ELEC 377 – Operating System

Week 2 – Class 2

Last Class

- Schedulers
- Process Creation
- Process Termination
- Interprocess Communication
- Message Passing

Today

- Finish Messaging
- Indirect vs Direct Messaging
- Threads
- Synchronization

Direct Communication

- Processes explicitly identify each other
- ◊ send(P,message)
- ◊ receive(Q, &message)

- addressing may be asymmetric
- ◊ send(P,message)
- ◊ receive(&id,&message)

- Advantages?

- Disadvantages?

Direct Communication

- Advantages?
  - Explicit
  - Simple
- Disadvantages?
- Limited modularity of the resulting process definitions
- Changing an ID of a process -> may need to examine all other process definitions.

Indirect Communication

- Mailboxes
  - ◊ each mailbox has a unique id
  - ◊ processes share the mailbox
- What if more than one process wants to receive a message from a mailbox
  - ◊ Only allow one process to read mailbox
  - ◊ First come – first serve
  - ◊ Multiple receivers
- Advantages?
- Disadvantages?

Indirect Communication

- Advantages?
  - More flexible
  - No question who received the message from a mailbox
- Disadvantages?
  - Operating system must provide mailbox mechanism
  - Mailbox ownership may be passed -> could result in multiple receivers.

Synchronization

- Coordination between sending and receiving process
  - ◊ blocking vs non-blocking
  - ◊ applies to both sender and receiver
  - ◊ blocking is synchronous
  - ◊ non-blocking is asynchronous

- if both send and receive are blocking -> rendezvous
  - ◊ Ada

Buffering

- Zero Capacity -> rendezvous
  - Queue of length 0
  - Sender blocks until message is received

- Bounded -> sender may need to wait or abandon the message
  - If queue is full, sender blocks.

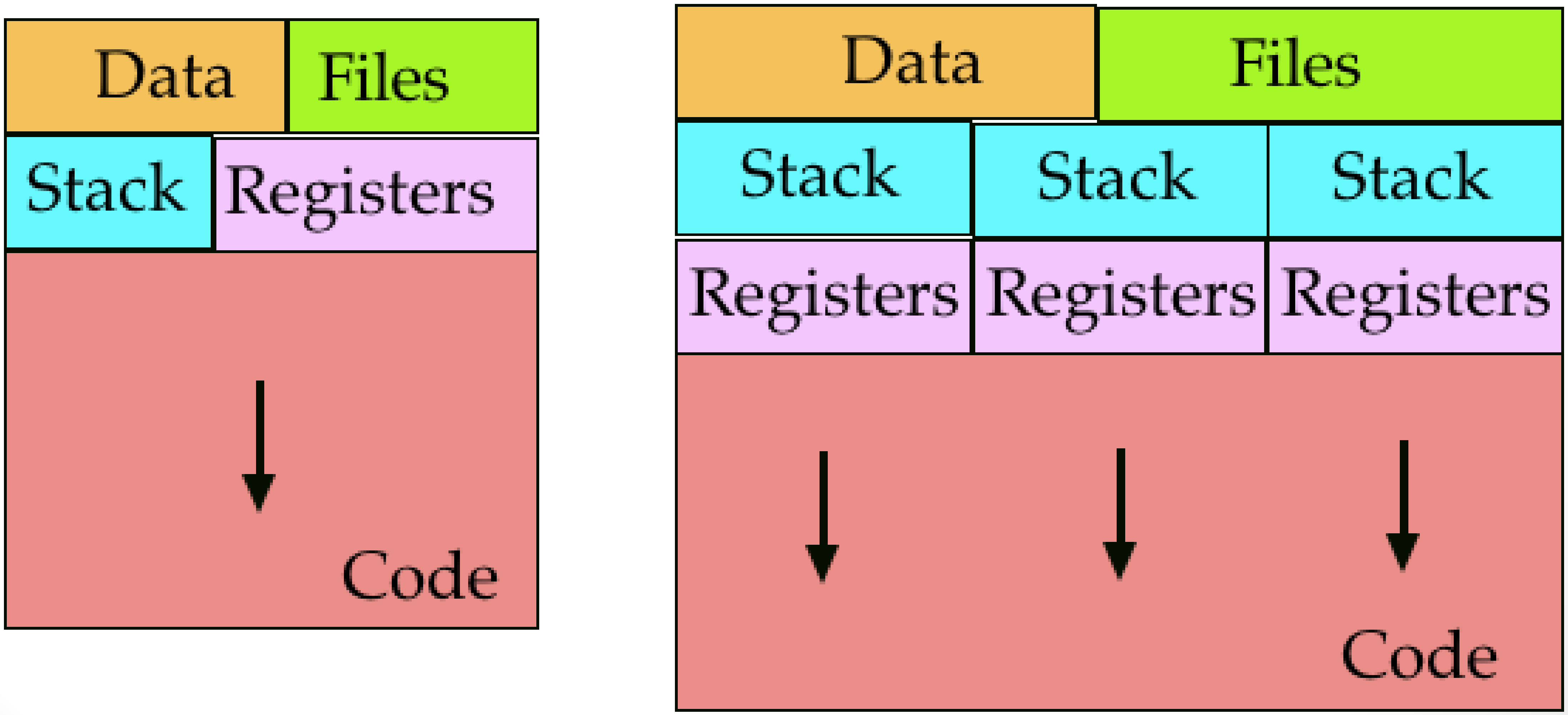
- Unbounded -> sender never needs to wait.

◊ resource intensive

◊ non-guaranteed delivery (IP/UDP)

Threads

- Lightweight Processes
  - Thread id, stack, registers, program counter
  - Memory Management Costs in context switches
- Traditional Process is a single thread



Why use Threads?

- Responsiveness
- Resource Sharing
- Economy
- Utilization of Multiprocessor/Multicore Architectures

User Threads

- Earliest Threads
- Threads implemented by a library (asm routines)
  - Operating System is unaware of the threads
- Advantages
  - Fast, no system call, simple scheduling
- Disadvantages
  - One thread blocks (I/O, IPC), all threads block
  - No Multiprocessor support
- Examples:
  - Early pthreads, Turing

Kernel Threads

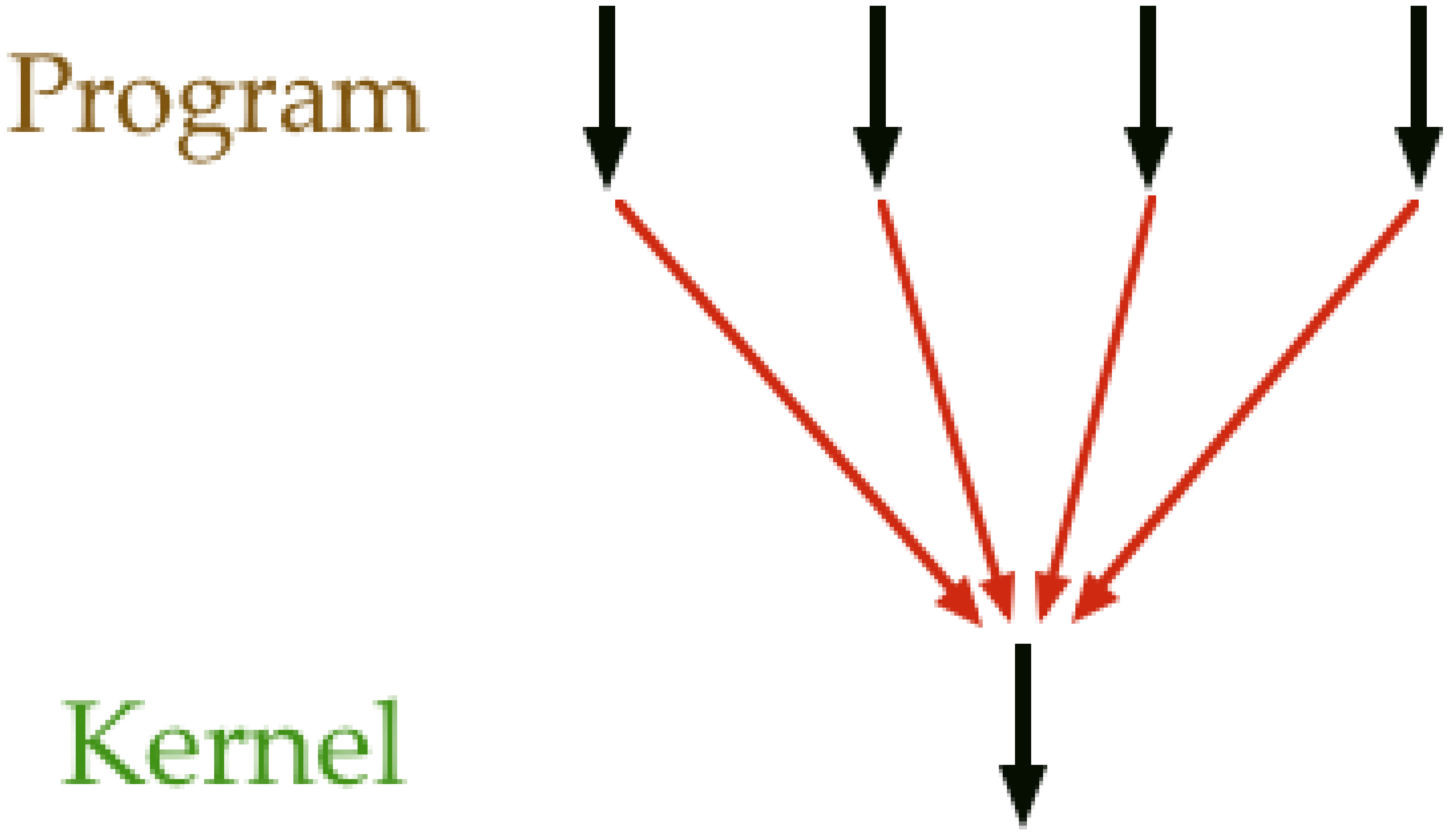
- Threads provided by the operating system
  - scheduled by the operating system
  - only difference is in context switch, and killing a given process kills all threads
- Advantages
  - OS only blocks thread doing a system call
  - MP support
- Disadvantage
  - not quite as fast as user level threads
  - Resource Intensive(each thread has a kernel entry)
- Provided by most modern operating systems

Thread Models

- Programmer sees threads, calls library to create thread.
  - Library may create a kernel thread (if supported)
  - Library may manage as a user level thread
- Mixture of user and kernel level threads possible

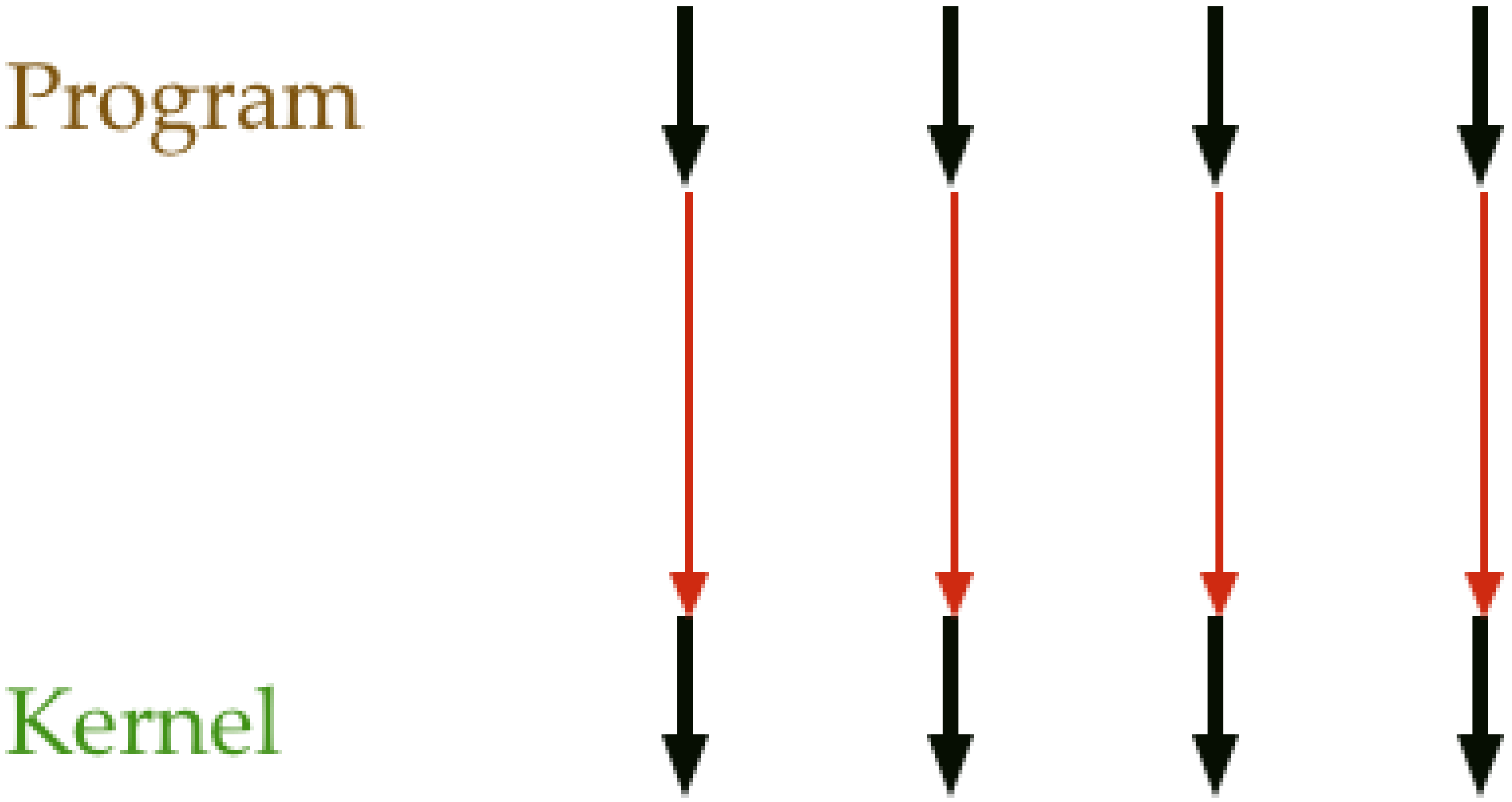
Many to One

- Only model on OS that do not support threads
- All threads mapped to a single kernel thread (process)



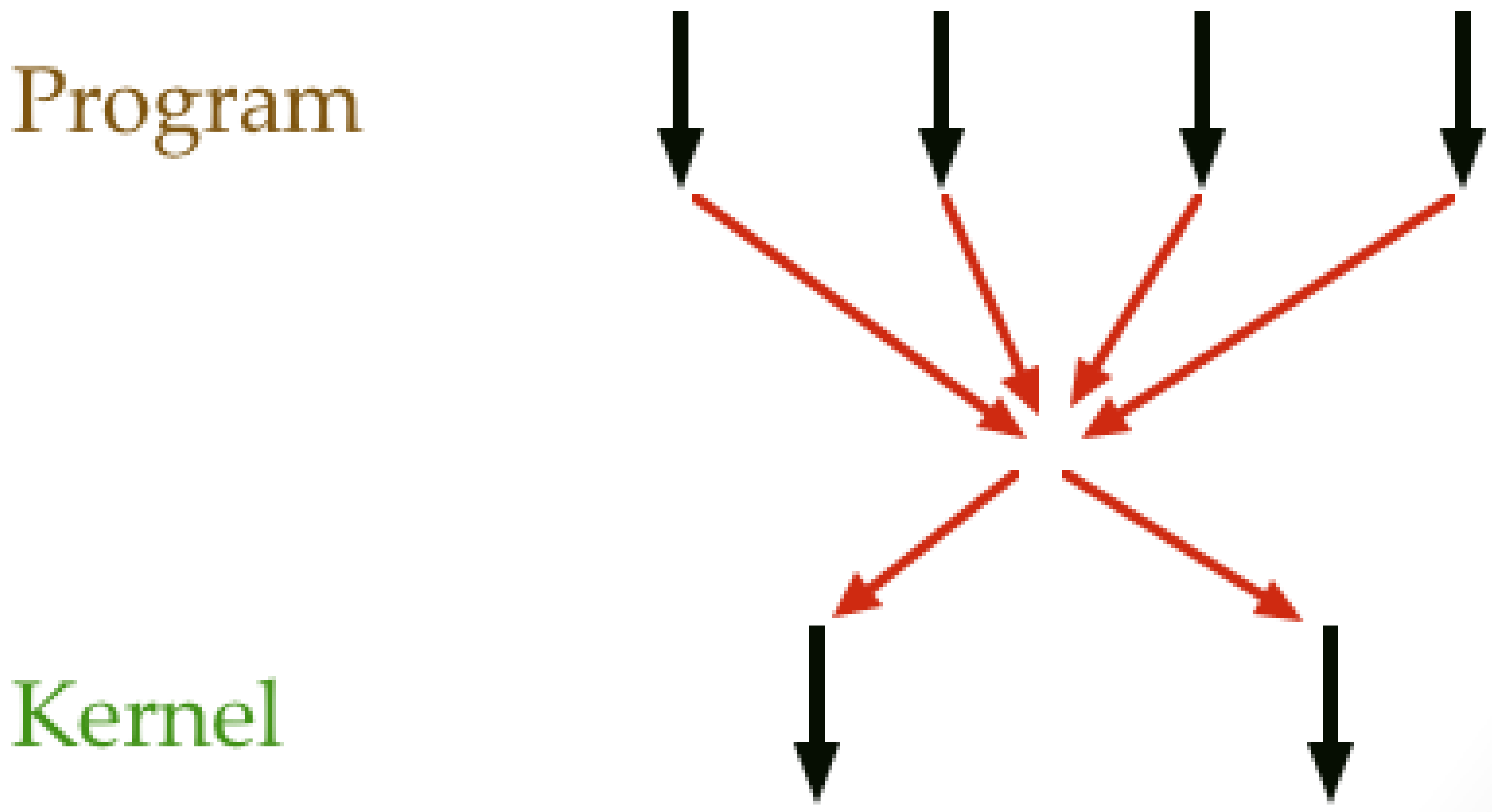
One to One

- OS must support threads
- Each program level thread gets a kernel thread



Many to Many

- Limit number of kernel threads
- More program threads than kernel threads
  - Thread library maps program threads to kernel



Threading Issues

- Processes Level System Calls (fork, exec)
- Thread Termination
  - Can a thread be terminated by another
  - Or does it have to terminate itself
- Signals (user level interrupts)
  - Which thread gets the signal
  - Resource Intensive(each thread has a kernel entry)
- Provided by most modern operating systems
- Thread Pools
- Thread Specific Data

Recent Advances

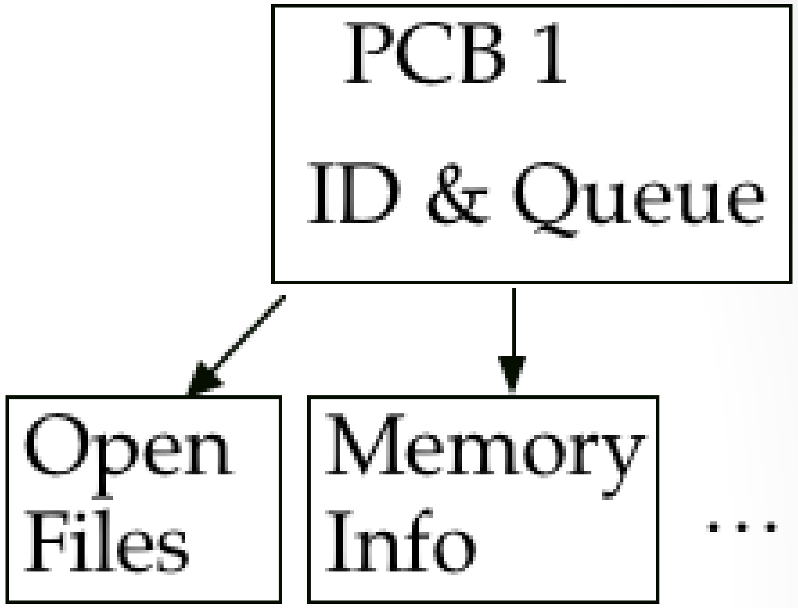
- Even threads as are too heavy weight for fine grained use multi core architectures.
  - Good for function/object level concurrency
  - But what about statement/block level concurrency?
  - Cost of thread construction/destruction
- Thread Pool Pattern built into language
  - Mechanism to farm out blocks of code to worker threads
  - Grand Central dispatch (apple)
  - Java Implementation (Java 7)

Thread Implementations

- Linux clone()
- Java threads

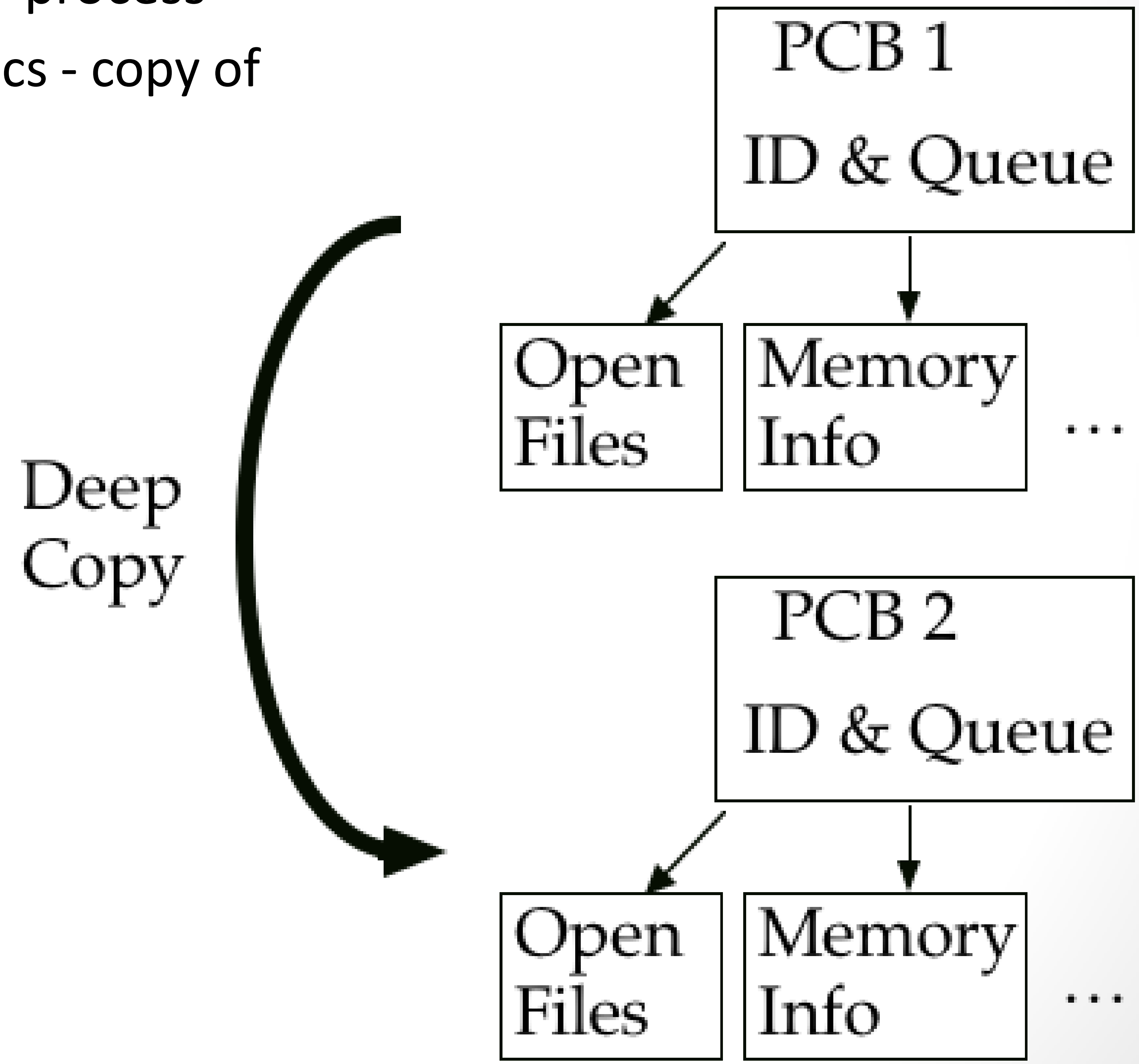
Linux clone() & fork()

- Linux is thread aware
- Every process has a PCB
  - linux PCB is not a single level structure
  - pointers to other structures



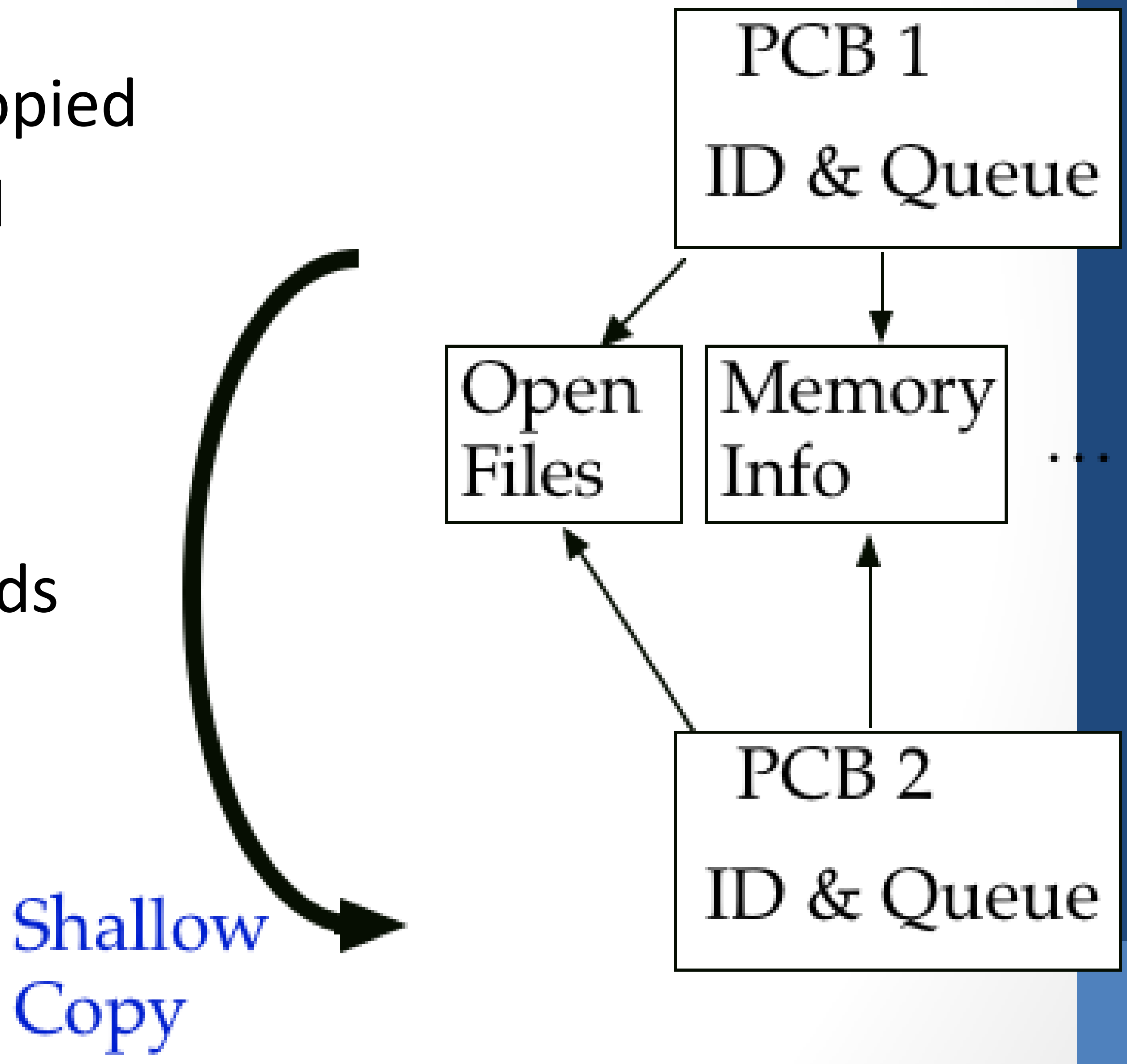
Linux clone() & fork()

- fork() starts another process
  - Unix fork semantics - copy of parent process



Linux clone() & fork()

- clone() starts another thread
  - New PCB only
- parameters determine what is copied
  - control over amount of shared information
  - share memory and not files
  - share files and not memory
- Mixture of kernel and user threads supported



Java Threads

- Java threads
  - Like Ada, Turing: threads are part of the language
  - Some threads already defined (garbage collector)
- Thread class
  - start() method
- Two ways to define code that thread will run
  - Extend Thread class and override run()
  - Implement Runnable interface and provide run()

Extend Thread

```
class Foo extends Thread {
    public void run() {
        for (i = 0; i < 10; i++){
            System.out.println(“bar”);
        }
    }
}
•••
  Foo bar = new Foo();
  Foo bat = new Foo();
•••
  bar.start();
  bat.start();
```

Implement Runnable

```
class Foo implements Runnable {
    public void run() {
        for (i = 0; i < 10; i++){
            System.out.println(“bar”);
        }
    }
}
•••
  Foo bar = new Foo();
  Foo bat = new Foo();
   Thread a = new Thread(bar);
   Thread b = new Thread(bat);
•••
  a.start();
  b.start();
```

JVM Virtual Machine

- Java is compiled to byte codes (0…255)
- Virtual machine is a hardware emulator
- Just-in-time compilers
- Threads implemented inside of the Virtual machine
  - Green Threads (user level threads)
  - Native Threads
  - Almost all of the thread implementations are available in different java implementations
  - Since I/O is provided by Java VM, don’t have to worry about one thread blocking the entire processes.

Process Synchronization

- Most Important Part of the Course and Text
- Concurrent access to shared resources
  - data inconsistency
  - need some mechanism to control access to shared resources

Synchronization Example

Account Deposit

…

account = account + deposit

…

Account Withdrawl

…

account = account - withdrawl

…

Synchronization Example

Account Deposit

```
…
mov account, reg1
// mov = get the memory value at the address and put it into
//register
add deposit, reg1
move reg1, account
…
```

Account Withdrawl

```
…
mov account, reg1
sub withdrawl, reg1
move reg1, account
…
```

Synchronization Example

Account Deposit

+$ 100

…

mov account, reg1

add deposit, reg1

move reg1, account

…

$ 5,243

1 $ 5,343 INT

3 $ 5,343

Account Withdrawl

-$ 100

…

mov account, reg1

sub withdrawl, reg1

move reg1, account

…

$ 5,243

2 $ 5,143 INT

4 $ 5,143

Process Synchronization

- Race Condition
  - Several process handle shared resources
  - Final value depends on who finishes first

- To prevent race conditions, concurrent processes must be synchronized
  - train signaling problem (James Burke)

Critical Sections

Account Deposit

…
mov account, reg1
add deposit, reg1 critical section
move reg1, account
…

Account Withdrawl

…
mov account, reg1
sub withdrawl, reg1 critical section
move reg1, account
…