Lec 3: Example use cases (continued)
Cloud computing
Example Use Cases

• Web architectures (last time)
  – Simple architectures and real-world architectures

• Cloud computing (today)
  – What it means and how it began
What Are Some Web Architectures?
1. The Client-Server Model

```
Client
   \--------\--------
   \        /        \
  requests   responses
   /        \        /
Server
```
2. The Three-Tiered Architecture

Client (browser)

user interface (front end)

application

database

Service

WAN

LAN

stateless

stateful
3. Real Architectures

So, What Did We Learn?

• Web architectures are complex
• But there are well-known solutions
• There are lots of tradeoffs and understanding workload is key in choosing the right solution at each layer

• Each layer has distinct hardware requirements and likely distinct bottlenecks

• What does the last observation tell us?
Example Use Cases

• Web architectures (last time)
  – Simple architectures and real-world architectures

• Cloud computing (today)
  – What it means and how it began
What is cloud computing?

• Computing technology in which data and/or computation are outsourced to a massive-scale, multi-user infrastructure that is managed by a third-party.

• Appeared gradually due to two important challenges facing the Web:
  – Scaling
  – Management

• I’ll tell you the story of how it appeared, so as to help you understand what it is
Around 1995: The Scaling Challenge

• The Web and e-commerce were gaining traction

– 1996 to 1997: eBay grew from 41,000 to 341,000 users!

• Their challenge: **how to scale?**
  – 1996 to 1997: eBay grew from 41,000 to 341,000 users!
Pre-1995 Answer: Big, Expensive Computers

• Example: eBay used Sun E-10000 “supermini”
  – 64 processors @250MHz, 64GB RAM, 20TB disk, ~$1M

• The good:
  – Easy to manage
  – Easy to program
  – Simple failure mode

• The bad:
  – Q: Any ideas?
Pre-1995 Answer: Big, Expensive Computers

• Example: eBay used Sun E-10000 “supermini”
  – 64 processors @250MHz, 64GB RAM, 20TB disk, ~$1M

• The good:
  – Easy to manage
  – Easy to program
  – Simple failure mode

• The bad:
  – Expensive
  – Single point of failure
  – No incremental scalability
1995: Berkeley Network of Workstations (NOW)

- Idea: Leverage many interconnected small, cheap, general-purpose machines for incremental scalability and reliability
  - Typical PC: 200 MHz CPU, 32MB RAM, 4GB disk
NOW-0

- 1994: NOW had 4 HP-735’s
NOW-1

- 1995: NOW had 32 Sun SPARC stations
NOW-2

• 1997: 60 Sun SPARC-2’s
• Build Inktomi app
Companies Adopt NOW

• Everybody builds their own clusters and grows them to handle more and more load
  – Examples: eBay, Amazon, Google, all .com bubble companies

• Similar to early days of electricity when everyone built their own generator

Q: What do you think happened next?
Late 1990s: The Manageability Challenge

• Hard to manage and program large clusters
  – How to write scalable distributed programs?
  – How to debug large-scale programs?
  – How to make services reliable?
  – How to architect the network infrastructure?
  – How to provision a cluster to handle peak load?
  – How to administer a huge number of computers?
  – ...

• Each company had to build own complex software
  – Like each of us building an OS from scratch!
Early 2000s: Scalable Cluster Primitives

• Very few technically strong companies create powerful scalable and reliable primitives for cluster management and programming

• Examples:
  – Google’s Map/Reduce
  – The Google File System (GFS)
  – Google’s Bigtable
  – Amazon’s Dynamo
  – Distributed debugging and tracing tools
  – Datacenter temperature regulators
  – Scalable distributed communication mechanisms
  – …
Mid 2000s: Three Valuable Commodities

• Giant-scale clusters with enormous excess capacity
  – Everybody provisioned for peak

Q: How big was a typical Google datacenter around 2005?
  a. 1,000 machines
  b. 5,000 machines
  c. 10,000 machines
  d. 50,000 machines
  e. 100,000 machines
Mid 2000s: Three Valuable Commodities

• Giant-scale clusters with enormous excess capacity
  – Everybody provisioned for peak

• Expertise for managing and operating clusters at low cost
  – “Economies of scale”

• Complex software to help program/manage clusters
  – Even full applications (e.g., Gmail, Google Calendar, etc.)

Q: What do you think happened next?
2006: Cloud computing

• AWS sells resources, expertise, and access to cloud primitives in a pay-for-what-you-use model
  – Resources: CPU, network bandwidth, persistent storage
  – Cloud primitives: Amazon S3, EC2, SQS, Map/Reduce, ...

• Google launches Google Apps for Your Domain
  – Customizable Gmail, Google Docs, Google Calendar under a custom domain (e.g., gmail.cs.columbia.edu)

• Google then launches the App Engine
  – Web hosting infrastructure (such infrastructures existed before, but didn't come with many primitives)

• Microsoft launches Azure in 2009
Advantages of cloud computing

- Low barrier of market entry for startups
- Cheaper, low-management email, calendars, CRM solutions
- New mobile applications
- Faster batch processing via parallelization across many machines
What do Clouds have to do w/distributed systems?

• Clouds are powered by (and sell) distributed, scalable systems, which can be used as building blocks for easy bootstrap of new applications and services
  – Often times, you hear about clouds as being great because you don't have to purchase machines upfront
  – I think their major advantage lies in fact in the scalable services they provide

• This is unlike prior Web hosting infrastructures that predated “clouds” by many years
  – Those offered (and some still do) bare-metal and no add-on value-added services
Next time

- Communication: remote procedure calls
- Homework 1 is due tomorrow

- Homework 2 will be out on Thursday
  - Start with the writing piece and then do the coding
  - It's long, so start coding soon
  - Next time TA will go over the YFS series
Distributed Systems
[Fall 2012]

Lec 3 (Part 2): OS Background
Processes, Threads, and Local Coordination
OS Background

• Topics:
  – Processes
  – Threads
  – Local coordination
    • Inter-process communication (or how processes coordinate)
    • Thread synchronization (or how threads coordinate)

• Is this an OS course?!?
  – No, but concepts are essential for distributed systems
  – They often have 1:1 relationships with distributed coordination concepts
Outline

• Processes
• Inter-process communication (IPC)
• Threads
• Thread synchronization

• Slide acknowledgements:
  – Dave Andersen (www.cs.cmu.edu/~dga/15-440/F10/lectures/04-work.pdf)
  – Jinyang Li (www.news.cs.nyu.edu/~jinyang/fa09/notes/ds-lec2.pdf)
What Is a Process?

- **Process**: an execution stream (or program) in the context of a particular process state
  - “Program in execution,” “virtual CPU”
- **Execution stream**: a stream of instructions
- **Process state**: determines effect of running code
  - **Registers**: general purpose, instruction pointer (program counter), floating point, …
  - **Memory**: everything a process can address, code, data, stack, heap, …
  - **I/O status**: file descriptor table, …
Program vs. Process

• Program != process
  – Program: static code + static data
  – Process: dynamic instantiation of code + data + more

• Program ⇔ process: no 1:1 mapping
  – Process > program: more than code and data
  – Program > process: one program runs many processes
  – Process > program: many processes of same program
The CPU

- Program counter
  - Retrieve next instruction
  - Interpret retrieved instruction
  - Interrupt signal?
    - yes
      - 1. Save counter and state
      - 2. Update them with new counter and state
    - no
  - state
Why Use Processes?

• Express concurrency
  – Systems have many concurrent jobs going on
  – E.g. Apache can spawn multiple processes to process requests in parallel on multiple CPUs and parallelize I/O…
  – OS manages concurrency

• General principle of divide and conquer
  – Decompose a large problem into smaller ones easier to think of well contained smaller problems

• Processes are isolated from each other
  – Sequential with well defined interactions
Address Spaces

• **Address Space (AS):** all memory a process can address
  – Really large memory to use
  – Linear array of bytes: [0, N), N roughly $2^{32} / 2^{64}$

• **Process $\Leftrightarrow$ address space:** $1:1$ mapping
  – Address space = protection domain

• **OS isolates** address spaces
  – One process can’t access another’s address space
  – Same pointer address in different processes point to different memory
Address Space Illustration

Process 1

Process 2
Practical Stuff: Using Processes

• Creating a new process: `fork()`
  – Makes an almost exact copy of calling process (PID changes, etc.)
  – New process has its own memory (although some of it is shared – copy-on-write)
  – How to tell difference between the two processes? Return value is 0 in child, child PID in parent.

• Executing a different program: `exec()`
  – Replaces the process’ image with a new one running the new program
#include <iostream>
#include <sys/wait.h>
#include <unistd.h>

using namespace std;

int main() {
    pid_t pid;
    int status, died;
    switch (pid = fork()) {
    case -1: cout << "can't fork\n";
            exit(-1);
    case 0 : execl("/usr/bin/date","date",0);  // this is the code the child runs
    default: died = wait(&status);  // this is the code the parent runs
    }
}
Outline

- Processes
- Inter-process communication (IPC)
- Threads
- Thread synchronization
Interprocess Communication

• Often, multiple processes are part of the same “program”
• Hence, they need to coordinate

• Example: Apache and its worker processes, each serving a request
  – Apache must send requests to each worker

• This is very similar to how processes (or tasks) in distributed systems must coordinate
IPC Models

Message passing  
Shared memory
Message Passing vs. Shared Memory

• Message passing
  – Why good? All *sharing is explicit* – less chance for error
  – Why bad? *Overhead*
    • Data copying, across protection domains (context switches)

• Shared memory
  – Why good? *Performance*
    • Set up shared memory once, then access w/o crossing protection domains
  – Why bad? Things change behind your back – *error prone*
IPC Example: UNIX Signals

• Signals
  – A very short message: just a small integer
  – A fixed set of available signals. Examples:
    • 9: kill
    • 11: segmentation fault

• Installing a handler for a signal: `signal()`
  – `sighandler_t signal(int signum, sighandler_t handler);`

• Send a signal to a process: `kill()`
  – `kill(pid_t pid, int signum)`
IPC Example: UNIX Pipe

- **int pipe(int fds[2])**
  - Creates a one way communication channel
  - `fds[2]` holds the returned two file descriptors
  - Bytes written to `fds[1]` will be read from `fds[0]`

```c
int pipefd[2];
pipe(pipefd); // error handling ignored
switch(pid = fork()) {
  case -1: perror("fork"); exit(1);
  case 0: close(pipefd[0]);
    // write to fd 1
    break;
  default: close(pipefd[1]);
    // read from fd 0
    break;
}
```
IPC Example: UNIX Shared Memory

• int `shmget`(key_t key, size_t size, int shmflg)
  – Create a shared memory segment; returns ID of segment
  – key: unique key of a shared memory segment, or IPC_PRIVATE

• int `shmat`(int shmid, const void *addr, int flg)
  – Attach shared memory segment to address space of calling process
  – shmid: id returned by `shmget()`

• int `shmdt`(const void *shmaddr);
  – Detach from shared memory

• Problem: synchronization! (similar concept as in threads)
Today

- Processes
- Inter-process communication (IPC)
- Threads
- Thread synchronization
Threads

• **Threads**: separate streams of executions that share an address space
  – Allow one process to have multiple points of execution, can use multiple CPUs

• **Per-thread state** (not shared across threads)
  – Program counter (EIP on x86)
  – Other registers
  – Stack

• Conceptually similar to processes, but **different**
  – Often called “lightweight processes”
Threads in Memory

When multiple threads exist, each must have a separate stack. This example shows two.

- Stack
- Heap
- BSS
- Data
- Text
- Uninitialized variables
- Initialized variables
- Instruction
Thread vs. Process

• Why threads?
  – Thread allows running code concurrently within a single process
  – Switching among threads is lightweight
  – Sharing data among threads requires no IPC

• Why processes?
  – Fault isolation: One buggy process cannot crash others
Why Multi-threaded Programming?

• Exploit multiple CPUs (multi-core) with little overhead

• Exploit I/O concurrency
  – Do some processing while waiting for disk, network, user

• Reduce latency of networked services
  – Servers serve multiple requests in parallel
  – Clients issue multiple requests in parallel

• Example:
  – In addition to multi-process support, Apache has multi-thread support, which is much more common.
Single-threaded servers do not fully utilize I/O and CPU.
Multi-threaded servers achieve I/O concurrency

Network usage

Disk usage

CPU usage
Practical Stuff: The `pthread` Library

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);`
  - Create a new thread to run `start_routine` on `arg`
  - `thread` holds the new thread’s id
  - Can be customized via `attr`

- `int pthread_join(pthread_t thread, void **value_ptr);`
  - Wait for thread termination, and retrieve return value in `value_ptr`

- `void pthread_exit(void *value_ptr);`
  - Terminates the calling thread, and returns `value_ptr` to threads waiting in `pthread_join`
Pthread Creation Example

void* thread_fn(void *arg) {
    int id = (int)arg;
    printf("thread %d runs\n", id);
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_fn, (void*)1);
    pthread_create(&t2, NULL, thread_fn, (void*)2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}

$ gcc -o threads threads.c -Wall -lpthread
$ ./threads
thread 1 runs
thread 2 runs
Thread Pools

• Problem:
  – Creating a thread for each request: costly
    • And, the created thread exits after serving a request
  – More user requests → more threads, server overload

• Solution: thread pool
  – Pre-create a number of threads waiting for work
  – Wake up thread to serve user request – faster than thread creation
  – When request done, don’t exit – go back to pool and wait
  – Limits the max number of threads

• Your YFS server will have thread pools
• Apache supports thread (and process) pools
Today

• Processes
• Inter-process communication (IPC)
• Threads
• Thread synchronization
The Problem

• Memory is shared across all threads
• Hence threads must coordinate so as to update shared memory correctly
Banking Example

int balance = 1000;
int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, withdraw, (void*)800);
    pthread_create(&t2, NULL, withdraw, (void*)800);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("All done: balance is $%d\n", balance);
    return 0;
}

void* withdraw(void *arg) {
    int amount = (int)arg;
    if (balance >= amount) {
        balance -= amount;
        printf("ATM gives user $%d\n", amount);
    }
}

Imagine that these threads are created in response to requests from ATM machines

What are possible results?
Results of Banking Example

$ gcc –Wall –lpthread –o bank bank.c

$ ./bank
  ATM gives user $800                  Result 1
  All done: balance is $200

$ ./bank
  ATM gives user $800
  ATM gives user $800                  Result 2
  All done: balance is $-600

$ ./bank
  ATM gives user $800
  ATM gives user $800
  ATM gives user $800                  Result 3
  All done: balance is $200

How are each of these achieved?
Schedule 1 (for Result 1)

Thread 1

if (balance >= amount)
    register = balance - amount;
    balance = register

Thread 2

if (balance >= amount)
    register = balance - amount;
    balance = register
Schedule 2 (for Result 2)

Thread 1

if (balance >= amount)
register = balance - amount;

balance = register

Thread 2

if (balance >= amount)
register = balance - amount;

balance = register
Schedule 3 (for Result 3)

Thread 1

if (balance >= amount)
    register = balance - amount;
    balance = register

Thread 2

if (balance >= amount)
    register = balance - amount;
    balance = register
Race Conditions

• Definition: a timing dependent error involving shared state

• Can be very bad
  – “Non-deterministic:” don’t know what the output will be, and it is likely to be different across runs
  – Hard to detect: too many possible schedules
  – Hard to debug: debugging changes timing so hides bugs (“heisenbug”)
Synchronization Mechanisms

- Multiple mechanisms, each solving a different problem
  - Locks
  - Condition variables
  - Semaphores
  - Monitors
  - Barriers

We’ll cover here briefly

Read in OS textbook or lectures:
http://www.cs.columbia.edu/~junfeng/12sp-w4118/lectures/l09-lock.pdf,

- Synchronization – both local and distributed – is used pervasively in distributed systems
  - Will use synchronization mechanisms in most labs
  - Will build distributed locking for Lab 1
  - MapReduce uses barriers to synchronize threads
  - ...
Locks

- Locks allow only one thread to pass through a "critical section" at any time
  - `lock(l)`: acquire lock exclusively; wait if not available
  - `unlock(l)`: release exclusive access to lock

```c
pthread_mutex_t l = PTHREAD_MUTEX_INITIALIZER;
void* withdraw(void *arg) {
    int amount = (int)arg;
    pthread_mutex_lock(l);
    if (balance >= amount) {
        balance -= amount;
        printf("ATM gives user $%d\n", amount);
    }
    pthread_mutex_unlock(l);
}
```
Common Pitfalls

- **Wrong lock granularity**
  - Too small granularity leads to races
  - Too large granularity leads to bad performance

- **Deadlocks**
  - Better bugs than race

- **Starvation**

- **Discussion of each is subject of another course...**
Processes, Threads, and Coordination in Distributed Systems

• All these topics are extremely relevant for distributed systems
  – Every server is multi-threaded
  – Servers need to coordinate, and they do so using similar methods as IPC, shared memory, locking, barriers, etc.

• In distributed systems, a process is often times called a “task” (unit of processing)
• A program is often times called a “job”
Next Time

• Inter-machine communication
  – Remote procedure calls
  – Semantics and complexities of RPCs
Appendix
Cool Process Internals: Copy-on-Write (CoW)

- CoW is a useful, general technique that shows up all over in systems
  - Mark parents’ memory read-only
  - Have child *share* parents memory instead of copying
  - If either one *writes* -- hey, it was read only! (CPU will raise an exception)
- *Now* give the child its own copy of the page of memory someone was writing