OS, Parallelism, and Cloud Services

Some slides thanks to Peter Chen, Harsha V. Madhyastha and Sugih Jamin

Outline (for two lectures)

- Today:
  - Cloud Services
  - Amazon Web Services
  - Operating systems basics
- Thursday:
  - OS virtualization
  - Network virtualization
  - Datacenters

Cloud Services Have Taken Over

- Most web services aren't run on dedicated hardware anymore

Netflix finishes its massive migration to the Amazon cloud

After move to Amazon, only the DVD business still uses traditional data center.

JON BRODKIN - 2/11/2016, 1

When Amazon's cloud storage fails, lots of people get wet

By MAE ANDERSON, AP TECHNOLOGY REPORTER
NEW YORK — Feb 28, 2017, 7:50 PM ET

Share with Facebook Share with Twitter

Cloud Services Have Taken Over

- They're run on Amazon Web Services (or Google or Azure) instead

What IS The Cloud?

- TLDR: No one knows
- But increasingly, it refers to compute services offered on hosted hardware/software
- Amazon offers incredible array of services for rent

- Amazon Athena
- Amazon API Gateway
- Amazon AppStream
- Amazon AppStream 2.0
- Amazon Chime
- Amazon Cloud Directory
- Amazon CloudSearch
- Amazon CloudWatch
- Amazon CloudWatch Events
- Amazon CloudWatch Logs
- Amazon Cognito
- Amazon DynamoDB
- Amazon EC2 Container Registry (ECR)

d?

edible array of services for

Amazon Athena	Amazon EC2 Container Service (ECS)
Amazon API Gateway	Amazon EC2 Systems Manager
Amazon AppStream	Amazon ElastiCache
Amazon AppStream 2.0	Amazon Elastic Block Store (EBS)
Amazon Chime	Amazon Elastic Compute Cloud (EC2)
Amazon Cloud Directory	Amazon Elastic File System (EFS)
Amazon CloudSearch	Amazon Elastic MapReduce
Amazon CloudWatch	Amazon Elasticsearch Service
Amazon CloudWatch Events	Amazon Elastic Transcoder
Amazon CloudWatch Logs	Amazon GameLift
Amazon Cognito	Amazon Glacier
Amazon DynamoDB	Amazon Inspector
Amazon EC2 Container Regi	Amazon Kinesis Analytics
	Amazon Kinesis Firehose

rvices for

Amazon Athena	Amazon EC2 Container Service (ECS)	Amazon Kinesis Streams
Amazon API Gateway	Amazon EC2 Systems Manager	Amazon Lightsail
Amazon AppStream	Amazon ElastiCache	Amazon Machine Learning
Amazon AppStream 2.0	Amazon Elastic Block Store (EBS)	Amazon Mobile Analytics
Amazon Chime	Amazon Elastic Compute Cloud (EC2)	Amazon Pinpoint
Amazon Cloud Directory	Amazon Elastic File System (EFS)	Amazon Polly
Amazon CloudSearch	Amazon Elastic MapReduce	Amazon QuickSight
Amazon CloudWatch	Amazon Elasticsearch Service	Amazon Redshift
Amazon CloudWatch Events	Amazon Elastic Transcoder	Amazon Rekognition
Amazon CloudWatch Logs	Amazon GameLift	Amazon Relational Database Service (RDS)
Amazon Cognito	Amazon Glacier	Amazon SimpleDB
Amazon DynamoDB	Amazon Inspector	Amazon Simple Email Service (SES)
Amazon EC2 Container Regi	Amazon Kinesis Analytics	Amazon Simple Notification Service (SNS)
	Amazon Kinesis Firehose	Amazon Simple Queue Service (SQS)

What IS The Cloud?

- Three rough “levels”
- **Infrastructure-as-a-service**, designed for admins
  - Machines, storage, network capacity; most of AWS
- **Platform-as-a-service**, designed for developers
  - Heroku, Twilio (GO BLUE)
- **Software-as-a-service**, designed for users
  - Gmail, Google Docs, Salesforce

- In 485 we’re mainly concerned with **IAAS**, though you will likely encounter PAAS in your future lives

Core Amazon Web Services

- Two services are the most important for us:
  - ***Elastic Compute Cloud*** ("EC2") offers machines for rent
  - ***Simple Storage Service*** ("S3") offers very basic Web storage
- An EC2 "instance" with 16 CPUs, 64GB costs $0.862 an hour.
  - That's $7,551.12 a year!
  - Dell will sell me something similar for $4493.00.
  - Why would I ever use AWS?

Advantages of IAAS

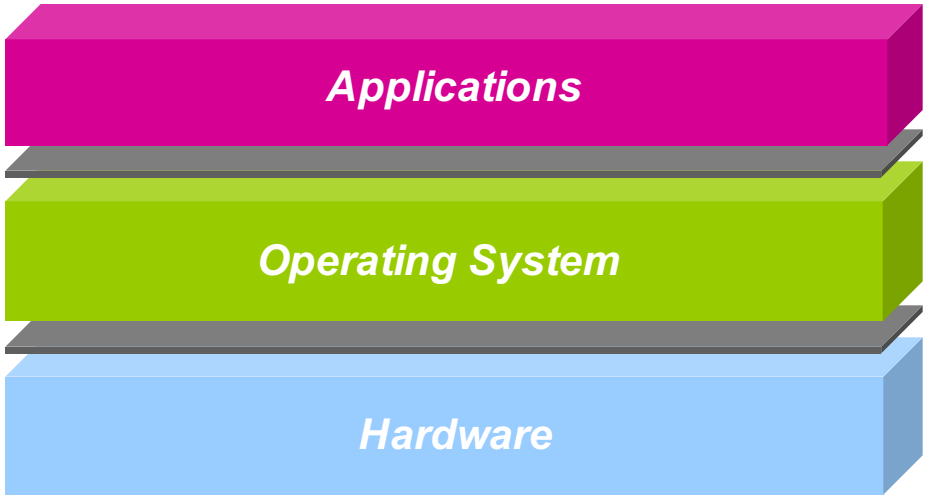
- Pay only for what you consume
  - LOTS of different sized machines
- Related, but not identical: no danger of misestimating load
- Amazon has scale: experienced admins, cheaper machine purchases, etc
- Spin up new machines in minutes
- It's hard to hire admin expertise

OS and Virtualization

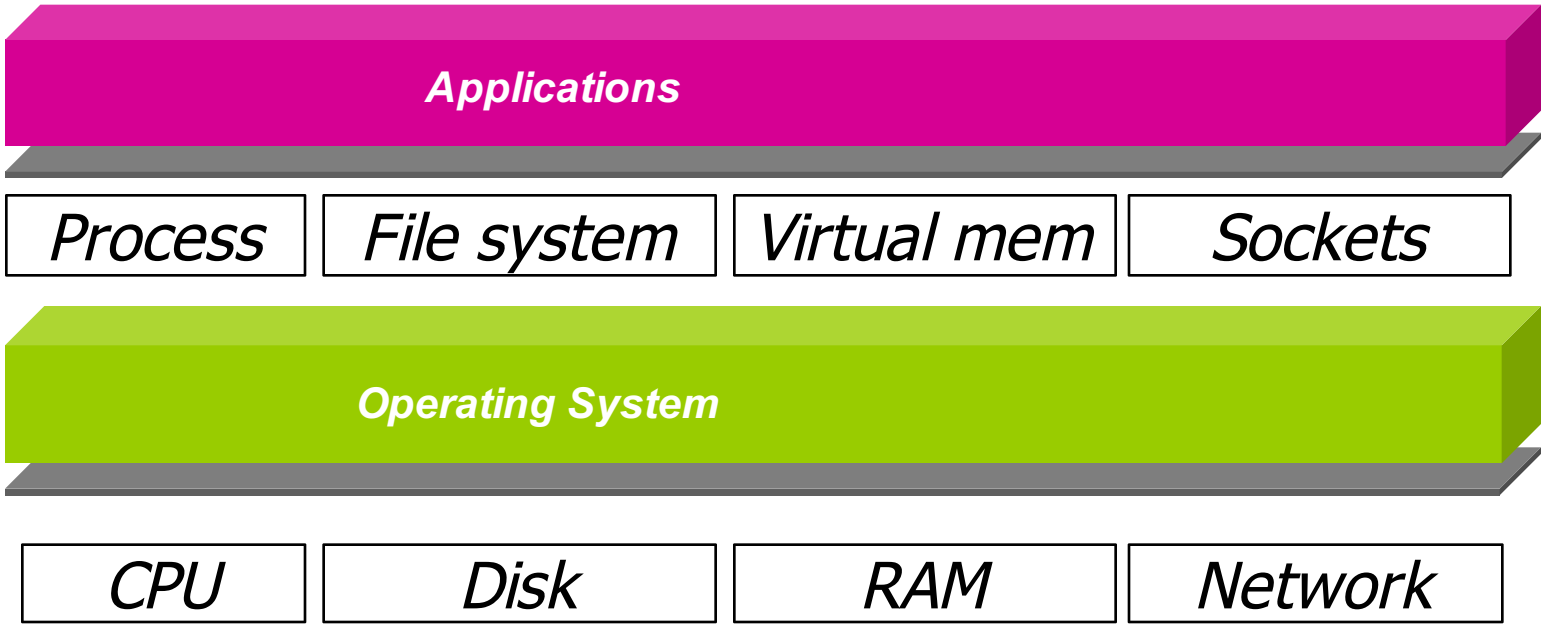
- A key part of AWS economics is **virtualization**
  - Amazon will rent you a HUGE array of different machines
  - These are often one big machine cut up into tiny parts, each on a separate virtual machine
- We will cover virtualization next lecture
- Today: what are the OS services we need to virtualize?

What does an OS do?

- Creates abstractions to make hardware easier to use
- Manages shared hardware resources



OS Abstractions



Managing Concurrency

- Source of OS complexity
  - Multiple users, programs, I/O devices, etc.
  - Originally for efficient use of H/W, but useful even now

- How to manage this complexity?
  - Divide and conquer
  - Modularity and abstraction

```
main() {
    getInput();
    computeResult();
    printOutput();
}
```

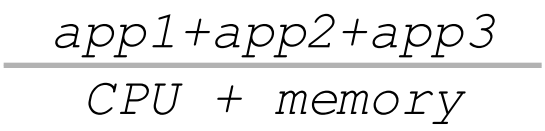
Outline

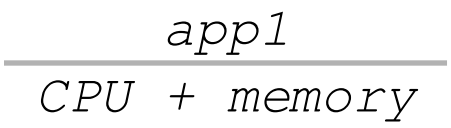
- Operating system overview
- Processes
- Threads
- Sockets

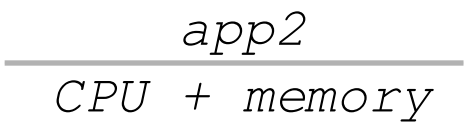
The Process

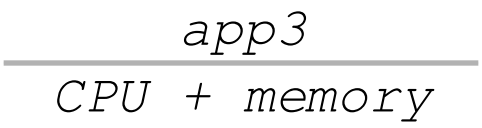
- The process is the OS abstraction for execution
  - Also sometimes called a job or a task
- A process is a program in execution
  - Programs are static entities with potential for execution

- For each area of OS, ask
  - What interface does hardware provide?
  - What interface does OS provide?









The Process

- What interface does hardware provide?
- Hardware interface: single computer (CPU & memory), executing instructions from a mix of many different applications

- What interface does OS provide?
- OS interface: several dedicated computers (one per application process)

- Single shared resource -> multiple private resources

Process example

```
from multiprocessing import Process
from time import sleep

def worker(worker_id):
    print("Hello from process {}".format(worker_id))
    sleep(10)

if __name__ == "__main__":
    for i in range(3):
        p = Process(target=worker, args=(i,))
        p.start()
```

```
$ python test_multiprocessing.py
Hello from process 0
Hello from process 1
Hello from process 2
```

Example

```
$ python test_process.py
Hello from process 0
Hello from process 1
Hello from process 2

$ ps -ax | grep python
16571 s001  S+     0:00.04 python test_multiprocessing.py
16572 s001  R+     1:59.49 python test_multiprocessing.py
16573 s001  R+     1:59.59 python test_multiprocessing.py
16574 s001  R+     1:59.68 python test_multiprocessing.py
```

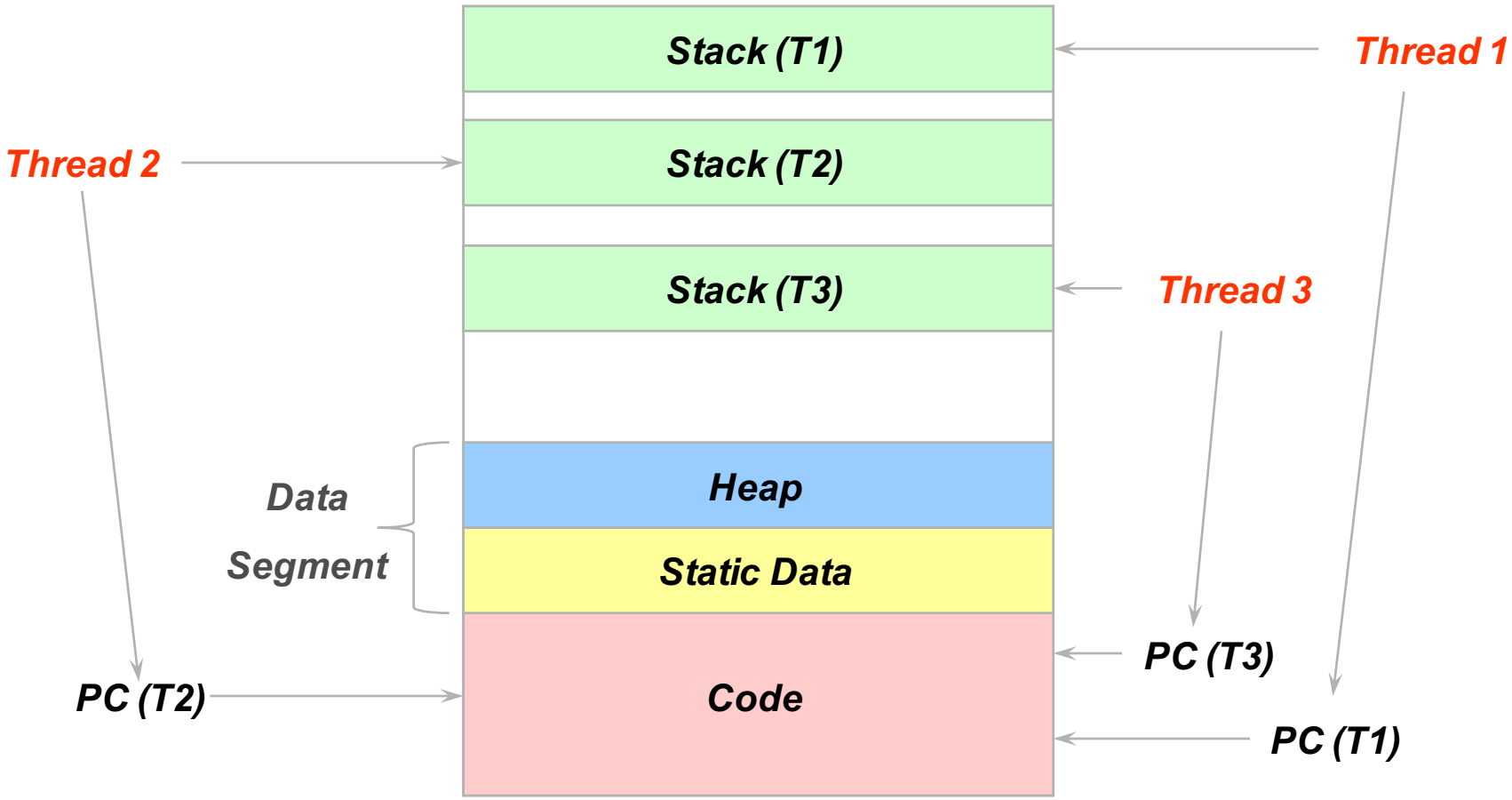
Process components

- A process, named using its process ID (PID), contains all the state for a program in execution
  - Set of threads (active)
  - An address space (passive)
- What's in the address space?

Shared across threads
- The code and input data for the executing program
- The memory allocated by the executing program
- Open files, network connections, etc.

Private to each thread
- An execution stack encapsulating the state of procedure calls
- The program counter (PC) indicating the next instruction
- A set of general-purpose registers with current values

Process address space



Review of stack frames

```
A(int tmp) {
    B(tmp);
}

B(int val) {
    C(val, val + 2);
    A(val - 1);
}

C(int foo, int bar) {
    int v = bar - foo;
}
```

Multiple threads

- Can have several threads in a single address space
  - Sometimes they interact
  - Sometimes they work independently
- What state is private to a thread?
  - Stack (and SP)
  - PC
  - Code
- What state is shared between threads?
  - Data segment

Thread example

```
from threading import Thread
from time import sleep


def worker(worker_id):
    print "Hello from thread {}".format(worker_id)
    sleep(10)

if __name__ == "__main__":
    for i in range(3):
        t = threading.Thread(target=worker, args=(i,))
        t.start()
```

```
$ python test_threading.py
Hello from thread 0
Hello from thread 1
Hello from thread 2
```

Example

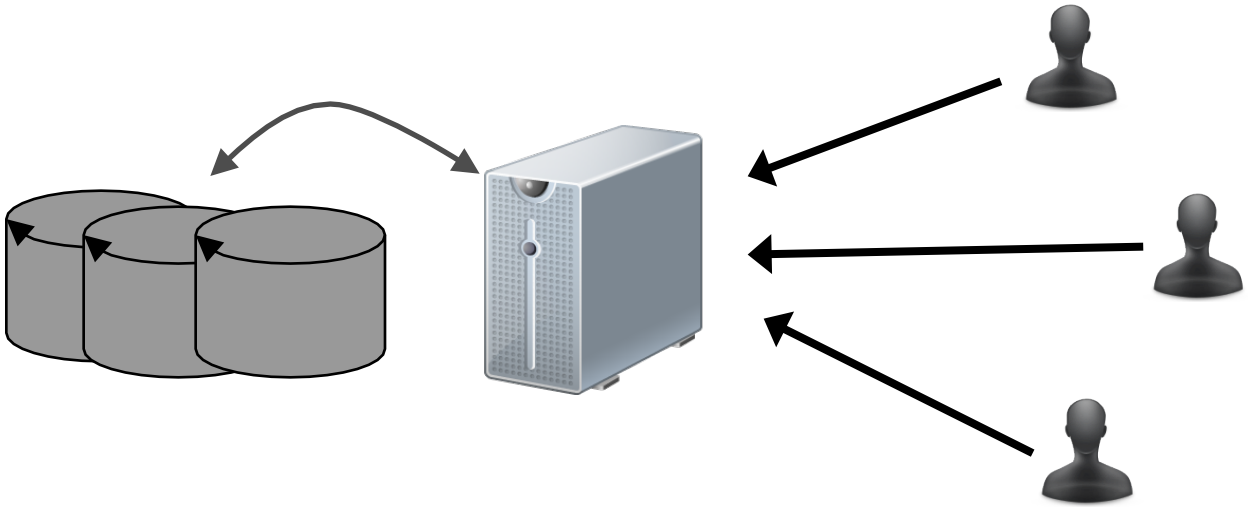
```
$ python test_threading.py
Hello from thread 0
Hello from thread 1
Hello from thread 2

$ ps -ax | grep python
16976 s001  R+     0:43.33 python test_threading.py

$ ps -axM | grep -A3 python
awdeorio        17006 s001    0.0 S    31T   0:00.01   0:00.02
                                         python test_threading.py
                17006        56.5 R    31T   0:18.65   0:10.97
                17006        56.2 R    31T   0:18.53   0:10.94
                17006        56.9 R    31T   0:18.71   0:11.17
```

Why do we need threads?

- Example: Web server
  - Receives multiple simultaneous requests
  - Reads web pages from disk to satisfy each request



Option 1: Handle one request at a time

- Example execution schedule:
  - Request 1 arrives
  - Server receives request 1
  - Server starts disk I/O 1a
  - Request 2 arrives
  - Server waits for I/O 1a to finish

- Easy to program, but slow
  - Why slow?
  - Can't overlap disk requests with computation, or with network receives

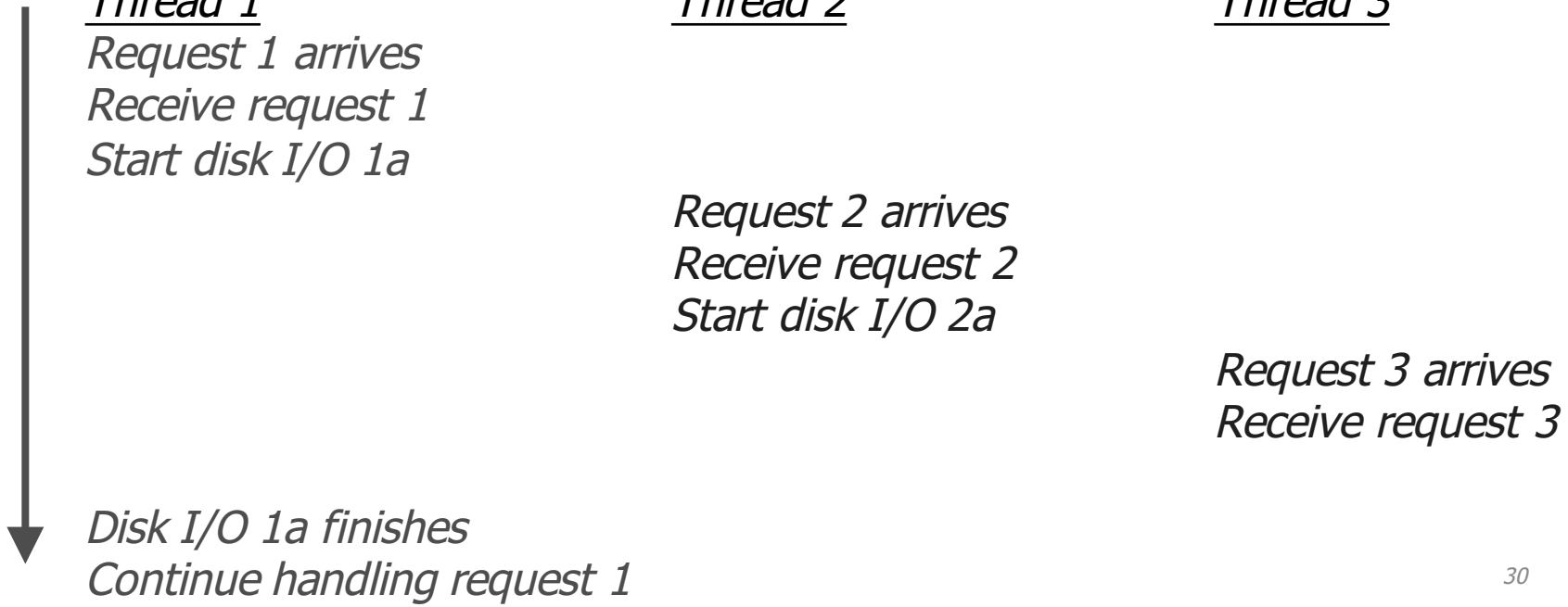
Option 2: Event-driven web server (asynchronous I/O)

- Issue I/Os, but don't wait for them to complete
  - Request 1 arrives
  - Server receives request 1
  - Server starts disk I/O 1a to satisfy request 1
  - Request 2 arrives
  - Server receives request 2
  - Server starts disk I/O 2a to satisfy request 2
  - Request 3 arrives
  - Disk I/O 1a finishes

Web server must remember
What requests are being served, and what stage they're in
What disk I/Os are outstanding (and which requests they belong to)

Multi-threaded web server

- One thread per request
  - Thread issues disk (or n/w) I/O, then waits for it to finish
  - Though thread is blocked on I/O, other threads can run
  - Where is the state of each request stored?



Benefits of threads

- Thread manager takes care of CPU sharing
  - Other threads can progress when one thread issues blocking I/Os
  - Private state for each thread

- Applications get a simpler programming model
  - The illusion of a dedicated CPU per thread

When are threads useful?

- Multiple things happening at once
- Usually some slow resource

- Examples:
  - Network server
  - Controlling a physical system (e.g., airplane controller)
  - Window system
  - Parallel programming

Ideal Scenario

- Split computation into threads
- Threads run independent of each other
  - Divide and conquer works best if divided parts are independent

Is independence of threads practical?

Dependence between threads

- Example 1: Microsoft Word
  - One thread formats document
  - Another thread spell checks document
- Example 2: Desktop computer
  - One thread plays World of Warcraft
  - Another thread compiles EECS project
- Two types of sharing: app resource or H/W
- Example of non-interacting threads?

Cooperating threads

- How can multiple threads cooperate on a single task?
  - Example: Ticketmaster's webserver
  - Assume each thread has a dedicated processor
- Problem:
  - Ordering of events across threads is non-deterministic
  - Speed of each processor is unpredictable

```
Thread A ---------------------------------------->
Thread B -   -   -   -   -   -   -   -   -   >
Thread C - - - - - - - - - - - - - - - - - - - - ->
```

- Consequences:
  - Many possible global ordering of events
  - Some may produce incorrect results

Non-deterministic ordering → Non-deterministic results

Thread 1
print ABC

Thread 2
print 123

- Printing example
  - Possible outputs?
    - 20 outputs: ABC123, AB1C23, AB12C3, AB123C, A1BC23, A12BC3, A123BC, 1ABC23, 1A2BC3, …
  - Impossible outputs?
    - ABC321
- Ordering within thread is sequential, but many ways to merge per-thread order into a global order
- What's being shared between these threads?
  - The output stream

Multiple writers

- Non deterministic results occur when multiple threads write the same data
- How to fix this?
- Option 1: avoid writing the same piece data (EECS 485)
- Option 2: take EECS 482!

When to use processes vs. threads

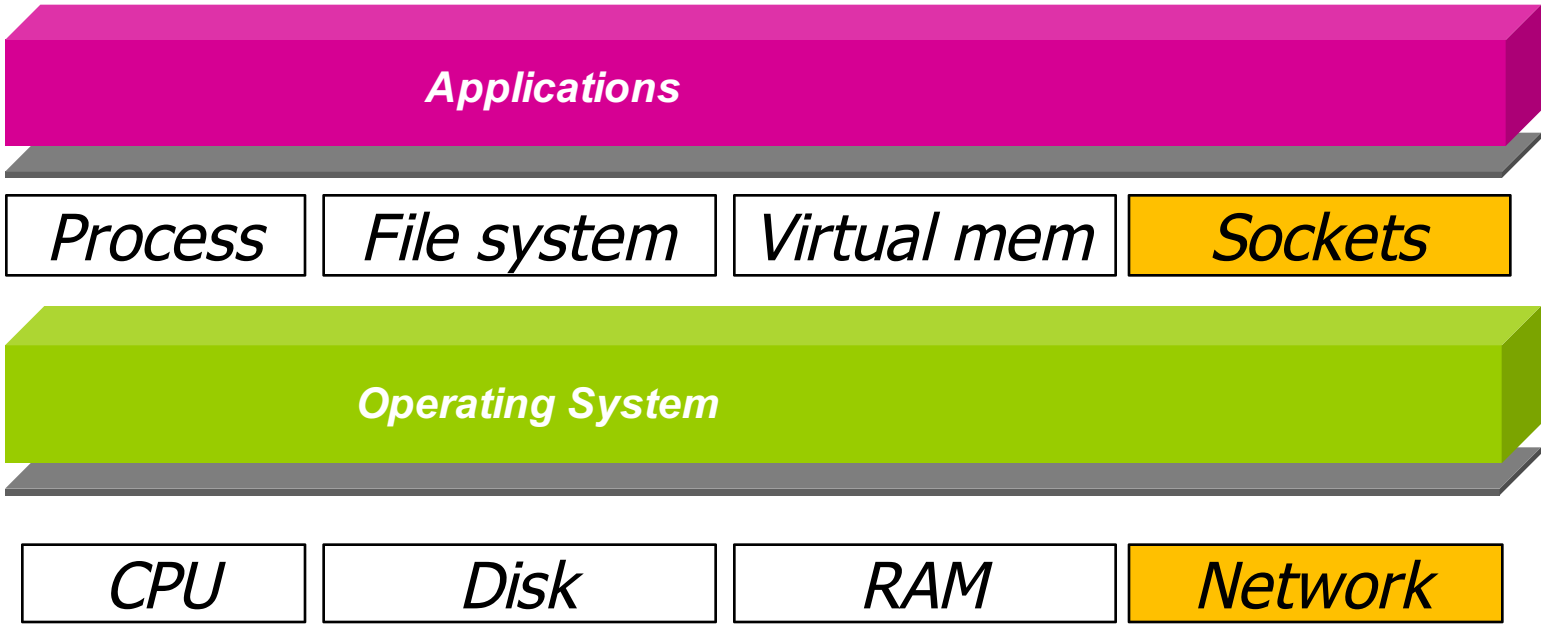
- **Threads** for small tasks
  - Like handling another request within an application
- **Processes** for "heavyweight" tasks
  - Like the execution of applications

- **Threads** share an address space
- **Processes** don't

Outline

- Operating system overview
- Processes
- Threads
- Sockets

The OS network abstraction



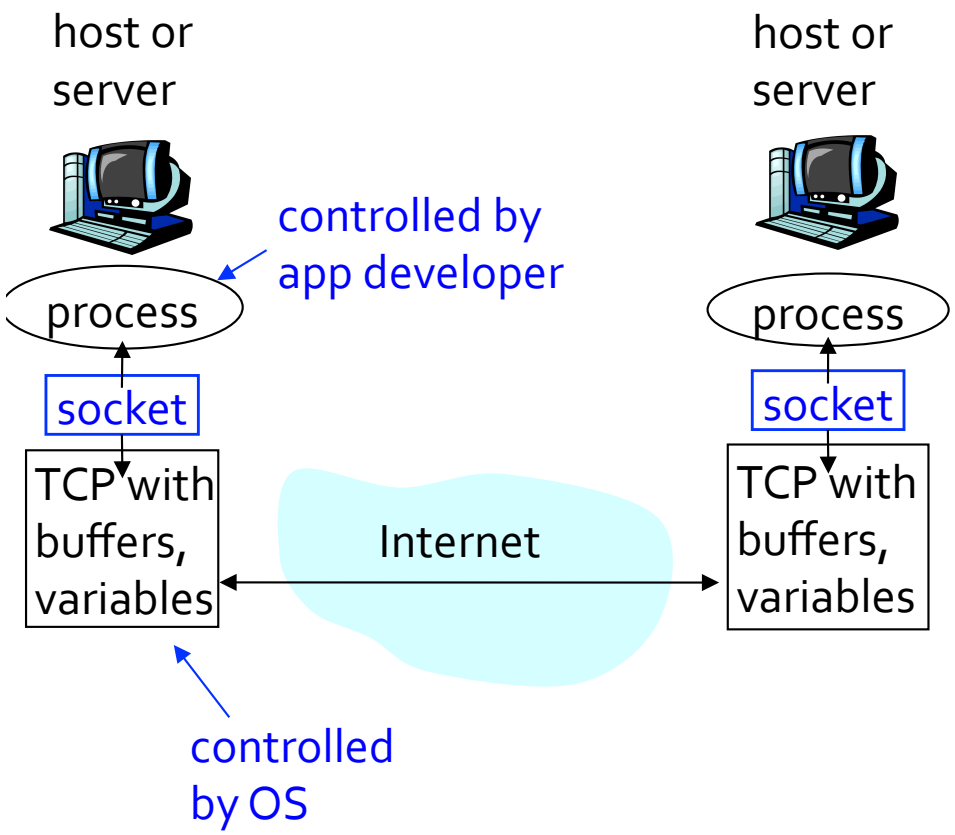
Sockets

- Remember, the OS creates abstractions to make hardware easier to use and manages shared hardware resources
- How do we share the network?

- A network **socket** is an endpoint of a connection
  - Identified by IP and port of both sender and receiver
- A **socket API** is provided by the operating system, and lets application programs use network sockets

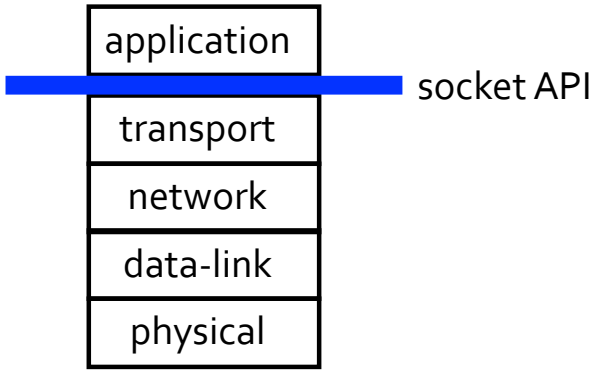
Sockets

- A process sends messages to its socket
- A process receives messages from its socket



Sockets API

- An API to access the network
- A host may support many simultaneous application processes, each with one or more sockets



Sockets at the command line

Server

```
$ nc -l localhost 9000
hello world
```

Client

```
$ echo "hello world" |
    nc localhost 9000
```

Socket example

```
$ echo "hello world" |
    nc localhost 9000
```

At the client

- Translate `localhost` into IP address `127.0.0.1`
- Decide to use the TCP protocol
- Create a socket
- `connect` to `127.0.0.1` at port `9000`
- Parcel out your message into packets
- `send` the packets out

Socket example

On the internet, packets were

- Transmitted
- Routed
- Buffered
- Forwarded, or
- Dropped

- NOTE: because this example uses `localhost` and `localhost`, your OS will avoid the internet

Socket example

```
$ nc -l localhost 9000
hello world
```

At the receiver (server)

- Receiver was started ahead of time
- Create socket
- Decide to use TCP
- `bind` the socket to port `9000`
- `listen` on the socket
- `accept` the connection request
- `recv` your packets

Socket server in Python

```
import socket
if __name__ == "__main__":
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # bind the socket to the server
    s.bind(("localhost", 9000))
    s.listen(5)

    while True:
        # connect
        clientsocket, address = s.accept()
        print "Server: connection from ", address

        # Receive a message
        message = clientsocket.recv(1024)
        clientsocket.close()
        print message
```

Details

- `AF_INET` : use IPV4 addresses
  - `AF_INET6` : use IPV6 addresses
- TCP / `SOCK_STREAM` is a connection-based protocol (AKA TCP)
  - UDP / `SOCK_DGRAM` is a datagram based protocol (AKA UDP)
- `listen(5)` queue up as many as 5 connect requests before refusing outside connections
  - If the rest of the code is written properly, that should be plenty

Socket server in Python

Server

```
$ python test_server.py
Server: connection from
('127.0.0.1', 60980)
hello world
```

Client

```
$ echo "hello world" |
    nc localhost 9000
```

Socket server in Python

Server

```
$ python test_server.py
Server: connection from ('127.0.0.1', 60980)
hello world
```

Client

- When a TCP server accepts a new client, it returns a new socket to communicate with the client
- Allows the server to communicate with multiple clients

Socket client in Python

```
import socket

if __name__ == "__main__":
    # create an INET, STREAMing socket
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # connect to the server
    s.connect(("localhost", 9000))

    # send a message
    message = "hello world!"
    s.sendall(message)
    s.close()
```

Socket client in Python

Server

```
$ nc -l localhost 9000
hello world
```

Client

```
$ python test_client.py
```

Socket client/server in Python

Server

```
$ python test_server.py
Server: connection from
('127.0.0.1', 60980)
hello world
```

Client

```
$ python test_client.py
```

Why do we care?

- Project 4 will use processes, threads and sockets to implement a Map Reduce server
- We need to do multiple things in parallel
  - Example: a master and N workers
    - N+1 processes
  - Example: a worker's task (a map function) and a worker's heartbeat
    - 2 threads
- We need to communicate between several machines
  - Master communicates with workers using sockets