Threads and Synchronization

(thanks to Doug Lea for some slides)

Thread Scheduling

• Once a new thread is created, how does it interact with existing threads?

• This is a question of scheduling:
  – Given \( N \) processors and \( M \) threads, which thread(s) should be run at any given time?
Thread Scheduling

- OS schedules a single-threaded process on a single processor
- Multithreaded process scheduling:
  - One thread per processor
    - Effectively splits a process across CPU’s
    - Exploits hardware-level concurrency
  - Many threads per processor
    - Need to share CPU in slices of time

Scheduling Example (1)

CPU 1
- p1
- p2

CPU 2
- p1
- p2

One process per CPU

p2 threads: blue
p1 threads: green
Scheduling Example (2)

Threads shared between CPU’s

CPU 1
- p1
- p2

CPU 2
- p1
- p2

Scheduling Consequences

- Concurrency
  - Different threads from the same application can be running at the same time on different processors
- Interleaving
  - Threads can be pre-empted at any time in order to schedule other threads
Thread Scheduling

- When multiple threads share a CPU, must decide:
  - When the current thread should stop running
  - What thread to run next
- A thread can voluntarily yield() the CPU
  - Call to yield may be ignored; don’t depend on it
- Preemptive schedulers can de-schedule the current thread at any time
  - Not all JVMs use preemptive scheduling, so a thread stuck in a loop may never yield by itself. Therefore, put yield() into loops
- Threads are de-scheduled whenever they block (e.g., on a lock or on I/O) or go to sleep

Thread Lifecycle

- While a thread executes, it goes through a number of different phases
  - New: created but not yet started
  - Runnable: is running, or can run on a free CPU
  - Blocked: waiting for I/O or on a lock
  - Sleeping: paused for a user-specified interval
  - Terminated: completed
Which Thread to Run Next?

- The scheduler looks at all of the runnable threads, including threads that were unblocked because
  - A lock was released
  - I/O became available
  - They finished sleeping, etc.
- Of these threads, it considers the thread’s priority. This can be set with `setPriority()`. Higher priority threads get preference.
  - Oftentimes, threads waiting for I/O are also preferred

Simple Thread Methods

- void start()
- boolean isAlive()
- void setPriority(int newPriority)
  - Thread scheduler might respect priority
- void join() throws InterruptedException
  - Waits for a thread to die/finish
Example: Threaded, Sync Alarm

```java
while (true) {
    System.out.print("Alarm\> ");

    // read user input
    String line = b.readLine();
    parseInput(line);

    // wait (in secs) asynchronously
    if (m != null) {
        // start alarm thread
        Thread t = new AlarmThread(m,tm);
        t.start();
        // wait for the thread to complete
        t.join();
    }
}
```

Simple Static Thread Methods

- `void yield()`
  - Give up the CPU
- `void sleep(long milliseconds)`
  - Sleep for the given period
  - Throws `InterruptedException`
- `Thread currentThread()`
  - Thread object for currently executing thread
- All apply to thread invoking the method
Daemon Threads

- void setDaemon(boolean on)
  - Marks thread as a daemon thread
  - Must be set before thread started
- By default, thread acquires status of thread that spawned it
- Program execution terminates when no threads running except daemons

Concurrency Issues

- Threads allow concurrent activities, which can be both good and bad!
- Two opposing design forces
  - **Safety**: “Nothing bad ever happens”
  - **Liveness**: “Something (useful) eventually happens”
- A safe system may not be live and a live system may not be safe. Balance is key.
Safe Objects

- Perform actions only when in consistent states
  - Don’t want one thread to access an object while another thread is modifying its internal state.

- This boils down to ensuring *object invariants* in the face of concurrent access.

Violating Safety

- Data can be shared by threads
  - Scheduler can interleave or overlap threads arbitrarily
  - Can lead to *interference*
    - Storage corruption (e.g., a *data race/race condition*)
    - Violation of representation invariant
    - Violation of a protocol (e.g., A occurs before B)
public class Example extends Thread {
    private static int cnt = 0;  // shared state
    public void run() {
        int y = cnt;
        cnt = y + 1;
    }
    public static void main(String args[]) {
        Thread t1 = new Example();
        Thread t2 = new Example();
        t1.start();
        t2.start();
    }
}

Data Race Example

static int cnt = 0;  // Shared state  cnt = 0

t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}

Start: both threads ready to run. Each will increment the global count.
Data Race Example

static int cnt = 0;  
*t1.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}* 

*t2.run() {
    int y = cnt;
    cnt = y + 1;
}* 

38

Data Race Example

static int cnt = 0;  
*t1.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}* 

*t2.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}* 

39

T1 executes, grabbing the global counter value into y.

T1 is pre-empted. T2 executes, grabbing the global counter value into y.
Data Race Example

```java
static int cnt = 0;  // Shared state   cnt = 1

t1.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}

t2.run() {
    int y = cnt;  y = 0
    cnt = y + 1;  // T2 executes, storing the incremented cnt value.
}
```

Data Race Example

```java
static int cnt = 0;  // Shared state   cnt = 1

t1.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}

t2.run() {
    int y = cnt;  y = 0
    cnt = y + 1;  // T2 completes.  T1 executes again, storing the old counter value (1) rather than the new one (2)!
}
```
But When I Run it Again?

Data Race Example

```java
static int cnt = 0;

t1.run() {
    int y = cnt;
    cnt = y + 1;
}

t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Start: both threads ready to run. Each will increment the global count.

Shared state cnt = 0
Data Race Example

```c
static int cnt = 0;

t1.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}

t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state: cnt = 0

T1 executes, grabbing the global counter value into y.

Data Race Example

```c
static int cnt = 0;

t1.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}

t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state: cnt = 1

T1 executes again, storing the counter value
Data Race Example

```
static int cnt = 0;  // Shared state  cnt = 1

    t1.run() {
        int y = cnt;  y = 0
        cnt = y + 1;
    }

t2.run() {
    int y = cnt;  y = 1
    cnt = y + 1;
}
```

T1 finishes.  T2 executes, grabbing the global counter value into y.

---

Data Race Example

```
static int cnt = 0;  // Shared state  cnt = 2

    t1.run() {
        int y = cnt;  y = 0
        cnt = y + 1;
    }

t2.run() {
    int y = cnt;  y = 1
    cnt = y + 1;  // T2 executes, storing the incremented cnt value.
}
```
What Happened?

• In the first example, t1 was preempted after it read the counter but before it stored the new value.
  – Violated an object invariant
• A particular way in which the execution of two threads is interleaved is called a schedule. We want to prevent this undesirable schedule.
• Undesirable schedules can be hard to reproduce, and so hard to debug.

Question

• If instead of
  
  ```
  int y = cnt;
  cnt = y+1;
  ```
• We had written
  
  ```
  – cnt++;
  ```
• Would the result be any different?
• Answer: NO!
  
  – Don’t depend on your intuition about atomicity
Question

• If you run a program with a race condition, will you always get an unexpected result?
  – No! It depends on the scheduler
  – ...i.e., which JVM you’re running
  – ...and on the other threads/processes/etc that are running on the same CPU

• Race conditions are hard to find

Atomicity

• We want to ensure that the code in the two threads is atomic
  – Operations A and B are atomic with respect to each other if, from the perspective of the thread executing A, when another thread executes B, either all of B has executed or none of it has.
  – An atomic operation is one that is atomic with respect to all operations, including itself, that operate on the same state.
Locks

• Commonly used for enforcing atomicity
  – Descends from semaphore construct in an OS.
• Only one thread can hold a lock
  – Other threads block until they can acquire it
  – The operation of acquiring a lock is atomic
    • Cannot have a race on lock operations themselves!
• Any Object subclass has (can act as) a lock
  – Called an intrinsic lock

Synchronized Statement

• synchronized (obj) { statements }
  – Obtains (a.k.a. acquires, locks) the obj intrinsic lock before executing statements in block
  – Releases (a.k.a. unlocks) the lock when the statement block completes, whether due to a break, return, exception, etc.
Avoiding Interference:
Synchronization

public class Example extends Thread {
    private static int cnt = 0;
    static Object lock = new Object();
    public void run() {
        synchronized (lock) {
            int y = cnt;
            cnt = y + 1;
        }
    }
    ...
}

Lock, for protecting the shared state
Acquires the lock;
Only succeeds if not held by another thread
Releases the lock

Applying Synchronization

int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}

Shared state cnt = 0
T1 acquires the lock
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  // y = 0
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

*Shared state*  
\[ \text{cnt} = 0 \]

*T1 reads cnt into y*

---

Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  // T1 is pre-empted.
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  // T2 attempts to acquire the lock but fails because it's held by T1, so it blocks
    }
}
```

*Shared state*  
\[ \text{cnt} = 0 \]
Applying Synchronization

int cnt = 0;

t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  y = 0
    }
}

t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}

Shared state  cnt = 1

T1 runs, assigning to cnt

T1 releases the lock and terminates
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  y = 0
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}

Shared state  cnt = 1
```

T2 now can acquire the lock.

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  y = 0
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  y = 1
    }
}

T2 reads cnt into y.
```
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
        y = 0
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
        y = 1
    }
}
```

Shared state  \( \text{cnt} = 2 \)

\[\text{T2 assigns cnt, then releases the lock}\]

More on Locks

- Intrinsic locks are reentrant
  - The thread can reacquire the same lock many times
  - Lock is released when object unlocked the corresponding number of times
- No way to attempt to acquire an intrinsic lock
  - Either succeeds, or blocks the thread
  - Java 1.5 java.util.concurrent.locks package defines separate locks with more operations
Synchronized Methods

• A method can be synchronized
  – Add synchronized modifier before return type

• Obtains the lock on object referenced by this before executing method
  – Releases lock when method completes

• For a static synchronized method
  – Locks the Class object for the class
    • Accessible directly, e.g. Foo.class
  – Not the same as this!