

# Code of the Day

Please mute yourself and turn off the video to save network bandwidth (I am using my hotel's internet).  
Thank you for your cooperation!

- $x \text{ modulo } y = x \% y$
- When  $y$  is a power of 2,  $x \text{ modulo } y = (x \& (y - 1))$  – Why???

0110010110 (406) modulo  
0001000000 (64) =

0110010110 &  
0000111111 (63) =  
0000010110 (22)

$$406 = 0110010110 = 2^1 + 2^2 + 2^4 + 2^7 + 2^8$$

# ECE 459

GPU Algorithm and System Design

[Fall 2025]

C++ Thread

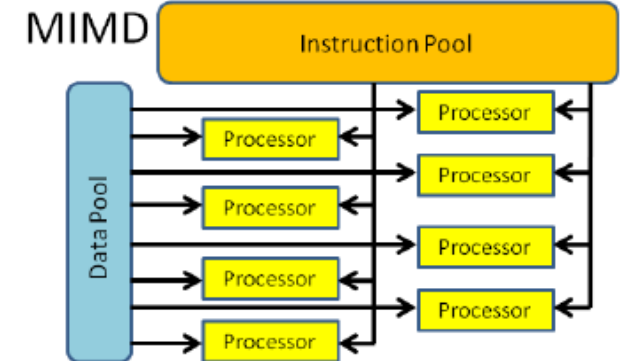
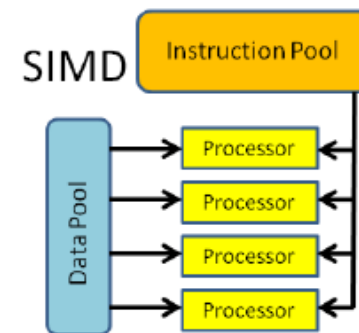
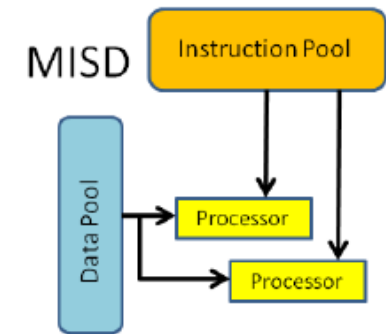
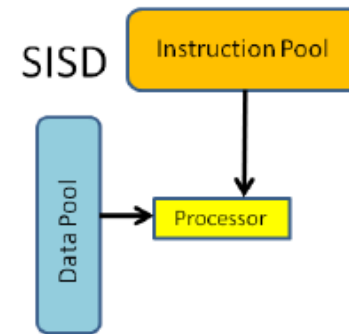
09/15/2025

# Before we get started...

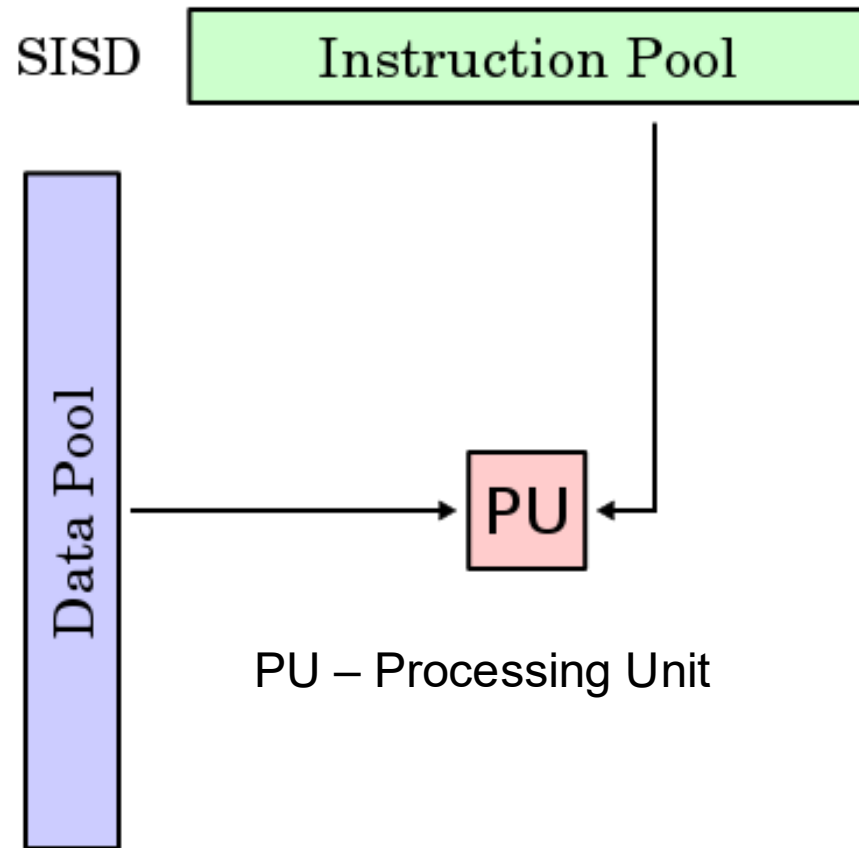
- Quick overview, things discussed last time
  - Three walls to push us to move towards parallel computing:
    - Memory wall: the speed gap between memory and processor continues to increase
    - ILP wall: instruction-level parallelism is great, but the complexity prevents it from moving forward
    - Power wall: the faster the hardware, the more power you need
  - Classification of parallel computing paradigms: SISD, SIMD, MISD, MIMD
    - A look at two computing platforms for parallel computing: multi-core and GPU chips
- Purpose of today's lecture:
  - Look into the basics of CPU-parallel computing and thread-level parallelism using C++ and OpenMP
- Miscellaneous
  - Lab assignment #1 due on **Friday 9/19 at 23:59 PM**

# Flynn's Taxonomy of Architectures

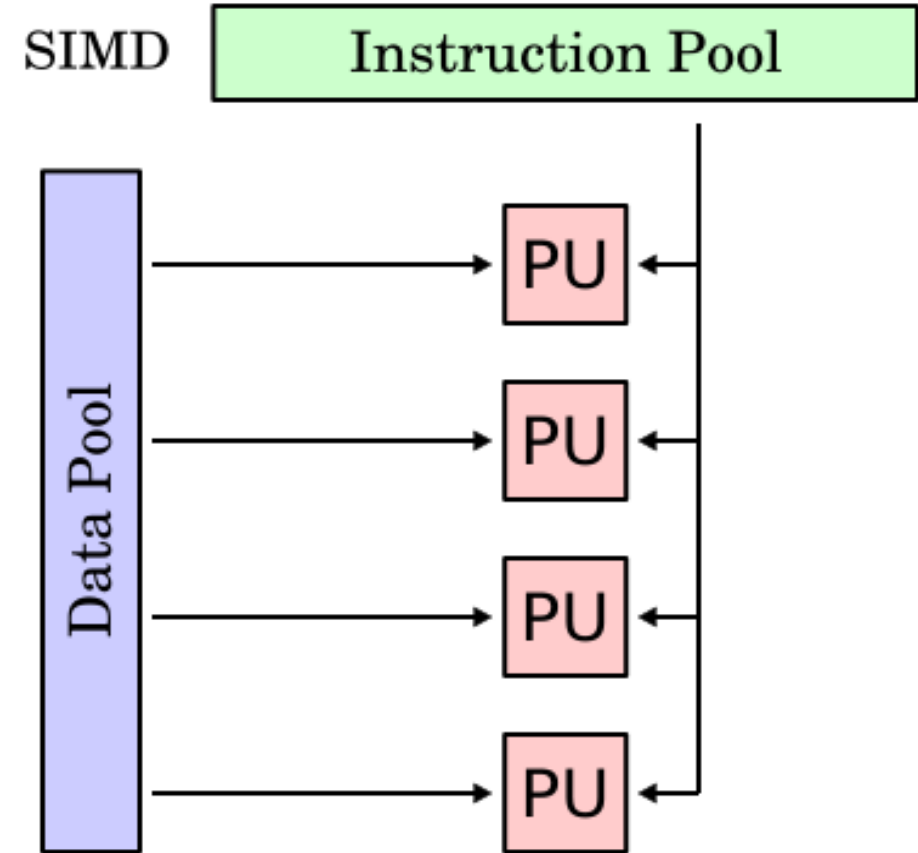
- A software-centric aspect to classify modern computer architectures
- Flynn's classification is based on how instructions are executed in relation to data
  - SISD - Single Instruction runs Single Data
  - SIMD - Single Instruction runs Multiple Data
  - MISD - Multiple Instructions run Single Data
  - MIMD - Multiple Instructions run Multiple Data



# Single Instruction Single vs Multiple Data: SISD vs SIMD



SISD is the most basic computer architecture in the single-core domain










SIMD is a parallelized architecture in the single-core domain

# A Short Side Trip: Why is CUDA calling its execution model SIMT?

- CUDA is inspired by SIMD but is actually called SIMT (single instruction multiple threads)
- SIMD: single instruction, multiple data
  - One instruction is issued and the same instruction is applied “multiple data”
  - This is done religiously by Intel and AMD architectures via streaming SIMD extensions (SSE)
    - Ex: Can create registers of 128-bit width and 256-bit width storing 4 and 8 integers (32 bits) at the same time
- SIMT: single instruction, multiple threads (for CUDA, only)
  - A *single* instruction is issued at each clock cycle where one or more threads can execute that instruction
    - In general, a warp of threads (32 threads) execute the same instruction in a *lock-step* fashion

# Opportunities for Efficiency Gains through Parallelism

- Outlined from the highest level  to the lowest level 

Cluster (distributed)		Group of nodes communicate through interconnect	MPI, Charm++, Chapel
Multi-Socket Node		Group of CPUs on the same node, talk through main mem.	OpenACC, OpenMP, MPI
Acceleration (GPU)		Compute devices accelerating parallel computation on one node	CUDA, OpenCL, OpenMP, OpenACC
Multi-Core Processor		Communication through shared caches and main mem.	OpenMP, TBB, pthreads, OpenACC
Vectorization		Higher operation throughput via special/fat registers	AVX, SSE, OpenMP
Pipelining		Sequence of instruction sharing functional units	Instruction-level
Superscalar		Non-sequence instructions sharing functional units	Instruction-level

We have full control →  We have little to no control → 

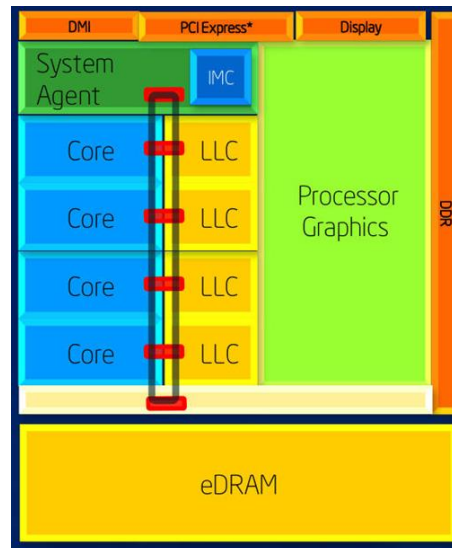
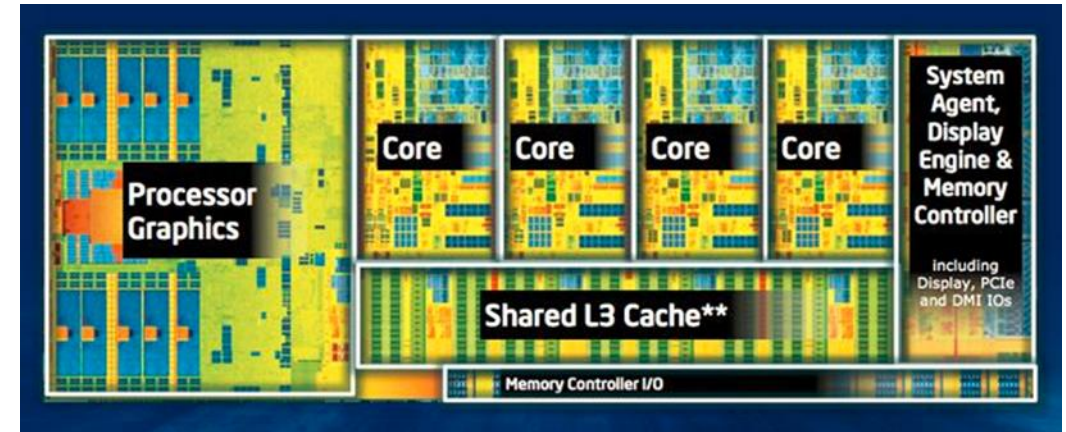
# Multi-core parallel Programming with `std::thread`



# Targeted Hardware: An Example of Intel Haswell CPU

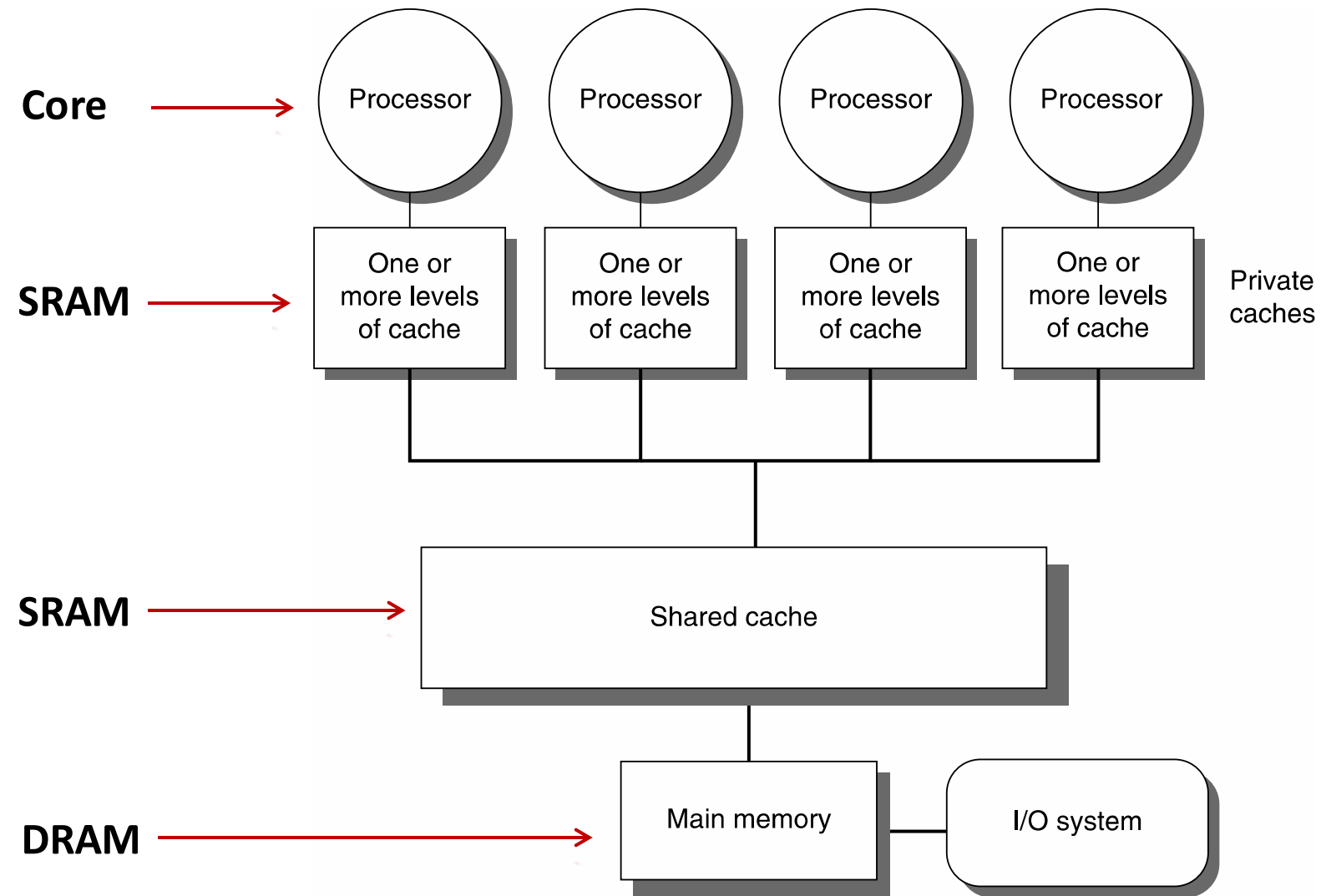
- Intel Haswell

- Released in June 2013
- 22 nm technology
- Transistor budget: 1.4 billions
- Typically comes in four cores
- Has an integrated GPU
- Sophisticated design for ILP acceleration
- Deep pipeline – 16 stages
- Superscalar
- Supports HTT



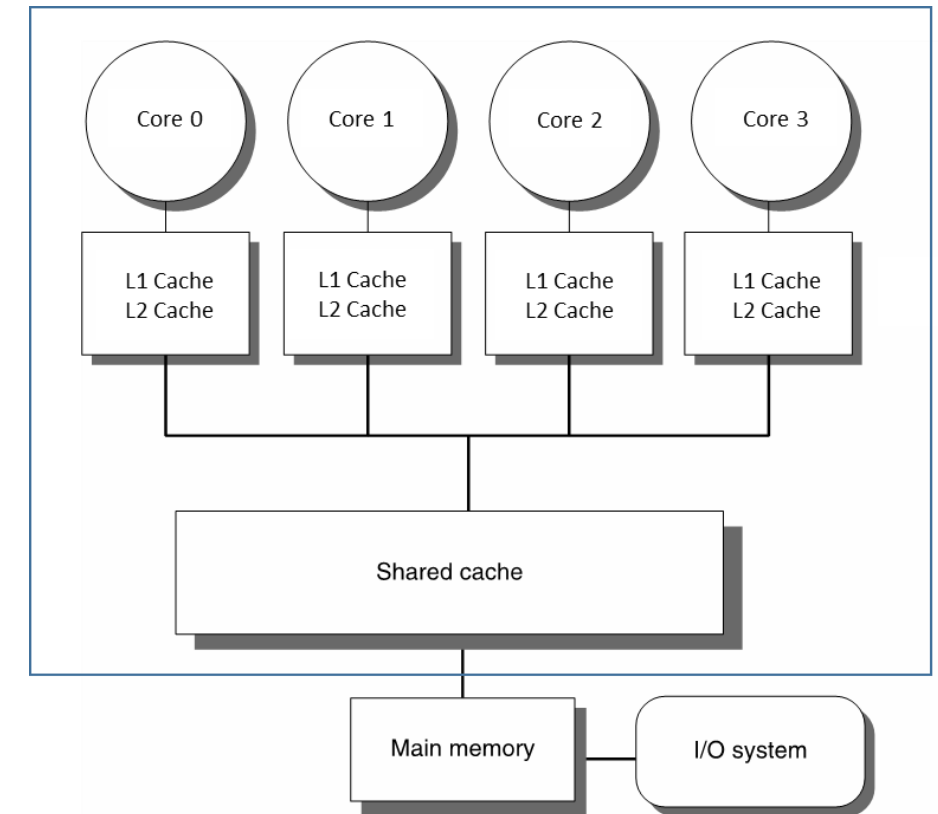
- LLC: last level cache (L3)
- Three clock domains for different purposes:
  - A core's clock ticks at 2.7 to 3.0 GHz but adjustable up to 3.7-3.9 GHz
  - Graphics processor ticking at 400 MHz but adjustable up to 1.3 GHz
  - Ring bus and shared L3 cache - a frequency that is close to but not necessarily identical to that of the cores

# Haswell is Based on Shared-memory Multiprocessor Architecture



# SMP: Symmetric Multi-Processing

- This type of shared-mem architecture is typically called *symmetric multi-processing* (SMP)
  - The chip is symmetric as far as its cores are concerned in its architecture
    - Core 0 sees the same amount of L1 cache as other cores
    - Core 0 sees the same amount of RAM as other cores
    - ... and so on so forth
- SMP offers several advantages for parallel programming
  - Simplified memory management
  - Easy to program on a single node
  - Single operating system instance
  - Transparent load balancing
  - Fault tolerance



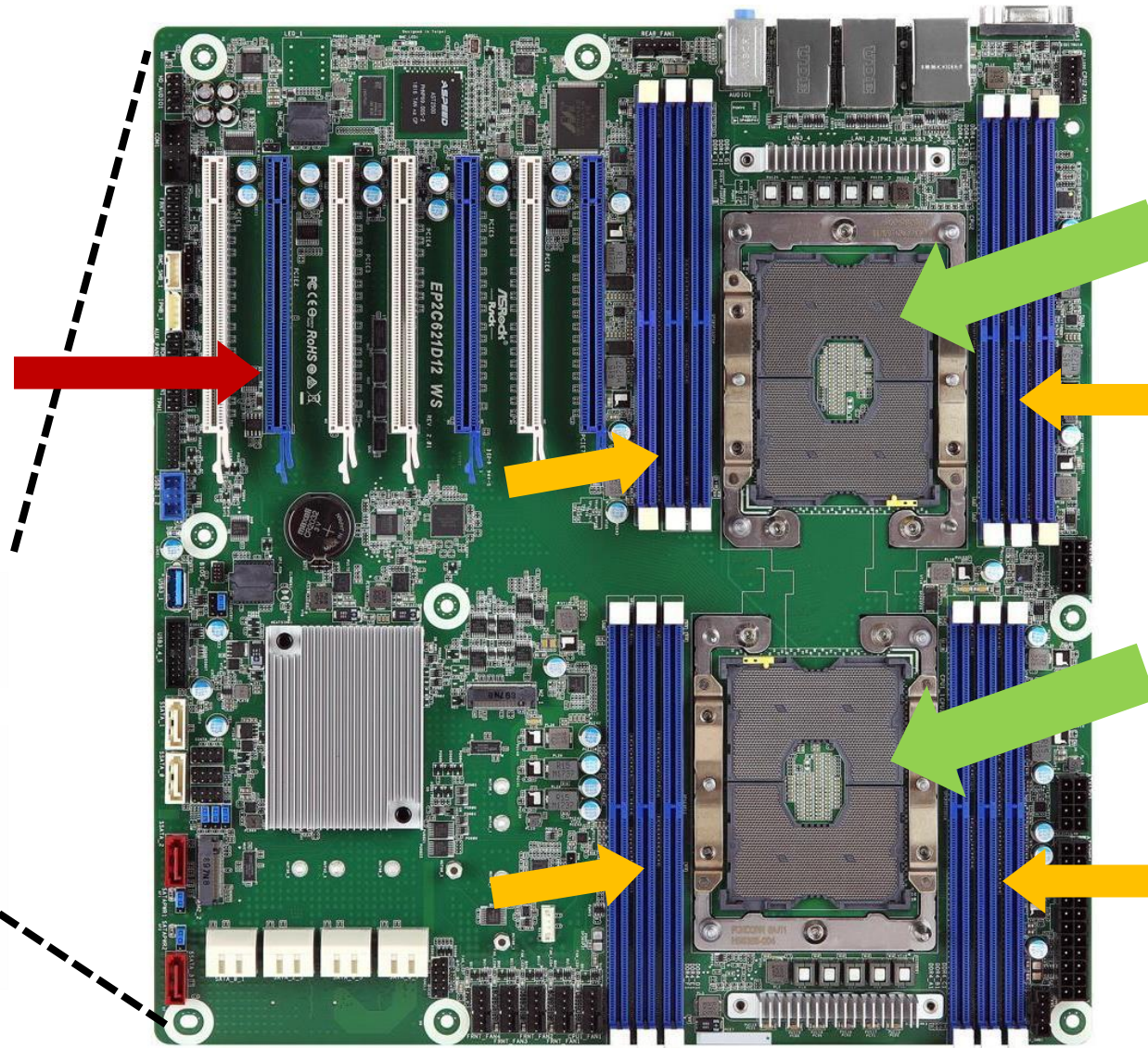
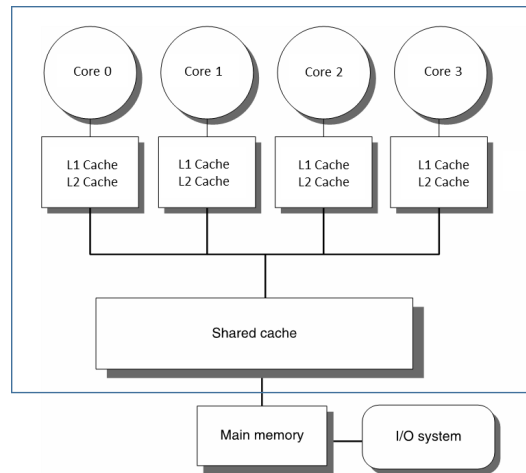


# Chip View of Where and What We Actually Program

[ASRock Rack EP2C621D12 WS EEB Server Motherboard LGA 3647 Intel C621]

Seven of PCI sockets to  
accommodate expansion  
of cards (e.g., GPU)

This is where actual  
CPU parallelism takes  
place (we're here)



→: Memory Bank

→: CPU Socket

→: PCI Socket

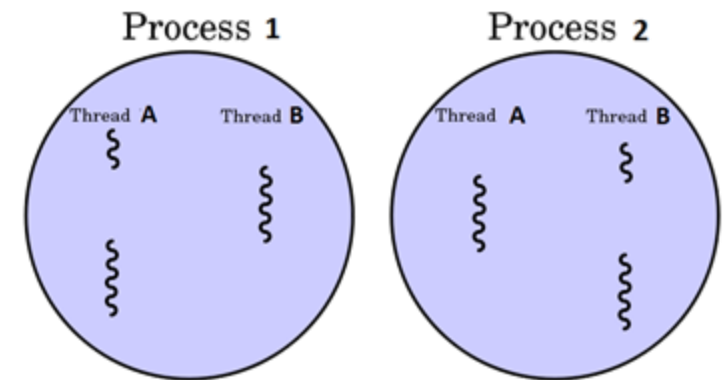
# Parallel Programming: Thread vs Process

- A thread is an independent execution sequence within a single process.
  - Each process starts with a main thread at which many child threads can be forked from
  - Each thread operates within the same process and share the same global data (e.g., text, data, and heap)
  - Each thread has their own memory stack (private memory space per thread)

```
int main() {...} // where main thread starts
```

- Processes vs Threads

- Process
  - Isolated virtual address spaces – 😊: security; ☹️: harder to share info
  - Harder to coordinate multiple tasks within the same program – ☹️
    - Ex: Python's parallelism is more at process level
- Threads
  - Shared virtual address space – ☹️: security; 😊: easier to share info
  - Easier to coordinate multiple tasks within the same program – 😊
    - Ex: C++'s parallelism is more at thread level



# C++ Threads (`std::thread`)

- `std::thread` in a C++ library to create and manipulate threads within a process
- A thread object can be spawned to run the given function with the given arguments

```
std::thread myThread(myFunc, arg1, arg2, ...);
```

- `myFunc`: the function the thread should execute asynchronously
  - `args`: a list of arguments (any length) to pass to the function upon execution – by default **passed by copy**
  - Once initialized with this construction, the thread can execute at any time!
- To pass arguments by references to a thread, use `std::ref()` function

```
void myFunc(int& x, int& y) {...}  
std::thread myThread(myFunc, std::ref(arg1), std::ref(arg2));
```

# C++ Threads (`std::thread`) – cont'd

- We can make an array of threads

```
// declare array of empty thread handles
std::thread friends[5];

// Spawn threads
for (size_t i = 0; i < 5; i++) {
    friends[i] = std::thread(myFunc, arg1, arg2);
}
```

- IMPORTANT: You must join all forked thread before leaving the main function

```
std::thread myThread(myFunc, arg1, arg2);
... // do some work
// Wait for thread to finish (blocks)
myThread.join();
```

# Atomic Function: `printf` vs `std::cout`

- An atomic function is the one that will be completed without being interrupted during its execution even when called concurrently from multiple threads
  - `printf` is atomic each call
  - `std::cout::operator <<` is atomic each call

Three function calls     `std::cout << "welcome" << " to" << " class"`

One function call     `printf("welcome to class")`

One function call     `std::cout << "welcome to class";`



# Thread-level Parallelism

- Threads allow a process to parallelize a problem across multiple cores
  - Main thread forks many child threads to process different partitions of a workload
- Consider a scenario where we want to process 250 images and have 10 cores
  - Let each thread help process images until none are left

```
std::thread processors[10];
size_t remainingImages = 250;
for (size_t i = 0; i < 10; i++) {
    processors[i] = std::thread(process, i, std::ref(remainingImages));
}
for (auto& proc: processors) {
    proc.join();
}
std::cout << "Images done!";
```

```
void process(
    size_t id,
    size_t& remainingImages
) {
    while (remainingImages > 0) {
        // processing image ...
        remainingImages--;
    }
}
```

# There is a Race Race Problem Here!

- Problem: threads could interrupt each other around continuation check (while)
  - 10 threads can simultaneously run `remainingImages > 0` and `remainingImages--`
- Why is this? `remainingImages > 0` test and `remainingImages--` aren't atomic
  - Recall a line of code can be compiled to multiple instructions (see Lectures 1 and 2)

`a[4] = delta + a[3];` //line of C code



```
lw  $t0, 12($s2)
add $t0, $s4, $t0
sw  $t0, 16($s2)
```

```
void process(size_t id, size_t& remainingImages) {
    while (remainingImages > 0) {
        // processing image ...
        remainingImages--;
    }
}
```

Thread 1 reading

Thread 2 writing

# Avoid Data Race: Mutex

- A mutex is a variable type that represents something like a “locked door”
- You can lock the door:
  - If it's unlocked, you go through the door and lock it
  - If it's locked, you wait until the person inside leaves and unlocks the door
- If you most recently locked the door, you can unlock the door
  - Door is now unlocked, another may go in now
- C++ offers a library to create a mutex for locking/unlocking a door



# std::mutex

- C++ `std::mutex` is a type used to enforce mutual exclusion, i.e., a critical section
- Mutexes are often called locks
  - To be very precise, mutexes are one kind of lock, there are others (read/write locks, reentrant locks, etc.), but we can just call them locks in this course, usually “lock” means “mutex”
- When a thread locks a mutex through `std::mutex::lock`
  - If the lock is unlocked the thread takes the lock and continues execution (cheap)
  - If the lock is locked, the thread blocks and waits until the lock is unlocked (not so cheap)
  - If multiple threads are waiting for a lock they all wait until lock is unlocked, one receives lock (expensive)
- When a thread unlocks a mutex through `std::mutex::unlock`
  - It continues normally; one waiting thread (if any) takes the lock and is scheduled to run
- This is a subset of the C++ mutex abstraction:

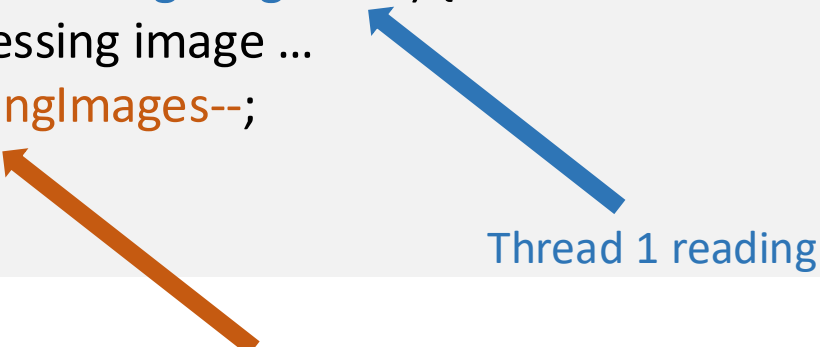
```
std::mutex mutex;  
mutex.lock();  
mutex.unlock();
```

How can we use `std::mutex` to correct our buggy program?

# Critical Section with `std::mutex`

- Create a `std::mutex` and pass it *by reference* (`std::mutex&`) to the thread
- The process function uses this lock to **protect the access** to `remainingImages`


```
void process(size_t id, size_t& remainingImages) {  
    while (remainingImages > 0) {  
        // processing image ...  
        remainingImages--;  
    }  
}
```



A blue arrow labeled "Thread 1 reading" points to the condition `remainingImages > 0`. An orange arrow labeled "Thread 2 writing" points to the decrement operation `remainingImages--`.

Thread 2 writing

There is still a tiny problem here...

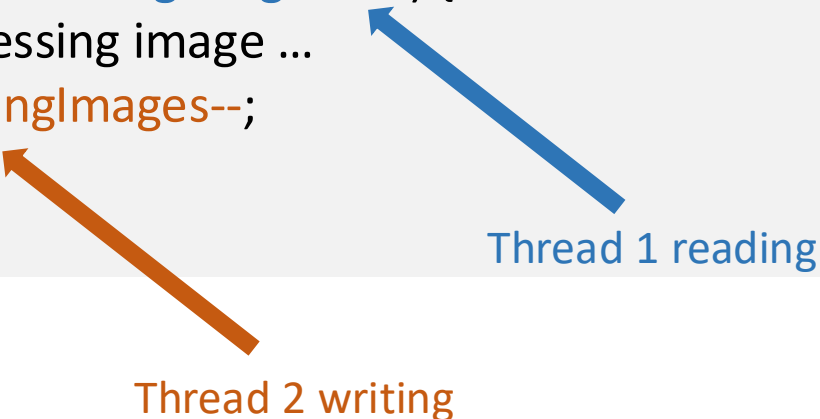


```
void process(size_t id, size_t& remainingImages,  
std::mutex& counterLock) {  
    while (true) {  
        counterLock.lock();  
        if (remainingImages == 0) {  
            return;  
        }  
        auto myimage = remainingImages--;  
        counterLock.unlock();  
        processImage(myimage);  
    }  
}
```

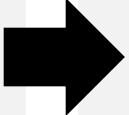
# Critical Section with `std::mutex`

- Create a `std::mutex` and pass it *by reference* (`std::mutex&`) to the thread
- The process function uses this lock to **protect the access** to `remainingImages`

```
void process(size_t id, size_t& remainingImages) {  
    while (remainingImages > 0) {  
        // processing image ...  
        remainingImages--;  
    }  
}
```



A blue arrow labeled "Thread 1 reading" points to the condition `remainingImages > 0`. An orange arrow labeled "Thread 2 writing" points to the decrement operation `remainingImages--`.

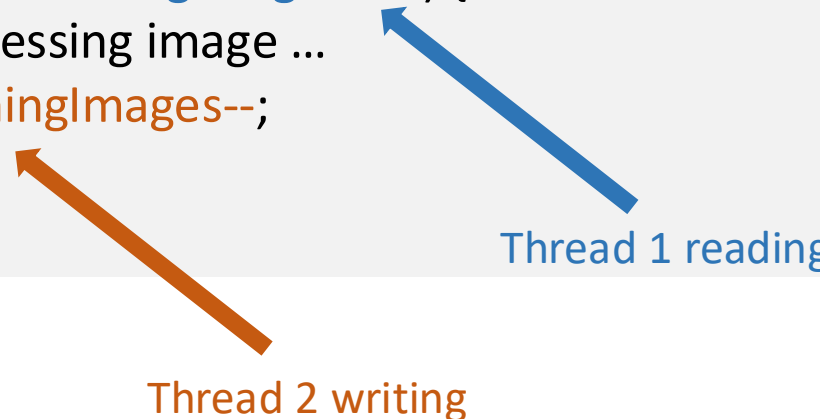


```
void process(size_t id, size_t& remainingImages,  
std::mutex& counterLock) {  
    while (true) {  
        counterLock.lock();  
        if (remainingImages == 0) {  
            counterLock.unlock();  
            return;  
        }  
        auto myimage = remainingImages--;  
        counterLock.unlock();  
        processImage(myimage);  
    }  
}
```

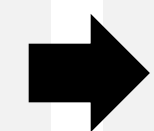
# Avoid Explicit Locking and Unlocking

- C++ provides a RAII-styled locker called `std::lock_guard` and `std::scoped_lock`
  - Acquire the lock on construction
  - Release the lock on destruction

```
void process(size_t id, size_t& remainingImages) {  
    while (remainingImages > 0) {  
        // processing image ...  
        remainingImages--;  
    }  
}
```



A blue arrow labeled "Thread 1 reading" points to the condition `remainingImages > 0`. An orange arrow labeled "Thread 2 writing" points to the decrement operation `remainingImages--`.



```
void process(size_t id, size_t& remainingImages,  
std::mutex& counterLock) {  
    while (true) {  
        { // RAII-styled lock  
            std::lock_guard lock(counterLock);  
            if (remainingImages == 0) {  
                return;  
            }  
            auto myimage = remainingImages--;  
        }  
        processImage(myimage);  
    }  
}
```

[Back to Basics: RAII in C++ - Andre Kostur - CppCon 2022](#)

# Problem that may Arise in Parallel Programming

- What if `processImage` returns an error?
  - If one thread has a SEGV the whole process will fail
- What if image processing time varies a lot (e.g, one image takes 100x longer than others)?
  - Need to figure out a way to estimate execution time for each image and try more intelligent scheduling
  - Perhaps, at the algorithm level, we need to further decompose `processImage` to independent tasks and assign more than one thread to run partitioned tasks in parallel
- What if there's a bug in your code, such that `processImage` enters an infinite loop?



# Some other Types of Mutex

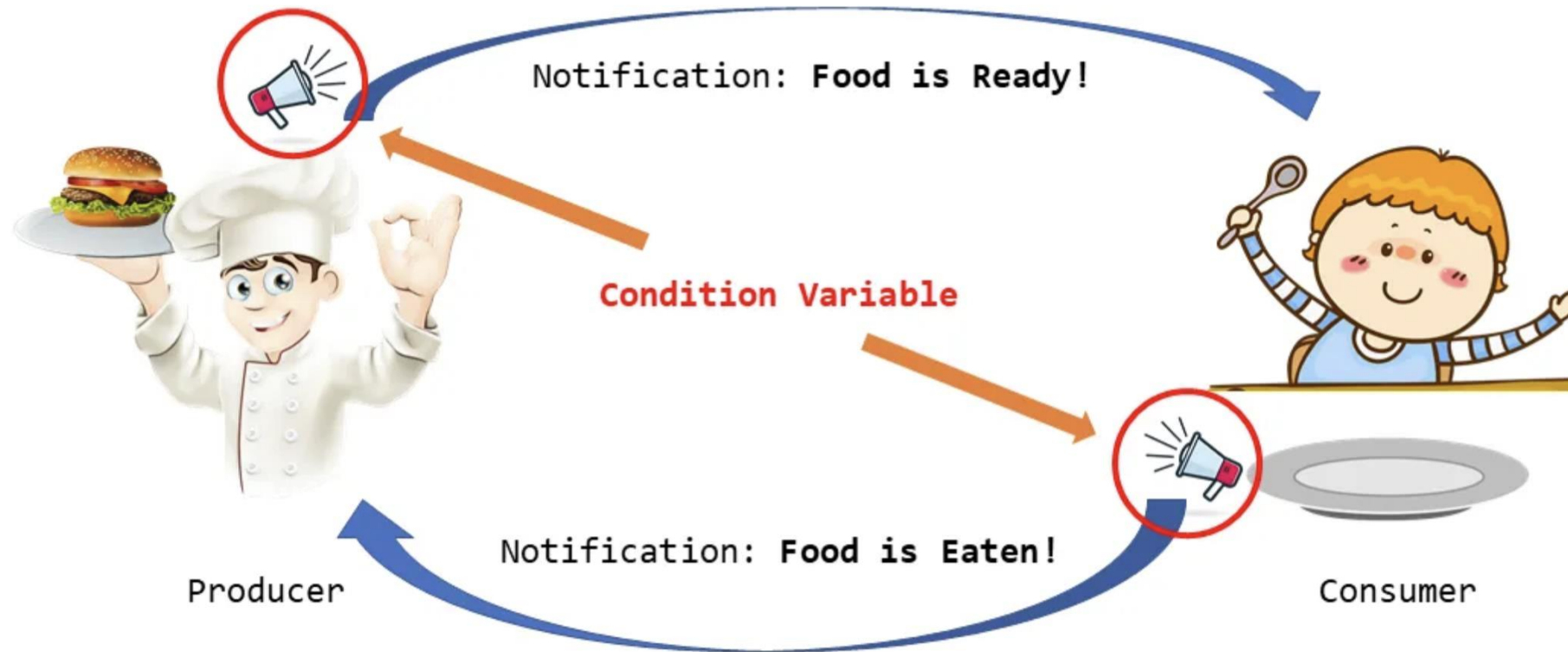
- `std::mutex`
  - A thread can lock the mutex only one time, and needs to unlock it to release it to other threads
  - A thread trying to lock the mutex more than one time will enter deadlock
    - The thread is waiting for itself to release the lock while trying to lock it
- `std::recursive_mutex`
  - A thread can lock the mutex multiple times, and needs to unlock it the same number of times to release it to other threads
- `std::timed_mutex`
  - A thread can `try_lock_for` / `try_lock_until`: if time elapses, don't take lock
  - Deadlocks if same thread tries to lock multiple times, like standard mutex
- In my experience, the standard `std::mutex` is good enough for most applications

# Impact of Parallelism on Cache

- How does this work with caches?
  - Each core typically has its own private caches (e.g., L1 and L2 caches)
    - Multiple cores can have their own views and copies of the same data variable
  - Caches need to be *coherent* -- if one core writes to a cache line that is also in another core's cache, the other core's cache line is invalidated:
    - This can become a performance problem – think about the contention of `remainingImages`
- Hardware provides optimized atomic memory operations, like compare and swap (CAS)
  - `atomic_add(var, 2)` – atomically add 2 to `var` from multiple threads
  - `CAS(source, target, new)` – If `source == target`, set `source` to `target` – *all atomic!*
  - Use this as a single bit to see if the lock is held and if not, take it
  - If the lock is held already, then enqueue yourself (in a thread safe way) and tell kernel to sleep you
  - When a node unlocks, it clears the bit and wakes up a thread

# Inter-thread Synchronization

- The producer-and-consumer pattern is a classical building block in parallel programming
  - One thread produces a task and another thread runs the produced task



- How can we enable the communication and synchronization between producer and consumer threads?

# Inter-thread Synchronization via `std::condition_variable`

- `std::condition_variable` is a synchronization primitive used with a `std::mutex` to block threads until other thread modifies the condition and notifies waiting threads
  - Often used to design producer-consumer algorithms in a multi-threaded environment
- The *producer* thread that intends to modify the shared variable must:
  - Acquire a `std::mutex`
  - Modify the shared variable while the lock is owned
  - Call `notify_one` or `notify_all` on the `std::condition_variable`
- The *consumer* thread that intends to wait on a `std::condition_variable` must:
  - Acquire a `std::unique_lock<std::mutex>` on the mutex used to protect the shared variable
  - Do one of the following:
    - Check the condition, in case it was already updated and notified
    - Wait on the `std::condition_variable`

# Example: Condition Variable

```
#include <condition_variable>

// mutex to block threads
std::mutex mtx;
std::condition_variable cv;
bool data_ready = false;

// producer function working as sender
void producer() {
    produce_work();
    {
        std::lock_guard<mutex> lock(mtx);
        data_ready = true;
    }
    // notify consumer when done
    cv.notify_one();
}
```

```
// consume what producer has produced
void consumer() {
    // locking
    {
        std::unique_lock<mutex> lock(mtx);
        // wait until the condition return true
        cv.wait(lock, [] () { return data_ready; });
    }
    consume_work();
}

int main() {
    std::thread consumer_thread(consumer);
    std::thread producer_thread(producer);
    consumer_thread.join();
    producer_thread.join();
    return 0;
}
```

# Pros and Cons of `std::thread`

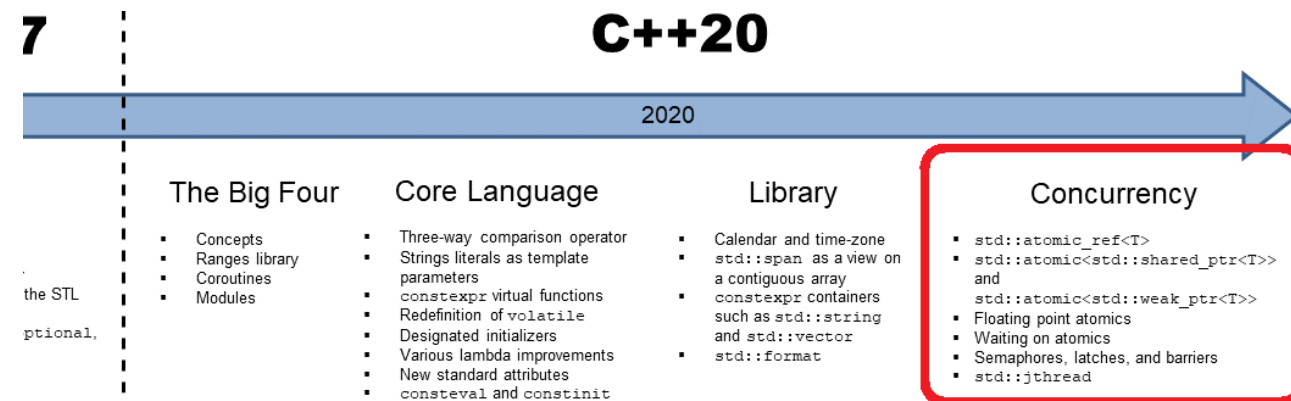
Advantages	Disadvantages
Sky is the limit – you have full control over thread	Everything is manual and explicit – low programming productivity of multi-threaded programming
Works well with mutexes and atomic operations to manage shared data	Requires manual synchronization, increasing risk of race conditions and deadlocks
Basic thread management is intuitive	Lacks higher-level features like thread pooling or task scheduling; more complex functionality requires libraries like <a href="#">Taskflow</a> , <a href="#">Boost</a> , <a href="#">TBB</a>
Efficient for parallel execution of tasks	For micro-tasks, overhead may outweigh benefits; better for medium-to-large workloads

# Other C++ Concurrency Primitives

- `std::jthread` – An RAII-styled thread (automatically joins at destruction)
- `std::latch` – A downward counter which can be used to synchronize threads
- `std::barrier` – A resettable downward counter to synchronize threads
- `std::future`/`std::promise` – A paired-object to design producer-consumer pattern
- `std::counting_semaphore` – A generalized, lightweight synchronization primitive
- `std::async` – A function to launch a task asynchronously



## [Back to Basics: C++ Concurrency - David Olsen - CppCon 2023](#)



# Taskflow: <https://taskflow.github.io/>

```
#include <taskflow/taskflow.hpp>
int main(){
    tf::Taskflow taskflow;
    tf::Executor executor;
    auto [A, B, C, D] = taskflow.emplace(
        [] () { std::cout << "TaskA\n"; },
        [] () { std::cout << "TaskB\n"; },
        [] () { std::cout << "TaskC\n"; },
        [] () { std::cout << "TaskD\n"; }
    );
    A.precede(B, C);
    D.succeed(B, C);
    executor.run(taskflow).wait();
    return 0;
}
```

