Introduction to Concurrency – Software and Hardware

Seminar in Concurrent Programming, Spring 2022

Based on Art of Multiprocessor Programming by Herlihy & Shavit

Reminders

- The different **threads** of a process share one memory space, and **synchronization** between the threads must be used to prevent interference between threads.
- A command that performs multiple actions (like a get and a set) as one uninterruptible operation is said to be **atomic**. Atomicity allows threads to protect their **critical sections** from other, interfering threads.

Reminders

- A **lock** is used to ensure atomicity. A lock can (usually) be acquired by one thread, and other threads that attempt to acquire the lock will be sent to wait until the thread that has the lock gives it away (unlocks).
- We can ensure atomicity with locks: lock before entering the critical section and unlock when it's done, that way only one thread can be in the critical section in any given time.

Software Implementations of Concurrency

- We will see implementations for concurrency in Java.
- If we have time, we will go over two additional programming languages: C# and C++.
- These languages handle concurrency differently but ultimately all three languages can be used for the same purposes.

Concurrency in Java

- Threads in Java are instances of the class **java.lang.Thread** or a subclass thereof. The constructor of this class receives a **Runnable** object, which can be implemented as a lambda expression (Java >= 8), as an anonymous class or as a class. This object specifies the function that will run for the thread.
- Example:

```
final String message = "Hello world from thread " + i;
thread = new Thread(new Runnable() {
  public void run() {
    System.out.println(message);
  }
});
```

Basic Threading

- Running threads is done by evoking the **start** method on a thread:

```
thread.start();
```

- Waiting for a thread to finish is done by evoking the **join** method.

```
thread.join();
```

“synchronized” methods

- Every object in Java has an implicit lock which can be used with the **synchronized** keyword.
- Only one thread can run a synchronized method for a given object at a given time. Other threads will wait for the first thread to finish when trying to call a synchronized method.
- The rest of the method is implemented naturally.

“synchronized” blocks

- The synchronized keyword can also be used to open a block that locks an object’s implicit lock at its start and unlocks it at its end:

```
synchronized(Singleton.class) {
  if (instance == null)
    instance = new Singleton();
}
```

- Here, the object used as the lock is the class object itself (unique to each class and used to identify the class’s fields and methods)

Dangers of “synchronized”

- Care should be taken when using **synchronized** to avoid deadlocks and other problematic situations.
- For example, a spinlock inside a synchronized method could cause a deadlock if the spinlock waits for a condition that can only be caused by a different synchronized method.
- One solution is using another way to unlock the implicit lock: **wait**.

“wait” and “notify”

- The wait method in Java unlocks a locked object and sends the thread to sleep (like a yield). When woken up, the thread must reacquire the lock.
- A thread can be woken up by the **notify** method. This method unlocks some thread that is currently waiting – there is no way to control which one. We can wake up all threads (eventually) using the **notifyAll** method.
- For example, a method removing an item from an empty queue can wait and be notified by the method adding an item to the queue.

“yield” and “sleep”

- In addition to wait and notify/notifyAll, there’s also **yield** and **sleep**
- yield – asks the scheduler to schedule something else (which it can ignore if there are no other threads).
- sleep – instructs the scheduler not to run the thread until the specified time elapses.

ThreadLocal

- Sometimes we would like that some memory will not be shared between the threads.
- This can be achieved using the **ThreadLocal<T>** class.
- Instances of ThreadLocal are not shared between threads. An important idiom using ThreadLocal is giving each thread a unique ID (see next slide).

Thread ID Idiom

- This is a common idiom using ThreadLocal: notice how nextID is shared but threadID is not.
- This way, a new thread can know how many threads came before, and still have its own unique threadID variable.

```
private static volatile int nextID = 0;
private static class ThreadLocalID extends ThreadLocal<Integer> {
  protected synchronized Integer initialValue() {
    return nextID++;
  }
}
private static ThreadLocalID threadID = new ThreadLocalID();
public static int get() {
  return threadID.get();
}
```

Randomization

- Randomization is another challenge in multithreading – the java.util.Random class, usually used in non-concurrent applications, can yield synchronization and predictable behavior when the same instance is used by multiple threads.
- The **ThreadLocalRandom** class has a separate random number generator for each thread, which can be accessed with ThreadLocalRandom.current().

Explicit Locks

- Threads can achieve mutual exclusion by using an implicit lock – a synchronized block, or by using an explicit lock – like the ones in java.util.concurrent.locks.

Atomic Variables

- There are also classes used to implement atomic variables directly, like AtomicInteger and AtomicReference<T>. These classes are found in the java.util.concurrent.atomic package.
- These classes have methods like **set** and **compareAndSet** that act like volatile writes and **get** which acts like a volatile read.

“final” fields

- In Java, fields defined with the **final** keyword cannot be changed. They are set in the constructor. However, if the constructor is improperly built, final fields can be observed to change.
- In order to prevent this and remove the need for synchronization for final fields, the “this” reference of an object must not be leaked in the constructor before it returns.

Conclusion - Software

- Threads in Java are implemented in java.lang.Thread.
- Every object in Java has a built-in implicit lock, which can be locked in a "synchronized" block. "wait" and "notify" allow further control of the lock.
- "yield" and "sleep" control scheduling of threads.
- ThreadLocal – non-shared memory.
- ThreadLocalRandom – java.lang.Random for concurrent applications

Conclusion - Software

- Explicit locks are found in java.util.concurrent.locks.
- AtomicReference – atomic class instances.
- Final fields don't require synchronization, if we don't leak "this" in the constructor.

Hardware Implementations of Concurrency

- We will now see some issues regarding hardware support for concurrency and we will discuss how this affects concurrent programming.
- Unlike single processor systems, efficient programming for multi-processors requires the programmer to know some details about the architecture of the system.

Motivation

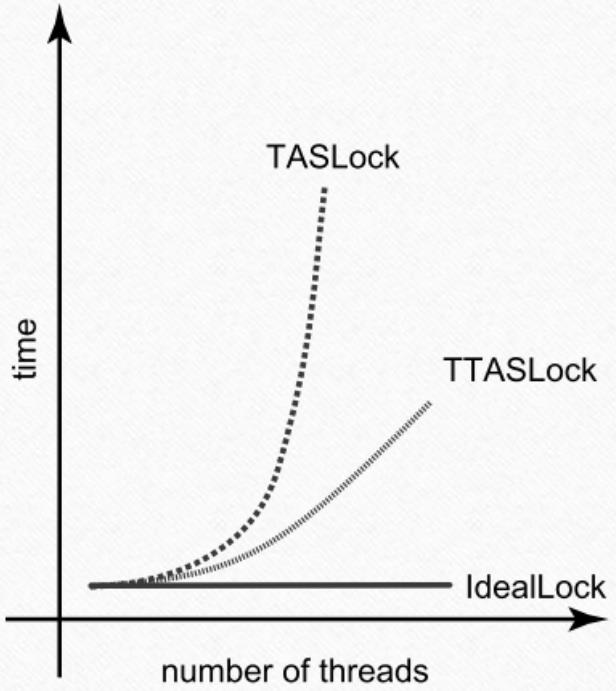
- Compare these two spin lock implementations:

```
public class TASLock implements Lock {
  ...
  public void lock() {
    while (state.getAndSet(true)) {} // spin
  }
  ...
}
```

```
public class TTASLock implements Lock {
  ...
  public void lock() {
    while (true) {
      while (state.get()) {}; // spin
      if (!state.getAndSet(true))
        return;
    }
  }
  ...
}
```

False Equivalence

- The two locks are not the same performance-wise:
- How could this be? They do essentially the same thing…
- In order to understand this, we need to take a closer look at memory.

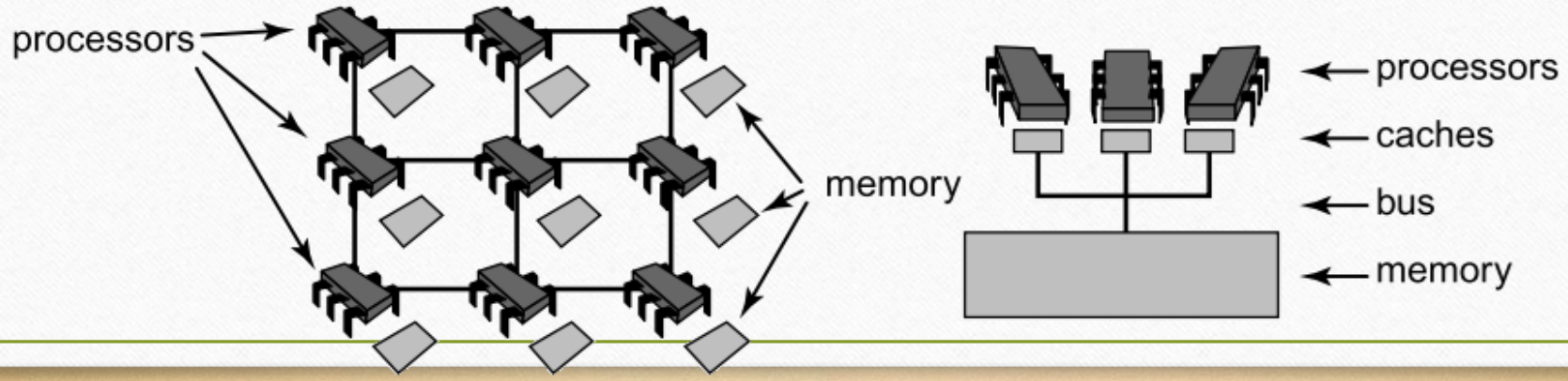


The Memory Pyramid

- Memory isn't just one chip – it is a whole hierarchy of progressively larger and slower components.
- The fastest components are used for **caching** – if a value is in the cache, we can save expensive memory access time.
- Caches work due to **locality**.

Architectures

- Processors in a system communicate using **interconnects**.
- SMP – each processor has a cache; the processors talk using a bus and share a memory chip.
- NUMA – each processor has its own memory; processors talk in a network



Contention

- **Sharing** or **memory contention** occurs when one processor tries to access a memory address cached by another processor.
- Problem: how do processors know if their cache is up to date?
- Solution: MESI protocol
- Bottom line: processors must invalidate each other's caching of a memory address when writing to that address.

MESI Protocol – Details

- Every cached block is in one of four states: Invalid, Exclusive, Shared, Modified.
- A block is in the Exclusive state if its value is valid, was not changed and is not cached on other processors.
- A block is in the Modified state if its value is valid, was changed and is not cached on other processors.

MESI Protocol – Details – Cont.

- A block is in the Shared state if its value is valid, was not changed and is cached on other processors.
- A block is in the Invalid state if its value is invalid.
- When writing to a block, a processor sends every other processor a message that invalidates that block's cache, changing its state to Invalid.

Spinning

- How bad is spinning on a multi-processor system?
- On an SMP architecture with no caches – very bad, the bus will be flooded with reads to memory.
- On a NUMA architecture with no caches – could be OK, if the address is in the local memory of the spinning processor.
- On any architecture with caches – not so bad, the processor just reads from its cache until it's invalidated (only two expensive memory reads).

Solution to the Puzzle

- Now, we can finally explain why the TTASLock is so much better than the TASLock.
- The TASLock always tries to write to memory, invalidating every other processor's cache and causing substantial traffic.
- The TTASLock tries to write to memory only when there is a chance it could stop spinning, producing no interconnect traffic.

Barriers

- Many processors have a write queue of data before it's written to memory.
- Write queues can create disparity between the expected value and the actual value in memory.
- Some processors have a **barrier** instruction that makes sure that all writes have been committed before continuing.

Synchronization Instructions

- Modern processor architectures must support synchronization primitives. Java and other languages use these primitives to ensure synchronization.
- CAS (Compare and Swap) – an instruction that receives a memory address and two values. It performs atomically:
  $\mathrm{CAS}(a, e, v)$ { if($^*a = e$) { $^*a = v$; return true } else return false }
- On Intel and AMD, this is called cmpxchg.

Synchronization Instructions – Cont.

- Another hardware synchronization primitive is LL/SC – load-linked and store-conditional.
- LL reads from an address, and SC tries to store a new value at an address. SC succeeds if the contents of the address have not changed since the last LL from the thread.
- LL/SC is supported by PowerPC, ARM, MIPS…

Efficiency Concerns

- Both LL/SC and CAS are about an order of magnitude slower than loads and stores.
- This is because, for example, LL/SC prevents out of order execution and includes a memory barrier.
- We should use atomic fields sparingly because they are often based on these expensive instructions.

Conclusion - Hardware

- Efficiency of various implementations of concurrent software is affected by the design of the system.
- SMP systems – one bus for all processors. NUMA systems – processors linked to each other.
- Writes to a shared memory location require all processors to invalidate their caches and therefore are significantly more expensive than reads.
- Barriers make sure that all cached writes are performed before the barrier.
- CAS and LL/SC are two synchronization instructions implemented in hardware and should be used sparingly.

If we have time… more PLs!

- We will now see additional software implementations of concurrency in two additional languages: C# and C++.
- The exact details are slightly different than they are in Java, but the essence is the same.

Concurrency in C#

- C#'s threading model is similar to Java's, and so we will not go into too much detail.
- Creating threads is done (like in Java) by instantiating the **System.Threading.Thread** class which can be done with an anonymous method specifying what the thread will run.
- Like in Java, threads must be started, which is done with the Start method. Threads can be waited for with the Join method.

Concurrency in C# - cont.

- Thread-local fields in C# are made simply by prefixing the field definition with the attribute [ThreadStatic].
- Example of ThreadID class in C#:

```
class ThreadID
{
    [ThreadStatic] static int myID;
    static int counter;
    public static int get()
    {
        if (myID == 0)
        {
            myID = Interlocked.Increment(ref counter);
        }
        return myID - 1;
    }
}
```

Concurrency in C# - cont.

- C# also has synchronized blocks, with the keyword “lock”:
- Unlike Java, the “lock” keyword cannot apply to a function but rather it always opens a block.
- In C# it is not possible to manually wait and notify on such a lock.

```
int GetAndIncrement()
{
    lock (this)
    {
        return value++;
    }
}
```

Monitors in C#

- In Java the "monitor" is always implicitly created, but in C# you must manually create it using the Monitor.Enter(this) and destroy it using Monitor.Exit(this).
- This semi-implicit lock can be waited on using Monitor.Wait(this) and notified using Monitor.Pulse(this) and Monitor.PulseAll(this).

```
public void Enq(T x)
{
    Monitor.Enter(this);
    try
    {
        while (tail - head == call.Length)
        {
            Monitor.Wait(this); // queue is full
        }
        calls[(tail++) % call.Length] = x;
        Monitor.Pulse(this);  // notify waiting dequeuers
    }
    finally
    {
        Monitor.Exit(this);
    }
}
```

Concurrency in C++

- C++ did not have native support for threads until version 11.
- Now, it has a concurrency model supporting threads, mutexes, atomic variables and condition variables.
- Threads in C++ are represented by **std::thread** objects. Constructing such an object also starts the thread (unlike in Java). Waiting for threads is done using the threads' **join** methods (like in Java).
- The constructor for std::thread can receive a lambda expression or a function pointer for the function that will run when the thread is started.

Locks in C++

- There are three most used types of locks in C++: **std::mutex**, **std::recursive_mutex**, **std::shared_mutex**.
- std::mutex has lock and unlock methods that do as you would expect, and also a try_lock method that tries to unlock the mutex but doesn't block the thread if it fails.
- std::recursive_mutex is similar to a regular mutex, but a thread already holding the lock can call the lock method again, and when unlocking the mutex, the thread will have to unlock it as many times as it locked it.

RAII and lock_guard

- C++ does not have “finally” blocks that can release a lock after a specified block completes and before anything else is done. However, it does have an idiom called RAII – “resource acquisition is initialization”.
- Using this idiom, constructors are like locking and destructors are like unlocking.
- We can use the class **std::lock_guard** – it will automatically lock the mutex when initialized and unlock it when destructed (example in the next slide).

lock_guard example

- The code opens a block with an initialization of a lock_guard guarding a mutex – this locks it.
- At the end of the block, the lock_guard is destructed, unlocking the mutex.

```
std::mutex m;
...
{
  std::lock_guard<std::mutex> g(m);
  // mutex m is locked
  if (i == 9)
    return; // releases m because g destructs
  f();
  // releases m because g destructs
}
```

Condition Variables

- C++ supports condition variables by using std::condition_variable. A condition variable can be made to wait on a condition by using the **wait** method on the variable (which accepts a function pointer or lambda and an associated mutex).
- Threads can ask another thread to check the value of a condition variable using the **notify_one** or **notify_all** methods on the variable.

Thread Local in C++

- In C++, a variable may have the thread_local modifier, which specifies that each thread should write and read to a separate instance of the variable.
- This is similar to ThreadLocal in Java, but here it is implemented as a keyword in the language.