Review

Last week

- Exceptions
Outline

- Concurrency
- Discussion of Final

Sources for today’s lecture:

PLP, 12
Barrett. Lecture notes, Fall 2008.
Lecture notes for CMU’s 15-440, Fall 2009: http://www.andrew.cmu.edu/course/15-440-sp09/
Concurrent

So far, we have focused on sequential programs.

But many programs are, in fact, *concurrent* and consist of more than one active execution context.

We call such an execution context a *thread*.

**Reasons for concurrency:**

- Reflection of logical structure of problem: many programs must keep track of more than one independent task at the same time
- To interact with multiple independent physical devices
- To increase performance

Note that we want concurrency even with a single processor core, e.g., to capture application domain, to overlap computation with I/O, to support multi-tasking, and so on. We absolutely must have it for multi- and many-core processors, which likely will be the most important source of performance increases for the foreseeable future.
Granularity of Concurrency

At the instruction level:

- Explicit through SIMD instructions, which apply a *Single Instruction on Multiple Data*.
- Implicit in superscalar core, which has duplicated execution units (e.g., several integer arithmetic units) and thus can perform several operations at the same time.

At the program level:

- Explicit with threads that coordinate through shared memory (Ada, Java, C#).
- Explicit with threads that are isolated from each other and coordinate through message passing (Erlang).
- Implicit through SIMD-like programming languages like APL.

Many older languages such as C, C++, or Fortran do not have built-in support for concurrency at all, but instead rely on libraries.
Creating Threads in JAVA

Implement Runnable:

```java
public class Worker implements Runnable {
    public void run() {
        // Do the real work here.
    }
}
```

```java
public void pullTrigger() {
    (new Thread(new Worker())).start();
}
```

The constructor invocation creates the new thread and the invocation of `start` begins its execution. The implementation of `Thread.start` internally calls the `Runnable`'s `run` method.
Extend Thread:

```java
public class Worker extends Thread {
    public void run() {
        // Do the real work here.
    }
}

public void pullTrigger() {
    (new Worker()).start();
}
```

The constructor invocation creates the new thread and the invocation of `start` begins its execution. The implementation of `Thread.start` internally calls the overridden `run` method.
Creating Threads in JAVA (iii)

Creating threads on demand takes time (and memory) and is wasteful if an repeatedly application creates and terminates threads (think servers).

A better solution is to pre-allocate some number of threads and parcel out tasks to threads that are currently not busy. This service is provided by a thread pool.

Before JAVA 1.5, developers had to write their own thread pool implementations, which can be rather tricky (believe me, I have done that). As of JAVA 1.5, support for so-called “executors” is provided in the java.util.concurrent package.
More Ways to Create Threads

Languages and libraries provide a variety of different ways to create new threads

- **Co-Begin**: A list of program blocks that are allowed to run in parallel
- **Parallel Loops**: Loops in which multiple iterations are allowed to run at the same time
- **Launch at Elaboration**: Threads declared with syntax similar to subroutines. Automatically created when the program runs.
- **Fork**: General mechanism to create a new thread
- **Join**: Wait until a thread created with `fork` is finished
- **Implicit receipt**: New thread created automatically in response to a request (on a server, for example)
- **Early Reply**: Like a procedure call, except subroutine executes `reply` instead of `return` at which point both the caller and callee continue in parallel
Each thread virtualizes a processor core. When the thread is actively executing, it is progressing in wall-clock time. When it is not executing, it has no way of observing that it is asleep — after all, it is not executing.

A thread can voluntarily give up its processor core:

```java
// Get the current thread.
Thread t = Thread.currentThread();

// Give up.
t.yield();
```

A little more conservatively, a thread can give up its core for a limited time:

```java
Thread t = Thread.currentThread();

// Bite the apple.
try {
    t.sleep(1000); // One second.
} catch (InterruptedException x) {
    // Hello Prince.
}
```
public class Account {
    private long balance;
    public void withdraw(long amount) throws Overdraft {
        if (amount > balance) throw new Overdraft();
        balance -= amount;
    }
    ...
}

Let's start with a balance of $1,000 in account checking and two threads A and B. What could possibly go wrong?
Race Conditions (ii)

```java
public class Account {
    private long balance;
    public void withdraw(long amount) throws Overdraft {
        if (amount > balance) throw new Overdraft();
        balance -= amount;
    }
}
```

Let's start with a balance of $1,000 in account checking and two threads A and B. What could possibly go wrong?

- A: call `checking.withdraw(800)`
- B: call `checking.withdraw(800)`
- A: pass balance check in line 4
- B: pass balance check in line 4
- A: update balance in line 5
- B: update balance in line 5

The remaining balance is $(600)$. Oops.
Race Conditions (iii)

1 public class Account {
2     private long balance;
3     public void withdraw(long amount) throws Overdraft {
4         if (amount > balance) throw new Overdraft();
5         balance -= amount;
6     }
7     ...
8 }

Let’s start with a balance of $1,000 in account checking and two threads A and B. What could possibly go wrong?

Even more insidiously, the statement on line 5 really is one memory read to get value of balance, one subtraction to compute the difference, and one memory write to update the value of balance. Consequently, the following thread interleaving is possible:

- A: read balance
- B: read balance
- A: compute and write new balance
- B: compute and write new balance

Thread A’s withdrawal succeeded but is not reflected in the account’s state at all. Double oops.
Memory Consistency Model

Modern processors have several levels of caches. Furthermore, some caches may be per-core and some caches may be shared across cores on a processor, but never across different processors. Ensuring that all processor cores see the same memory contents at all times requires consistency traffic between caches and memory.

The illusion of perfect order is called sequential consistency. I.e., all writes become visible to all cores in the same order and all of a core’s writes become visible in the performed order.

Sequential consistency is intuitive but expensive to maintain. Consequently, most modern architectures feature more relaxed consistency models and require explicit synchronization instructions to enforce ordering.

As a result, memory races can be very very tricky and may even differ across architectures.
Monitors

A monitor is an object that ensures \textit{mutual exclusion} for its methods, i.e., only one thread can execute a method at a given time.

In Java, every object has an implicit monitor. To guarantee mutual exclusion, you simply annotate a method with the \texttt{synchronized} modifier:

\begin{verbatim}
public class Account {
    private long balance;

    public synchronized void withdraw(long amount) throws Overdraft {
        if (amount > balance) throw new Overdraft();
        balance -= amount;
    }

    ...
}
\end{verbatim}

Now every withdrawal is guaranteed to execute \textit{atomically}, as one indivisible unit.
Synchronized Blocks

To provide more flexibility in using monitors, Java also supports synchronized blocks, which explicitly specify the monitor object:

```java
synchronized (monitor) {
    ...
}
```

Again, every object implicitly has a monitor and thus can be used to ensure mutual exclusion in a synchronized block.

Integrating support for synchronized methods and blocks into the language is a Good Thing™. Both can be implemented with a `lock`, which is acquired before entering the method or block and which is released when exiting the method or block. By only supporting synchronized methods and blocks, the implementation can automatically ensure that all lock acquire and release operations are always well-matched — even when exiting the method or block through an exception.
The Trouble with JAVA Monitors

A monitor object always uses itself to synchronize methods.

A synchronized block, however, can use any JAVA object, including an object that has synchronized methods.

Consequently, an ill-behaved program can synchronize on a monitor and then loop forever, thus preventing the monitor from ever executing synchronized methods again.

The problem is a lack of encapsulation for JAVA monitors. Instead of using synchronized methods, it is cleaner to use an explicit and private lock object:

```java
public class Monitor {
    private Object lock = new Object();

    public void exampleMethod(...) {
        synchronized (lock) {
            ...
        }
    }

    ...
}
```
Lock Everything?

So, if race conditions are such a danger for concurrent programs, why not protect all data and operations with locks, locks, and more locks?

- Locks can be rather expensive, since each synchronized method or block must acquire the underlying lock before executing any code and then release that lock again before exiting from the method or block (and do that even when exiting through an exception).

- Locks easily lead to deadlock...
public void m1() {
    synchronized (l1) { ...
        synchronized (l2) {
            ...
        }
    }
}

public void m2() {
    synchronized (l2) { ...
        synchronized (l1) {
            ...
        }
    }
}

Consider two threads A and B. Thread A calls m1() and thread B calls m2(). What could possibly go wrong?


Consider two threads A and B. Thread A calls \texttt{m1()} and thread B calls \texttt{m2()}. What could possibly go wrong?

I'm glad you asked: A:1, A:2, B:8, B:9, and then?
Avoiding Deadlock

Some options:

- Follow well-established algorithms and heuristics, such as using only one lock at a time or always acquiring locks in the same order.

- Switch to transactional memory, which replaces explicit locks with atomic blocks. All code in an atomic block either commits, i.e., executes as one indivisible operation, or aborts, i.e., appears not to have executed at all. These semantics support arbitrary nesting of atomic blocks, thus eliminating deadlock.

But:

- Deadlock avoidance algorithms and heuristics can be hard to follow and may simply not apply at all.

- Transactional memory is not yet available in mainstream languages. Furthermore, transactional memory implementations need to be able to undo all operations of an atomic block, which is impossible for I/O.
Condition Variables

Sometimes, concurrent code needs to wait for a condition to become true. For example, a queue’s take operation requires the queue to contain at least one element.

One solution is to poll: repeatedly perform the condition’s test in a loop. But polling means that a processor core is busy without doing much useful work. This is fine for short durations, but not such a great idea when polling takes a while. Furthermore, polling code must not hold any locks, making it harder to get right.

A better solution are condition variables, which support three key operations:

- **wait** makes the current thread wait, optionally with a timeout.

- **notify** wakes up some already waiting thread.

- **notifyAll** wakes up all waiting threads.

Every condition variable has an associated lock, which must be held when invoking either of the three operations. Each condition variable operation, in turn, temporarily relinquishes the lock so that other threads can make progress.

Each JAVA object not only has an implicit monitor but also an implicit condition variable.
Condition Variables in JAVA

public class Queue<E> {

    ...

    public E take() {
        synchronized (cond) {
            while (isEmpty()) {
                try {
                    cond.wait();
                } catch (InterruptedException e) {}  
            }
        }
        ... // Remove element from internal storage.
    }

    public void give(E e) {
        synchronized (cond) {
            ... // Add e to internal storage.
            cond.notify();
        }
    }

}

Note that condition variables really are library support for concurrency, at least in JAVA.
JAVA Concurrency in One Slide

Three “simple” facts of concurrent life:

- Every object is a monitor and supports **synchronized** methods
- Every object is a lock and supports **synchronized** blocks
- Every object also is a condition variable supporting **wait**, **notify**, and **notifyAll**

JAVA 1.5 adds many more options through the `java.util.concurrent`, `java.util.concurrent.atomic`, and `java.util.concurrent.locks` packages.

Other design points are certainly possible; see, for example, Clark Barrett’s slides from the Fall 2008 for (more than you ever wanted to know about) ADA.

Also, programming with threads is inherently hard: race conditions, deadlock, livelock, priority inversion, and so on are a fact of concurrent life.
Implementing Threads

A minimal threading implementation needs per-thread storage for registers etc. This is typically called the *thread control block*.

It also needs to know which thread is currently running and which threads are ready to run, i.e., the ready queue.

Finally, each condition variable needs a queue of threads that are waiting on that condition.

The *scheduler* then uses these data structures to save the old thread’s state, to select which thread to run next, and to restore the new thread’s state.

But: We still need locks...
Implementing locks with memory read and write instructions is possible but complex and slow. More powerful atomic instructions help:

- **int** TAS(<location>) {
  
  if (*<location> == 0) {
    *<location> = 1;
    return 0;
  } else {
    return 1;
  }
}

- **int** CAS(<location>, <expected>, <updated>) {
  
  if (*<location> == <expected>) {
    *<location> = <updated>;
    return 0;
  } else {
    return 1;
  }
}

Note that \(\text{TAS}(x) == \text{CAS}(x, 0, 1)\).
Implementing a Lock

With CAS, implementing a lock becomes straight-forward:

```c
void acquire(<lock>) {
    while (CAS(<lock>, 1, 0)) ;
}

void release(<lock>) {
    *<lock> = 1;
}
```

Voilà, we have a *spin lock*.

Note that we can avoid all this complexity if our threading implementation uses *cooperative scheduling* instead of *preemption*. Under the latter, a hardware interrupt periodically invokes the scheduler, hence we need to use atomic instructions. Under the former, each thread periodically calls into the scheduler. That simplifies matters (no possibility of interruption) but also complicates matters (developers need to write well-behaved code).