Lect. 15: Real-World Concurrent Programming

Xin Liu

Florida State University

xliu15@fsu.edu

CIS 5370 Computer Security

https://xinliulab.github.io/cis5370.html

Outline

Concurrency Control and Synchronization

- Splinlock
- Producer-Consumer Problem
- Condition Variables
- Semaphores

Real-World Concurrent Programming

- World Wide Web
- High-Performance computing (HPC)
- Data Center
- AI

Concurrency Control and Synchronization

Spinlocks

- A spinlock is a simple lock where a thread constantly checks for lock availability.
- Imagine a single key to a critical section. The first thread to acquire the key can enter.
- Hardware instructions ensure atomic key exchange.

Understanding Spinlocks Thoroughly

The single atomic instruction (xchg) ensures no race condition.

```
// 0 means unlocked, 1 means locked
int lock = 0;

void acquire_lock(int *lock) {
    while (*lock != 0) { }
    *lock = 1;
}

void release_lock(int *lock) {
    *lock = 0;
}

void *foo(void *arg) {
    acquire_lock(&lock);
    // Critical section: Do work here ...
    release_lock(&lock);
    return NULL;
}
```

```
// 0 means unlocked, 1 means locked
int lock = 0;

int xchg(int *addr, int newval) {
    int result;
    asm volatile (
        "lock xchg %0, %1"
        : "+m" (*addr), "=a" (result)
        : "1" (newval)
        : "cc"
    );
    return result;
}

void acquire_lock(int *lock) {
    while (xchg(lock, 1)) { }

    }

void release_lock(int *lock) {
    xchg(lock, 0);
    }

void *foo(void *arg) {
    acquire_lock(&lock);
    // Critical section: Do work here ...
    release_lock(&lock);
    return NULL;
}
```

Rules for Acquiring a Lock

- **Grab first, verify later:** Don't bother checking if the lock is free. Just grab it and verify its status later.
- **Be fast:** Grab that lock as quickly as possible before anyone else does.

Spinlocks

Spinlocks

- A spinlock is a simple lock where a thread constantly checks for lock availability.
- Imagine a single key to a critical section. The first thread to acquire the key can enter.
- Hardware instructions ensure atomic key exchange.

Performance Issue

- Spinlocks can cause inefficiency, especially if many threads compete for the same lock, leading to frequent context switches (Grab first, verify later).
- If a thread holding the lock is swapped out, all other threads continue busy-waiting, wasting CPU resources, because the CPU still considers them active (either in the Running or Ready to Run state).

Mutexes and Futexes

Mutexes

- The lock is managed by the OS kernel.
- When a thread attempts to acquire a mutex that is already locked, the OS puts the thread to sleep (blocked state) instead of busy-waiting.
- The kernel wakes up the thread when the lock becomes available, preventing it from wasting CPU time while waiting for the lock.

Futexes

- A futex is a combination of spinlocks and mutexes.
- It starts with spinning and escalates to a kernel-based mutex when needed.
- This hybrid approach improves performance by reducing both busy-waiting in user space and context switches to the kernel.

Example: Mutex with 3 Threads (Sleep and Wake-up)

1. **Thread X** acquires the lock first and enters the critical section.
2. **Thread Y** and **Thread Z** attempt to acquire the lock but go into a sleep (blocked state) since the lock is already held by **X**.
3. Once **X** finishes and releases the lock, the OS wakes up **Y**, typically following a first-come, first-served policy (FIFO) or priority-based scheduling.
4. After Thread **Y** completes its critical section and releases the lock, the OS wakes up **Z**, which then acquires the lock.

- The waking mechanism is managed by the OS, which monitors the release of the lock and uses it as the signal to wake the next waiting thread.

The Essence of Concurrency Programming

- Collaborative relationships are a combination of Competition Relationships and Dependency Relationships

Competition Relationships

- Involves access and modification of shared resources within threads

- When threads are independent
  - The main concern is to avoid Competition Relationships
  - Use synchronization mechanisms like Spinlocks and Mutex Locks
  - Ensure only one thread accesses the shared resource at a time
  - Avoid data inconsistency and race conditions

- Focus on safe access within threads

Dependency Relationships

- Involves execution order and causal relationships between threads
- When one thread must complete before another can execute
  - Use mechanisms like Condition Variables and Semaphores
  - Control the execution order of threads
  - Satisfy logical dependency requirements
- Focus on correct coordination between threads

Real-World Application

Core Question

- How do you coordinate multiple threads to handle tasks efficiently in real-world systems?

Example: E-commerce Platform Order Processing System

- **Order Validation:** Check product inventory, user balance, and coupon validity.
- **Payment Processing:** Deduct from user accounts or process third-party payments.
- **Inventory Update:** Deduct product stock to prevent overselling.
- **Logistics Arrangement:** Generate shipping orders and arrange delivery.
- **Notify Users:** Send confirmation emails or SMS to users.

Real-World Application (Cont.)

Challenges and Solutions

- **Managing Shared Resources (Competition)**:
  - Multiple threads updating inventory or user balances may cause race conditions.
  - **Solution:** Use Mutex locks to ensure only one thread modifies shared resources at a time.
  - Also, use transactions to roll back in case of failures, ensuring data consistency.
- **Managing Dependencies Between Threads (Dependency)**:
  - Notification threads must wait until order processing is complete.
  - **Solution:** Use condition variables or semaphores to signal thread progress and control execution order.
  - Task queues can be used to arrange execution based on dependencies.

Modeling Concurrent Problems

Producer-Consumer Problem

- A fundamental synchronization problem that allows you to solve 99.9% of real-world concurrency issues.

Dining Philosophers Problem

- Another classic problem that demonstrates how multiple entities share limited resources (like CPUs).

Key Tools

Condition Variables

- A flexible synchronization primitive that allows threads to wait until a specific condition is met.

Semaphores

- A more rigid mechanism used to control access to shared resources by multiple threads.

Producer "O" and Consumer "X"

Producer:

- Produces an item ("O")
- Waits if storage is full
- Must be synchronized with the consumer

Consumer:

- Consumes an item ("X")
- Waits if no item is available
- Synchronization ensures no consumption before production

- We need to ensure that the symbols ("O" and "X") are printed in a valid sequence:
- Example:
  - $n = 3$, OOOXXOXXOOO (valid)
  - $n = 3$, OOOOXXXX, OOXXX (invalid)

- Involves two types of threads: Producers (generate data) and Consumers (process data)
- Producers don't overflow the buffer and consumers don't try to consume data that's not yet available.

Challenges:

- Synchronization and mutual exclusion
- Managing dependencies and inter-thread communication

Initial Attempt

- Ensure the condition is met using mutex locks.
  - Link of Code: **Producer-Consumer Example Code**
- Stress Testing
  - Link of Code: **Stress Test Checker Code**
  - Command: `./a.out 2 | python3 pc_checker.py 2`
- **Bad News:**
  - After running the program for several hours, it actually failed!
  - The issue is difficult to reproduce and to fix.
  - Concurrent programming is highly challenging.
- **Good News:**
  - The problem occurred while it was in your hands.
  - Avoid taking shortcuts and always stick to the most reliable methods.

Condition Variables: A Universal Synchronization Method

The Essence of Synchronization

- The essence of synchronization is ensuring that multiple threads or processes reach a **known state** at the same time, so that they can proceed in coordination.

Example:

- Imagine two people (threads) trying to meet for dinner (a task).
- One is playing a game (task A), and the other is fixing a bug (task B).
- They can't start dinner (synchronized task) until both have finished their tasks (**known state**).
- Even if one person finishes earlier, they must wait for the other.

Core Concept

- The core of synchronization is waiting for all necessary conditions to be met before proceeding together.

Synchronization Example

- From the very beginning when you started working with threads, you were already using synchronization.
- Can you find which part is synchronization?

```
pthread_t t1, t2;

pthread_create(&t1, NULL, foo, NULL);
pthread_create(&t2, NULL, foo, NULL);

pthread_join(t1, NULL);
pthread_join(t2, NULL);
```

Synchronization Example (Cont.)

- From the very beginning when you started working with threads, you were already using synchronization.
- Can you find which part is synchronization?
  - `pthread_join` ensures that the main thread waits for the other threads to finish before continuing.
  - This is a form of synchronization because it guarantees that all threads reach a known state (completion) before the program proceeds.

```
pthread_t t1, t2;

pthread_create(&t1, NULL, foo, NULL);
pthread_create(&t2, NULL, foo, NULL);

pthread_join(t1, NULL);
pthread_join(t2, NULL);
```

Problems with Initial Attempt

```
void *Tproduce(void *arg) {
  while (1) {
retry:
    pthread_mutex_lock(&lk);
    if (count == n) {
      pthread_mutex_unlock(&lk);
      goto retry;
    }
    count++;
    printf("O");
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

```
void *Tconsume(void *arg) {
  while (1) {
retry:
    pthread_mutex_lock(&lk);
    if (count == 0) {
      pthread_mutex_unlock(&lk);
      goto retry;
    }
    count--;
    printf("X");
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

Problems with Initial Attempt (Cont.)

```
void *Tproduce(void *arg) {
  while (1) {
retry:
    pthread_mutex_lock(&lk);
    if (count == n) {
      pthread_mutex_unlock(&lk);
      goto retry;
    }
    count++;
    printf("O");
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

```
void *Tconsume(void *arg) {
  while (1) {
retry:
    pthread_mutex_lock(&lk);
    if (count == 0) {
      pthread_mutex_unlock(&lk);
      goto retry;
    }
    count--;
    printf("X");
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

- **Busy Waiting:** Both producer and consumer continuously retry when the buffer is full or empty. This leads to a waste of CPU resources.
- **Resource Contention:** Multiple threads constantly lock and unlock the same mutex without meaningful progress when conditions are not met, causing unnecessary contention.
- **High CPU Utilization:** The goto retry causes the threads to remain in a tight loop, consuming CPU cycles even when they should be waiting.

Why Avoid Busy Waiting?

”Haste makes waste.”

Constant spinning and busy waiting lead to errors. Slowing down with condition variables reduces mistakes.

Tip 1: pthread_cond_wait

```
void *Tproduce(void *arg) {
  while (1) {
    pthread_mutex_lock(&lk);
    while (count == n) {
      pthread_cond_wait(&not_full, &lk);
    }
    count++;
    printf("O");
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

```
void *Tconsume(void *arg) {
  while (1) {
    pthread_mutex_lock(&lk);
    while (count == 0) {
      pthread_cond_wait(&not_empty, &lk);
    }
    count--;
    printf("X");
    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

- `pthread_cond_wait`: A thread goes to sleep and releases the mutex while waiting for a condition (e.g., buffer not empty/full).
- **Important**: `pthread_cond_wait` must be used with a mutex.
  - The thread must first acquire the mutex lock before calling `pthread_cond_wait`.
    - `pthread_cond_wait` only handles waiting for a condition to be met, it does not handle acquiring the lock.

Tip 2: pthread_cond_signal

```
void *Tproduce(void *arg) {
  while (1) {
    pthread_mutex_lock(&lk);
    while (count == n) {
      pthread_cond_wait(&not_full, &lk);
    }
    count++;
    printf("O");
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

```
void *Tconsume(void *arg) {
  while (1) {
    pthread_mutex_lock(&lk);
    while (count == 0) {
      pthread_cond_wait(&not_empty, &lk);
    }
    count--;
    printf("X");
    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

- `pthread_cond_signal`: Wake up one waiting thread when the condition is met (e.g., an item is produced or consumed).
  - Which thread is woken up?
    - If multiple threads are waiting, the OS decides which thread to wake up based on a scheduling policy, usually first-come, first-served (FIFO) or priority-based.

Tip 3: You can also use pthread_cond_broadcast

```c
void *Tproduce(void *arg) {
  while (1) {
    pthread_mutex_lock(&lk);
    while (count == n) {
      pthread_cond_wait(&not_full, &lk);
    }
    count++;
    printf("O");
    pthread_cond_broadcast(&not_empty);
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

```c
void *Tconsume(void *arg) {
  while (1) {
    pthread_mutex_lock(&lk);
    while (count == 0) {
      pthread_cond_wait(&not_empty, &lk);
    }
    count--;
    printf("X");
    pthread_cond_broadcast(&not_full);
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

- `pthread_cond_broadcast`: Wake up all waiting threads when the condition is met.
  - When to use `pthread_cond_broadcast`?
    - Use `pthread_cond_broadcast` when a global state changes that affects all threads.

The Most Important Tip: Two Condition Variables!

```
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
```

- Avoid waking the same type of thread:
  - Producers should not wake other producers, and consumers should not wake other consumers.
  - Producer thread:
    - Waits on `not_full` when the buffer is full.
    - Signals `not_empty` after producing an item, allowing consumers to wake up and consume.
  - Consumer thread:
    - Waits on `not_empty` when the buffer is empty.
    - Signals `not_full` after consuming an item, allowing producers to wake up and produce.
- Link of Code: **Single Condition Varaible Example Code**

Deadlock with Single Condition Variable Example

```
pthread_cond_t buffer_change = PTHREAD_COND_INITIALIZER;
```

- Scenario:
  - Buffer size (n = 1)
  - 2 producer threads (P1, P2) and 2 consumer threads (C1, C2)
  - The buffer is empty and C1 and C2 are sleeping
  - P2 is also sleeping due to the buffer being full previously.
- Process:
  - P1 produces an item, filling the buffer (count = 1), then signals 'buffer_change' (P1 is ready to run and not sleeping)
  - The signal wakes up P2
  - P2 is woken up, but finds the buffer is full, so P2 goes back to sleep without sending any signal
  - P1 is scheduled by the OS, but P1 also finds the buffer is full and goes to sleep without sending any signal
  - The OS may now try to schedule C1 or C2, but they are still sleeping, waiting for the signal that hasn't been sent
- Result:
  - All threads are now in a sleeping state, resulting in deadlock

Cause of Single Condition Variable Deadlock

- All threads rely on a signal to wake up, rather than automatically waking when the condition becomes true.
- A single condition variable may wake up the same type of thread repeatedly.
- No further signals can be sent, leading to deadlock.

- Role of the Operating System:
  - Manages thread scheduling and CPU time allocation
  - Does not manage thread synchronization or signal passing
  - Cannot wake threads
- Thread Communication:
  - Synchronization happens through condition variables (signals) and mutexes
  - Signals must be explicitly sent and received between threads
  - Proper signal passing is critical for correct thread coordination

Why Two Condition Variables Prevent Deadlock

- A producer's 'not_empty' signal only wakes consumers.
- A consumer's 'not_full' signal only wakes producers.

- At least one thread type can always proceed and change the buffer state
- Eliminates the possibility of all threads waiting at the same time

Limitations of Condition Variables

- Imagine a buffer with 5 slots, initially empty / full.
- If 5 producer / consumer threads want to produce / consume 'O', a condition variable only allows one thread to produce / consume at a time.
- But what if we want multiple threads to produce / consume 'O' concurrently?

Semaphores

- **Semaphore** is a synchronization mechanism used to control access to shared resources in concurrent systems.
- It acts as an integer counter that tracks the availability of a limited number of resources.
- It can allow multiple threads to enter the critical section simultaneously.
    - However, you must ensure that there are no race conditions when multiple threads are in the critical section. If there are no such issues, semaphores can be used effectively.

Semaphore Operations

- Semaphores were first introduced by **Edsger W. Dijkstra** in the 1960s.
- Semaphores operate similarly to **condition variables**, allowing threads to wait and be signaled based on certain conditions.

Semaphores have two primary operations:

- **P operation** (from Dutch *proberen*, meaning ”to try”):
  - Decreases the semaphore’s value by 1.
  - If the value becomes negative, the thread performing the P operation is blocked until the semaphore’s value becomes positive.
- **V operation** (from Dutch *verhogen*, meaning ”to increment”):
  - Increases the semaphore’s value by 1.
  - If there are any blocked threads, the V operation wakes up one of them.

Code Comparison

```
// Condition Variables
void *Tproduce(void *arg) {
  while (1) {
    pthread_mutex_lock(&lk);
    while (count == n) {
      pthread_cond_wait(&not_full, &lk);
    }
    count++;
    printf("O");
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}

void *Tconsume(void *arg) {
  while (1) {
    pthread_mutex_lock(&lk);
    while (count == 0) {
      pthread_cond_wait(&not_empty, &lk);
    }
    count--;
    printf("X");
    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&lk);
  }
  return NULL;
}
```

```
// Semaphores
void *Tproduce(void *arg) {
  while (1) {
    P(&empty_sem);


    pthread_mutex_lock(&mutex);
    printf("O");
    pthread_mutex_unlock(&mutex);


    V(&full_sem);
  return NULL;
}

void *Tconsume(void *arg) {
  while (1) {
    P(&full_sem);


    pthread_mutex_lock(&mutex);
    printf("X");
    pthread_mutex_unlock(&mutex);

    V(&empty_sem);
  }
  return NULL;
}
```

- Link of Code: **Semaphores Example Code**

Semaphores vs Condition Variables: Key Differences

- **Resource Management:**
  - Semaphores have a built-in counter to manage resource availability.
  - Condition variables do not track resource availability. The programmer must manage resource state manually.
- **Wait/Wake Mechanism:**
  - Semaphores use the **P (wait)** and **V (signal)** operations to automatically handle the blocking and unblocking of threads.
  - Condition variables use **pthread_cond_wait()** to put a thread to sleep and **pthread_cond_signal()** or **pthread_cond_broadcast()** to wake up waiting threads.
- **Mutex Usage:**
  - Semaphores can be used with or without a mutex, allowing multiple threads to access the critical section simultaneously based on the semaphore's value.
  - Condition variables must be used with a mutex, typically allowing only one thread in the critical section at a time, even if multiple threads are woken up.

Semaphores without Mutex

```
void *Tproduce(void *arg) {
  while (1) {
    P(&empty_sem);
    printf("O");
    V(&full_sem);
  }
  return NULL;
}
```

```
void *Tconsume(void *arg) {
  while (1) {
    P(&full_sem);
    printf("X");
    V(&empty_sem);
  }
  return NULL;
}
```

- Link of Code: **Semaphores without Mutex Example Code**

Considerations for Semaphore Usage

- If you plan to implement more complex buffer operations (e.g., actually storing data instead of just printing characters), you will need to use a mutex to avoid race conditions.
- While semaphores may seem convenient, they become less effective as more rules are added, making them harder to manage.
- It's often better to use **condition variables** for complex synchronization needs.

Real-World Concurrent Programming

Visual Studio Code: A Web Application

Did you know? Visual Studio Code is a Web App!

- Visual Studio Code (VS Code) is built using Electron, which is essentially a web browser.
- It runs inside a Chromium engine, meaning it is just like a web page!
  - The editor is a web application running locally.
  - The entire UI is powered by HTML, CSS, and JavaScript.
- Many extensions and features interact with VS Code just like a website interacts with a backend.
  - The backend is built with Node.js, handling interactions.

Haven't Tried Visual Studio Code Yet?

I've heard Cursor is pretty good too! I've been using it recently...

The Web 2.0 Era (1999)

- The Internet brought people closer together.
- ”Users were encouraged to provide content, rather than just viewing it.”
- You can even find early hints of ”Web 3.0”/Metaverse in this period.

From Web 1.0 to Web 2.0: The Rise of Ajax

Asynchronous JavaScript and XML (Ajax, 1999)

- Revolutionized the Web:
  - Allowed web pages to update content without reloading the entire page.
  - Made dynamic, interactive applications possible in the browser.
  - Enabled background communication with the server, improving user experience.
- How does it work?
  - JavaScript sends a request to the server asynchronously.
  - The server responds with data, which JavaScript processes.
  - The webpage updates dynamically by modifying the DOM.
- Surprising Fact: It wasn't JSON!
  - Early Ajax applications often used XML, not JSON.
  - Why? Many backend applications (especially Java) primarily used XML.

jQuery: Making JavaScript Easier (2006)

Why jQuery?

- JavaScript was powerful but messy—working with the DOM was difficult.
- jQuery simplified JavaScript and made it more readable.
- Cross-browser compatibility—jQuery handled inconsistencies between browsers.

Key Features of jQuery:

- Easier DOM Manipulation:
  - Example: Replacing all ‘<h3>‘ elements with ”XXX”:

```
$('h3').replaceWith('XXX');
```

- Built-in Animation & Effects
- Simplified Ajax Requests

Impact:

- jQuery made it easy for developers to create interactive websites.
- It paved the way for modern front-end frameworks like React, Angular, and Vue.

Then, Everything Can Be Done in the Browser

- HTML + CSS made applications more flexible and faster than traditional GUI programming.
- This even led to the creation of ChromeOS.

Do you remember?

- GTK, Qt, MFC... Who used those? (Me 😢)

Why Did Web Replace Traditional GUI Frameworks?

Web (HTML + CSS + JavaScript) vs. Traditional GUI (GTK, Qt, MFC)

- Cross-Platform & No Installation Required
  - Traditional GUI frameworks require separate implementations for Windows, Linux, and Mac.
  - Web apps run on any device with a browser, no installation needed.
- Higher Development Efficiency
  - Traditional GUI apps require manual UI component design.
  - Web frameworks (React, Vue, Angular) provide reusable components.
- Easier Distribution & Maintenance
  - Traditional apps require packaging (EXE/DMG) and manual updates.
  - Web apps update instantly on the server—no user action required.

Challenges

- Threads became popular in the 1990s.
- Thread synchronization is difficult and error-prone.

Solution: Event-based concurrency (Dynamic Computation Graphs)

- Allows computation nodes to be created at runtime.
  - Examples: Network requests, timers.
- **No parallel execution of computation nodes**
  - Most time is spent on network access; browser-side computation is minimal.
- Uses events as fundamental scheduling units.
  - Events can be observed inside the browser!

More one Features and Challenges

Features:

- Not very complex
- Minimal computation required
  - The DOM tree is not too large (humans can't handle huge trees anyway)
  - The browser handles rendering the DOM tree for us
- Not too much I/O, just a few network requests

Challenges:

- Too many programmers, especially for beginners
- Expecting beginners to handle multithreading with shared memory would lead to a world full of buggy applications!

Why is Web Concurrency Unique?

- JavaScript in browsers is single-threaded.
- Blocking operations would freeze the entire page!
- Solution: Event-driven concurrency (Event Loop).

Event Loop: Managing Asynchronous Execution

- Main thread executes JavaScript code sequentially.
- Asynchronous operations (e.g., network requests, timers) are sent to the browser API.
- Once completed, these tasks re-enter the JavaScript engine via an event queue.

Example: What gets printed first?

```
console.log("1");

setTimeout(() => console.log("2"), 0);

Promise.resolve().then(() => console.log("3"));

console.log("4");
```

Expected Output:

- `1` -> Executed first (synchronous).
- `4` -> Executed second (synchronous).
- `3` -> Executed third (Promise.then() -> microtask queue).
- `2` -> Executed last (setTimeout -> macrotask queue).

Why?

- JavaScript first executes all synchronous code.
- Then it processes microtasks (Promise callbacks).
- Finally, it executes macrotasks (setTimeout, I/O events).

Microtask vs. Macrotask Queue

- **Microtasks** (higher priority): Promise callbacks, `queueMicrotask()`.
- **Macrotasks**: setTimeout, setInterval, I/O operations.

Key Takeaway: The Event Loop ensures JavaScript remains responsive while handling asynchronous tasks efficiently.

Single-Threaded + Event Loop

Asynchronous with minimal but sufficient concurrency:

- Single thread, global event queue, sequential execution (run-to-complete)
- Time-consuming APIs (Timer, Ajax, etc.) return immediately
- When conditions are met, a new event is added to the queue

Example: Chained Ajax Calls

```
$.ajax( { url: 'https://xxx.yyy.zzz/login',
  success: function(resp) {
    $.ajax( { url: 'https://xxx.yyy.zzz/cart',
      success: function(resp) {
        // do something
      },
      error: function(req, status, err) { ... }
    }
  },
  error: function(req, status, err) { ... }
);
```

Solution: Asynchronous Event Model

Advantages:

- Concurrency model is greatly simplified
  - Function execution is atomic (no parallel execution, reducing the chance of concurrency bugs)
- APIs can still run in parallel
  - Suitable for web applications where most time is spent on rendering and network requests
  - JavaScript code only ”describes” the DOM Tree

Disadvantages:

- Callback hell (the infamous ”spaghetti code”)
- As seen in the previous example, nesting 5 levels deep makes the code nearly unmaintainable

Asynchronous Programming: Promise

Definition:

- The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Promise: An Embedded Language for Describing Workflows

- **Chaining:**

```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // scripts are loaded, we can use functions declared there
  })
  .catch(err => { ... });
```

- **Fork-join:**

```
a = new Promise((resolve, reject) => { resolve('A') });
b = new Promise((resolve, reject) => { resolve('B') });
c = new Promise((resolve, reject) => { resolve('C') });
Promise.all([a, b, c]).then(res => { console.log(res) });
```

Advantages of Promise

- Readability: Promise chaining improves code readability by avoiding deeply nested callbacks, making asynchronous operations easier to follow.
- Error Handling: Provides a clear and structured way to handle errors through .catch(), reducing complexity compared to traditional callback error handling.
- Control Flow: Promises enable better control over the execution order of asynchronous tasks, ensuring that steps are completed in sequence.
- Flexibility: Easily integrates with modern JavaScript features like async/await for even cleaner and more readable code.

Writing Asynchronous Code Like Synchronous Code

Problem with Callbacks and Promises:

- Callback Hell: Nested functions become unreadable.
- Promises improve structure but still require chaining.

Solution: `async/await` (ES8)

- `async` functions always return a Promise.
- `await` pauses execution until the Promise resolves.
- Looks synchronous, but runs asynchronously!

Async-Await

```
async function fetchData() {
  console.log("Start");
  let response = await fetch('https://example.com/data');
  let data = await response.json();
  console.log("Data␣loaded:", data);
}
console.log("Before␣fetch");
fetchData();
console.log("After␣fetch");
```

Understanding Execution Order:

- `"Before fetch"` -> Executed first.
- `"Start"` -> Executed second.
- `fetch()` runs asynchronously, doesn't block execution!
- `"After fetch"` -> Executed third.
- Once the request completes, `"Data loaded:"` is printed.

Async-Await: Even Better

async function:

- Always returns a Promise object
- `async_func()` - fork
- `await promise` - join

```
A = async () => await $.ajax('/hello/a');
B = async () => await $.ajax('/hello/b');
C = async () => await $.ajax('/hello/c');

hello = async () => await Promise.all([A(), B(), C()]);

hello()
  .then(window.alert)
  .catch(res => { console.log('fetch failed!') });
```

From ”Frontend” to ”Full Stack”

ECMAScript 2015 (ES6)

- Standardized JavaScript, resolving the ”library wars” chaos.
- The rise of open-source ecosystems fueled frontend innovation.

Modern Frontend Technologies

- Frontend Frameworks: Angular, React, Vue
- Full Stack Development: Express.js, Next.js
- CSS Frameworks: Bootstrap, TailwindCSS
- Beyond the Browser: Electron (VS Code)
  - Web technologies power desktop applications.

Frontend technologies are no longer just for browsers; they have expanded to backend and even desktop applications!

The Wheels of History Rolling Forward

PC -> **Web ->** **Web 2.0 (UGC) ->** **AI (AGI)**

- Frameworks drive technological advancements.
- We need high-level abstractions to express real-world human needs.
  - **Simplicity and Clarity** -> Attracts a large number of industry developers.
  - **Flexibility and Generalization** -> Enables the construction of diverse applications.

Standalone Computers -> **Internet ->** **Mobile Computing ->** **???**

- Opportunities and Uncertainty
- Risks and Rewards

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

First Supercomputer (CRAY-1 from Los Alamos National Laboratory in 1976)

Features of HPC

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/**)
    - No. 1

Static Partitioning in Traditional Computation (Machine-Thread Two-Level Task Decomposition)

- The Producer-Consumer Model solves most problems.
- MPI – "Message Passing Interface" for distributed computing.
- OpenMP – "Multi-platform shared-memory parallel programming (C/C++ and Fortran)."

Example: OpenMP Parallelization

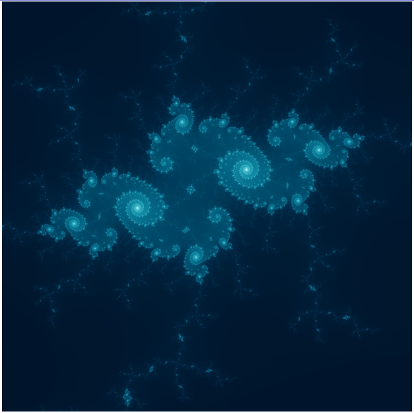
```
#pragma omp parallel num_threads(128)
for (int i = 0; i < 1024; i++) {
   // Parallel execution
}
```

Challenges in HPC:

- Network latency, power consumption, stability, and scalability.
- Hardware-software toolchains.

Example: Mandelbrot Set

- $Z_{n+1}^2 = Z_n^2 + C$
- Each point in the Mandelbrot set iterates independently and is only influenced by its complex coordinate.
- Link of Code: **Mandelbrot Set Code**



- While the number of cores is not the only factor, it is the most critical factor for determining thread execution efficiency.
- Core count helps estimate the system's computational capacity and parallel processing capabilities.
- Therefore, it is a key factor in HPC.

Concurrent Programming in Data Centers

Google Data Center

Features of Data Center

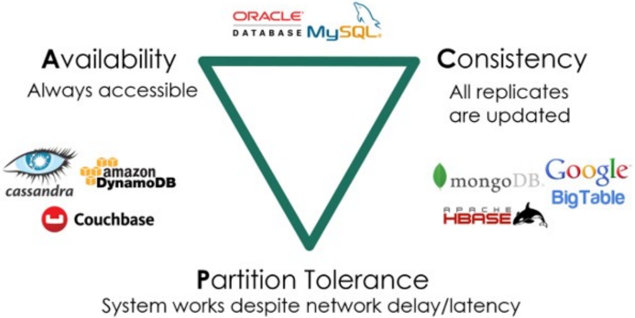
Data Center

"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

- Serving massive, geographically distributed requests
- Data must remain consistent (Consistency)
- Services must always be available (Availability)
    - Must tolerate machine failures (Partition Tolerance)



We foucus on a single machine

How to Maximize Parallel Request Handling with a Single Machine

- Key Metrics: QPS, Tail Latency, ...

Maximizing Parallel Request Handling: Threads

Advantages:

- True parallelism with multiple cores, enabling multiple execution flows.
- OS-level scheduling, allowing independent tasks to be managed efficiently by the operating system.
- Well-supported by most programming languages and operating systems.

Disadvantages:

- Higher overhead due to system calls and context switching.
- Limited by the number of cores, potentially leading to contention and inefficiencies with too many threads.
- Memory overhead due to thread stacks and system resources.

- Link of Code: **Thread Example Code**

Maximizing Parallel Request Handling: Coroutines

Advantages:

- More lightweight than threads, as they don't require system calls or context switches.
- Link of Code: **Coroutine Example Code**

Why are Coroutines More Lightweight?

- **User-Space Scheduling:** Coroutines are managed in user space and do not require kernel intervention, avoiding the overhead of system calls.
- **Minimal Context Switching:** Switching between coroutines requires saving only a small amount of information:
  - **Execution Position:** The point in the code where the coroutine yields or resumes (similar to the program counter in threads).
  - **Local Variables and Stack Frame:** The current state of local variables and the execution stack.
- **Comparison with Threads:** Threads require the operating system to save and restore more extensive context:
  - All CPU registers, including general-purpose and floating-point registers.
  - Program counter and stack pointer, which determine the execution position and stack location.
  - Thread-specific kernel data structures, which manage the thread's scheduling and other metadata.

Disadvantages of Coroutines

- No true parallelism
  - A single thread can only run one coroutine at a time.
  - Although multiple coroutines can exist within the same thread, only one is active at any given moment.
  - The quick switching between coroutines creates the illusion of concurrency.
- **Blocking Operations:**
  - If a coroutine encounters a blocking operation (e.g., system calls), it blocks the entire thread.
  - This means all other coroutines in the same thread are also blocked, causing a significant performance issue.
- Requires manual yielding: Developers must manually control when a coroutine yields, which can lead to more complex code management.
- Less well-supported: Coroutines are not as universally supported across programming languages as threads.

Maximizing Parallel Request Handling: Go

Goroutines = Threads + Coroutines

Advantages:

- Extremely lightweight
- Enable true parallel execution on multiple cores
- Ideal for high-concurrency systems with minimal developer management
    - Efficient CPU utilization, achieving near 100% performance

Disadvantages:

- Go runtime's scheduling decisions are opaque, making it harder to control execution flow.
- Debugging and profiling goroutines can be more difficult due to their lightweight nature and runtime control.
- Not available in all languages, limited to the Go ecosystem.

Why Goroutines = Threads + Coroutines?

- **When a Goroutine encounters a blocking system call:**
  - Automatically converts blocking system calls (e.g., file I/O) into non-blocking operations.
  - Moves the Goroutine off the current thread and schedules another Goroutine to continue execution.
  - This ensures that no Goroutine blocks the entire system, maximizing concurrency.
- Link of Code: **Goroutine Example Code**

Concurrent Programming in the Age of AI

NVIDIA DGX-1 (2016)

8 × Tesla V100: The Computational Core of DGX-1

- **DGX-1 is a complete AI supercomputer** designed by NVIDIA.
- It integrates **8 Tesla V100 GPUs** into a single system.
- These GPUs are interconnected using **NVSwitch**, providing high-speed GPU-to-GPU communication (300GB/s).
- Compared to standalone GPUs, DGX-1 includes:
  - **2 × Intel Xeon CPUs** for coordination.
  - **512GB DDR4 RAM** for system memory.
  - **15TB NVMe SSD** for high-speed storage.
  - Optimized power and cooling system (3.2kW power consumption).
- **Performance:** 170 TFLOPS @ 3.2kW
- **Comparison:** CRAY-1: 138 MFLOPS @ 115kW

NVIDIA DGX B200 (2024)

8 × Blackwell GPUs: The Computational Core of DGX B200

- **DGX B200 is the latest AI supercomputer** designed by NVIDIA.
- It integrates **8 Blackwell GPUs** into a single system.
- These GPUs are interconnected using **NVLink and NVSwitch**, providing ultra-high-speed communication.
- Compared to standalone GPUs, DGX B200 includes:
  - **Optimized AI acceleration** for large-scale training and inference.
  - **High-bandwidth memory (HBM)** for faster data access.
  - Advanced **power and cooling solutions** for efficient operation.
- **Performance:**
  - **72 PFLOPS (Training), 144 PFLOPS (Inference)** @ 14.3kW

Computation Behind Large Language Models

"Attention Is All You Need"

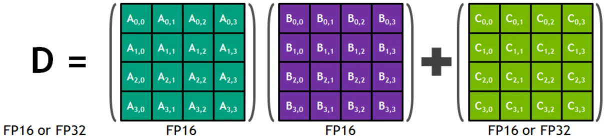
LLM Visualization

Single Instruction, Multiple Data

- **Tensor Instructions (Tensor Core):** Mixed Precision

$$A \times B + C$$

  - A single instruction performs a 4×4 matrix operation.

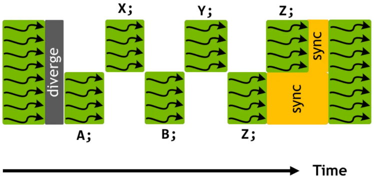


- **x86 SIMD Evolution:**
  - MMX (MultiMedia eXtension, 64-bit MM) $\rightarrow$ SSE (Streaming SIMD Extensions, 128-bit) $\rightarrow$ AVX (Advanced Vector eXtensions, 256-bit) $\rightarrow$ AVX512 (512-bit)

Single Instruction, Multiple Threads

- **One PC (Program Counter)** controls **32 execution flows simultaneously**.
    - The number of logical threads can be even larger.
- Each execution flow has its own **registers**.
    - Three registers (*x*, *y*, and *z*) are used to store the "thread ID".
- Then, **a massive number of threads!**

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



Takeaways

- Most of the synchronization problems you will face are just variations of the **Producer-Consumer problem**.
- Mastering **condition variables** is enough to handle most real-world scenarios.
- The rest is just icing on the cake.

Takeaways

- Web
  - Focus: Usability
  - Pattern: Single Thread + Event Loop
  - Technologies: Promise
- High-Performance Computing
  - Focus: Task Decomposition
  - Pattern: Producer-Consumer
  - Technologies: MPI / OpenMP
- Data Centers
  - Focus: System Calls
  - Pattern: Threads-Coroutines
  - Technologies: Goroutine
- AI
  - Focus: Parallel Computation & Scalability
  - Pattern: Data Parallelism + Model Parallelism
  - Technologies: SIMIT / CUDA / TensorRT