CIS 455/555: Internet and Web Systems

Server Architectures

September 13, 2021
Agenda for the next two lectures

- Brief discussion of the Lampson paper
- The client/server model
- An example service: Web/HTTP
- Servers with threads
- Consistency issues with shared resources
- Event-driven servers
- State, and where to keep it
- Extensible servers
- Multi-tier services and replication
worker(connection) {
    request = connection.read();
    if (request == "GET <document>") {
        data = filesystem.read(document);
        connection.write("HTTP/1.1 200 OK");
        connection.write(data);
    } else {
        connection.write("HTTP/1.1 400 Bad req");
    }
    close(connection);
}

main() {
    socket = listen(port 80);
    while (true) {
        connection = accept(socket);
        (new Thread).run(worker, connection);
    }
}
The trouble with threads

Source: SEDA paper (Welsh et al., SOSP 2001)

- Throughput, tasks/sec
- Latency, msec
- Number of threads
Thread pools

- What happens if we have too many threads?
  - Thrashing can occur

- Idea: Keep a thread pool
  - Very commonly used - e.g., in many Apache products, including some versions of the web server
  - Fixed number of threads

- What happens if there are too many requests?
  - The dispatcher thread puts requests into a queue (key aspect of thread pools!)
  - The worker threads dequeue requests and process them

- What if the system is not fully saturated?
  - Some threads in the pool may be idle
  - object.wait(), notify(), notifyAll()
Example with Thread Pools

class Dispatcher

void main()
{
    ServerSocket svr = new ServerSocket(1234);
    WorkerPool workerPool = new WorkerPool();
    while (true)
        Socket s = svr.accept(); /* blocks */
        workerPool.handle(s);
}


class WorkerPool {
    List<Worker> workers;
    BlockingQueue<Socket> queue;
    WorkerPool(int size) {
        /* fill list with workers */
        for (Worker w : workers) w.start();
    }
    void handle(Socket s) {
        queue.put(s);
    }
}

class Worker extends Thread {
    void run() {
        while (true) {
            Socket s = pool.queue.take(); /* blocks */
            /* read, write, ... */
            s.close();
        }
    }
}
Recap: Servers with threads

Why do we need this?

- Multiple requests can be processed concurrently
- Available resources (CPU, disk, ...) are multiplexed between requests

How do we build a server with threads?

- Basic pattern: Dispatcher thread + worker threads
- Thread pools increase efficiency and avoid thrashing
- Thread pools need a queue and some synchronization (to suspend workers when idle, and to reactivate them when more work arrives)
Plan for today

- Brief discussion of the Lampson paper
- The client/server model
- An example service: Web/HTTP
- Servers with threads
- Consistency issues with shared resources
- Event-driven servers
- State, and where to keep it
- Programming abstractions for Web servers
- Multi-tier services and replication
Why do we need synchronization?

- Now that we have more than one thread, life gets more complicated
  - Key issue: **Shared resources**, e.g., variables that can be accessed by more than one thread

- Simple example: Accounting system in a bank
  - Maintains the current balance of each customer's account
  - Customers can transfer money to other customers

```c
void transferMoney(customer A, customer B, int amount)
{
    showMessage("Transferring "+amount+" to "+B);
    int balanceA = getBalance(A);
    int balanceB = getBalance(B);
    setBalance(B, balanceB + amount);
    setBalance(A, balanceA - amount);
    showMessage("Your new balance: "+(balanceA-amount));
}
```
Why do we need synchronization?

1) \text{B} = \text{Balance}(\text{Bob})
2) \text{A} = \text{Balance}(\text{Alice})
3) \text{SetBalance}(\text{Bob}, \text{B} + 100)
4) \text{SetBalance}(\text{Alice}, \text{A} - 100)

1) \text{A} = \text{Balance}(\text{Alice})
2) \text{B} = \text{Balance}(\text{Bob})
3) \text{SetBalance}(\text{Alice}, \text{A} + 500)
4) \text{SetBalance}(\text{Bob}, \text{B} - 500)

What can happen if this code runs concurrently?

Alice's balance: $200$ $200$ $700$ $700$ $100$
Bob's balance: $800$ $900$ $900$ $300$ $300$
Problem: Race condition

What happened?

- **Race condition**: Result of the computation depends on the exact timing of the two threads of execution, i.e., the order in which the instructions are executed
- **Reason**: Concurrent updates to the same state
How common is this?

- Depends on how much interaction there is between server processes/requests

- Let's consider:
  - google.com
  - amazon.com
  - facebook.com
  - Fortnite
Goal: Consistency

- **What should have happened?**
  - Intuition: It shouldn't make a difference whether the requests are executed concurrently or not

- **How can we formalize this?**
  - Need a *consistency model* that specifies how the system should behave in the presence of concurrency
Sequential consistency:

- The result of any execution is the same as if the operations of all the cores had been executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program.

- How do we build systems that achieve this?
Mutual exclusion

How can we achieve better consistency?
- Key insight: Code has a critical section where accesses from other cores to the same resources will cause problems

Approach: Mutual exclusion
- Enforce restriction that only one core can execute the critical section at any given time
- What does this mean for scalability?

```c
void transferMoney(customer A, customer B, int amount) {
    showMessage("Transferring "+amount+" to "+B);
    int balanceA = getBalance(A);
    int balanceB = getBalance(B);
    setBalance(B, balanceB + amount);
    setBalance(A, balanceA - amount);
    showMessage("Your new balance: "+(balanceA-amount));
}
```
Locking

void transferMoney(customer A, customer B, int amount) {
    showMessage("Transferring "+amount+" to "+B);
    int balanceA = getBalance(A);
    int balanceB = getBalance(B);
    setBalance(B, balanceB + amount);
    setBalance(A, balanceA - amount);
    showMessage("Your new balance: "+(balanceA-amount));
}

• Idea: Implement locks
  • If LOCK(X) is called and X is not locked, lock X and continue
  • If LOCK(X) is called and X is locked, wait until X is unlocked
  • If UNLOCK(X) is called and X is locked, unlock X

• How should we insert locks into this code?
  • Option #1: One lock around the critical section
  • Option #2: One lock per variable (A's and B's balance)
  • Pros and cons? Other options?
Refresher: Monitors in Java

- For shared resources, use `synchronized` to gain a monitor on the object to be 'locked'
  - `synchronized` methods lock the entire object
  - `synchronized(obj){...} ` blocks lock their argument

- Implicit condition variables
  - Call `wait()` to block a thread while in the monitor
  - Call `notify()` or `notifyAll()` to unblock waiting threads
Locking helps!

1) **LOCK**(Bob)
2) **LOCK**(Alice)
3) B=Balance(Bob)
4) A=Balance(Alice)
5) SetBalance(Bob,B+100)
6) SetBalance(Alice,A-100)
7) **UNLOCK**(Alice)
8) **UNLOCK**(Bob)

1) **LOCK**(Alice)
2) **LOCK**(Bob)
3) A=Balance(Alice)
4) B=Balance(Bob)
5) SetBalance(Alice,A+500)
6) SetBalance(Bob,B-500)
7) **UNLOCK**(Bob)
8) **UNLOCK**(Alice)

Alice's balance: $200
Bob's balance: $800

Change:

<table>
<thead>
<tr>
<th>Time</th>
<th>Alice's balance</th>
<th>Bob's balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$200</td>
<td>$800</td>
</tr>
<tr>
<td>2</td>
<td>$200</td>
<td>$900</td>
</tr>
<tr>
<td>3</td>
<td>$200</td>
<td>$900</td>
</tr>
<tr>
<td>4</td>
<td>$200</td>
<td>$900</td>
</tr>
<tr>
<td>5</td>
<td>$600</td>
<td>$900</td>
</tr>
<tr>
<td>6</td>
<td>$600</td>
<td>$900</td>
</tr>
<tr>
<td>7</td>
<td>$600</td>
<td>$900</td>
</tr>
<tr>
<td>8</td>
<td>$600</td>
<td>$400</td>
</tr>
</tbody>
</table>

© 2021 A. Haeberlen, Z. Ives, V. Liu
Problem: Deadlock

1) LOCK(Bob)
2) LOCK(Alice)
3) B=Balance(Bob)
4) A=Balance(Alice)
5) SetBalance(Bob,B+100)
6) SetBalance(Alice,A-100)
7) UNLOCK(Alice)
8) UNLOCK(Bob)

1) LOCK(Alice)
2) LOCK(Bob)
3) A=Balance(Alice)
4) B=Balance(Bob)
5) SetBalance(Alice,A+500)
6) SetBalance(Bob,B-500)
7) UNLOCK(Bob)
8) UNLOCK(Alice)

Alice
Bob

$100
$500

1
2

Time

1 2

blocked (waiting for lock on Alice)

blocked (waiting for lock on Bob)

Neither processor can make progress!
What to do about deadlocks

Many possible solutions, including:

- **Design for no locking!**

- **Single lock:** Threads ask for a ‘bank’ lock rather than an per-account lock
  - Consequences for scalability?

- **Resource hierarchy:** Acquire in a specific order, i.e., Alice’s lock before B’s lock
  - Problem?

- **Deadlock recovery:** See that I can’t make progress, yield
Other issues with shared resources

- **Livelock**: Each thread begins a series of actions, discovers it cannot continue because actions by other threads have interfered, and begins again, endlessly
  - Example: Two people meeting in a narrow corridor
  - For instance, what does this mean for timeouts?

- **Starvation**: Several threads are competing for resources and, due to adverse scheduling, one of them never gets what it needs to continue
  - Example: 'Low-priority' patient in a waiting room when the doctors are always busy with 'high-priority' emergencies
Shared resources in Java

Suppose we share a resource across threads
  - Example: Log file

What can happen if threads write concurrently?

For shared resources, use `synchronized` to gain a monitor on an object to be "locked"
  - `synchronized` method locks the entire object
  - `synchronized (obj) { ... }` locks only its argument
Recap: Shared resources

- When multiple concurrent threads share resources, we can get into trouble
  - Inconsistencies, race conditions, ...

- We can avoid trouble in several ways
  - Example: Explicit locking; monitors (Java 'synchronized' blocks, wait()/notify()/notifyAll())

- But we have to be very careful
  - Deadlock, livelock, starvation, ...
What does this mean for HW1?

- Concurrency brings substantial complexity!
  - Suggestion: Implement in multiple stages (e.g., start with the single-threaded code from the last lecture, then add support for multiple pages, then add the thread pool, ...)
  - Avoid rewriting everything all the time - need to have an architecture **before** you start
    - Which classes will there be at the end? How will they interact?
    - What features will you need, and where will they go?
    - Do read the entire assignment and the entire spec **first**!
  - Test early and often - e.g., by writing "dummies" of some classes first, and then replacing them with working ones

- Start as soon as the assignment comes out!
  - It's ok to finish early - but a strange concurrency bug on the day of the deadline could mean trouble!
Plan for today

- Brief discussion of the Lampson paper
- The client/server model
- An example service: Web/HTTP
- Servers with threads
- Consistency issues with shared resources
- Event-driven servers
- State, and where to keep it
- Programming abstractions for Web servers
- Multi-tier services and replication
Event-driven programming

- Basically, a programmer-specified way of breaking up tasks
  - You've probably seen it if you've done GUI programming
  - But it is also used to multitask

- Based on an event queue and event handlers
  - Each task is broken into a series of events
  - Each event has a handler that does some work and potentially enqueues another event
  - "Local state" is generally kept in the event
  - What are the events that would occur in a web server?
Example: Event-based web server

```c
handleNewConnection(e) { startReading(e.connection); }
handleRequestRead(e) {
    if (e.request == "GET <document>") {
        issueFilesystemRead(document);
    } else {
        issueWrite(e.connection, "HTTP/1.1 400 Bad req");
    }
}
/* other handlers go here */

main() {
    EventQueue q;
    while (true) {
        e = q.getNextEvent();
        case e of {
            NewConnection: handleNewConnection(e);
            RequestArrived: handleRequestRead(e);
            FileReadCompleted: handleFileRead(e);
            AllDataWritten: handleDataWritten(e);
        }
    }
}
```
Threads vs events

- For this case:
  - No throughput degradation under load
  - Peak throughput is higher

Source: SEDA paper (Welsh et al., SOSP 2001)
Example: Event-based web server

```java
handleNewConnection(e) { startReading(e.connection); }
handleRequestRead(e) {
    if (e.request == "GET <document>") {
        issueFilesystemRead(document);
    } else {
        issueWrite(e.connection, "HTTP/1.1 400 Bad req");
    }
}
/* other handlers go here */

main() {
    EventQueue q;
    while (true) {
        e = q.getNextEvent();
        case e of {
            NewConnection: handleNewConnection(e);
            RequestArrived: handleRequestRead(e);
            FileReadCompleted: handleFileRead(e);
            AllDataWritten: handleDataWritten(e);
        }
    }
}
```
Continuations

- What if, in response to some event, we must perform a blocking system call?
  - Example: Request arrives; now we need to read the file from disk (blocking read() call) and send it back to the client
  - What would happen if we called read() in the event handler?
  - Solution: Write two event handlers:
    - Handler A parses the request and issues a non-blocking read (using a special system call)
    - Handler B is called when the read completes and sends data to client

- What if handler A has some state that handler B needs to know?
  - Must be saved explicitly in a continuation
Continuations: Example

```c
class AppWithThreads {
    void readRequest(char *filename) {
        FILE *file = fopen(filename, "r");
        int blocksTotal = getNumberOfBlocks(file);
        byte *buffer = (byte*) malloc(blocksTotal * blockSize);
        for (int i=0; i<blocksTotal; i++)
            blockingRead(file, &buffer[i*blockSize], blockSize);
        /* do something with the file that has been read */
    }
}
```

- OK to block here (other threads can continue)

- With threads, you can write linear code
  - If an operation blocks the current thread, the kernel can (transparently) save its state, switch to another thread, and later resume the thread when the operation completes
  - What can be done when events are used?
Continuations continued

class EventDrivenApp {
    struct {
        FILE *file;
        byte *buffer;
        int blocksRead, blocksTotal;
    } continuation;

    void handleNewReadRequest(char *filename) {
        continuation.file = fopen(filename, "r");
        continuation.blocksTotal = getNumberOfBlocks(continuation.file);
        continuation.buffer = (byte*) malloc(continuation.blocksTotal * blockSize);
        continuation.blocksRead = 0;
        issueNonblockingRead(continuation.file, continuation.buffer, blockSize);
    }

    void handleBlockReadComplete() {
        continuation.blocksRead ++;
        if (continuation.blocksRead < continuation.blocksTotal)
            issueNonblockingRead(continuation.file, continuation.buffer + continuation.blocksRead*blockSize, blockSize);
        else
            /* do something with the file that has been read */
    }
}

© 2021 A. Haeberlen, Z. Ives, V. Liu
University of Pennsylvania
Shared resources in event handlers

- For a single-threaded event server, don’t need *synchronized* blocks
  - We control when each event handler gives up a resource, hence we control interleaving of requests and their modification to shared resource

- But still may need to maintain flags or other information for situations when a resource is used across events
Concurrency and debugging

- A critical issue: how do we debug concurrent programs?

- Options:
  - `printlns` (be sure to tag with thread/context info)
    - Actual control flow may often surprise you!
    - Logs (see Apache log4j)
  - Selective breakpoints (remember to do 'binary search')

- Are threads or events easier to debug?
Event-driven programming interfaces

```c
handleNewConnection(e) { startReading(e.connection); }
handleRequestRead(e) {
    if (e.request == "GET <document>") {
        issueFilesystemRead(document);
    } else {
        issueWrite(e.connection, "HTTP/1.1 400 Bad req");
    }
}
/* other handlers go here */

main() {
    EventQueue q;
    while (true) {
        e = q.getNextEvent();
        case e of {
            NewConnection: handleNewConnection(e);
            RequestArrived: handleRequestRead(e);
            FileReadCompleted: handleFileRead(e);
            AllDataWritten: handleDataWritten(e);
        }
    }
}
```
Event-driven programming interfaces

Lambda functions in Java – see
http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

```java
class Task {
    public Object state;
    public Socket socket;
    public Function<Task, Task> op;
    public Task (Object state, …) { … }
}

sock.configureBlocking(false);
while (true) {
    SocketChannel sc = sock.accept();
    if (sc != null) {
        tasks.add(new Task(null, sc.socket(), (Task t) -> { … } ));
    }
    if (!tasks.empty()) {
        t = tasks.pop(); Task t2 = t.op.apply(t);
        if (t2 != null) tasks.add(t2);
    }
}
```

Lambda functions in Java – see
http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html
Recap: Event-driven programming

- A different way of structuring servers
  - Tasks are broken up into small events
  - Key elements: Event queue, event handlers, dispatch loop
  - Usually a single thread per core

- Tradeoffs different from multithreaded servers
  - Less need for synchronization (only one running thread!)
  - Higher performance; less degradation under load
  - Can be more difficult to program (e.g., continuations)

- Are threads and events mutually exclusive?