Mutual Exclusion

CS 475, Spring 2019
Concurrent & Distributed Systems

With material from Herlihy & Shavit, Art of Multiprocessor Programming
Review: Processes vs Threads

• Context Switching
  • Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).
  • When switching processes, **all** of that data needs to get flushed out (by the OS)
  • Threads share the same address space: no need to do this switch
Review: Processes vs Threads

• How threads and processes are similar
  • Each has its own logical control flow.
  • Each can run concurrently.
  • Each is context switched.

• How threads and processes are different
  • Threads share code and data, processes (typically) do not.
  • Threads are somewhat less expensive than processes.
  • Process control (creating and reaping) is (ballpark!) twice as expensive as thread control.
Review: Threads in Java

```java
public static void main(String[] args) throws InterruptedException {
    Thread t = new Thread(
        new Runnable() {
            @Override
            public void run() {
                //This code will now run in a new thread
                System.out.println("Hello from the thread!");
            }
        }
    );
    t.start();
    System.out.println("Hello from main!");
    t.join();
}
```

What is the output of this code?

- #1 Hello from the thread!
  Hello from main!

- #2 Hello from main!
  Hello from the thread!

This is a race condition
Review: Splitting up the work

• The problem: What if we have thousands of tasks to do simultaneously, should we make a new thread for each?
  • No (lots of overhead, probably too many threads)
• The answer: think about work as tasks and not threads
  • Threads will magically appear to do your tasks
• Tasks -> Runnable and Callable objects
  • ExecutorService handles taking tasks and running them
Today

• Mutual exclusion, formally
• Definitions for time, ordering
• Locks, and locking algorithms
• Reading: H&S 2.1-2.3
• Note: HW1 posted: https://www.jonbell.net/gmu-cs-475-spring-2019/homework-1/
Why is Concurrent Programming so Hard?

- Try preparing a seven-course banquet
  - By yourself
  - With one friend
  - With twenty-seven friends …
- Before we can talk about programs
  - Need a language
  - Describing time and concurrency
• “Absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external.” (I. Newton, 1689)

• “Time is, like, Nature’s way of making sure that everything doesn’t happen all at once.” (Anonymous, circa 1968)
Events

- An event $a_0$ of thread A is
  - Instantaneous
  - No simultaneous events (break ties)
Threads

- A thread $A$ is (formally) a sequence $a_0, a_1, ...$ of events
  - “Trace” model
  - Notation: $a_0 \rightarrow a_1$ indicates order
Example Thread Events

• Assign to shared variable
• Assign to local variable
• Invoke method
• Return from method
• Lots of other things …
Threads are State Machines

Events are transitions
States

- Thread State
  - Program counter
  - Local variables
- System state
  - Object fields (shared variables)
  - Union of thread states
Concurrency

• Thread A
Concurrency

• Thread A

• Thread B
Interleavings

• Events of two or more threads
  – Interleaved
  – Not necessarily independent (why?)
Intervals

- An interval $A_0 = (a_0, a_1)$ is
  - Time between events $a_0$ and $a_1$
Intervals may Overlap

\[
\begin{align*}
A_0 &\quad a_0 & a_1 & B_0 & b_0 & b_1 \\
\end{align*}
\]
Intervals may be Disjoint

\[ a_0 \quad A_0 \quad a_1 \]

\[ b_0 \quad B_0 \quad b_1 \]

\[ \text{time} \]
Interval $A_0$ precedes interval $B_0$
• Notation: $A_0 \rightarrow B_0$

• Formally,
  - End event of $A_0$ before start event of $B_0$
  - Also called “happens before” or “precedes”
• Remark: $A_0 \rightarrow B_0$ is just like saying
  - 1066 AD $\rightarrow$ 1492 AD,
  - Middle Ages $\rightarrow$ Renaissance,
• Oh wait,
  - what about this week vs this month?
Precedence Ordering

- Never true that $A \rightarrow A$
- If $A \rightarrow B$ then not true that $B \rightarrow A$
- If $A \rightarrow B \land B \rightarrow C$ then $A \rightarrow C$
- Funny thing: $A \rightarrow B \land B \rightarrow A$ might both be false!
Partial Orders
(you may know this already)

• **Irreflexive:**
  - Never true that \( A \rightarrow A \)

• **Antisymmetric:**
  - If \( A \rightarrow B \) then not true that \( B \rightarrow A \)

• **Transitive:**
  - If \( A \rightarrow B \) & \( B \rightarrow C \) then \( A \rightarrow C \)
Total Orders
(you may know this already)

• Also
  - Irreflexive
  - Antisymmetric
  - Transitive

• Except that for every distinct A, B,
  - Either $A \to B$ or $B \to A$
Repeated Events

while (mumble) {
    a_0; a_1;
}

k-th occurrence of event $a_0$

$k$-th occurrence of interval $A_0 = (a_0, a_1)$
Implementing a Counter

public class Counter {
    private long value;

    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}

Make these steps indivisible using locks
Locks (Mutual Exclusion)

public interface Lock {
    public void lock();
    public void unlock();
}
Locks (Mutual Exclusion)

public interface Lock {
  public void lock();
  public void unlock();
}

acquire lock
Locks (Mutual Exclusion)

public interface Lock {
    public void lock();
    public void unlock();
}

acquire lock
release lock
Using Locks

```java
public class Counter {
    private long value;
    private Lock lock;
    public long getAndIncrement() {
        lock.lock();
        try {
            int temp = value;
            value = value + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
```
Using Locks

```java
public class Counter {
    private long value;
    private Lock lock;
    public long getAndIncrement() {
        lock.lock();
        try {
            int temp = value;
            value = value + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
```
public class Counter {
    private long value;
    private Lock lock;
    public long getAndIncrement() {
        lock.lock();
        try {
            int temp = value;
            value = value + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
Using Locks

```java
public class Counter {
    private long value;
    private Lock lock;
    public long getAndIncrement() {
        lock.lock();
        try {
            int temp = value;
            value = value + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
```

Critical section
Mutual Exclusion, Formally

- Let $CS_i^k \leftarrow \text{be thread i’s k-th critical section execution}$
Mutual Exclusion, Formally

- Let $CS_i^k$ be thread i’s $k$-th critical section execution
- And $CS_j^m$ be thread j’s $m$-th critical section execution
Mutual Exclusion, Formally

- Let $CS_i^k$ be thread $i$’s $k$-th critical section execution
- And $CS_j^m$ be thread $j$’s $m$-th execution
- Then either
  - $\rightarrow$ $\rightarrow$ $\rightarrow$ or $\rightarrow$ $\rightarrow$ $\rightarrow$
Mutual Exclusion, Formally

• Let $CS_i^k$ be thread i’s k-th critical section execution
• And $CS_j^m$ be thread j’s m-th execution
• Then either
  - $CS_i^k \rightarrow CS_j^m$ or $CS_j^m \rightarrow CS_i^k$
Mutual Exclusion, Formally

- Let $CS_i^k \leftarrow \rightarrow$ be thread i’s k-th critical section execution
- And $CS_j^m \leftarrow \rightarrow$ be thread j’s m-th execution
- Then either
  - $CS_i^k \rightarrow CS_j^m$
  - $CS_j^m \rightarrow CS_i^k$

Aka: it is guaranteed that one critical section happens before the other
Deadlock-Free

• If some thread calls \texttt{lock()}
  – And never returns
  – Then other threads must complete \texttt{lock()} and \texttt{unlock()} calls infinitely often

• System as a whole makes progress
  – Even if individuals starve
Starvation-Free

• If some thread calls lock()
  – It will eventually return
• Individual threads make progress
Locking in Java

• Most locks are reentrant: if you hold it, and ask for it again, you don’t have to wait (because you already have it)

• Basic primitives:
  • synchronized{}

• Plus…
  • Lock API... lock.lock(), lock.unlock()

• The preferred way
Live programming: Locks
Synchronized methods in Java

```java
public synchronized static void increment()
{
    i = i + 1;
}
```

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()`.
**Synchronized methods in Java**

```java
public synchronized static void increment()
{
    i = i + 1;
}
```

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()`

```java
public synchronized static void incrementOther()
{
    j = j + 1;
}
```

Result: Before entering `incrementOther()`, thread gets a lock on the Class object of `incrementOther()`

Problem?
Synchronized blocks in Java

• Can also use *any* object as that monitor

```java
static Object someObject = new Object();
public static void increment()
{
    synchronized(someObject){
        i = i + 1;
    }
}

static Object someOtherObject = new Object();
public static void incrementOther()
{
    synchronized(someOtherObject){
        j = j + 1;
    }
}
```

Now, two different threads could call `increment()` and `incrementOther()` at the same time.
Live programming: Synchronized
Implementing Locks: Peterson’s Algorithm

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}

public void unlock() {
    flag[i] = false;
}
```
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {
    }
}

public void unlock() {
    flag[i] = false;
}
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}
public void unlock() {
    flag[i] = false;
}
Peterson's Algorithm

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}

public void unlock() {
    flag[i] = false;
}

```

- **Announce I'm interested**
- **Defer to other**
- **Wait while other interested & I'm the victim**
Peterson’s Algorithm

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}

public void unlock() {
    flag[i] = false;
}
```

- **Announce I’m interested**
- **Defer to other**
- **Wait while other interested & I’m the victim**
- **No longer interested**
Peterson’s Alg: Mutual Exclusion

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}
```

- If thread 1 in critical section,
  - `flag[1]=true`,
  - `victim = 0`

- If thread 0 in critical section,
  - `flag[0]=true`,
  - `victim = 1`

Cannot both be true, hence yes: it is safe!
Peterson’s Alg: Deadlock Free

```java
public void lock() {
    ... 
    while (flag[j] && victim == i) {};
}
```

- Thread blocked
  - only at `while` loop
  - only if it is the victim
- One or the other must not be the victim
Peterson's Alg: Starvation Free

- Thread $i$ blocked only if $j$ repeatedly re-enters so that
  \[ \text{flag}[j] == \text{true} \quad \text{and} \quad \text{victim} == i \]
- When $j$ re-enters
  - it sets victim to $j$.
  - So $i$ gets in

```java
public void lock() {
    \text{flag}[i] = \text{true};
    \text{victim} = i;
    \text{while} (\text{flag}[j] && \text{victim} == i) \{ \}
}

public void unlock() {
    \text{flag}[i] = \text{false};
}
```
Locking Granularity

- BIG design question in writing concurrent programs: how many locks should you have?
- Example: Distributed filesystem
  - It would be *correct* to block all clients from reading *any* file, when one client writes a file
  - However, this would not be performant at all!
  - It would be much better to instead lock on *individual files*
  - *More locks* -> *more complicated semantics and tricky to avoid deadlocks, races*
Amdahl's Law

• Identifies performance gains from adding additional cores to an application that has both serial and parallel components
• $S$ is serial portion
• $N$ processing cores
• That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
• As $N$ approaches infinity, speedup approaches $1 / S$
• Serial portion of an application has disproportionate effect on performance gained by adding additional cores
Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 2.17 = \frac{1}{1 - 0.6 + \frac{0.6}{10}}
\]
Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}
\]
Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = \frac{1}{1 - 0.9 + \frac{0.9}{10}} = 5.26
\]
Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}
\]
The Moral

• Making good use of our multiple processors (cores) means
• Finding ways to effectively parallelize our code
  – Minimize sequential parts
  – Reduce idle time in which threads *wait* without
  – This will be a constant theme throughout the course!
This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.