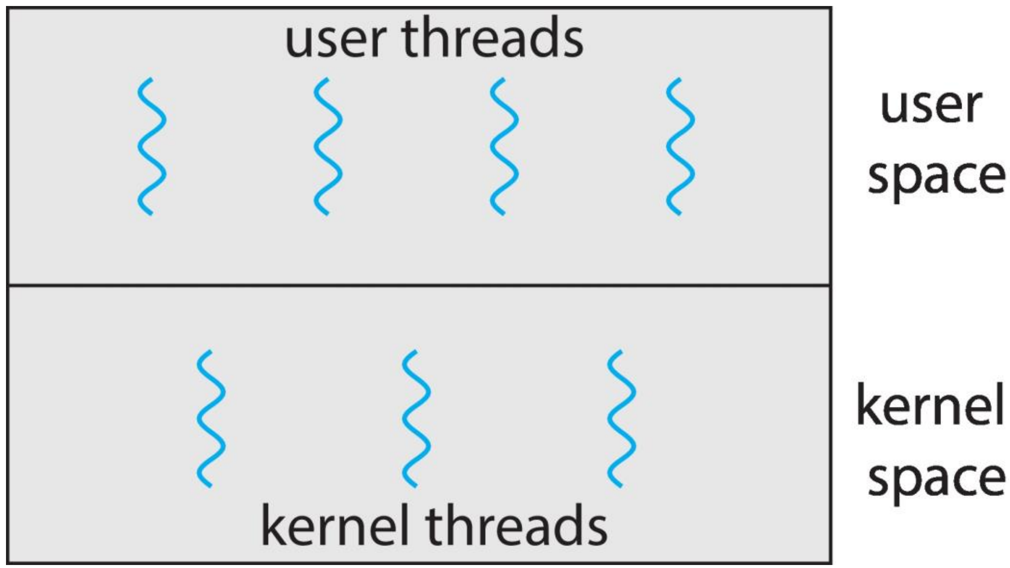
Recap: Thread

- A sequential execution stream
  - The smallest CPU scheduling unit
  - Can be programmed as if it owns the entire CPU
    - Although in actual operation, multiple threads are sharing the same CPU, the operating system uses time slicing or other scheduling mechanisms to let each thread take turns using the CPU. However, the programmer can assume that each thread has its own CPU resources, because the operating system switches threads so quickly and efficiently that it gives the impression that each thread is running independently.
    - Illusion of multiple CPUs on a single-CPU machine
  - An infinite loop within a thread won't halt the system
    - The operating system has the ability to allocate CPU resources to other threads through scheduling mechanisms, preventing a single thread from monopolizing all CPU time.

Recap: Concurrency

- Allows multiple applications to run at the same time
  - Analogy: juggling

Recap: User and Kernel Threads



Independent Threads

- No states shared with other threads
- Deterministic computation
  - Output depends solely on the input
  - Same input always produces the same output
- Reproducible
  - Output does not depend on the order and timing of other threads
  - Scheduling order does not matter

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux & Mac OS X)

- pthread.h defines the interface for pthreads

Example 1: Pthreads Basics (Cont.)

https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_5_independent_and_cooperating_threads/ex_1_pthread_basics.c

Example 1: Pthreads Basics (Cont.)

- The program utilizing pthread.h can be written to take advantage of multiple processors!

- The operating system will automatically place threads on different processors.

- You can observe the CPU usage exceeding 100%!

Concurrent Programming in HPC

- The World's Most Expensive Sofa
  - The First Supercomputer (1976)
  - Single-processor system
  - 138 million FLOPs (Floating Point Operations per Second)
    - 40 times faster than IBM 370 at the time
    - Slightly better than embedded chips today
  - Processed large data sets with one instruction

Features of HPC

"A technology that harnesses the power of supercomputers or computer clusters to solve complex problems requiring massive computation."

(IBM)

- Computation-Centric
  - System Simulation: Weather forecasting, energy, molecular biology
  - Artificial Intelligence: Neural network training
  - Mining: Pure hash computation
  - TOP 500 (https://www.top500.org/)
    - 1st: Frontier (8, 699,904 cores, 1206 PFLOS)

Main Challenges of HPC

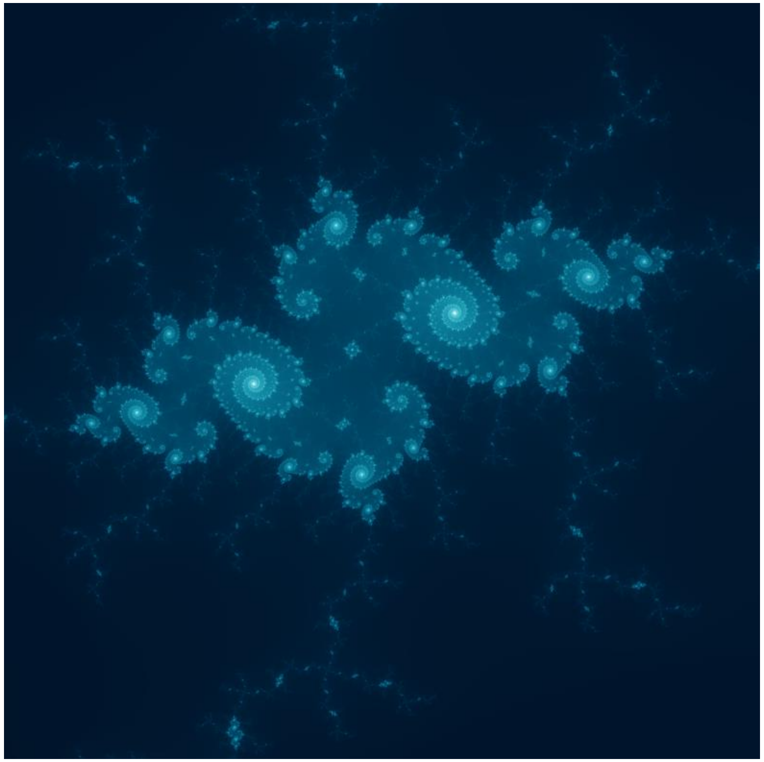
- How to Break Down Computation Tasks?
  - Computation Graphs need to be easy to parallelize
    - Task decomposition happens on two levels: machine and thread
  - Parallel and Distributed Computation: Numerical Methods

- How Do Threads Communicate?
  - Communication happens not only between nodes/threads but also with any shared memory access
  - MPI - "a specification for developers and users of message-passing libraries"
  - OpenMP - "multi-platform shared-memory parallel programming in C/C++ and Fortran“

Example 2: Mandelbrot Set

$$z_{n+1} = z_n^2 + c$$

Each point in the Mandelbrot set iterates independently and is only influenced by its complex coordinate.



https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_5_independent_and_cooperating_threads/ex_2_mandelbrot.c

Although the number of cores is not the only factor, it is indeed the most critical factor in determining thread execution efficiency. The core count allows us to estimate the system's computational capacity and parallel processing capabilities..

Cooperating Threads

- Shared states (address space -> memory)
  - The Root of All Evil
- Nondeterministic
  - Output depends on **input and other factors**
- Nonreproducible
  - Same input can produce **different outputs** in different runs
  - Influenced by factors such as thread scheduling, randomness, or external environment

Example 3: Share Memory

- https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_5_independent_and_cooperating_threads/ex_3_share_memory.c

So, Why Allow Cooperating Threads?

So, Why Allow Cooperating Threads?

- Shared resources
  - e.g., a single processor
- Speedup
  - Occurs when threads use different resources at the same times
- Modularity
  - An application can be decomposed into threads

However, Something Terrifying is Approaching..

- In a multiprocessor system, threads may execute code simultaneously.

- What will happen if two threads execute x++ at the same time?

Atomic Operations

- Atomicity refers to an operation or a sequence of operations that either completes fully or does not execute at all, without being interrupted by other operations during execution.
- It guarantees the indivisibility of an operation, ensuring that it cannot be partially completed or interrupted by another thread or process

- Key Characteristics:
  - Indivisibility: Atomic operations cannot be divided; no other thread or process can see or modify the operation's intermediate state.
  - Completeness: An atomic operation either fully succeeds and completes all its tasks, or it does not execute at all. There is no partial state.
  - No Interference: In a multi-threaded environment, atomic operations are not affected or interrupted by other threads.

Examples of Atomic Operations

- Simple Operation:
  - On most processors, an operation like int x = 1; is atomic because it involves a single memory action that cannot be interrupted.

- Non-Atomic Operation:
  - Operations like x++ involve multiple steps:
    - Read the current value of x.
    - Increment x by 1.
    - Write the new value back to x.
  - Example 4:

https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_5_independent_and_cooperating_threads/ex_4_x%2B%2B.c

Race Condition

- Race conditions occur when threads share data, and their results depend on the timing of their execution.

- If we replace the x++ operation in C with a single assembly instruction…
  - Example 5: https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_5_independent_and_cooperating_threads/ex_5_x%2B%2B_assembly.c

- The atomicity of the operation is still lost, as x++ is not a single instruction but a series of steps (load, increment, store).

Important

Using State Machines to Detect Bugs in Multithreading

- In multithreaded programs, the execution order of threads can vary, leading to different potential outcomes.
- A state machine allows us to represent all possible execution states of a program and explore how each thread interacts with the others.
- By systematically going through all possible states (execution orders), we can identify whether there are any bugs or unexpected behaviors in the program.
- This approach is particularly useful for detecting race conditions, deadlocks, and other concurrency-related issues.

Example 6: All Possible Execution Orders

- If threads share data, the final values are not as obvious

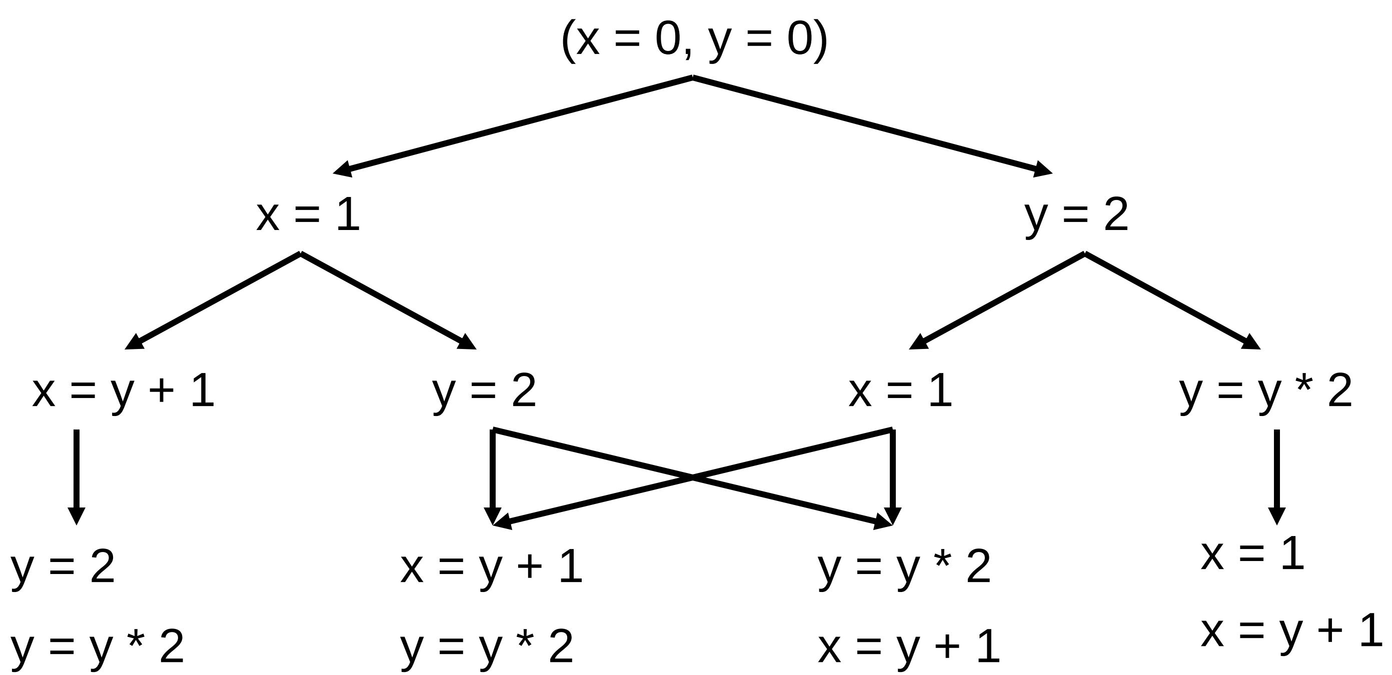
Thread A	Thread B
x = 1;	y = 2;
x = y + 1;	y = y * 2;

- What are the indivisible operations?
  - Example 6:

    https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_5_independent_and_cooperating_threads/ex_6_exec_order.c

Example 6: All Possible Execution Orders

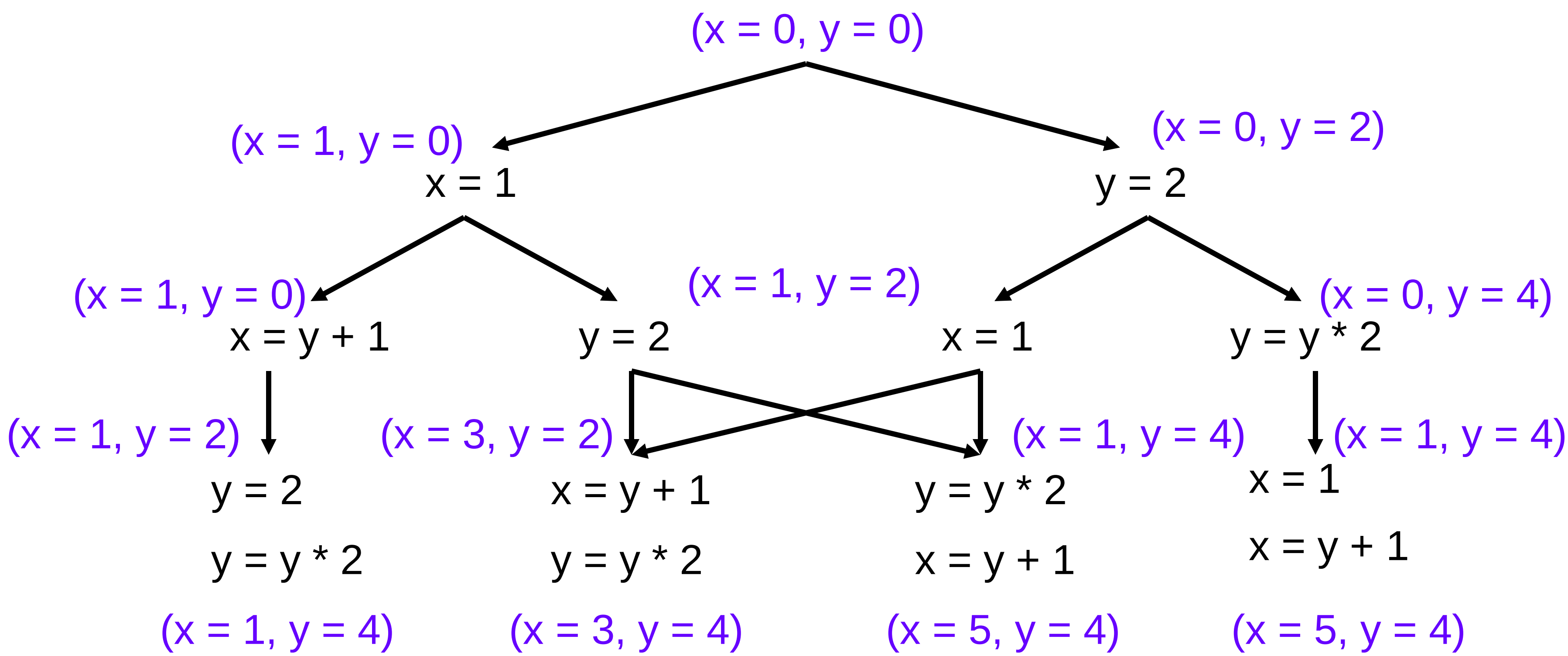
Thread A	Thread B
x = 1;	y = 2;
x = y + 1;	y = y * 2;



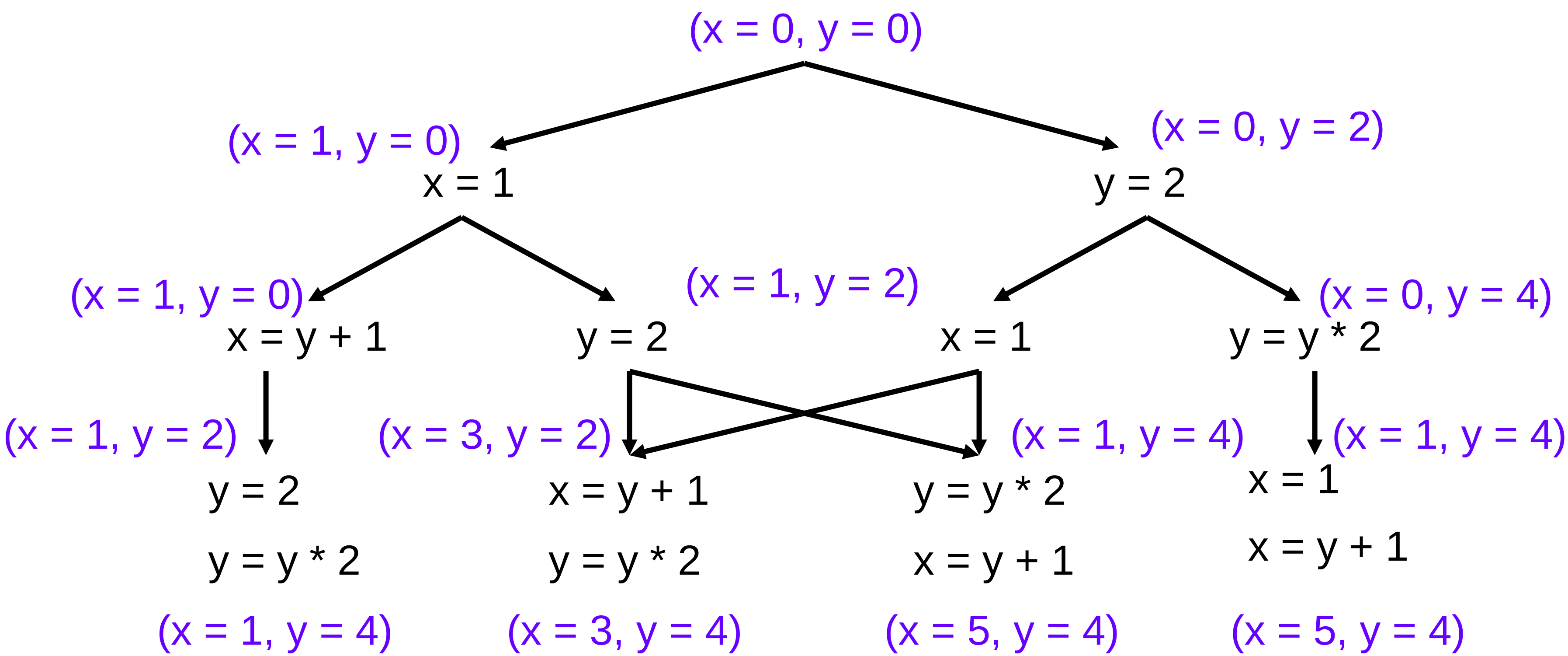
A decision tree

Example 6: All Possible Execution Orders

Thread A	Thread B
x = 1;	y = 2;
x = y + 1;	y = y * 2;



Example 6: All Possible Execution Orders



Note that some outcomes may have very low probabilities of occurring. Therefore, you might need to increase the number of iterations significantly to observe more diverse results. Even so, some outcomes may never appear due to their extremely low likelihood.

To explore all possible outcomes, more sophisticated software tools may be needed to systematically analyze and verify all possibilities.

Important

Loss of Atomicity in Modern Multiprocessor Systems

- The basic assumption that "a program (or even a single instruction) exclusively executes on the processor" no longer holds true in modern multiprocessor systems.

- Single Processor, Multithreading:
  - A thread may be interrupted and switched to another thread during execution.
- Multiprocessor, Multithreading:
  - Threads are executed truly in parallel.

- Historical Context (1960s):
  - There was a race to implement atomicity (mutual exclusion) in shared memory systems.
  - Almost all implementations were flawed until Dekker's Algorithm, which could only ensure mutual exclusion between two threads.

Concurrent Programming in Data Centers

- Google Data Center

Features of Data Center

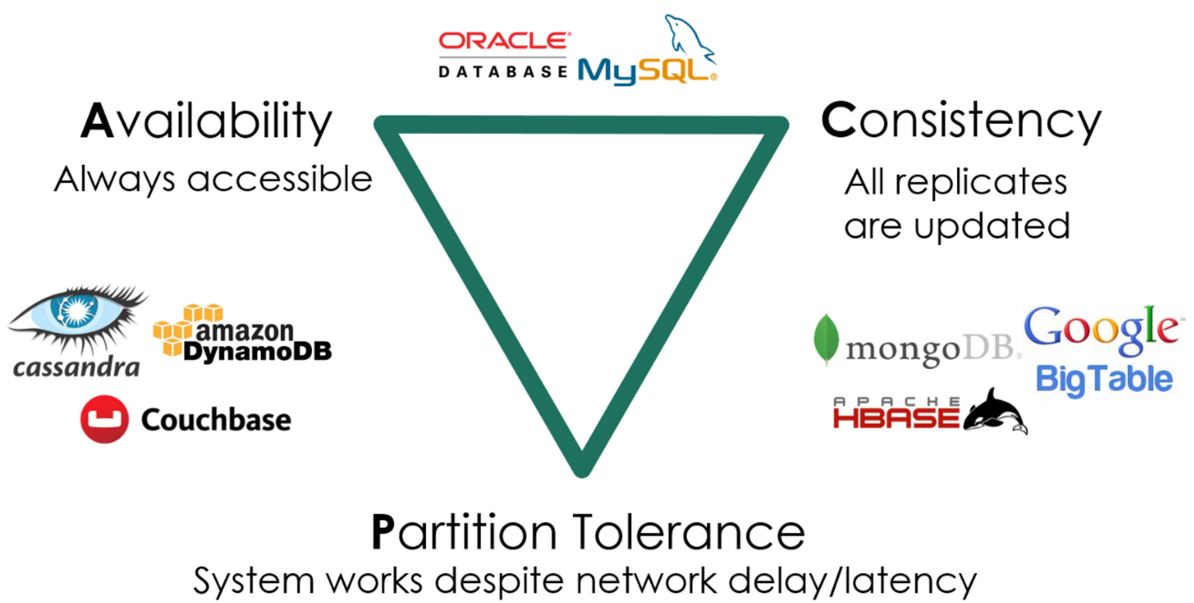
"A network of computing and storage resources that enable the delivery of shared applications and data."

(CISCO)

- Data-Centric (Storage-Focused) Approach
  - Originated from internet search (Google), social networks (Facebook/Twitter)
  - Powers various internet applications: Gaming/Cloud Storage/ WeChat/Alipay/...
- The Importance of Algorithms/Systems for HPC and Data Centers
  - You manage 1,000,000 servers
  - A 1% improvement in an algorithm or implementation can save 10,000 servers

Main Challenges of Data Center

- Highly Reliable and Low-Latency Data Access in Multi-Replica Systems
  - Serving massive, geographically distributed requests
  - Data must remain consistent (Consistency)
  - Services must always be available (Availability)
  - Must tolerate machine failures (Partition Tolerance)



How to Maximize Parallel Request Handling with a Single Machine

- Key Metrics: QPS, Tail Latency, …

- Tools We Have
  - Threads

```
thread(start = true) {
    println("${Thread.currentThread()} has run.")
}
```

  - Coroutines
    - Multiple execution flows that can be paused/resumed (M2 - libco)
    - More lightweight than threads (no system calls, thus no OS state)
  - GO
    - Threads + Coroutines

Concurrent Programming Around Us

- Web 2.0 Era (1999)
  - The Internet that connects people more closely.
  - "Users were encouraged to provide content, rather than just viewing it."
  - You can even find some traces of "Web 3.0" / Metaverse.

- What Enabled Today's Web 2.0?
  - Concurrent Programming in Browsers: Ajax (Asynchronous JavaScript + XML)
  - HTML (DOM Tree) + CSS
    - Represent everything you can see
  - JavaScript
    - Modify the page content
    - Connect local and server resources
  - You have the whole world at your fingertips!

Features of Human-Computer Interaction

- As few concurrent tasks as possible, but just enough to meet requirements.

- One thread, a global event queue, and sequential execution.

- Run-to-completion: Each task runs to completion before the next one starts.

Concurrent Programming - Real-world Applications

- High-Performance Computing
  - Focus: Task Decomposition
  - Pattern: Producer-Consumer
  - Technologies: MPI / OpenMP
- Data Centers
  - Focus: System Calls
  - Pattern: Threads-Coroutines
  - Technologies: Goroutine
- Human-Computer Interaction
  - Focus: Usability
  - Pattern: Event-Stream Graph
  - Technologies: Promise.

Motivating Example: Too Much Milk

- Two robots are programmed to maintain the milk inventory at a store…
- They are not aware of each other's presence…

Robot: Dumb

Robot: Dumber

Motivating Example: Too Much Milk

Dumb	Dumber
10:00 Look into fridge: Out of milk	

Motivating Example: Too Much Milk

Dumb

10:00 Look into fridge:

Out of milk

10:05 Head for the warehouse

Dumber

Motivating Example: Too Much Milk

Dumb

10:05 Head for the warehouse

Dumber

10:10 Look into fridge: Out of milk

Motivating Example: Too Much Milk

Dumb

Dumber

10:10 Look into fridge: Out of milk

10:15 Head for the warehouse

Motivating Example: Too Much Milk

Dumb	Dumber
	10:15 Head for the warehouse
10:20 Arrive with milk	

Motivating Example: Too Much Milk

Dumb

Dumber

10:15 Head for the warehouse

10:20 Arrive with milk

Motivating Example: Too Much Milk

Dumb		Dumber
10:20	Arrive with milk	
10:25	Go party	

Motivating Example: Too Much Milk

Dumb

10:20 Arrive with milk

10:25 Go party

Dumber

10:30 Arrive with milk: "Uh oh…"

Definitions

- ***Synchronization***: uses atomic operations to ensure cooperation among threads
- ***Mutual exclusion***: ensures one thread can do something without the interference of other threads
- ***Critical section***: a piece of code that only one thread can execute at a time

More on Critical Section

- A ***lock*** prevents a thread from doing something
  - A thread should lock before entering a critical section
  - A thread should unlock when leaving the critical section
  - A thread should wait if the critical section is locked
    - Synchronization often involves waiting

Too Much Milk: Solution 1

- Two properties:
  - Only one robot will go get milk
  - Someone should go get the milk if needed
- Basic idea of solution 1
  - Leave a note (kind of like a lock)
  - Remove the note (kind of like a unlock)
  - Don’t go get milk if the note is around (wait)

Too Much Milk: Solution 1

```
if (no milk) {
 if (no note) {
    // leave a note;
    // go get milk;
    // remove the note;
 }
}
```

Too Much Milk: Solution 1

Dumb

```
10:00 if (no milk) {
```

Dumber

Too Much Milk: Solution 1

Dumb

```
10:00 if (no milk) {
```

Dumber

```
10:01 if (no milk) {
```

Too Much Milk: Solution 1

Dumb

```
10:00 if (no milk) {
```

Dumber

```
10:01 if (no milk) {
10:02   if (no note) {
```

Too Much Milk: Solution 1

Dumb

```
10:00 if (no milk) {


10:03   if (no note) {
```

Dumber

```

10:01 if (no milk) {
10:02   if (no note) {
```

Too Much Milk: Solution 1

Dumb

```
10:00 if (no milk) {


10:03   if (no note) {
10:04     // leave a note
```

Dumber

```
10:01 if (no milk) {
10:02   if (no note) {
```

Too Much Milk: Solution 1

Dumb

```
10:03   if (no note) {
10:04     // leave a note
```

Dumber

```
10:01 if (no milk) {
10:02   if (no note) {
10:05     // leave a note
```

Too Much Milk: Solution 1

Dumb

```
10:03   if (no note) {
10:04     // leave a note

10:06     // go get milk
```

Dumber

```
10:02   if (no note) {

10:05     // leave a note
```

Too Much Milk: Solution 1

Dumb		Dumber	
10:03	`if (no note) {`		
10:04	`// leave a note`		
		10:05	`// leave a note`
10:06	`// go get milk`		
		10:07	`// go get milk`

Too Much Milk: Solution 2

- Okay…solution 1 does not work
- The notes are posted too late…
- What if both robots begin by leaving their own notes?

Too Much Milk: Solution 2

```
// leave a note;
if (no note from the other) {
 if (no milk) {
    // go get milk;
 }
}
// remove the note;
```

Too Much Milk: Solution 2

Dumb

`10:00 // leave a note`

Dumber

Too Much Milk: Solution 2

Dumb

```
10:00 // leave a note
```

Dumber

```
10:01 // leave a note
```

Too Much Milk: Solution 2

Dumb

```
10:00 // leave a note

10:02 if (no note from
 Dumber) {…}
```

Dumber

```
10:01 // leave a note
```

Too Much Milk: Solution 2

Dumb

```
10:00 // leave a note

10:02 if (no note from
 Dumber) {…}
```

Dumber

```
10:01 // leave a note

10:03 if (no note from Dumb)
  {…}
```

Too Much Milk: Solution 2

Dumb

```
10:00 // leave a note

10:02 if (no note from
  Dumber) {…}

10:04 // remove the note
```

Dumber

```
10:01 // leave a note

10:03 if (no note from Dumb)
  {…}
```

Too Much Milk: Solution 2

Dumb

```
10:00 // leave a note

10:02 if (no note from
 Dumber) {…}

10:04 // remove the note
```

Dumber

```
10:01 // leave a note

10:03 if (no note from Dumb)
  {…}
```

Too Much Milk: Solution 2

Dumb

```
10:02 if (no note from
 Dumber) {…}

10:04 // remove the note
```

Dumber

```
10:01 // leave a note

10:03 if (no note from Dumb)
  {…}

10:05 // remove the note
```

Too Much Milk: Solution 2

Dumb

```
10:02 if (no note from
 Dumber) {…}

10:04 // remove the note
```

Dumber

```
10:01 // leave a note

10:03 if (no note from Dumb)
  {…}

10:05 // remove the note
```

Too Much Milk: Solution 2

- Solution 2 does not work
- The notes are found too late…
- What if both robots wait for the other to leave a note?

Too Much Milk: Solution 3

Dumb

```
// leave Dumb’s note
while (Dumber’s note) { };
if (no milk) {
 // go get milk
}
// remove Dumb’s note
```

Dumber

```
// leave Dumber’s note
if (no Dumb’s note) {
  if (no milk) {
    // go get milk
  }
}
// remove Dumber’s note
```

Too Much Milk Solution 3

- How do we verify the correctness of a solution?
- Test arbitrary interleaving of locking and checking locks
    - In this case, leaving notes and checking notes

Dumber Challenges Dumb: Case 1

Dumb	Dumber
`// leave Dumb's note`	
`while (Dumber's note) { };`	
	`// leave Dumber's note`
`if (no milk) {`	
`  // go get milk`	
`}`	
	`if (no Dumb's note) {`
	`}`
	`// remove Dumber's note`
`// remove Dumb's note`	

Time

Dumber Challenges Dumb: Case 2

Dumb

```
// leave Dumb's note

while (Dumber's note) { };



if (no milk) {
  // go get milk
}
// remove Dumb's note
```

Dumber

```

// leave Dumber's note

if (no Dumb's note) {
}
// remove Dumber's note
```

Time

Dumber Challenges Dumb: Case 3

Dumb

```
// leave Dumb's note

while (Dumber's note) { };

if (no milk) {
  // go get milk
}
// remove Dumb's note
```

Dumber

```
// leave Dumber's note
if (no Dumb's note) {
}
// remove Dumber's note
```

Time

Dumb Challenges Dumber: Case 1

Dumb

```
// leave Dumb’s note
while (Dumber’s note) { };
if (no milk) {
}
// remove Dumb’s note
```

Dumber

```
// leave Dumber’s note
if (no Dumb’s note) {
  if (no milk) {
    // go get milk
  }
}
// remove Dumber’s note
```

Time

Dumb Challenges Dumber: Case 2

Dumb

```
// leave Dumb’s note

while (Dumber’s note) { };

if (no milk) {
  // go get milk
}
// remove Dumb’s note
```

Dumber

```
// leave Dumber’s note

if (no Dumb’s note) {
}
// remove Dumber’s note
```

Time

Dumb Challenges Dumber: Case 3

Dumb

```
// leave Dumb's note
while (Dumber's note) { };
if (no milk) {
  // go get milk
}
// remove Dumb's note
```

Dumber

```
// leave Dumber's note
if (no Dumb's note) {
}
// remove Dumber's note
```

Time

Lessons Learned

- Although it works, Solution 3 is ugly
  - Difficult to verify correctness
  - Two threads have different code
    - Difficult to generalize to N threads
  - While Dumb is waiting, it consumes CPU time (***busy waiting***)
- More elegant with higher-level primitives

lock→acquire();

if (no milk) { // go get milk }

lock→release();

Takeaways

- Multithreading Program = State Machine
  - Shared Memory
  - Non-deterministic selection of thread execution
- pthread.h
  - pthread_create
  - pthread_join
- Let Go of Your Old Understanding of "Programs"
  - Non-atomic, can reorder, not immediately visible
  - Ad hoc synchronization considered harmful (OSDI'10)
- Draw out all states to understand the program
  - Of course, doing this manually is exhausting!