Frequently asked questions from the previous class survey

- Vector clocks
  - What if we don’t know the number of processes?
  - Multicasting: Is every process getting every message?
  - How are events representative of causality?
- Matrix clocks: Why?
- Threads
  - Why do they have to wait?
  - Does Heap update result in I/O?
  - Where is stack located in the PCB?
- Why are clock speeds not increasing?
- Matrix clocks: Why?

Topics covered in this lecture

- Thread management
  - Risks and safety considerations
  - Server threading architectures
  - Distributed Servers

Difficult issues in writing multithreaded programs

- Key issues relate to the:
  - Sharing of objects
  - Techniques used for thread coordination and cooperation
- Each thread’s local variables are private to it
  - Remember, threads have their own stack
- However, threads are not given copies of
  - static (class) variables or
  - object instance variables

Threads retrieving work from a shared work-queue

- Race conditions may arise when threads manipulate data structures concurrently
- Queued requests may be lost or duplicated unless the threads’ manipulations are coordinated carefully

Things to watch for with threads
Risks of Threads:
- Safety Hazards
- Liveness Hazards
- Performance Hazards

Defining thread safety
- Relates to correctness
  - Class conforms to its specifications
  - Define invariants constraining object’s state
  - Postconditions that describe effects of operations
- Class is thread-safe when it continues to behave correctly
  - When accessed by multiple threads

Thread safety is about managing access to shared, mutable state
- An object’s state is any data that can affect its externally visible behavior
- Shared: Can be accessed by multiple threads
- Mutable: Value can change during the object’s lifetime

Some rules about thread-safety
- Encapsulate any needed synchronization
  - Clients should not have to provide one for accesses
- Stateless objects are always thread-safe

Guarding state variables
- To preserve consistency
  - Update state variables in a single atomic action
- For every invariant that involves more than one variable
  - All variables must be guarded by the same lock

Threads: Liveness hazard
- Safety
  - Nothing bad happens
- Liveness
  - Something good eventually happens
- Liveness failures
  - Deadlocks
  - Starvation
  - Livelocks
Threads: Performance hazard

- Liveness relates to something good happening **eventually**
- But what if eventually is not good enough?
  
  Responses should be quick

Some examples of issues in a threaded environment

Sequence generation example

```java
public class Sequence {
    private int value;
    public int getNext() {
        return value++;
    }
}
```

Must return a unique value

Unlucky execution sequence in a multithreaded setting

Race Condition

```
value=9
9+1=10
value=10
```

```
value=9
9+1=10
value=10
```

Returning to the sequence generation example

Guarded by this

```java
public class Sequence {
    private int value;
    public synchronized int getNext() {
        return value++;
    }
}
```

Guarding state with private locks

Guarded by myLock

```java
public class PrivateLock {
    private final Object myLock = new Object();
    private Widget widget;
    public void someMethod() {
        synchronized (myLock) {
            // access or modify state widget
        }
    }
}
```
Lazy Initializations

```java
public class LazyIntRace {
    private ExpensiveObject instance;

    public ExpensiveObject getInstance() {
        if (instance == null) {
            instance = new ExpensiveObject();
        }
        return instance;
    }
}
```

Only one instance should be created.

Using thread-safe objects

```java
public class Sequence {
    private final AtomicLong count = new AtomicLong(0);
    public long getNext() {
        return count.incrementAndGet();
    }
}
```

In the absence of synchronization compiler, CPU, and runtime can do weird stuff

```java
public class VisibilityTest {
    private static boolean ready;
    private static int number;

    public static class ReaderThread extends Thread {
        public void run() {
            while(!ready) {
                Thread.yield();
            }
            System.out.println(number);
        }
    }

    public static void main() {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

1. May loop forever
2. Could print 0
Write to `ready` may be visible BEFORE the write to `number`

Other concurrency pitfalls: Be careful about how much you synchronize

- Thread is not blocked, but cannot make progress
- Keep retrying operation that will always fail
- Transactional applications
- Rollback transaction if message cannot be processed
- Put it back to the head of the queue
- Buggy message handler cannot handle a message
  - Message continually dequeued and requeued
  - Poison message problem
- Other concurrency pitfalls: Be careful with how much you synchronize
  - You don't want to end up with sequential execution of threads
  - This will happen if you overkill on the synchronization
  - Watch for performance as well
Threads in distributed systems:
Multithreaded clients

- Hide communication latencies
  - Initiate communications
  - Immediately do something else
- Web browsers
  - As soon as main HTML page is fetched
  - Display it
  - Activate threads to retrieve other data types

Several connections can be opened simultaneously
- To the same server
  - If the server is overloaded, things get even slower
- To replicated servers
  - Data transfer in parallel
  - Much faster rendering of content

Multithreaded Servers

- Simplifies server code
- Easier to develop servers that exploit parallelism
- E.g.: Handling concurrent connections
  - Each connection managed by a different thread
  - Multiple connections handled by a pool of threads

Worker pool architecture

- Server creates a fixed pool of worker threads to process requests
  - Pool is initialized when server starts up
- Incoming requests are placed into a queue
  - Workers retrieve requests (work units) from the queue and process them
Managing priorities in the worker pool?
- Introduce multiple queues
- Worker threads scan queues in the order of descending priority

Disadvantages of the worker pool model
- Number of worker threads is fixed
  - So, threads in the pool may be too few to adequately cope with the rate of requests
  - Need to account for coordinated accesses to the shared queue

Thread-per-request architecture
- Worker thread is spawned for each incoming request
  - Worker thread destroys itself after processing request
- Advantages:
  - Threads do not contend for the shared work-queue
  - Throughput is potentially maximized
- Disadvantage
  - Overhead for thread creation and destruction operations

Thread-per-connection architecture
- Associates a thread per connection
- New worker thread created when a client makes a connection
  - Destroyed when client closes the connection
- Client may make many requests over the connection

Thread-per-object architecture
- Associate a thread with each remote object
- A separate thread receives requests and queues them
  - But there is a queue per-object

Thread-per-connection & Thread-per-object
- Advantages
  - Server benefits from lower thread management overheads compared to thread-per-request
- Disadvantages
  - Clients may be delayed when a worker thread has several outstanding requests, but another thread has no work to perform
Why prefer multiple threads over multiple processes?

- Threads are cheaper to create and manage than processes
- Resource sharing can be achieved more efficiently between threads than processes
- Threads within a process share the address space of the process
- Switching between threads is cheaper than for processes
- **BUT …** threads within a process are not protected from one another

Other costs for processes

- When a new process is created to perform a task there are other costs
  - In a kernel supporting virtual memory the new process will incur page faults
    - Due to data and instructions being referenced for the first time
  - Hardware caches must acquire new cache entries for that particular process

Contrasting the costs for threads [1/2]

- With threads these overheads may also occur but they are likely to be smaller
- When thread accesses code & data that was accessed recently by other threads in the process?
  - Automatically take advantage of any hardware or main memory caching

Contrasting the costs for threads [2/2]

- **Switching** between threads is much faster than that between processes
- This is a cost that is incurred many times throughout the lifecycle of the thread or process

Server Design Issues

September 8, 2015

CS555: Distributed Systems [Fall 2015]

Dept. Of Computer Science, Colorado State University
Server Design Issues

- **Iterative Servers**
  - Handles request
  - Returns response to requesting client

- **Concurrent Servers**
  - Pass request to a separate thread/process
    - Multithreaded server
  - Await new incoming request

The endpoint issue

- Clients send their requests to an endpoint
  - Port to which a server listens to
- But how do clients know about a port?
  - Globally assign endpoints for well-known ports
    - Internet Assigned Numbers Authority (IANA)
      - FTP (TCP, 21), HTTP (TCP, 80)

Implementing each service with a separate server could waste resources

- Instead of having multiple servers awaiting client requests
  - Have a single super-server
- INETD daemon on Unix
  - Listens to several ports for Internet services
    - Pop3 (110), FTP (21), Telnet (23)
  - When request comes in:
    1. Fork process to handle it
    2. Process exits once done

Designing Servers: Support interruption

- Terminate client session
  - Server will eventually detect connection loss (TCP)
- Send out-of-band data
  - Data to be processed before any other client data
  - But how can we send this out-of-band data?
    1. Send to a different port
    2. Reuse same connection
      - TCP urgent data e.g. socket.sendUrgentData(int data)

Tracking State in Servers

- Stateless servers
- Stateful servers
Stateless servers

- No state information about clients
  - E.g. Web Servers
- Usually some state is maintained
  - Log of documents accessed by client
  - But if this is lost, there should be no disruption of service
- Soft state: track state for a limited time
  - When timer elapses, revert to default behavior

Stateful servers

- Maintain persistent information on clients
- Use this to improve performance
  - Real and perceived
- Special measures needed to recover from failures

Stateful servers: A file server example

- Allows client to maintain local copy of file
  - Even for updates to the file
  - Maintain (client, file) tuples to track file state
  - Identify who has the most recent version of file
  - If server crashes it must recover the (client, file) entries

A hybrid approach: Have the client send its state to the server

- Cookies serve this purpose for Web pages
  - Tells a site about the pages accessed by a user
  - Use this to decide how to manage client
  - Sent back to browser every time state info changes
- Cookies don't stay where they are baked!

The contents of this slide-set are based on the following references