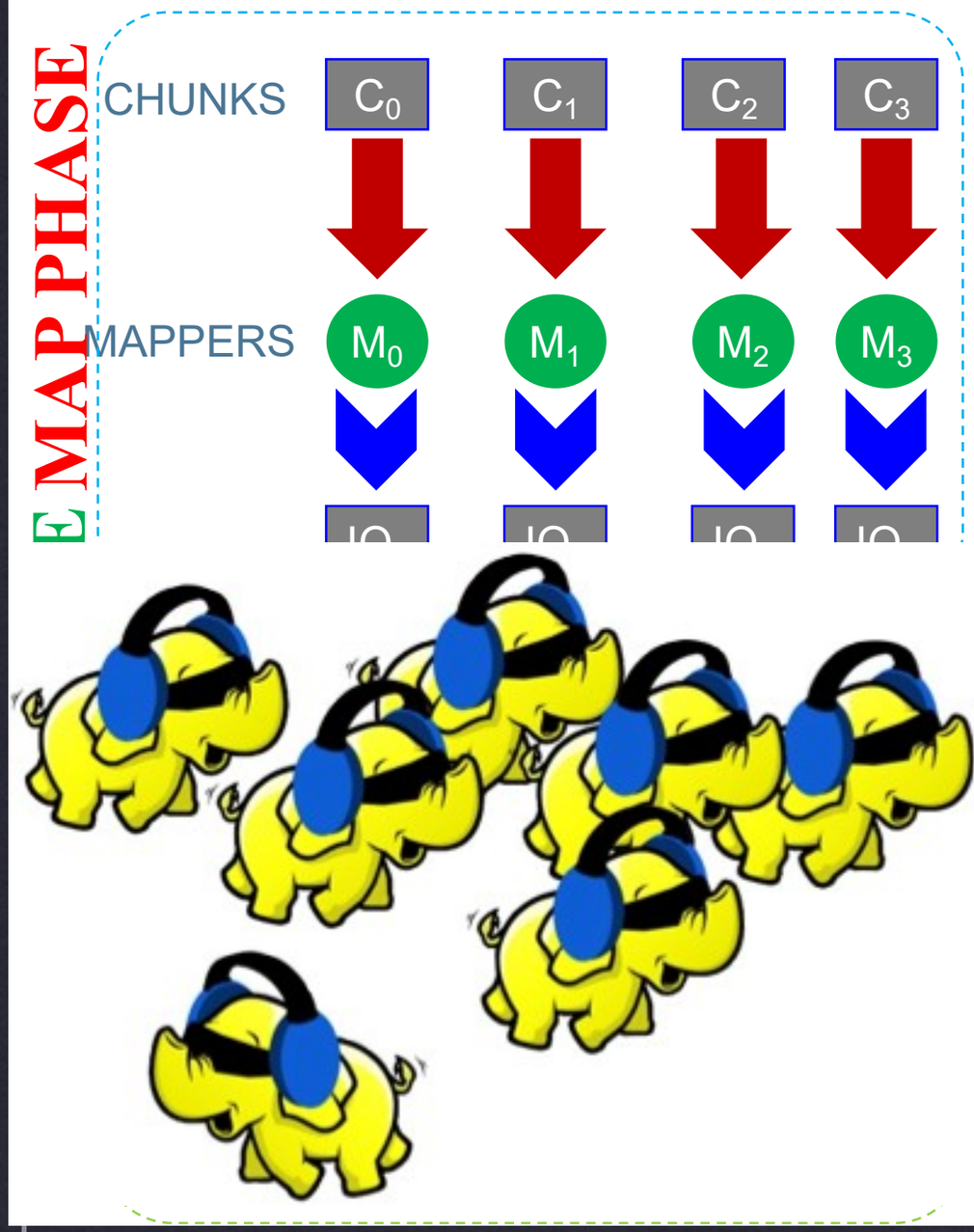
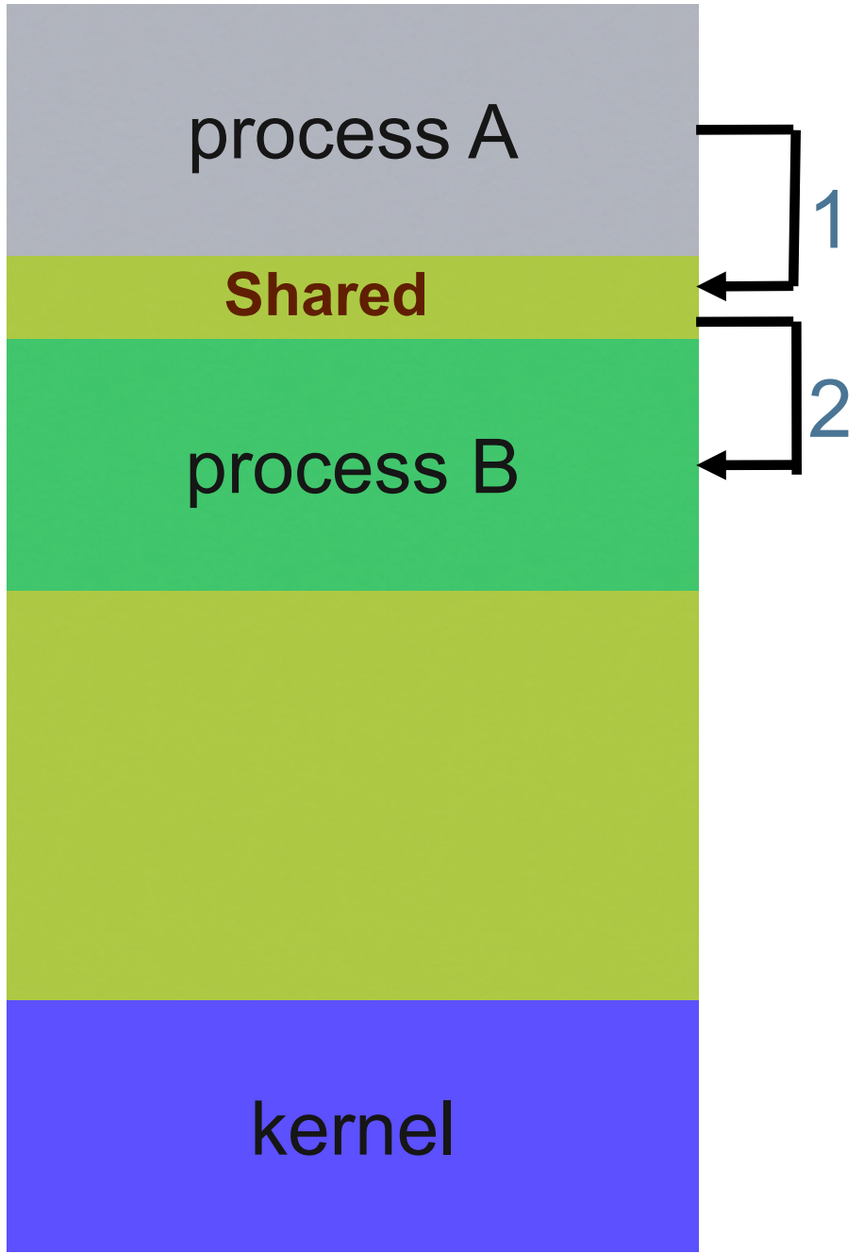




distributed systems

PROCESSES, THREADS, LOCAL SYNCH

Activity: Distributed Systems Olympics

- Form groups of three for a debate
- Write down three examples of distributed systems
- Decide as a group on your favorite
- Tell me what your favorite is and why?

Awesome example

- 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
- Systems talk: Google
- 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 ***process state***
- Process state: Everything running code ***can affect*** or ***be affected by***
  - **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 and static data
  - Process: dynamic instantiation of code and data
- Program : process: no 1:1 mapping
  - Process is more than code and data
  - one program can invoke many processes
  - 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
  - Independent sequences of executions
  - Has its own **address space**

Address Space

- 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

Address Spaces

How do you have all these processes with their own addresses space executing on the same CPU?

Context switching

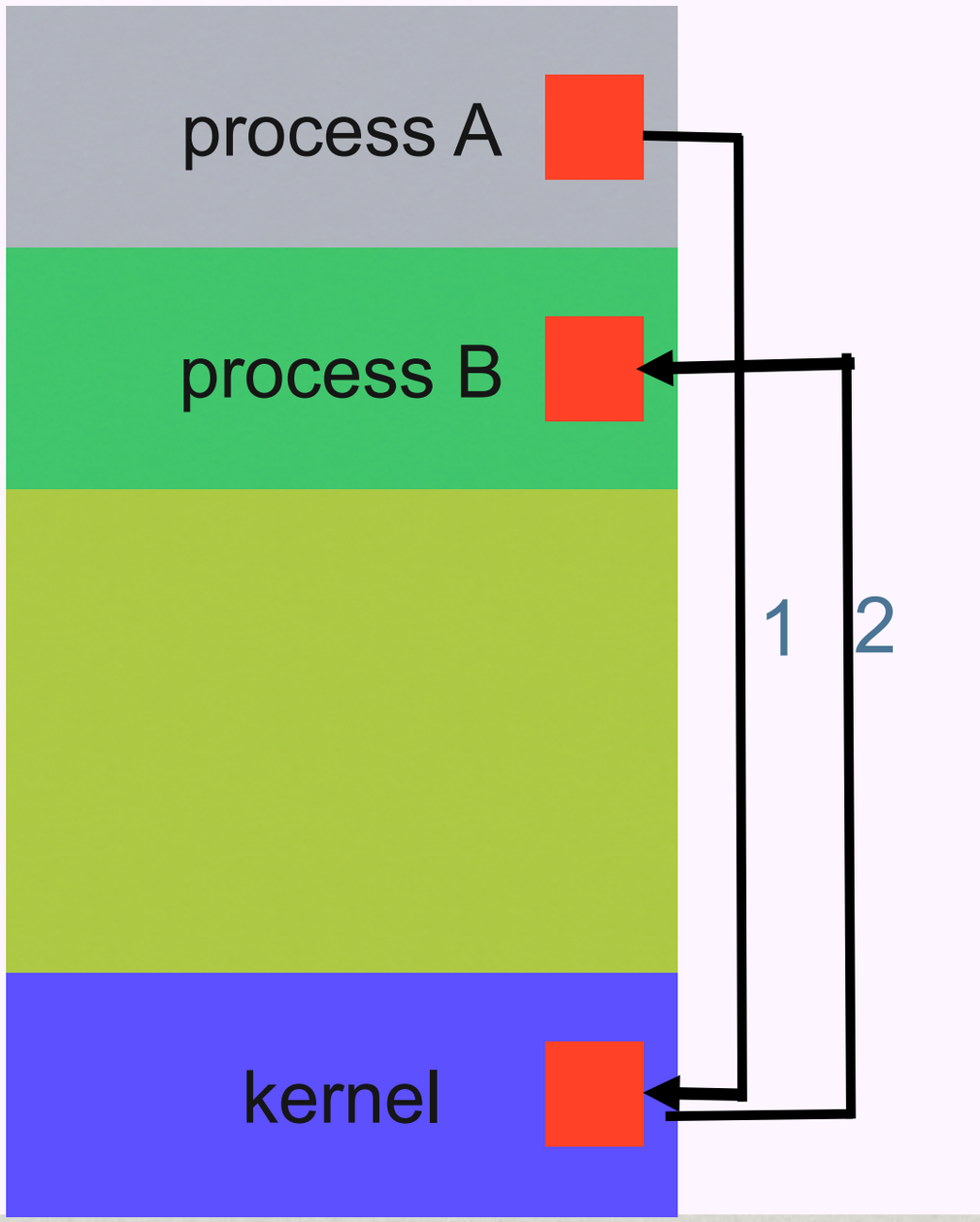
- Storing and restoring the state of a process such that execution can be resumed; crucial for multitasking
  - processes can wait for I/O or some other synchronization operation: other processes can run!
  - need to save and load register and memory maps, update tables, etc.
- Performance
  - **Expensive**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

Inter-process Communication Example

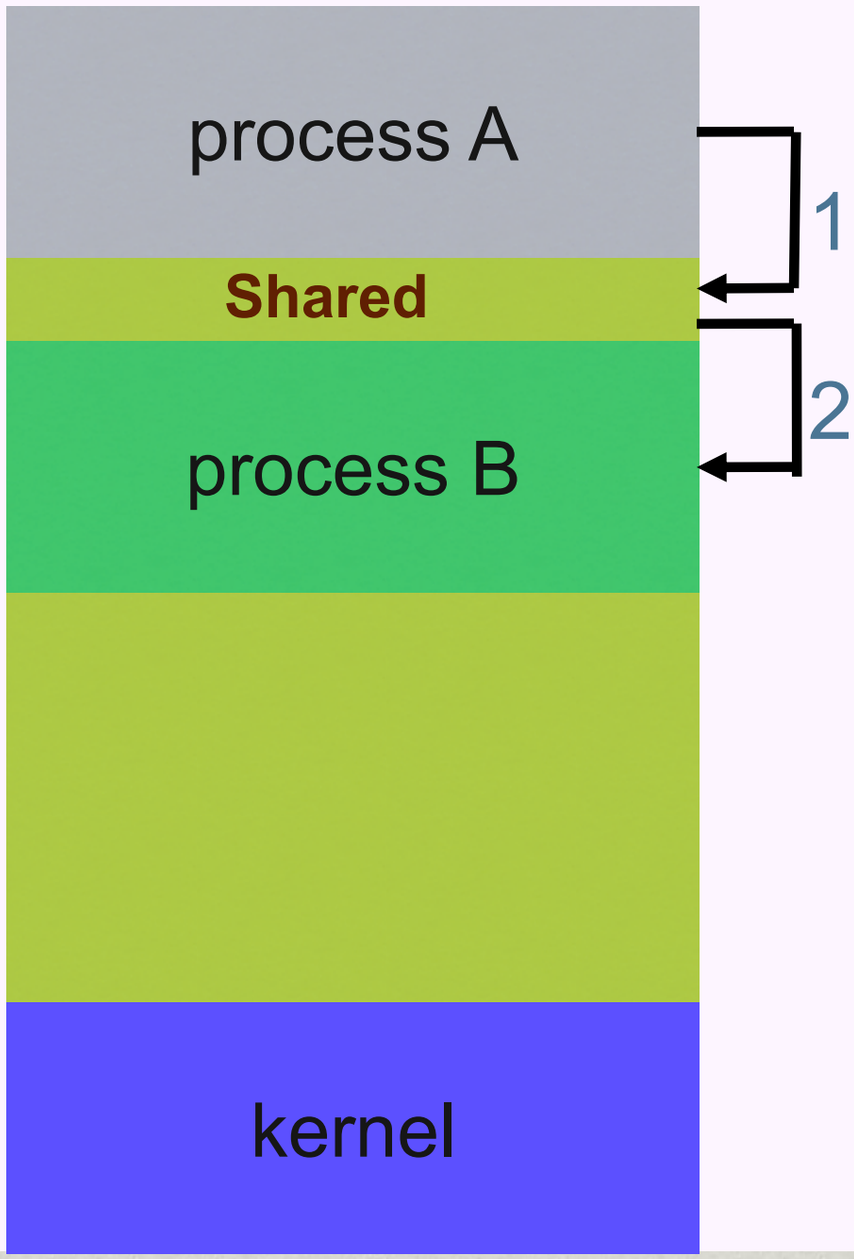
- Multiple processes are part of the same program
  - they need to coordinate
- Examples
  - grep “DS is awesome” files | sort
  - servers like Apache spawn child processes to handle requests
  - producer-consumer - like communication

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**

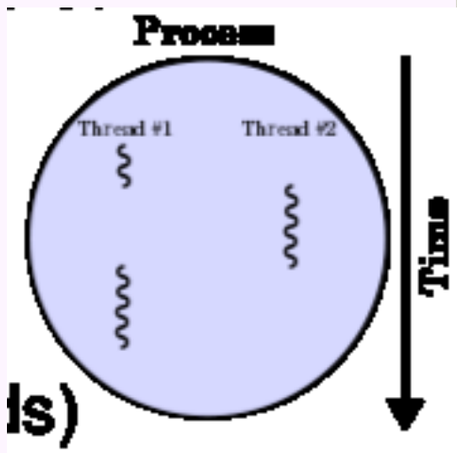
```
int main (int argc, char *argv[]) {
int nthreads, tid;
/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
{
  /* Obtain thread number */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);
  /* Only master 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**
  - **Heap and resources like open flies**
- Thread state (not shared)
  - Program counter
  - Other registers
  - Stack (***each thread has its own separate stack***)
- Conceptually similar to processes, but different – Often called "**lightweight processes**"

Threads vs processes

- Threads
  - allow running code concurrently within a single process
  - Switching among threads is lightweight
  - Sharing data among threads requires no IPC

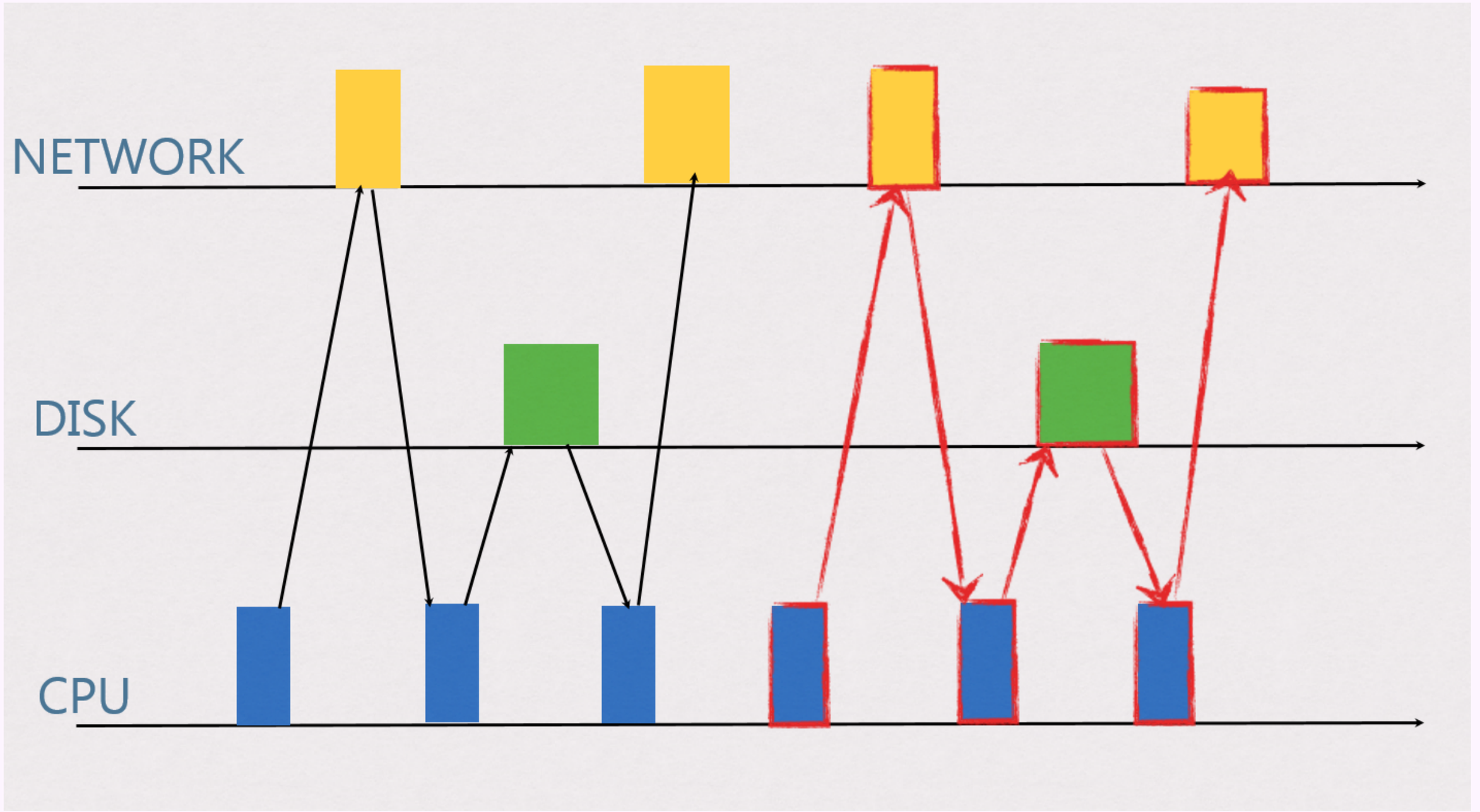


- Processes
  - Fault isolation: One buggy process cannot crash others - see Chrome implementation

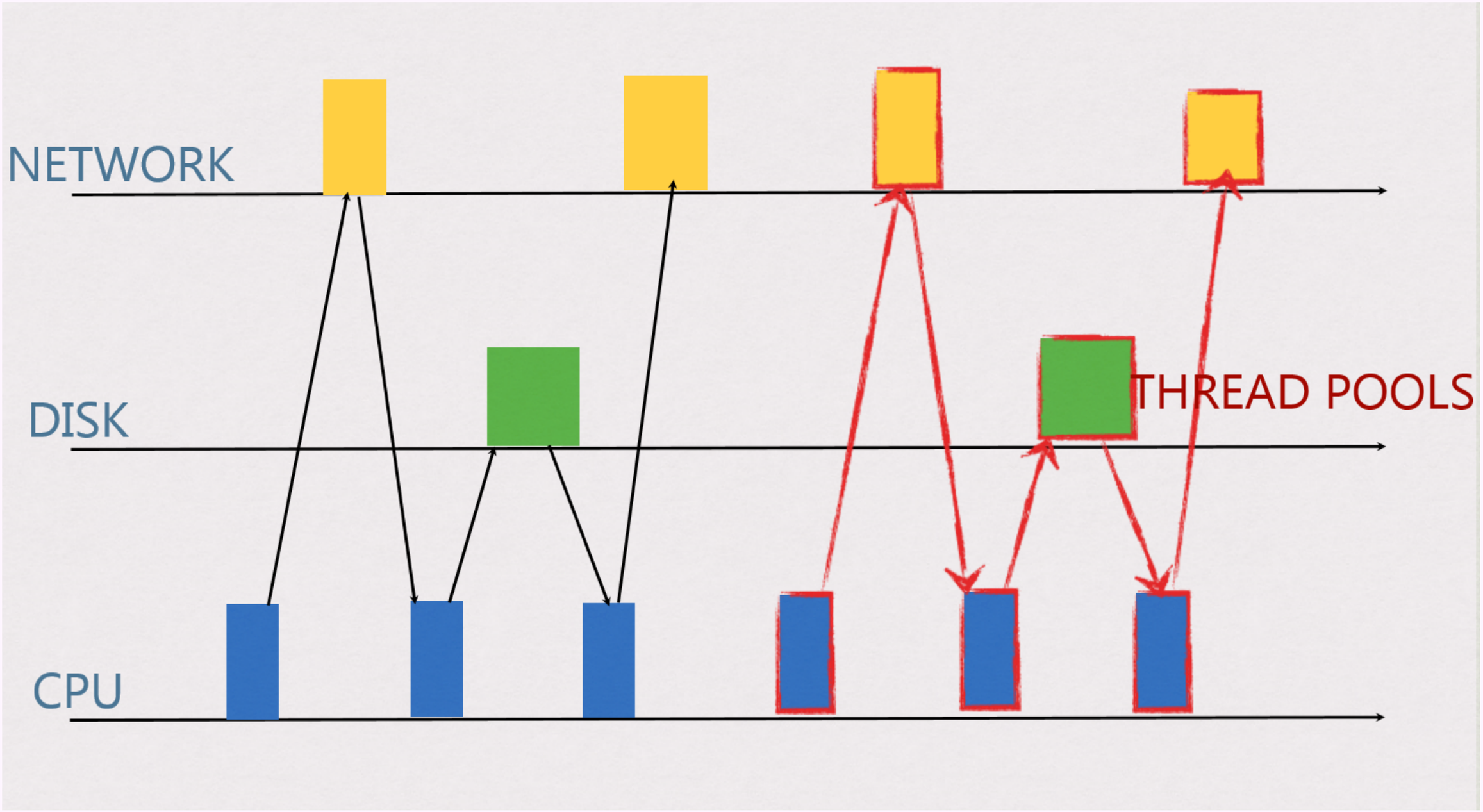
Multi-threaded programming

- **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:
  - **What if your e-mail client is not multi-threaded?**

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) {     }
```

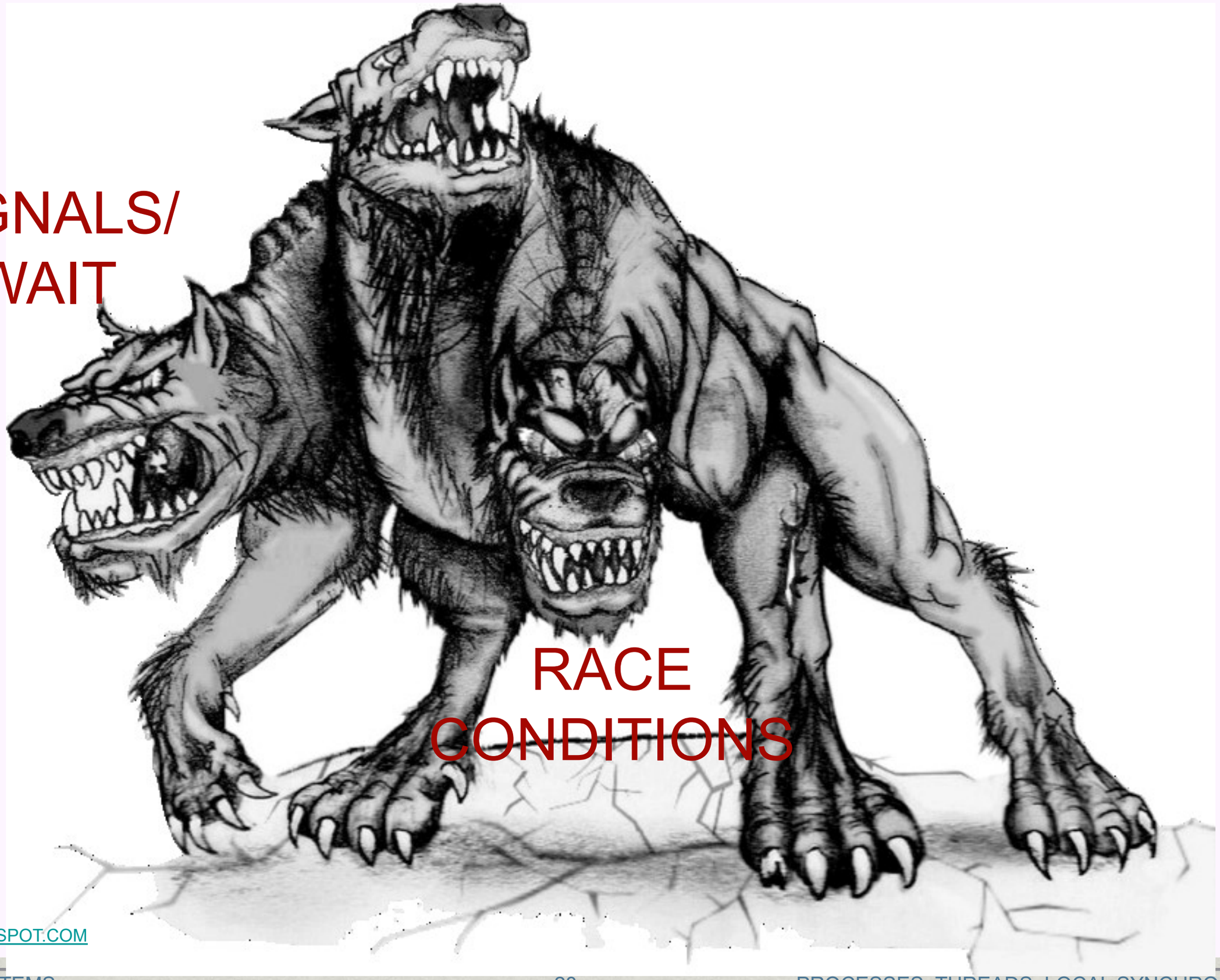
```
public clientThread(Socket clientSocket, clientThread[]
threads) {
    this.clientSocket = clientSocket;
    this.threads = threads;
    maxClientsCount = threads.length;
  }

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

Thread synchronization

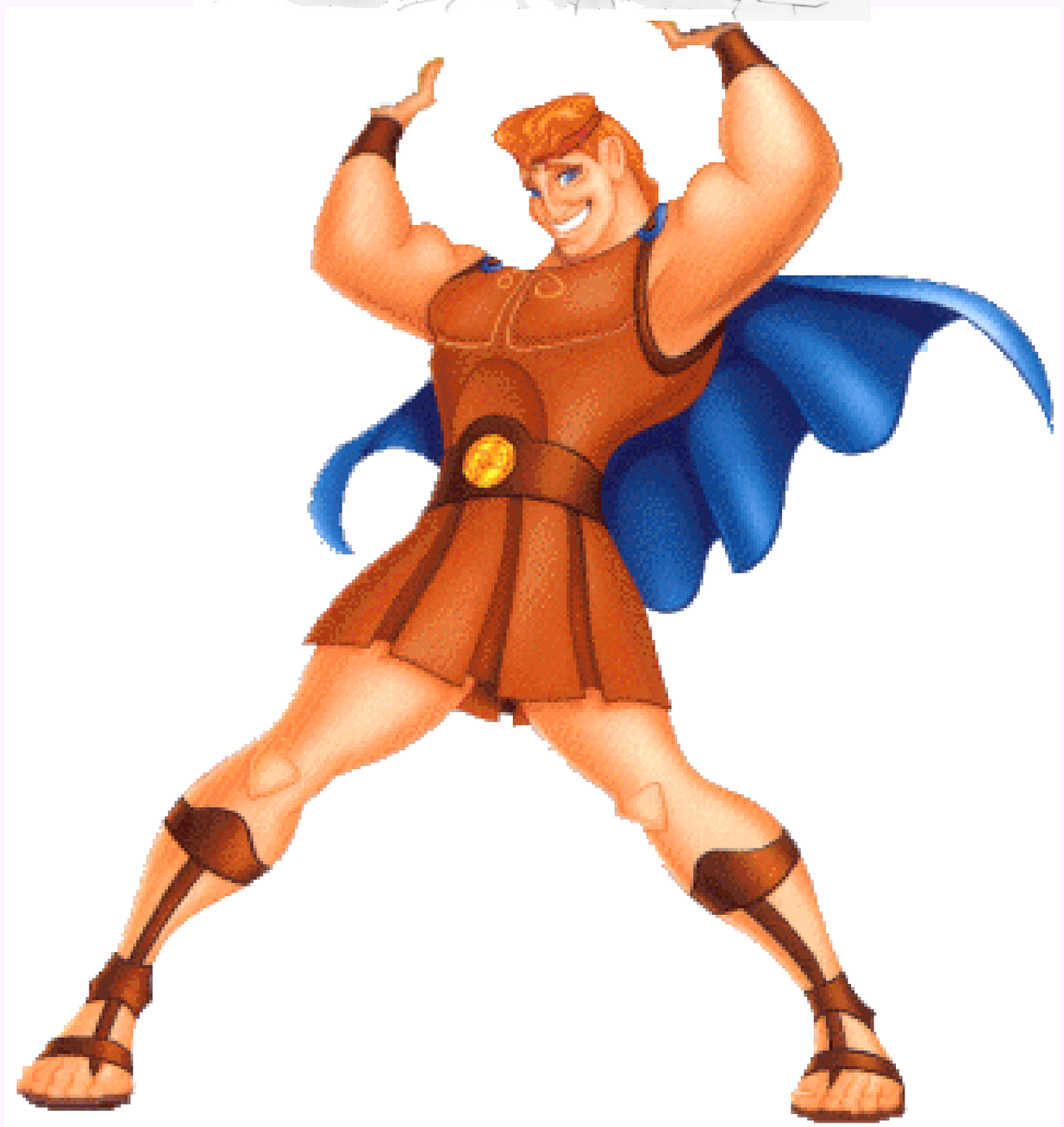
- **Memory** is shared across all threads
- Coordination is needed!

DEADLOCK

SIGNALS/
WAIT



RACE
CONDITIONS



http://www.athenacollege.com.au

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;}
```

Result? Vote!

a)200

b)-600

c)1000

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

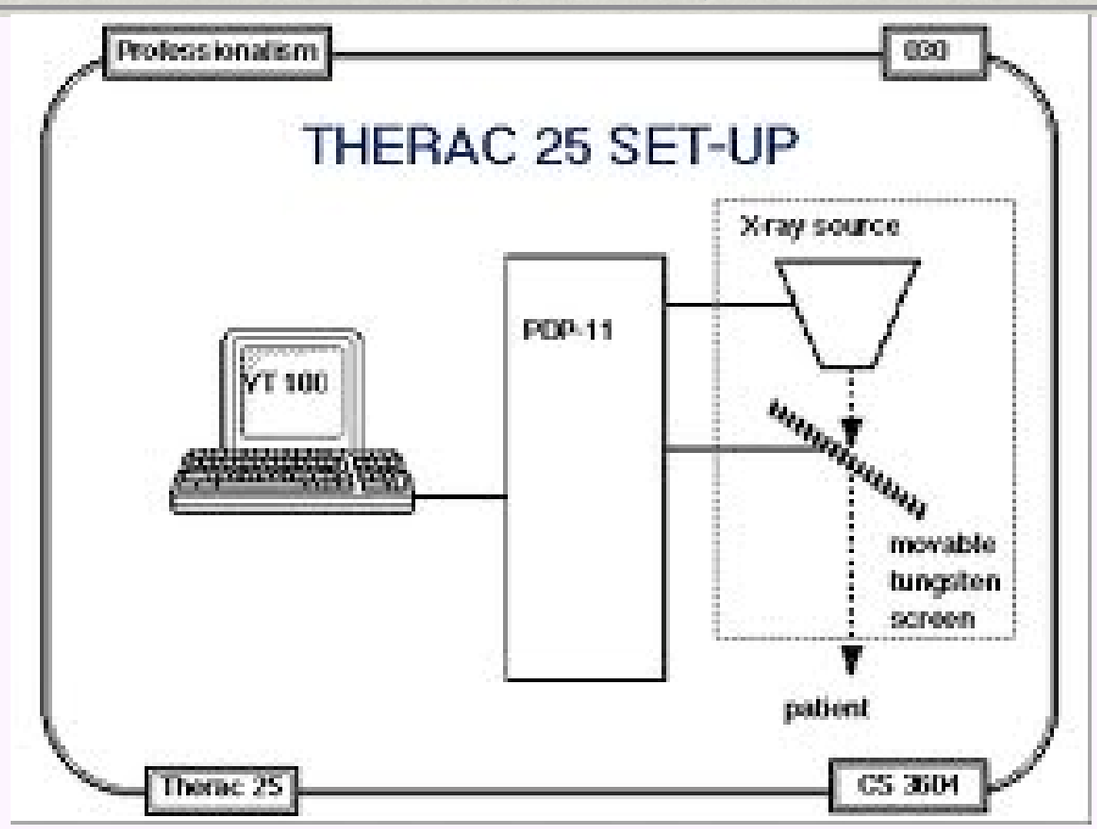
```
if (balance >= amount) {
  balance -= amount;
  printf ...
```

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



d.edu/class/spring2003/cmsc838p/Misc/therac.pdf

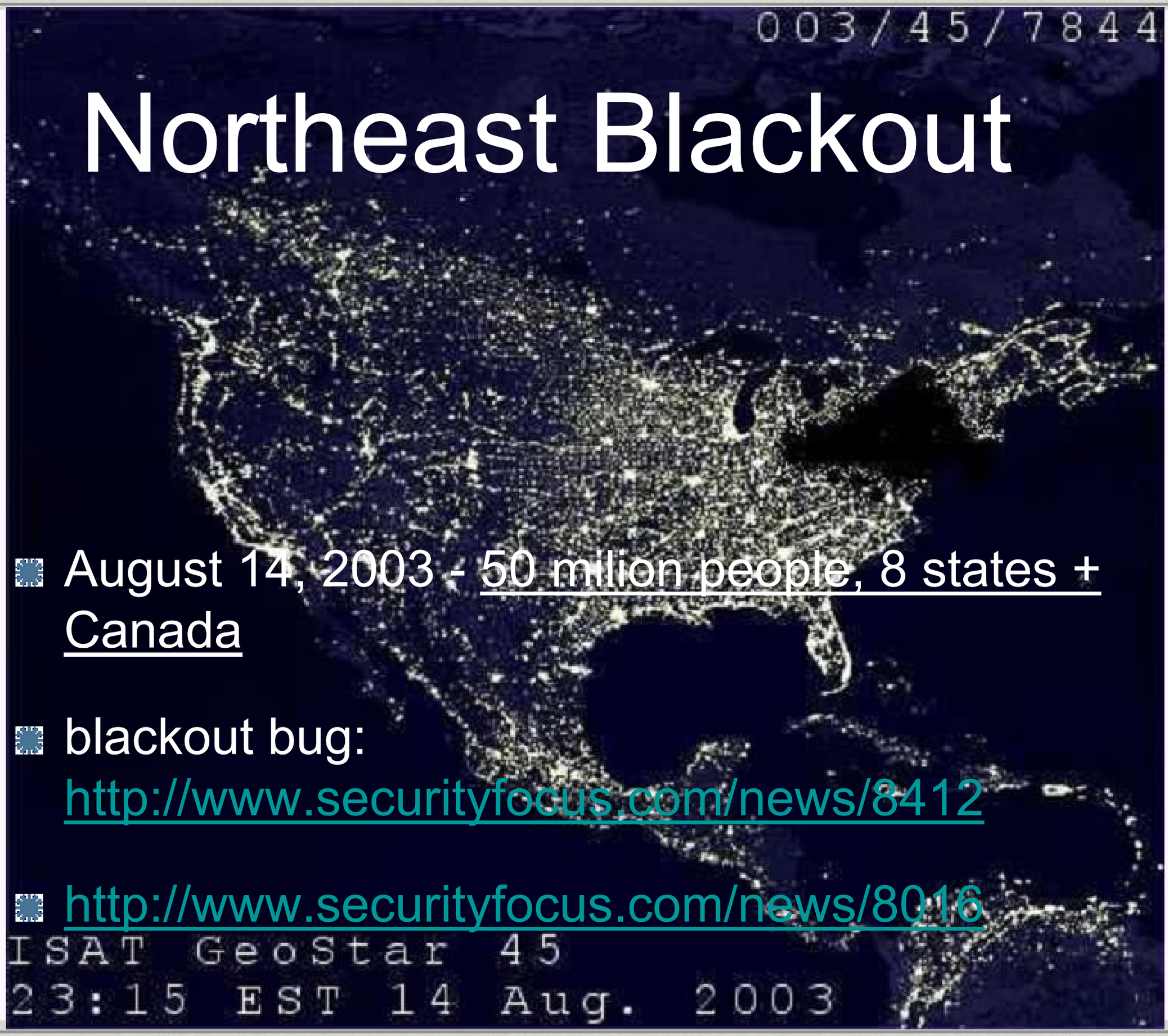
No results Options

arbitrates non-treatment-related communication between the therapy system and the operator.

- Snapshot (run periodically by the scheduler). Snapshot captures preselected parameter values and is called by the treatment task at the end of a treatment.
- Hand-control processor (run periodically).
- Calibration processor. This task is responsible for a package of tasks that let the operator examine and change system setup parameters and interlock limits.

It is clear from the AECL documentation on the modifications that the software allows concurrent access to shared memory, that there is no real synchronization aside from data stored in shared variables, and that the "test" and "set" for such variables are not indivisible operations. Race conditions resulting from this implementation of multitasking played an important part in the accidents.

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

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?

Example

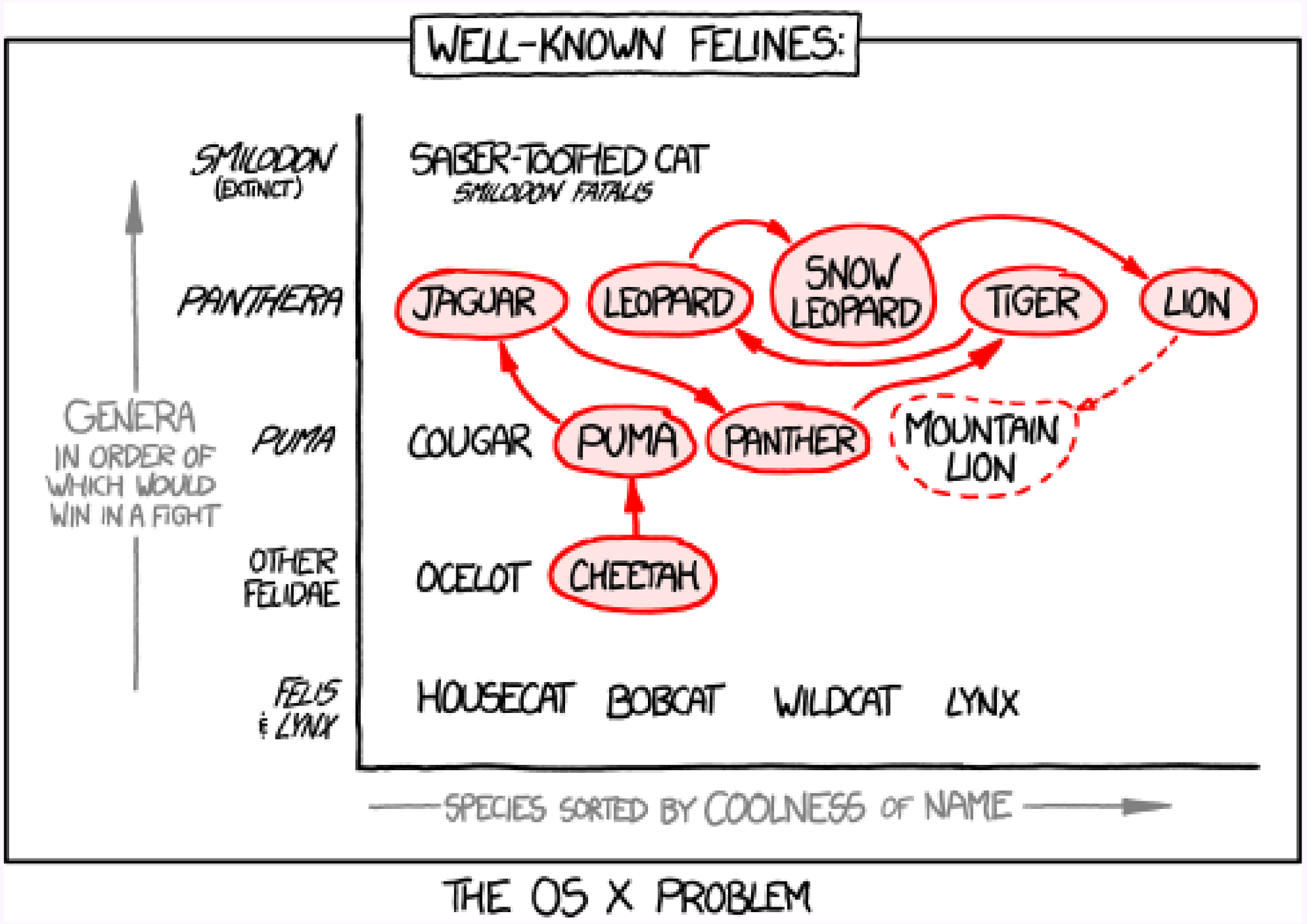
```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }}
```

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 communicate with multiple peers/clients (Assignment 1 and Assignment 2)
  - need to coordinate: similar to IPC, shared memory, locking, barriers, etc. (Assignment 2)
- When testing, we will use multi-threaded programs to start multiple clients (Assignment 1)



Systems talk (xkcd 1056)

Google

Google Search I'm Feeling Lucky

What does Google do?!

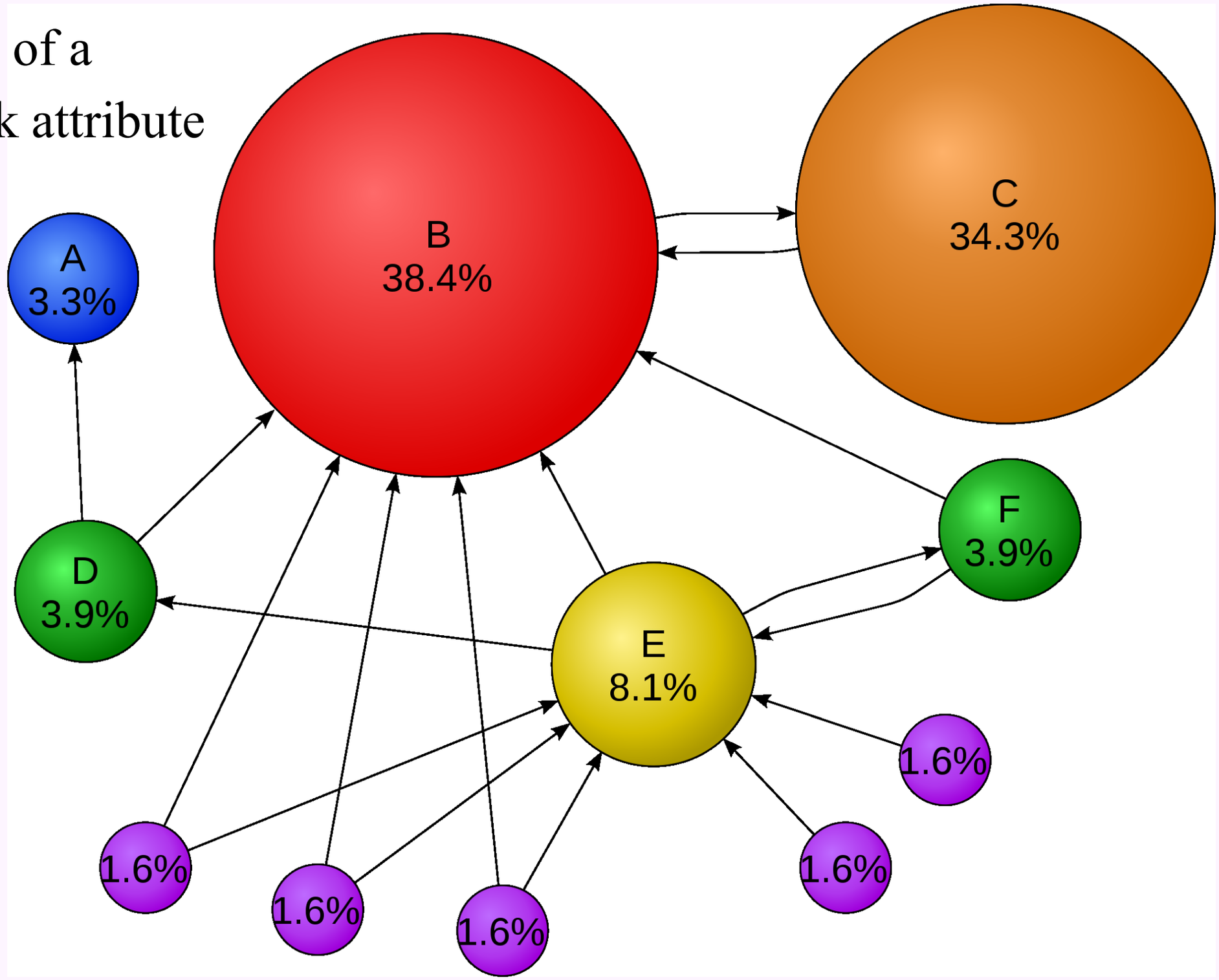
- Indexing the web
- Image recognition & rendering
  - Face blurring, business names,
- Maps, route finding
- Mail, managing blogs, …
- Manages code bases
- **Advertisements**
- $$$$$...$$

Indexing the Web

- Crawlers
- Variations of an algorithm called PageRank
  - Rank each document in a hyperlinked set of documents
  - Measure its relative importance within the set
  - Spice it up with semantic understanding of relationships between words

Indexing the Web

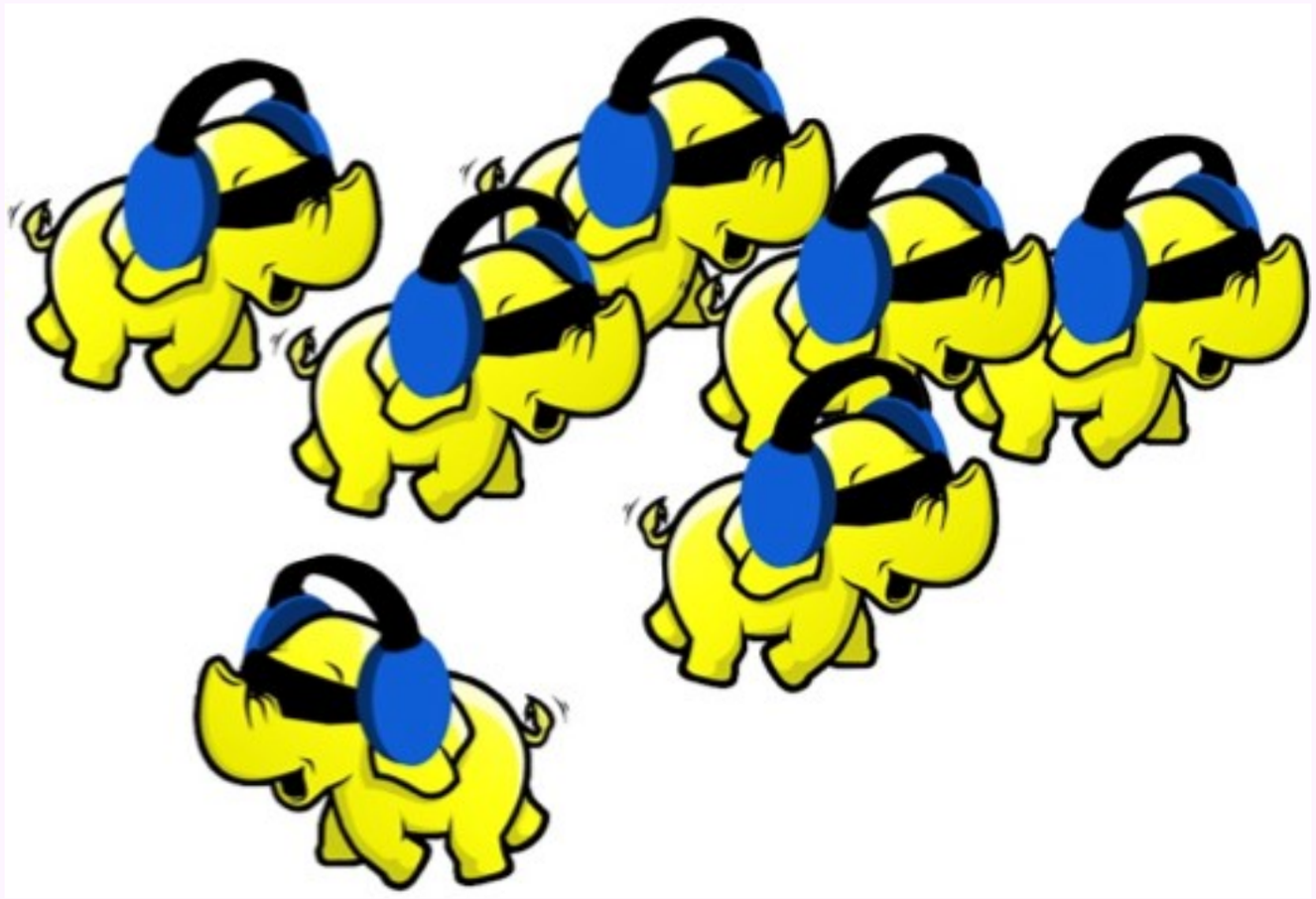
Example of a
page rank attribute



What is MapReduce?

- A programming model for data processing
- Supports distributed computing on large data sets on multiple machines (clusters, public or private clouds)
- Ability to scale to 100s or 1000s of computers, each with several processor cores
- How large an amount of work?
  - Web-scale data on the order of 100s of GBs to TBs or PBs
  - Input data set will not likely fit on a single computer's hard drive
  - eg: Google sorted 1 PB on 4,000 computers in 6 hrs 02 minutes in 2008

MapReduce



MANY COMPUTERS, LITTLE COMMUNICATION, FAILURES

MapReduce Typical Application

- Read (A LOT OF) data

1. MAP
    - (extract data you need from each record)
2. Sort data
3. REDUCE
    - (aggregate, summarize, filter, transform extracted data)
4. Write the results

MapReduce Example

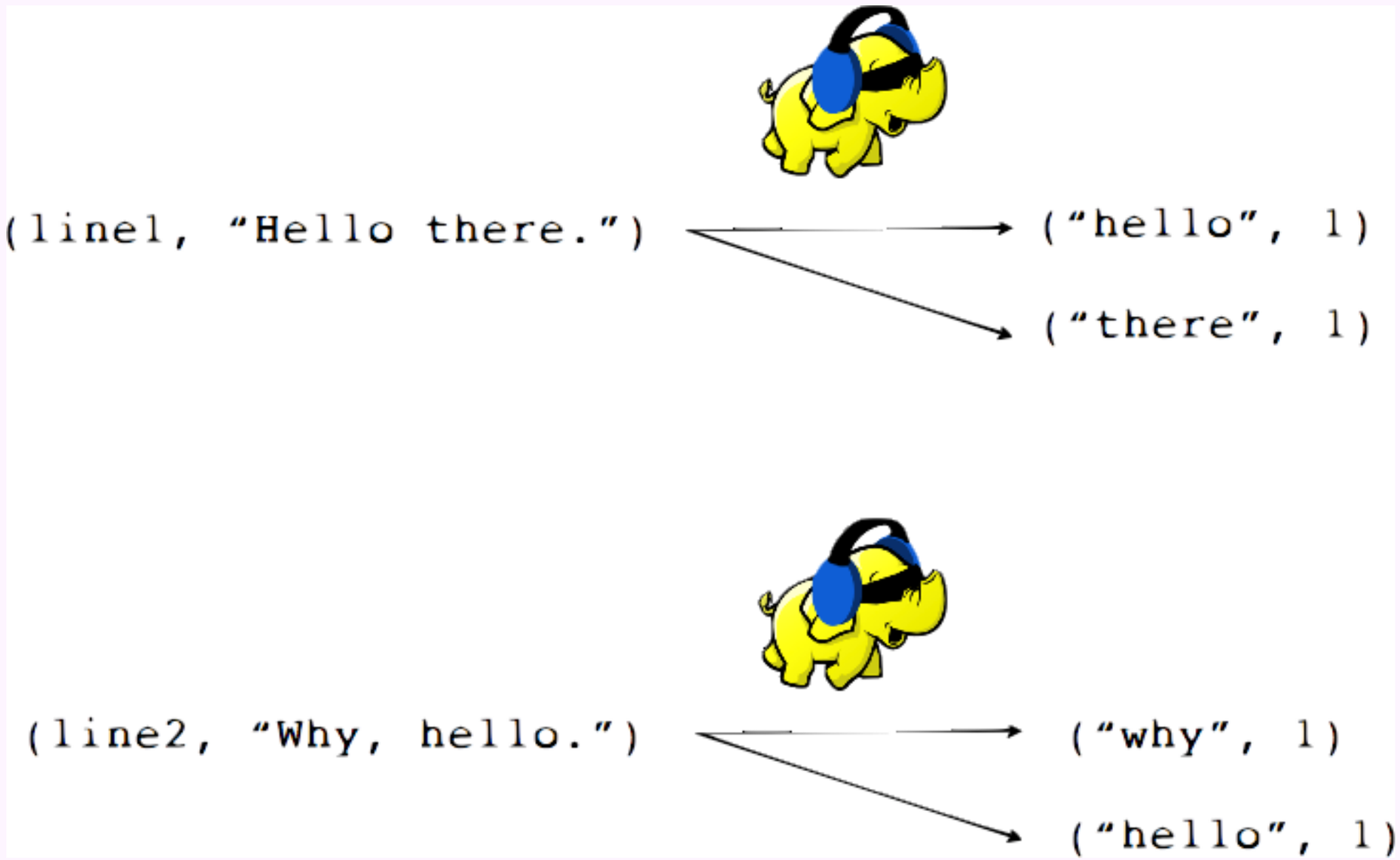
Count how many times each word appears in a set of documents

```
void map (String doc_name, String doc_content) {

    for each word in doc_content

        send_intermediate(word, "1");

}

void reduce (String word, List partialCounts) {

    int sum = 0;

    for each pc in partialCounts

        sum = sum + int(pc);

    send(word, String(sum));}
```

1. Map Phase

- Specify operations on key-value pairs
- Each elephant works on an input pair, does not know others exist

1. Map Phase


(linel, "Hello there.")
("hello", 1)
("there", 1)
(line2, "Why, hello.")
("why", 1)
("hello", 1)

2. Sort Phase

(key1, value289)

(key1, value43)

(key1, value3)

...

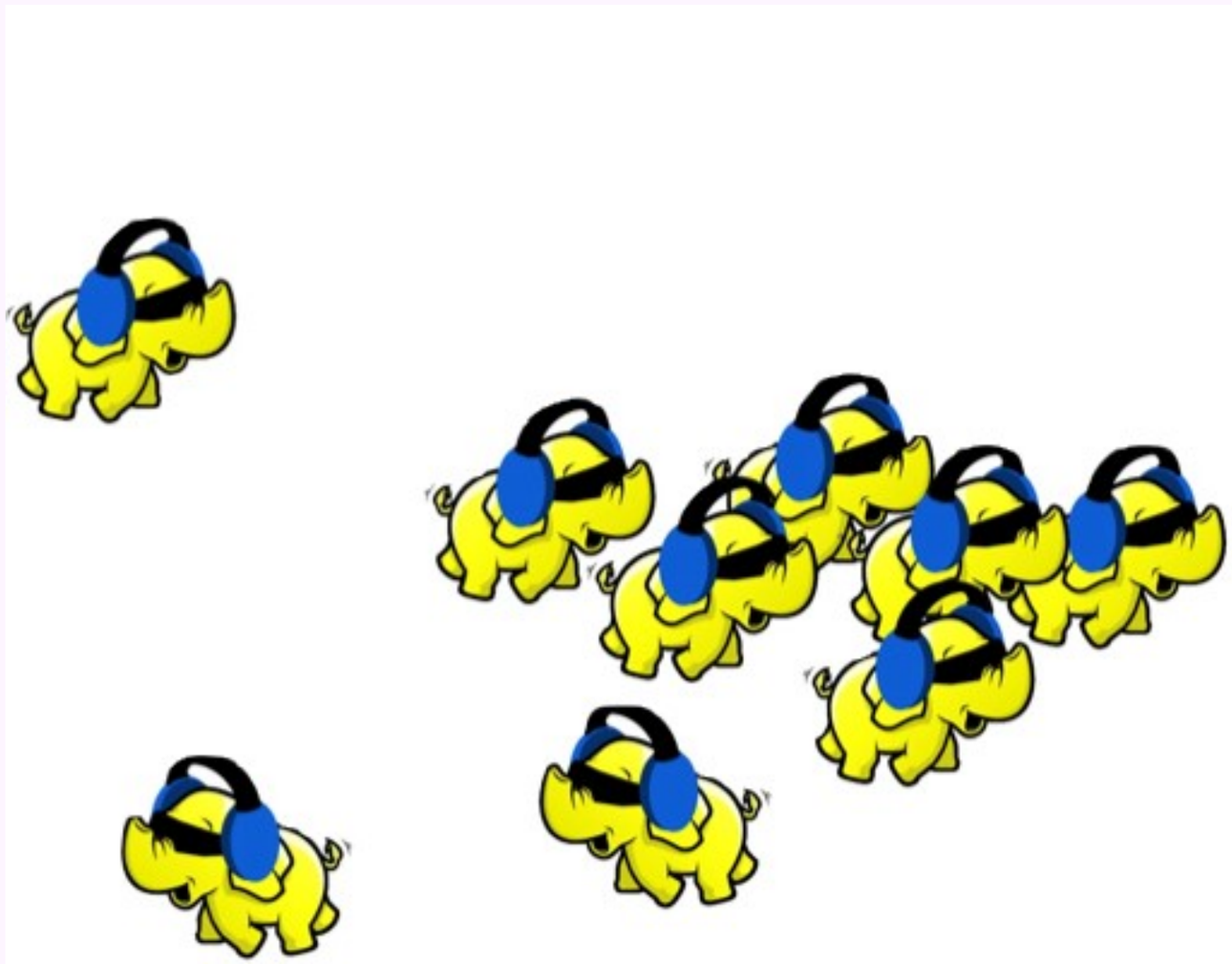
(key2, value512)

(key2, value11)

(key2, value67)

...

2. Sort Phase



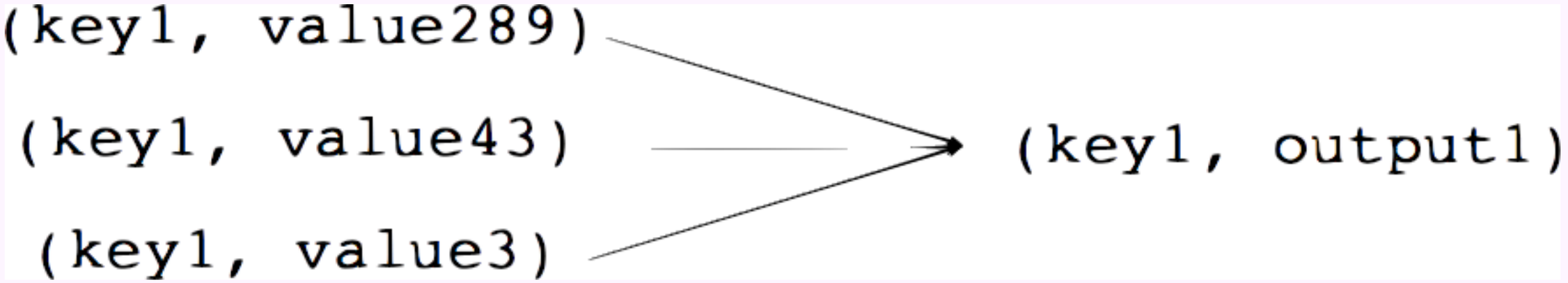
("hello", 1)

("hello", 1)

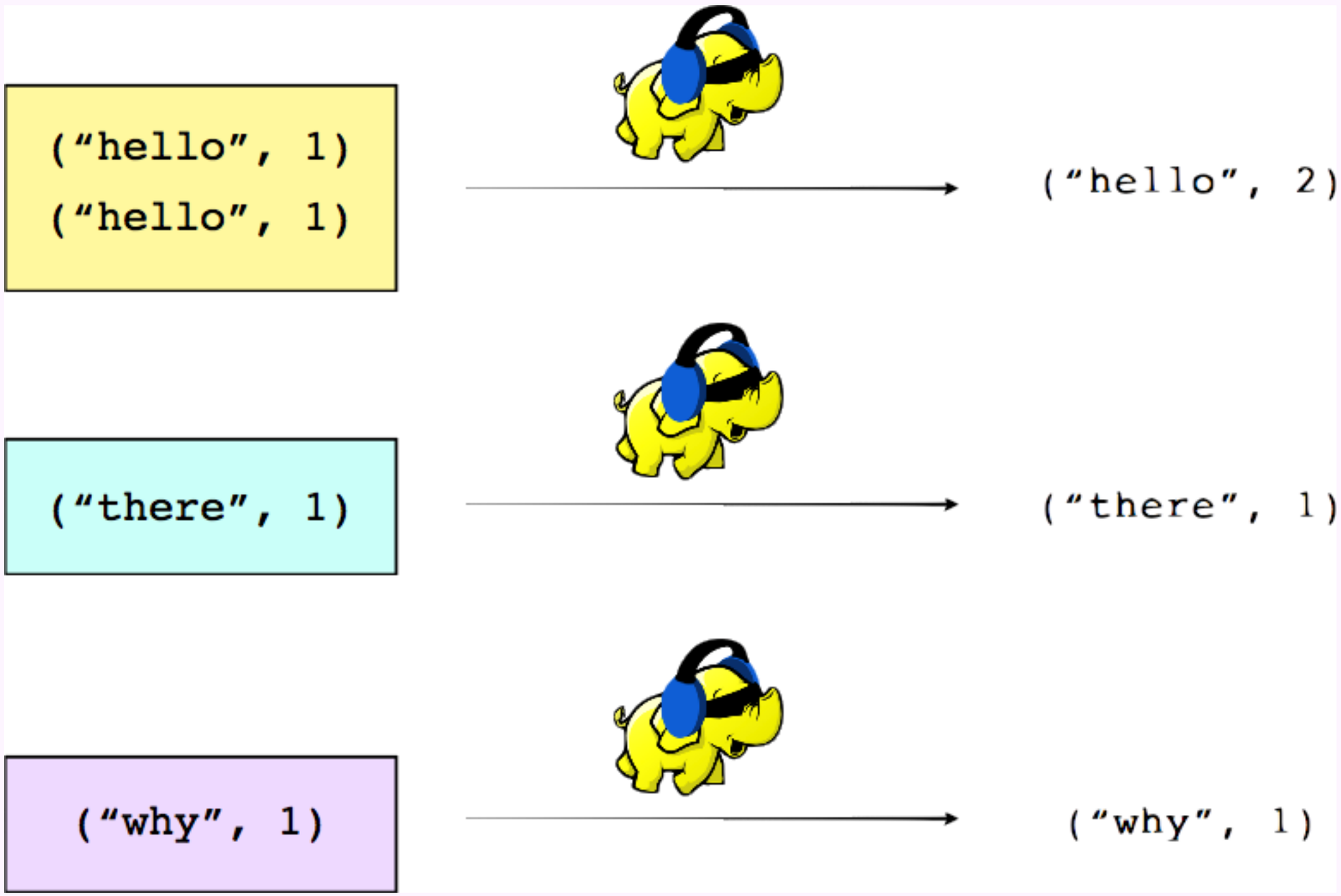
("there", 1)

("why", 1)

3. Reduce Phase



3. Reduce Phase



Parallelism

- **map** function runs in parallel
  - The programmer specifies the chunks of data for each map worker to allow parallelism
  - Each map worker creates intermediate local values from each input data set
- **reduce** function runs in parallel
  - The number of reduce workers determines the MapReduce performance

Parallelism

- All data sets (chunks of data) are processed independently
  - reduce cannot start until map is completely finished
- Master program creates tasks based on the location of the data, and tries to send map tasks to close machines to minimize network communication