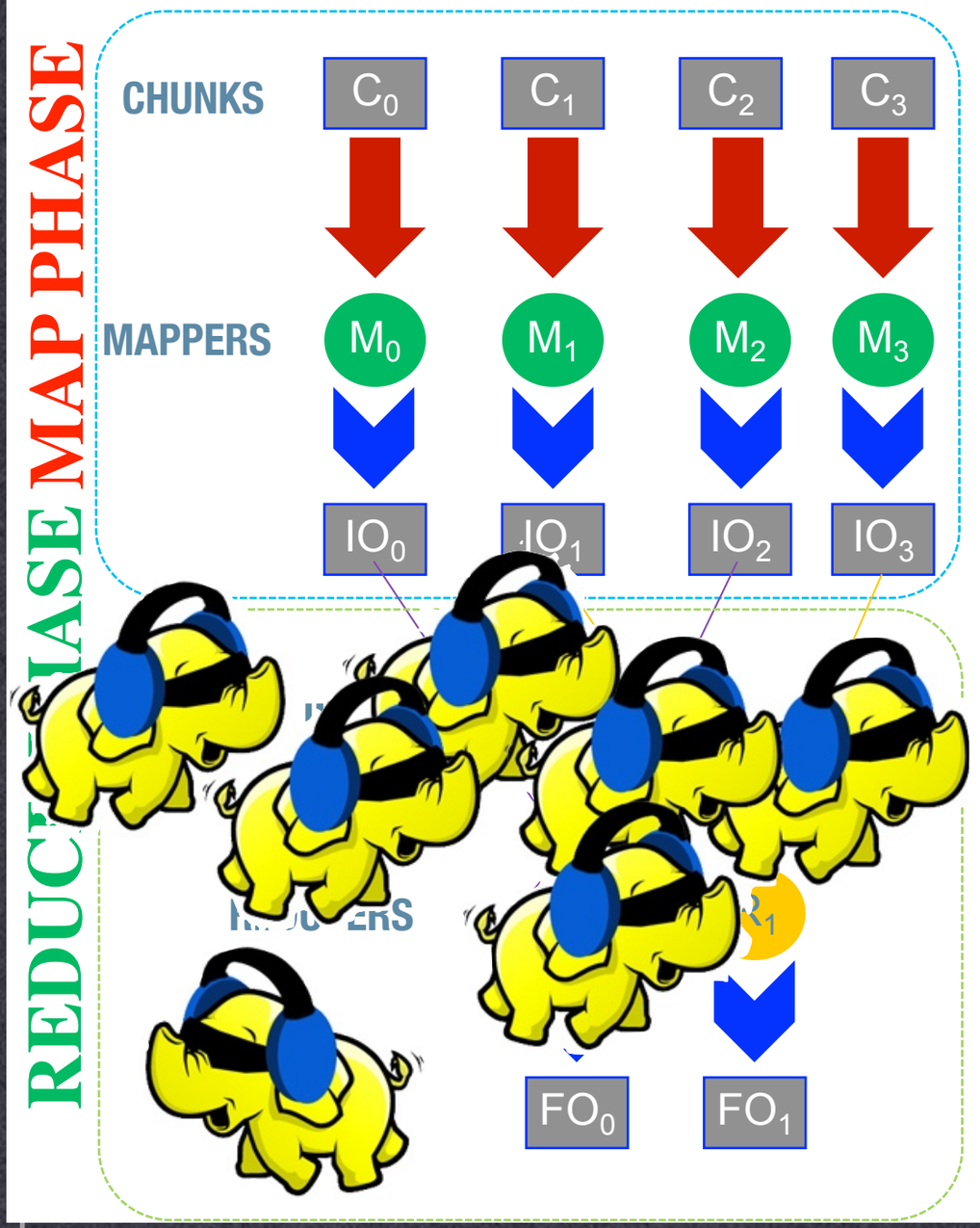
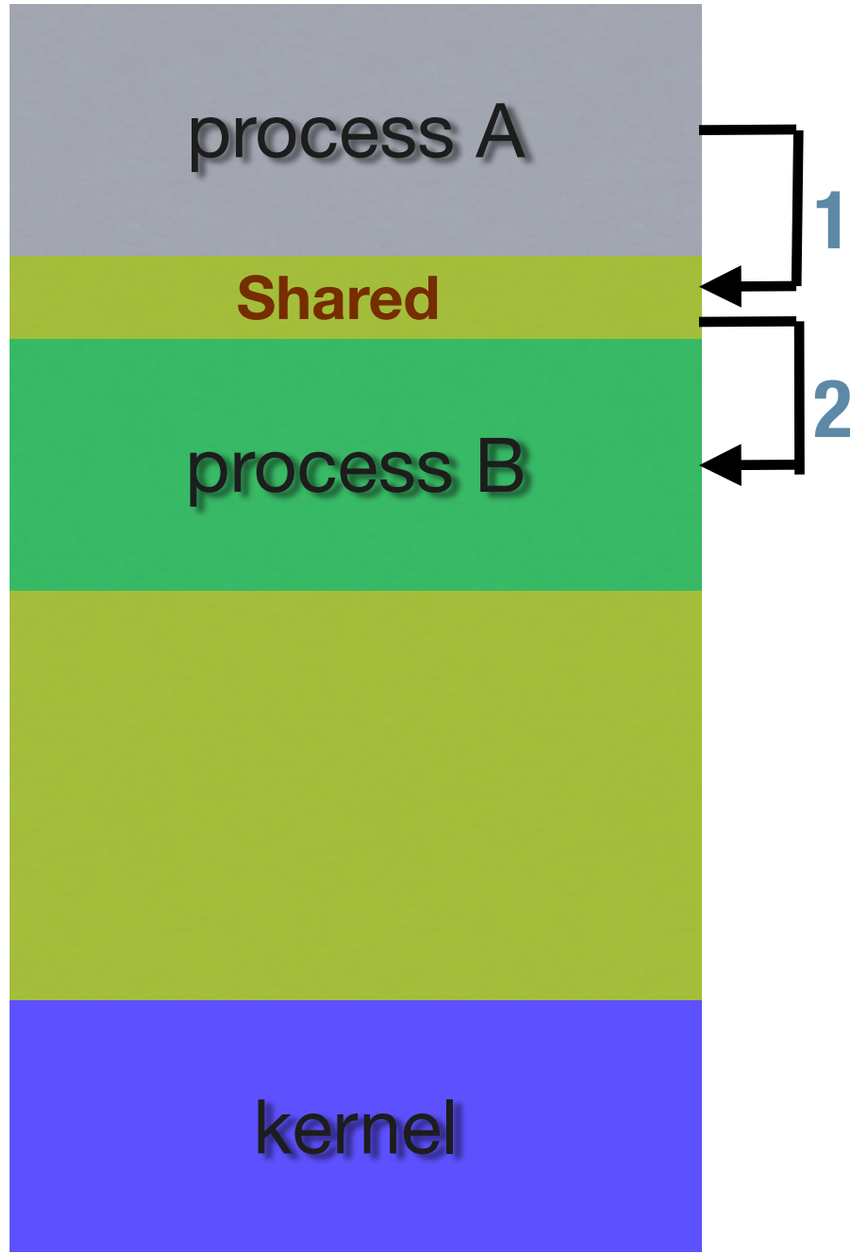


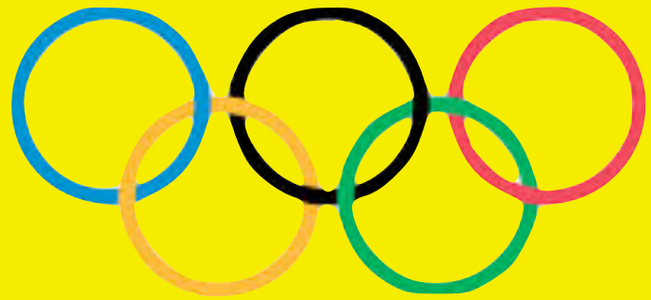




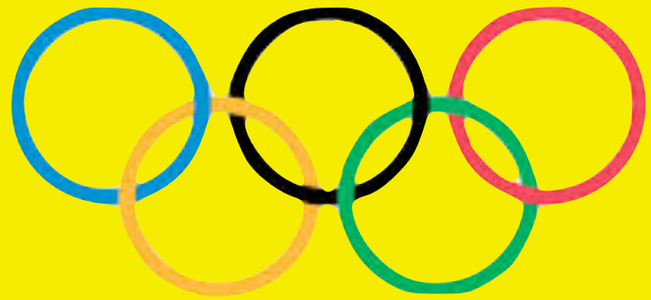

# DISTRIBUTED SYSTEMS

## PROCESSES, THREADS, LOCAL SYNCH

# Activity: Distributed Systems Olympics



- Form groups of three
- Write down three examples of distributed systems
- Decide as a group on your favorite and tell me why
- Tell me what your favorite is
- Give me the piece of paper – we will talk about your rejects at the end of the course



- Asynchronous, lagging, communication
- Communication is not instantaneous
  - time matters
- Failure is not an option
- http://www.youtube.com/watch?v=h2I8AoB1xgU

# Welcome (back to Earth)!

- Last's week lecture introduced the course
- Grand challenges in distributed systems
  - synchronization
  - transparency
  - heterogeneity
  - openness

# Welcome (back to Earth)!

- Today's lecture will be all about processes, threads, local coordination & communication
- Is this an OS course?!
  - No
  - Concepts are nevertheless essential

# Program, process, threads

- What is a program?
- What is a process?

# What is a process?

- Process: an execution stream (or program) in the context of a particular state
- 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

# Why processes?


- Express concurrency
  - a web server can spawn multiple processes to take care of requests in parallel
  - managed by OS
- Follows divide and conquer
- Processes are isolated from each other
  - work sequentially
  - well-defined interfaces

# Address Spaces

- All memory a process can address
- One process, one address space
- OS performs the isolation
  - one process cannot access other's address space
  - same pointer address in different process point to different memory

# Context switching

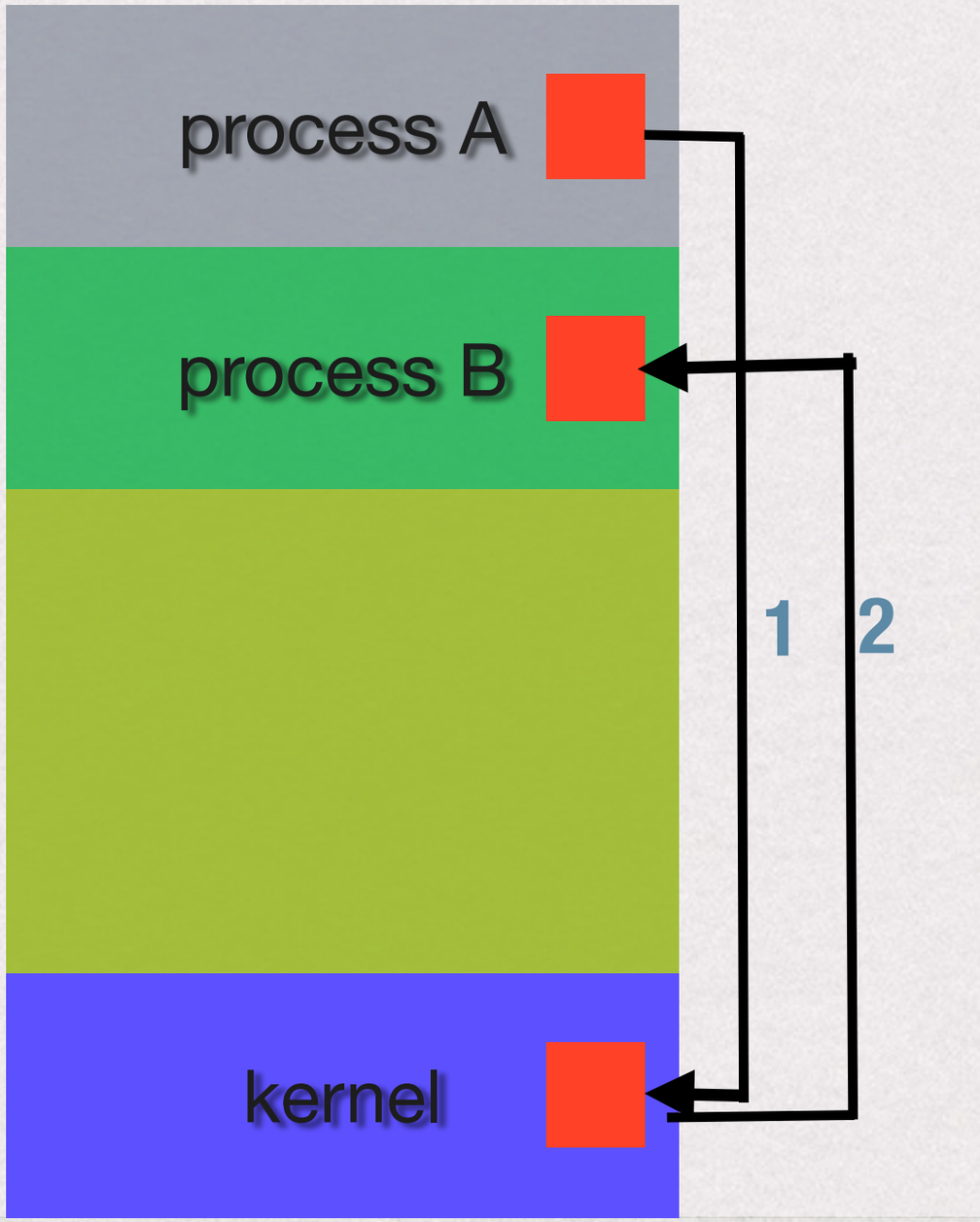
- Storing and restoring the state of a process such that execution can be resumed; crucial for multitasking
  - processes can wait for I/O or some other synchronization operation: other processes can run!
  - need to save and load register and memory maps, update tables, etc.
- Performance
  - Expensive
  - Needs to be limited

# Inter-process Communication Example

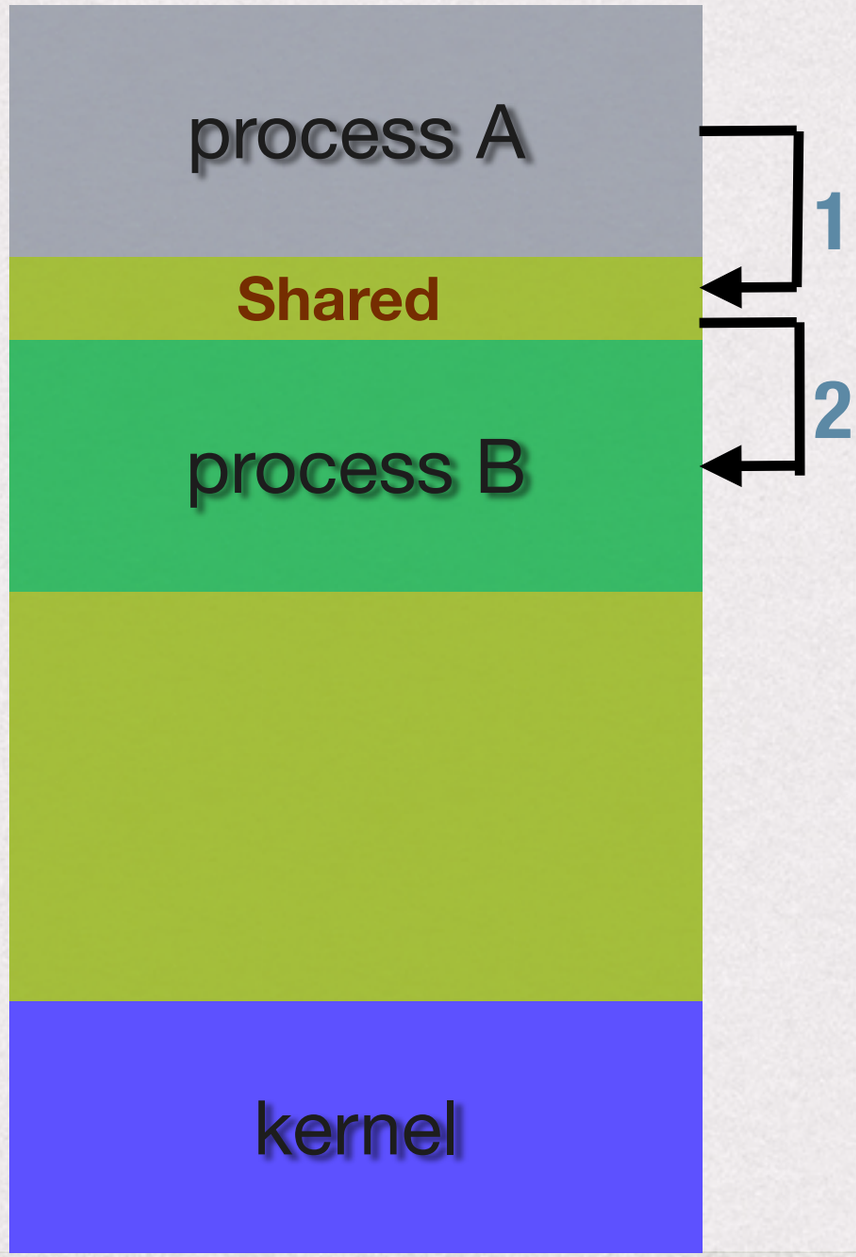
- Multiple processes are part of the same program
  - they need to coordinate
- Examples
  - # grep “DS is awesome” files | sort
  - servers like Apache spawn child processes to handle requests
  - producer-consumer - like communication

# Fundamental Architectures

**Message Passing**




**Shared Memory**



# Message Passing vs. Shared Memory

- Message passing
  - Good: all sharing is explicit
  - Bad: large overhead (copying data, context switching)
- Shared memory
  - Good: less overhead
  - Bad: no control about what changes on the shared space

Talk is cheap.

# Show me the code. MPI

```
 int argc;
   char *argv[];
{
  int rank, size;


  MPI_Init (&argc, &argv);     /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);     /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);      /* get number of processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

Talk is cheap.

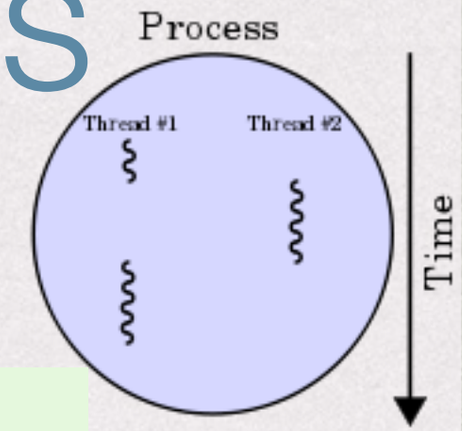
# Show me the code. OpenMP

```
int main (int argc, char *argv[]) {

int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */

#pragma omp parallel private(nthreads, tid)

 {

/* Obtain thread number */

tid = omp_get_thread_num();

printf("Hello World from thread = %d\n", tid);

/* Only master thread does this */

if (tid == 0)

{

 nthreads = omp_get_num_threads();

printf("Number of threads = %d\n", nthreads);

}

} /* All threads join master thread and disband */

}
```

# Threads

- Separate streams of execution that share one address space
- Thread state (not shared)
  - Program counter
  - Other registers
  - Stack
- Conceptually similar to processes, but different – Often called “lightweight processes”
- Threads in memory: each thread must have its own separate stack

# Threads vs processes

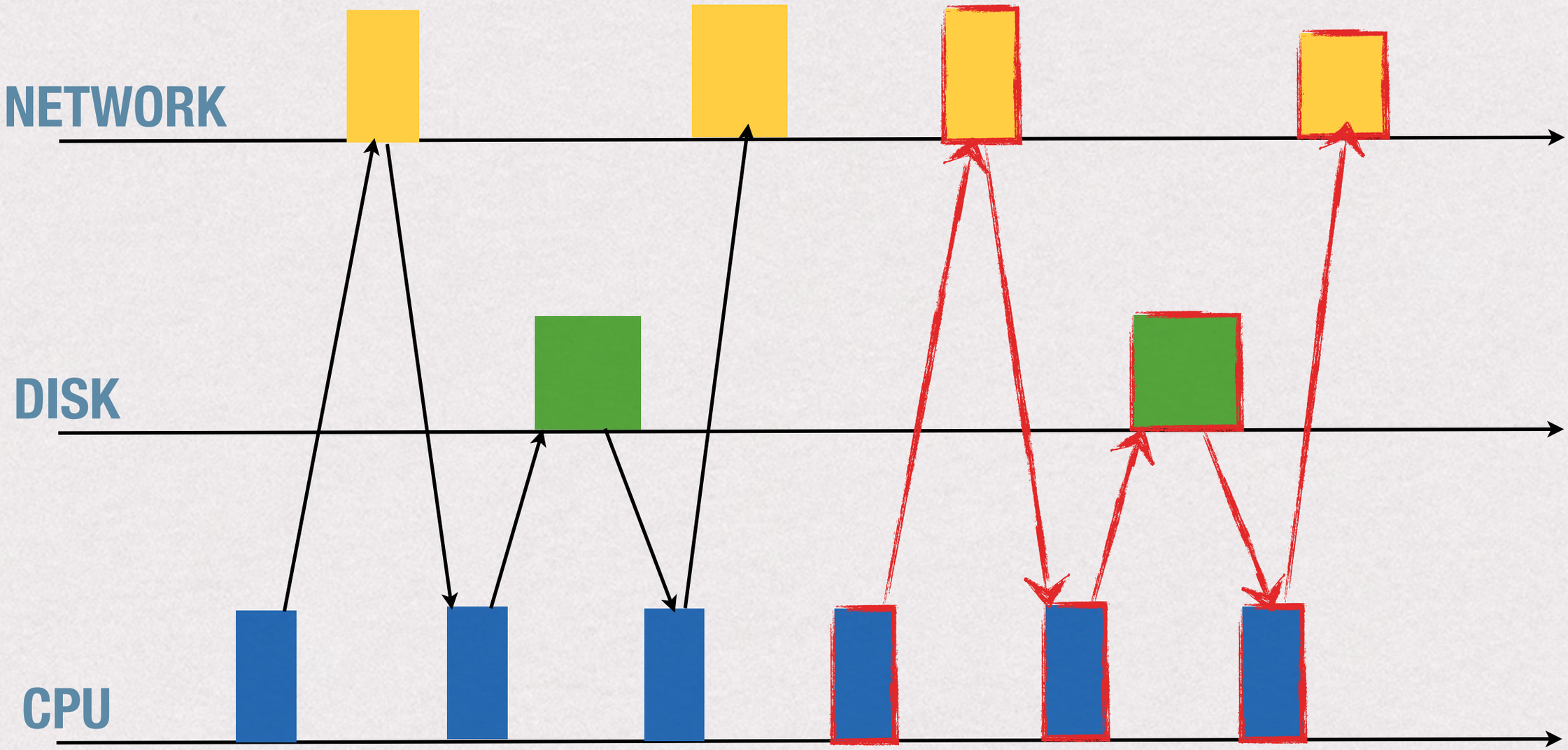



- Threads
  - allow running code concurrently within a single process
  - Switching among threads is lightweight
  - Sharing data among threads requires no IPC
- Processes
  - Fault isolation: One buggy process cannot crash others - see Chrome implementation

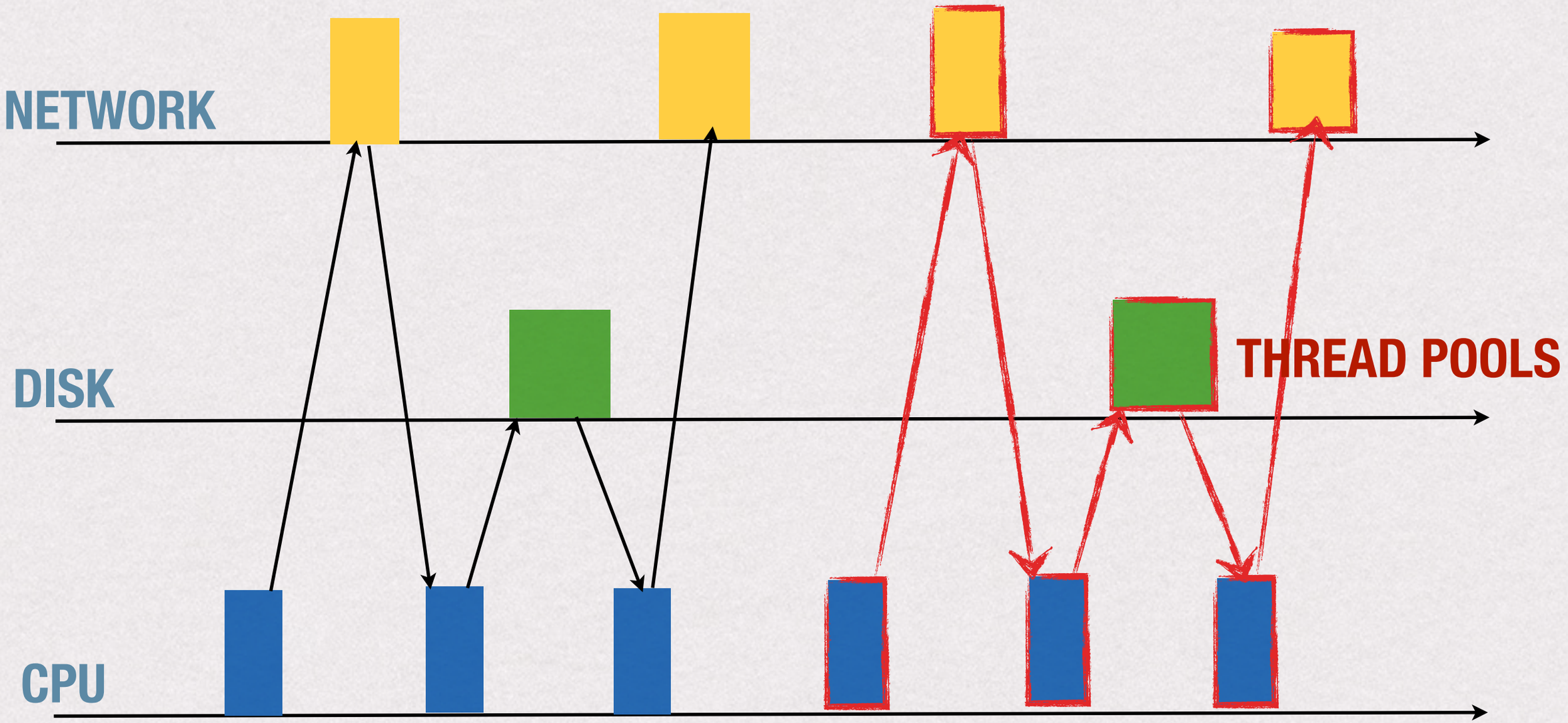
# Multi-threaded programming

- 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:
  - threads in a web application: sending email after registration

# Single-threaded server



# Multi-threaded server



# How?

```
while (true) {
        try {
          clientSocket = serverSocket.accept();
          int i = 0;
          for (i = 0; i < maxClientsCount; i++) {
            if (threads[i] == null) {
              (threads[i] = new clientThread(clientSocket, threads)).start();
              break;
            }
          }
          if (i == maxClientsCount) {
            PrintStream os = new PrintStream(clientSocket.getOutputStream());
            os.println("Server too busy. Try later.");
            os.close();
            clientSocket.close();
          }
        } catch (IOException e) {      }
```

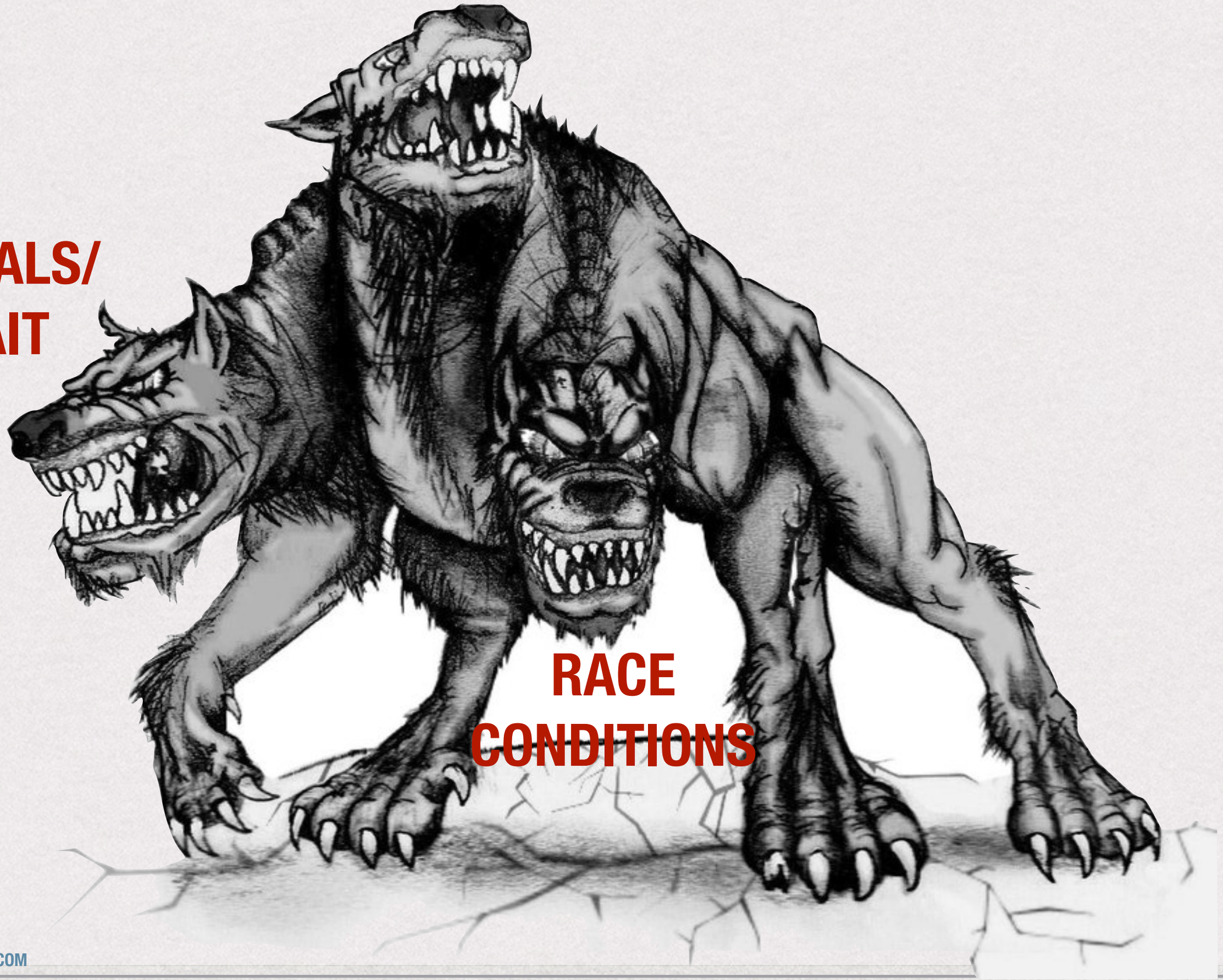
```
 public clientThread(Socket clientSocket, clientThread[] threads)
{
    this.clientSocket = clientSocket;
    this.threads = threads;
    maxClientsCount = threads.length;
  }

  public void run() {
    int maxClientsCount = this.maxClientsCount;
    clientThread[] threads = this.threads;
    //do magick!!!
}
```

# Thread synchronization

- Memory is shared across all threads
- Coordination is needed!

# DEADLOCK

# SIGNALS/ WAIT



# RACE CONDITIONS

# Example

```
int balance = 1000;

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

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;}
```

# Discuss

a) 200

b) -600

c) 1000

# Result? Vote!

a) 200

b) -600

c) 1000

e) https://www.mentimeter.com/app/presentation/alh3h6g8fmeuek7yv5sxm9py7ewhy7bh/9nwggkzz5hhk

# Schedule

```
matterhorn:resources gecko$ ./bank

ATM gives user $800

All done: balance is $200
```

## THREAD 1

```
if (balance >= amount) {

        balance -= amount;
```

## THREAD 2

```
if (balance >= amount) {

        balance -= amount;
```

# Schedule

```
matterhorn:resources gecko$ ./bank

ATM gives user $800

ATM gives user $800

All done: balance is $-600
```

## THREAD 1

```
if (balance >= amount) {

        balance -= amount;

        printf ...
```

## THREAD 2

```
if (balance >= amount) {

        balance -= amount;

        printf ...
```

# Schedule

```
matterhorn:resources gecko$ ./bank
ATM gives user $800
ATM gives user $800
All done: balance is $200
```

## THREAD 1

```
if (balance >= amount) {

        balance -= amount;

        printf ...
```

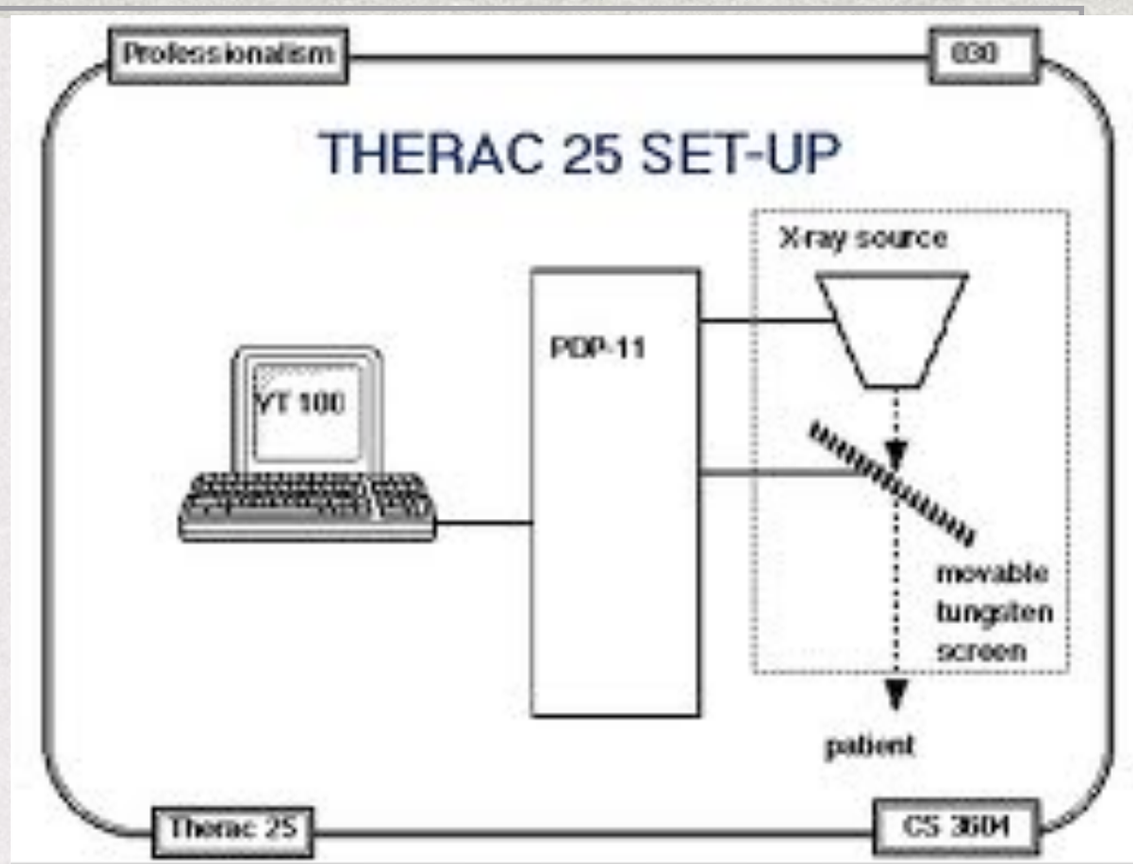
## THREAD 2

```
if (balance >= amount) {

        balance -= amount;

        printf ...
```

# Race conditions

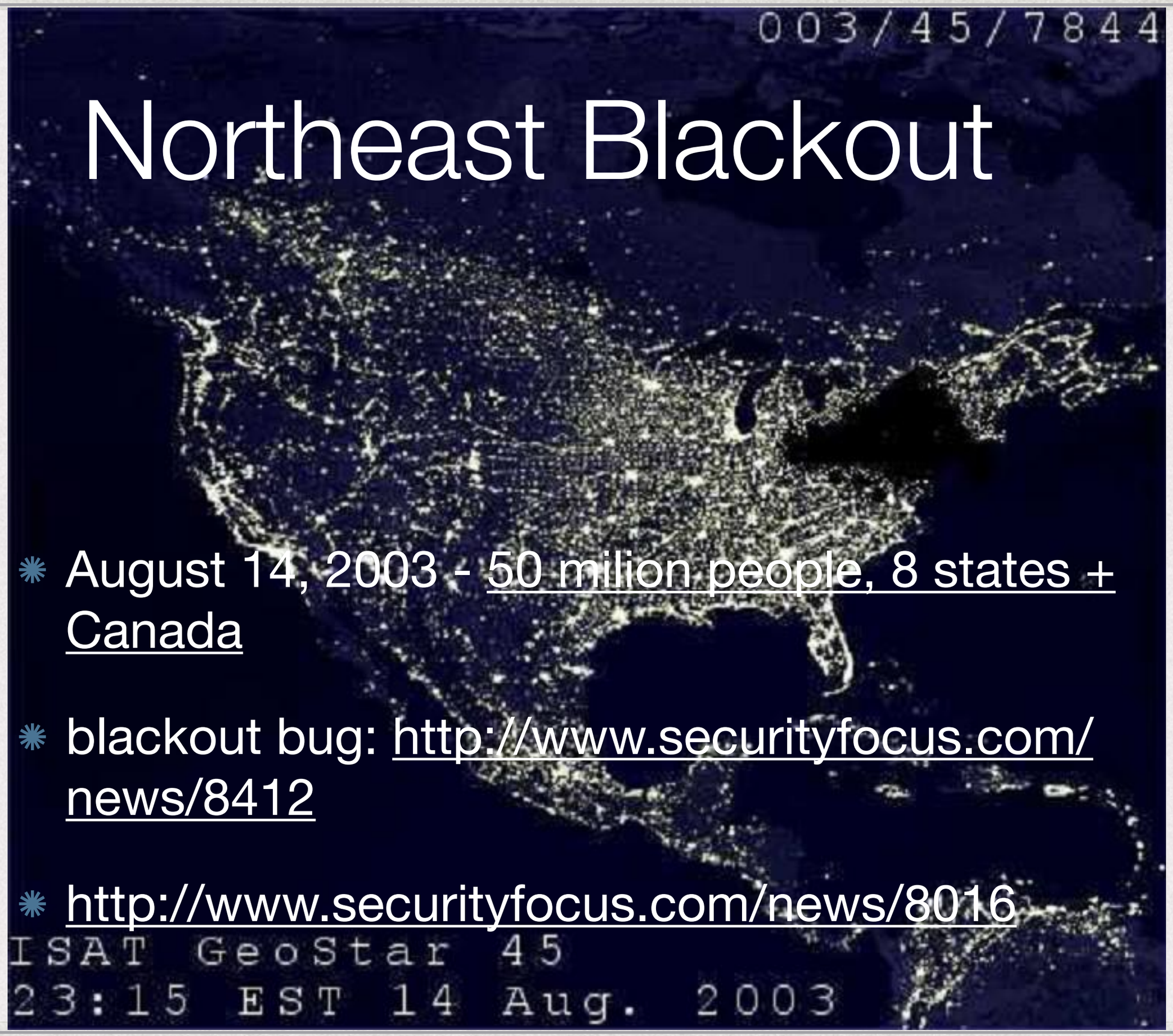
- Definition: a timing dependent error involving shared state
- Can be disastrous
  - Non-deterministic: don't know what the output will be, and it is likely to be different across runs
  - Hard to detect: too many possible schedules
  - Hard to debug: debugging changes timing so hides bugs ("heisenbug")
    - W. Heisenberg: the act of observing a system fundamentally changes its state

# Therac 25




- Radiation therapy machine
- Race condition gave patients 100 times intended radiation dose

# Northeast Blackout




- August 14, 2003 - 50 milion people, 8 states + Canada
- blackout bug: http://www.securityfocus.com/news/8412
- http://www.securityfocus.com/news/8016

# Discuss

a) How would you solve synchronisation problems?

# Synchronization Mechanisms

- Used extensively in distributed systems
- Multiple mechanisms, each solving a different problem
  - Locks
  - Condition variables
  - Semaphores
  - Monitors
  - Barriers

# Locks

- Allow only one thread to pass through a critical section at any time
  - lock: acquire lock exclusively; wait if not available
  - unlock: allow other threads to have access to lock

# Locks in Java

```
public class SynchronizedCounter {

    private int c = 0;

    public synchronized void increment() {

        c++;

    }

    public synchronized void decrement() {

        c--;

    }

    public synchronized int value() {

        return c;

    }}
```

http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

# Common pitfalls

- Wrong lock granularity
  - Too small: leads to races - why?
  - Too large: leads to bad performance - why?
- Deadlocks
  - Better bugs than race
- Starvation

# What does this have to do with DS?

- Every server is multi-threaded
  - need to coordinate: similar to IPC, shared memory, locking, barriers, etc.
- When testing, we will use multi-threaded programs to start multiple clients