Operating Systems: Threads

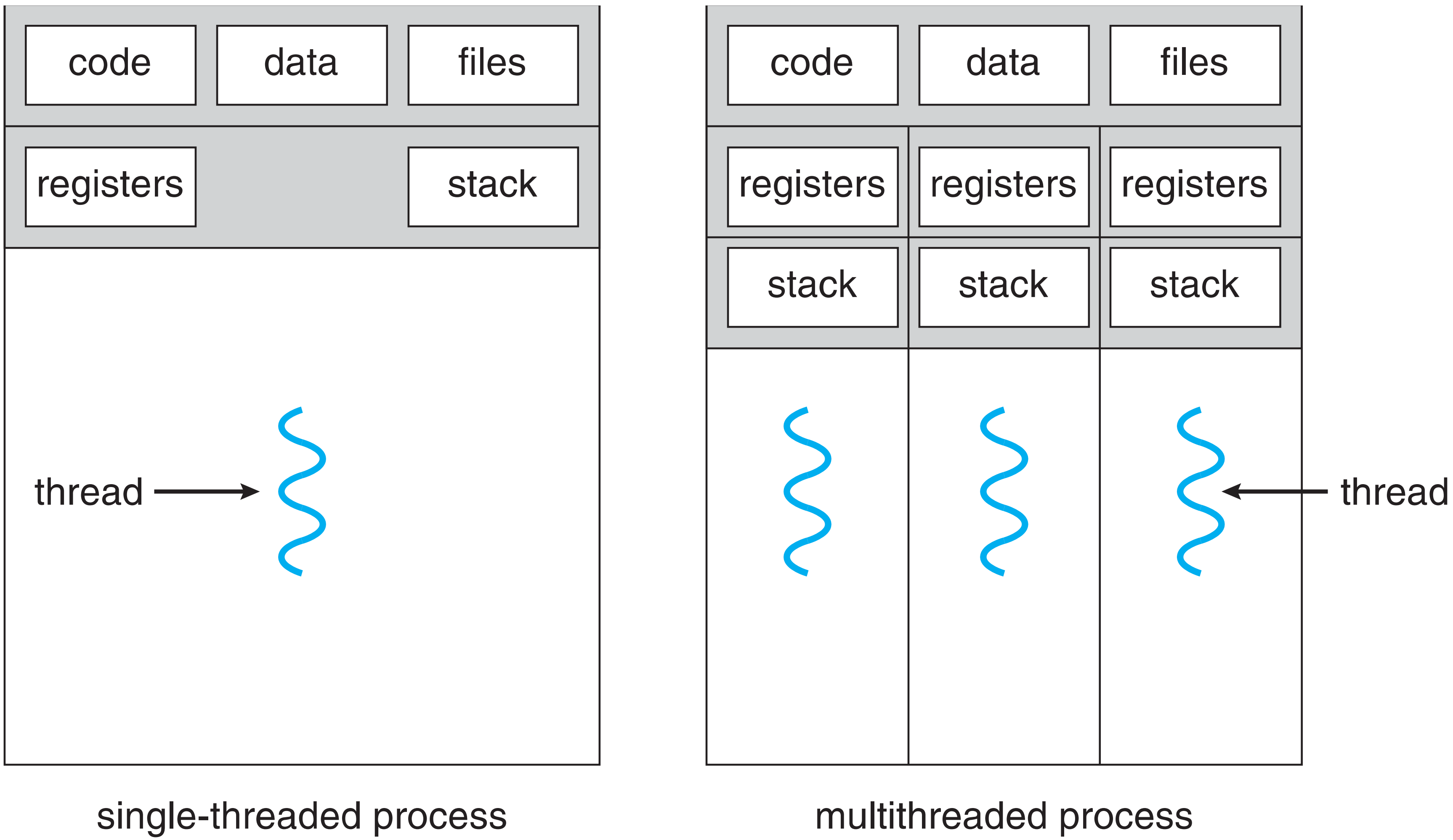
CSC-4320/6320 –Summer 2014

Dustin Kempton (dkempton1@cs.gsu.edu)

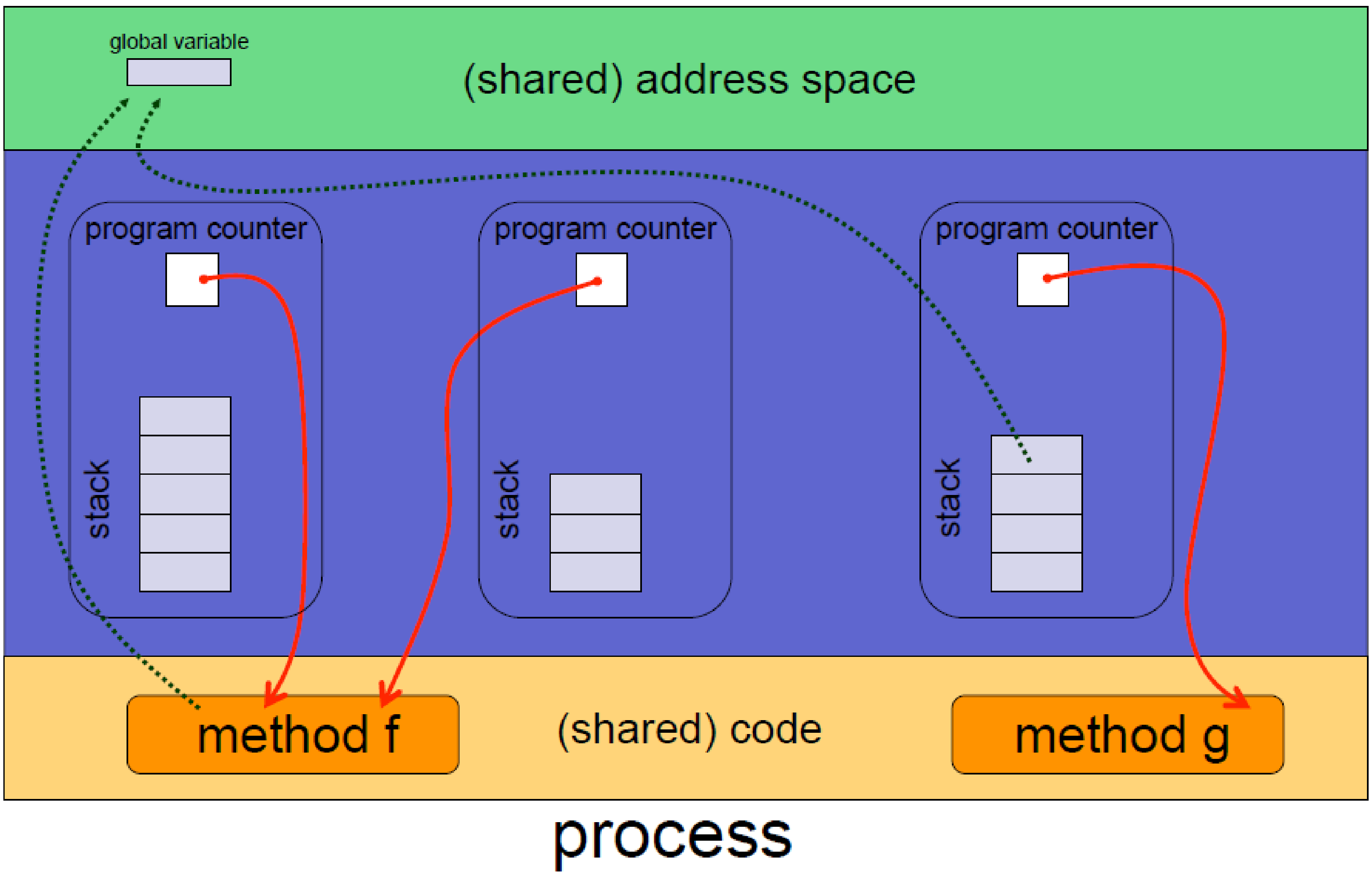
Definition

- A thread is a basic unit of CPU utilization within a process
- Each thread has its own
  - Thread ID
  - Program Counter
  - Register Set
  - Stack
- It shares the following with other threads within the same process
  - Code section
  - Data section
  - The heap (dynamically allocated memory)
  - Open files and signals
- **Concurrency:** A multi-threaded process can do multiple things at once

The Typical Figure



A More Detailed Figure



Advantages of Threads?

- Economy:
  - Creating a thread is cheap
    - Much cheaper than creating a process
  - Context-switching between threads is cheap
    - Much cheaper than between processes
- Resource Sharing:
  - Threads naturally share memory
    - With processes you have to use possibly complicated IPC (e.g., Shared Memory Segments)
  - Having concurrent activities in the same address space is very powerful
    - But fraught with danger

Advantages of Threads?

- Responsiveness
  - A program that has concurrent activities is more responsive
    - While one thread blocks to answer a client request in a client-server implementation
  - This is true of processes as well, but with threads we have better sharing and economy
- Scalability
  - Running multiple “threads” at once uses the machine more effectively
    - e.g., on a multi-core machine
  - This is true of processes as well, but with threads we have better sharing and economy

Drawbacks of Threads

- One drawback of thread-based concurrency compared to process-based concurrency: If one thread fails (e.g., a segfault), then the process fails
  - And therefore the whole program
- This leads to process-based concurrency
  - e.g., The Google Chrome Web browser
  - See http://www.google.com/googlebooks/chrome/
  - Sort of a throwback to the pre-thread era
    - Threads have been available for 20+ years
    - Very trendy recently due to multi-core architectures

Drawbacks of Threads

- Threads may be more memory-constrained than processes
  – Due to OS limitation of the address space size of a single process
- Threads do not benefit from memory protection
  – Concurrent programming with Threads is hard
    - But so is it with Processes and Shared Memory Segments
  – We will see this in later chapters

Multi-Threading Challenges

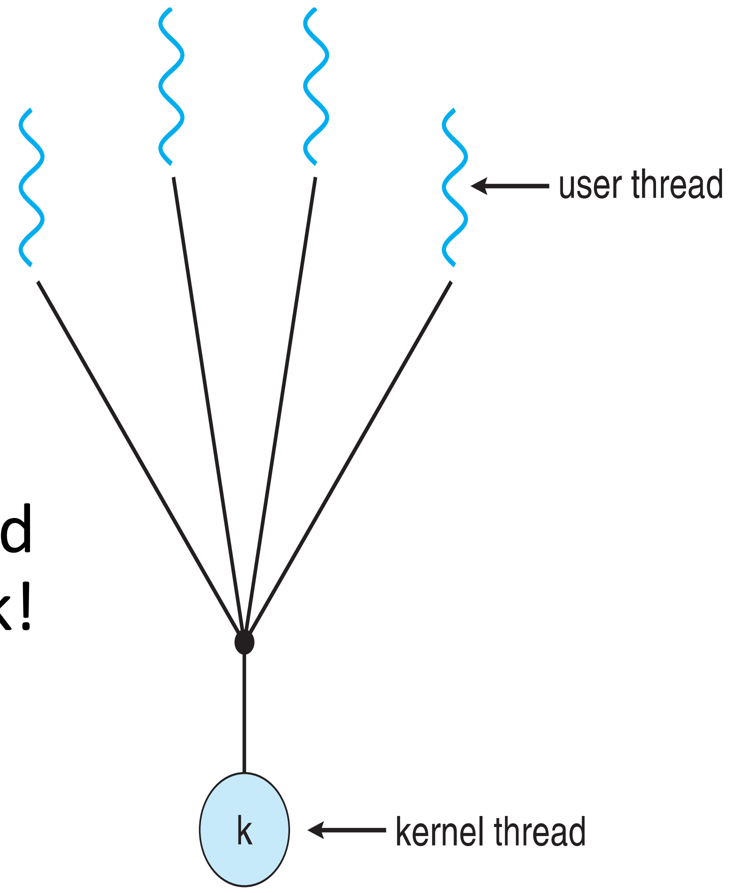
- Typical challenges of multi-threaded programming
  - Dividing activities among threads
  - Balancing load among threads
  - Split data among threads
  - Deal with data dependency and synchronization
  - Testing and Debugging
- Section 4.2 talks a little about this
  - All of you will most likely write multi-threaded code on multi-core architectures

User Threads vs. Kernel Threads

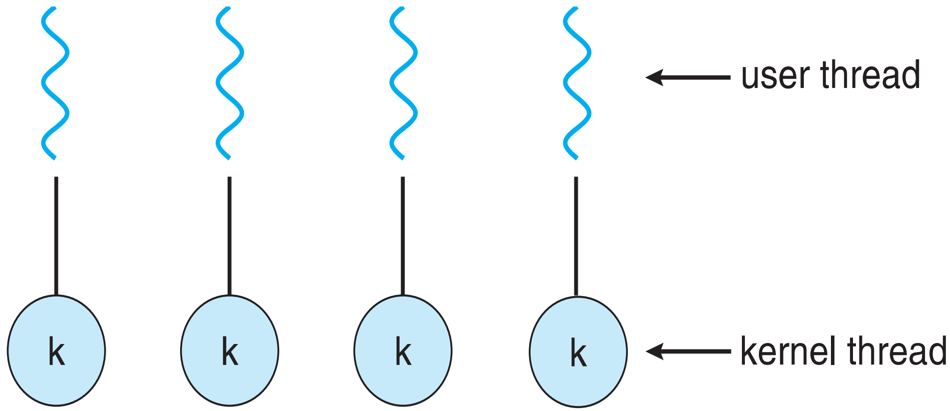
- Threads can be supported solely in User Space
  - Threads are managed by some user-level thread library without OS support
  - Less system calls –more efficient
- Threads can also be supported in Kernel Space
  - The kernel has data structures and functionality to deal with threads
  - Most modern OSes support kernel threads
    - In fact, Linux doesn't really make a difference between processes and threads (same data structure)
- Tradeoffs: Kernel thread handling incurs more overhead. User threads stops the application on every blocking call

Many-to-One Model

- Advantage: multi-treading is efficient and low-overhead
  - No syscalls to the kernel
- Major Drawback #1: cannot take advantage of a multi-core architecture!
- Major Drawback #2: if one thread blocks, then all the threads block!
- Examples (User-level Threads):
  - Java Green Threads
  - GNU Portable Threads



One-to-One Model

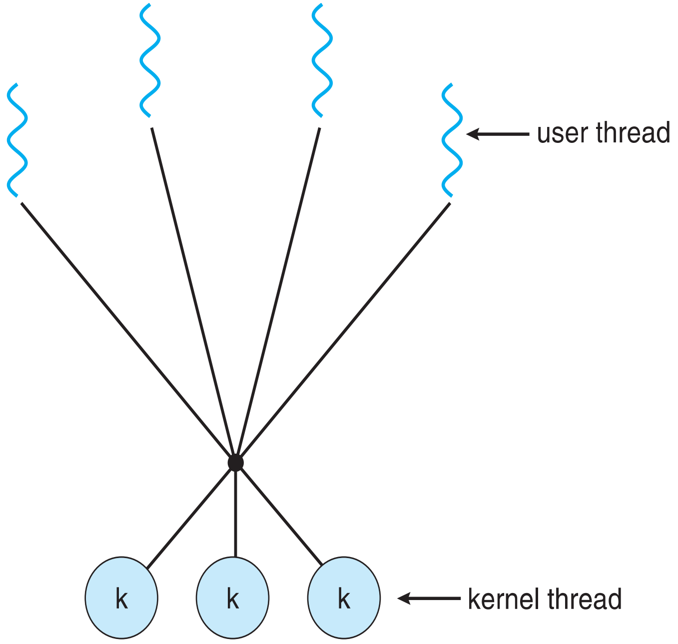


- Removes both drawbacks of the Many-to-One Model
  - Blocking OS calls don’t suspend applications
- Creating a new thread requires work by the kernel
  - Not as fast as in the Many-to-One Model
  - Upper limit on the total number of threads (more so than before)

- Example:
  - Linux
  - Windows
  - Solaris 9 and later

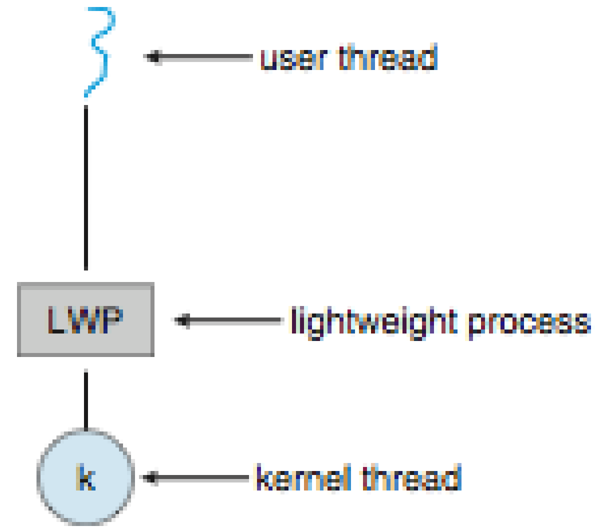
Many-to-Many

- Kernel thread pool assigned to an application and managed by a thread library
- Advantages
  - Eliminates user thread number limit
  - Applications don’t suspend on blocking OS calls
  - Increased thread efficiency while maintaining concurrency
- Examples:
  - Solaris 9 and earlier
  - Win NT/2000 with the ThreadFiber package



Two level threads: A many-to-many model.
The kernel maps threads onto processors,
And the run-time library maps user
Threads onto kernel threads

Many-to-Many Thread Scheduling



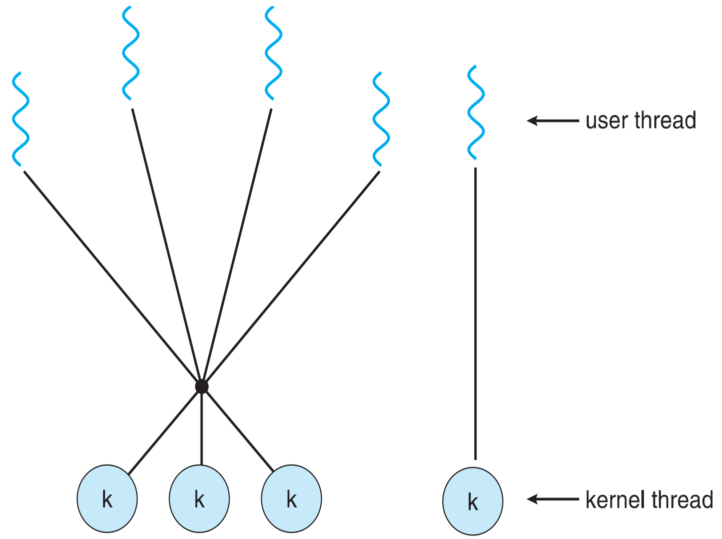
Issue: How many kernel threads?

- Too many means extra OS overhead
- Too few means processes block

1. The kernel assigns a group of kernel threads to a process
2. Up-calls: kernel→thread library
   a) If the process is about to block
   b) If a blocked kernel thread becomes ready
   c) Thread allocations to be released (freed)

Note: A kernel process sitting between user and kernel threads assists with thread management. These are often called light weight processes (LWP)

Two-Level Model



- The user can say: “Bind this thread to its own kernel thread”

- Example:
  - IRIX, HP-UX, Tru64 UNIX
  - Solaris 8 and earlier

Threading Issues

- Does spawning a new process spawn all threads or only the one that is executing? Example: fork() duplicates all threads, exec() replaces the process with a single thread.
- How do we cancel a thread? What if it is working with system resources? Approaches: Asynchronous or Synchronous cancellation
- How do threads communicate? Linux: Signals to notify a thread of an event, Windows: Asynchronous procedure calls that function like callbacks. Linux: clone() options determine which resources are shared between threads.
- How do we created data that is local to a specific thread? Answer: Thread specific data
- How are threads scheduled for execution? One possibility: Light weight processes

Thread Libraries

- Thread libraries provide users with ways to create threads in their own programs
  - In C/C++: Pthreads
    - Implemented by the kernel
  - In C/C++: OpenMP
    - A layer above Pthreads for convenient multithreading in “easy” cases
  - In Java: Java Threads
    - Implemented by the JVM, which relies on threads implemented by the kernel

Pthreads Example

```
static pthread_mutex_t mtx;

void main(int argc, char argv[]){
    thread_t[] handles;
    int t;
    int threads = strtoi(argv[1], NULL, 10);
    mtx = pthread_mutex_init(&mtx, NULL);
    handles = malloc(threads*sizeof(pthread_t));

    for(t=0;t<threads;t++)
        pthread_create(&handles[t], NULL, addThem, (void*)&t);

    for(t=0;t<threads; t++)
        pthreads_join(handles[t], NULL);

    printf("Total= %d\n", sum);
    free(handles);
    pthread_mutex_destroy(&mtx);

}
```

```
void* addThem(void *rank){
    int myRank = (int)(*rank);
    double mySum = 0;
    int i;
    int myN = 10000/myRank;
    int first = myN*myRank;
    int last = first+myN;

    for(i=first; i<last; i++)
        sum+=i;

    pthread_mutex_lock(&mtx);
    sum+=mySum;
    pthread_mutex_lock(&mtx);
}
```

OpenMP Example

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

    double sum = 0.0;
    int last = 10000;
    int threads = strtoi(argv[1], NULL, 10);

    #pragma omp parallel num_threads(threads) reduction(+:sum)
    for(i=0; i<last; i++)
        sum+=i;

    printf("Total = %d\n", sum);
}
```

Note: OpenMP is a popular industry-wide thread standard. To compile with OpenMP enabled in GCC just add the –fopenmp directive in the g++ or gcc compile command

Java Threads

- Java threads are managed by the JVM and generally utilize an OS provided thread-based library
- Java threads may be created by:
  - Implementing the runnable interface
  - Extending the Thread class
- Java threads terminate when they leave the run method

```
public interface Runnable
{
    public abstract void run();
}
```

Extending the Thread Class

- To create a thread, you can extend the thread class and override its “run()” method

```
class MyThread extends Thread {
    public void run() {
       . . .
    }
    . . .
}

MyThread t = new MyThread();
```

Example

```
public class MyThread extends Thread {
    public void run() {
        for (int i=0; i<10; i++) {
            System.out.println("Hello world #"+i);
        }
    }
    . . .
}

myThread t = new MyThread();
```

Spawning a thread

- To launch, or spawn, a thread, you just call the thread's start() method
- Warning: Do not call the run() method directly to launch a thread
  - If you call the run() method directly, then you just call some method of some object, and the method executes
    - Not a big deal, but probably not what you wanted
  - The start() method, which you should not override, doe all the thread launching
    - It launches a thread that starts its execution by calling the run() method

Example

```
public class MyThread extends Thread {
    public void run() {
      for (int i=0; i<5; i++) {
          System.out.println("Hello world #"+i);
      }
    }
}

public class MyProgram {
  public MyProgram() {
     MyThread t = new MyThread();
      t.start();
  }
  public static void main(String args[]) {
     MyProgram p = new MyProgram();
  }
}
```

What happens

- The previous program runs as a Java process
  - That is, a thread running inside the JVM
- When the start() method is called, the main thread creates a new thread
- We now have two threads
  - The “main”, “original” thread
  - The newly created thread
- Both threads are running
  - The main thread doesn’t do anything
  - The new thread prints messages to the screen and exits
- When both threads terminate, the process terminates
- Let’s have the first thread do something as well…

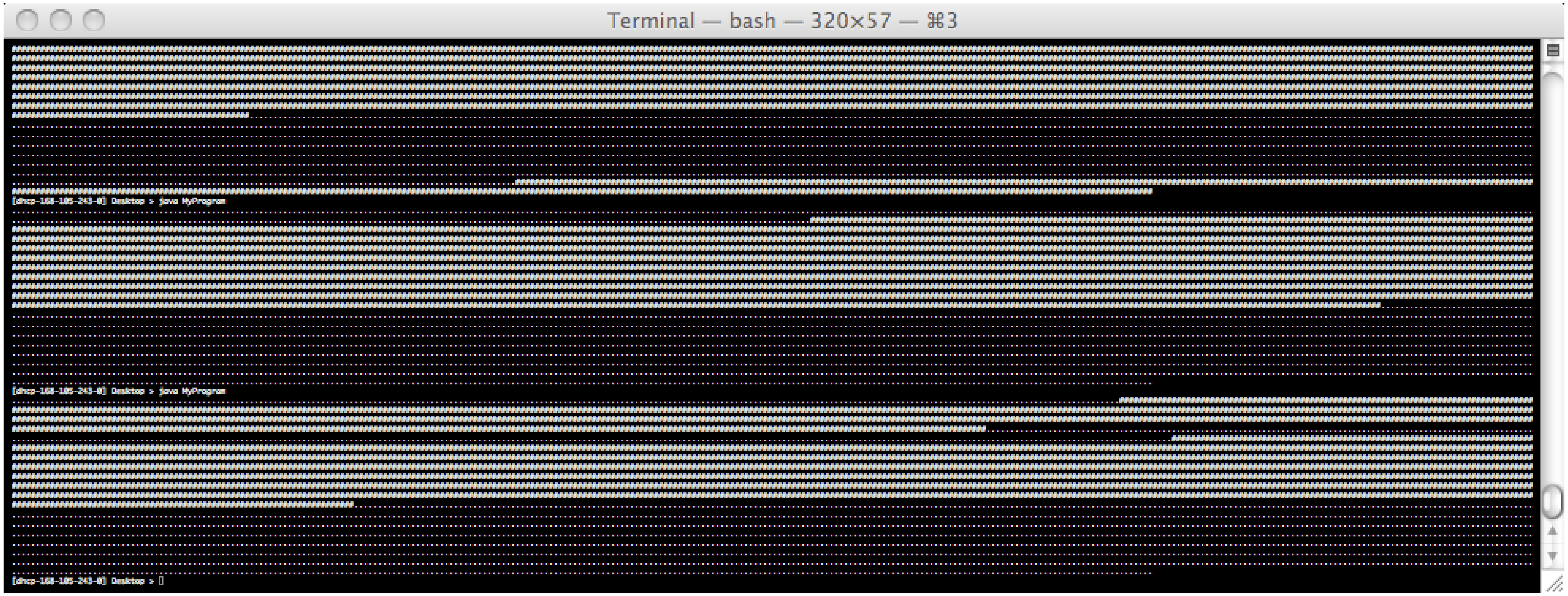
Example

```
public class myThread extends Thread {
    public void run() {
       for (int i=0; i<5; i++)
           System.out.println(“Hello world #“+i);
    }
}

public class MyProgram {
  public MyProgram() {
     MyThread t = new MyThread();
      t.start();
      for (int i=0; i<5; i++)
          System.out.println(“Beep ”+i);
  }
  public static void main(String args[]) {
     MyProgram p = new MyProgram();
  }
}
```

What happens?

- Now we have the main thread printing to the screen and the new thread printing to the screen
- Question: what will the output be?
- Answer: Impossible to tell for sure
  - If you know the implementation of the JVM on your particular machine, then you may be able to tell
  - But if you write this code to be run anywhere, then you can not expect to know what happens
- Let’s take a look at what happens in a program in which one thread prints “#” and the other prints “.” 1000 times each

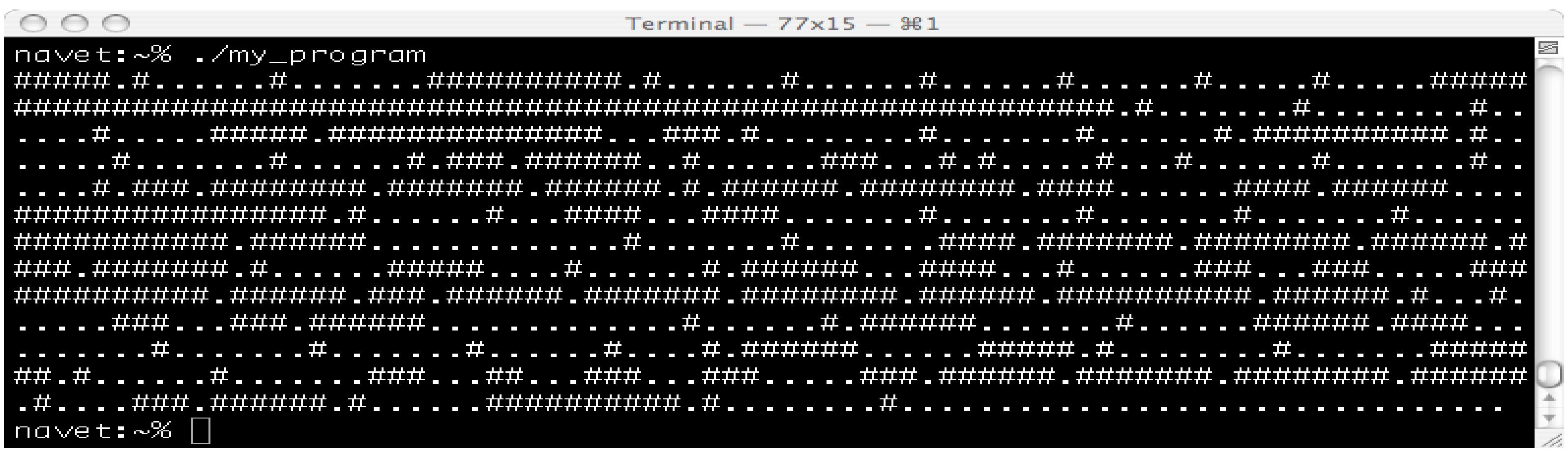
Three Sample Output



- Non-deterministic execution
- Somebody decides when a thread runs
  - You run for a while, now you run for a while,…
- This is called thread scheduling

Thread Programming

- Major Challenge: You cannot make any assumption about thread scheduling
  - Here is an example with C on Linux (no JVM)



- Major Difficulty: you may not be able to reproduce a bug each execution is different (Very frustrating!)

Thread Cancellation

Example: Kill the loading of a Web page in a browser

- Asynchronous: immediately cancel the thread. Can lead to problems if the thread has partially completed a critical function and thread owned resources may not be reclaimed
- Deferred: A thread periodically checks for termination at cancellation points. The thread then cancels itself.
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation only requests cancellation, the actual cancellation depends on the thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Thread Pools

- A group of threads created in advance that await work to do
  - Advantages
    - Eliminate overhead of thread creation and destruction
    - Applications can control the size of the pool
    - Avoids creating an unlimited number of threads which can tax the system
  - Java Options
    - Single thread executor – pool of size 1 Executors.newSingleThreadExecutor()
    - Fixed thread executor-pool of fixed size. Executors.newFixedThreadPool(int nThreads)
    - Cached thread pool –pool of unbounded size Executors.newCachedThreadPool()

Note: Dynamic thread pools adjust the number of threads based on system load

Thread Pool Example

Import Java.util.concurrent

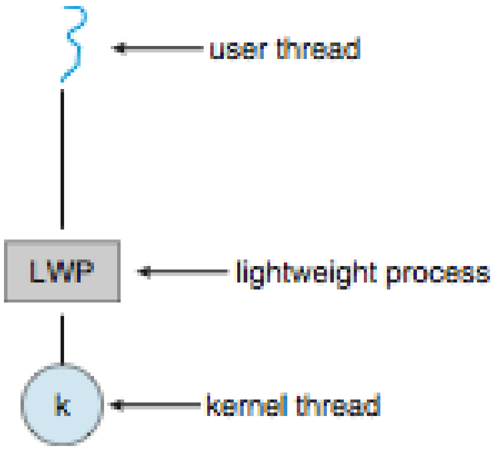
```
public class SomeThread implements Runnable
{  public void run()
   { System.out.println(new Date()); }
}

public class Pool
{ public static void main(String[] args)
  { int tasks=Integer.parseInt(args[0].trim());
    ExecutorService pool =
              Executors.newCachedThreadPool();
    for (int i=0; i<numTasks; i++)
           pool.execute(new SomeThread());
    pool.shutdown();
}
```

Thread Specific Data

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to **`static`** data
  - TLS is unique to each thread

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads



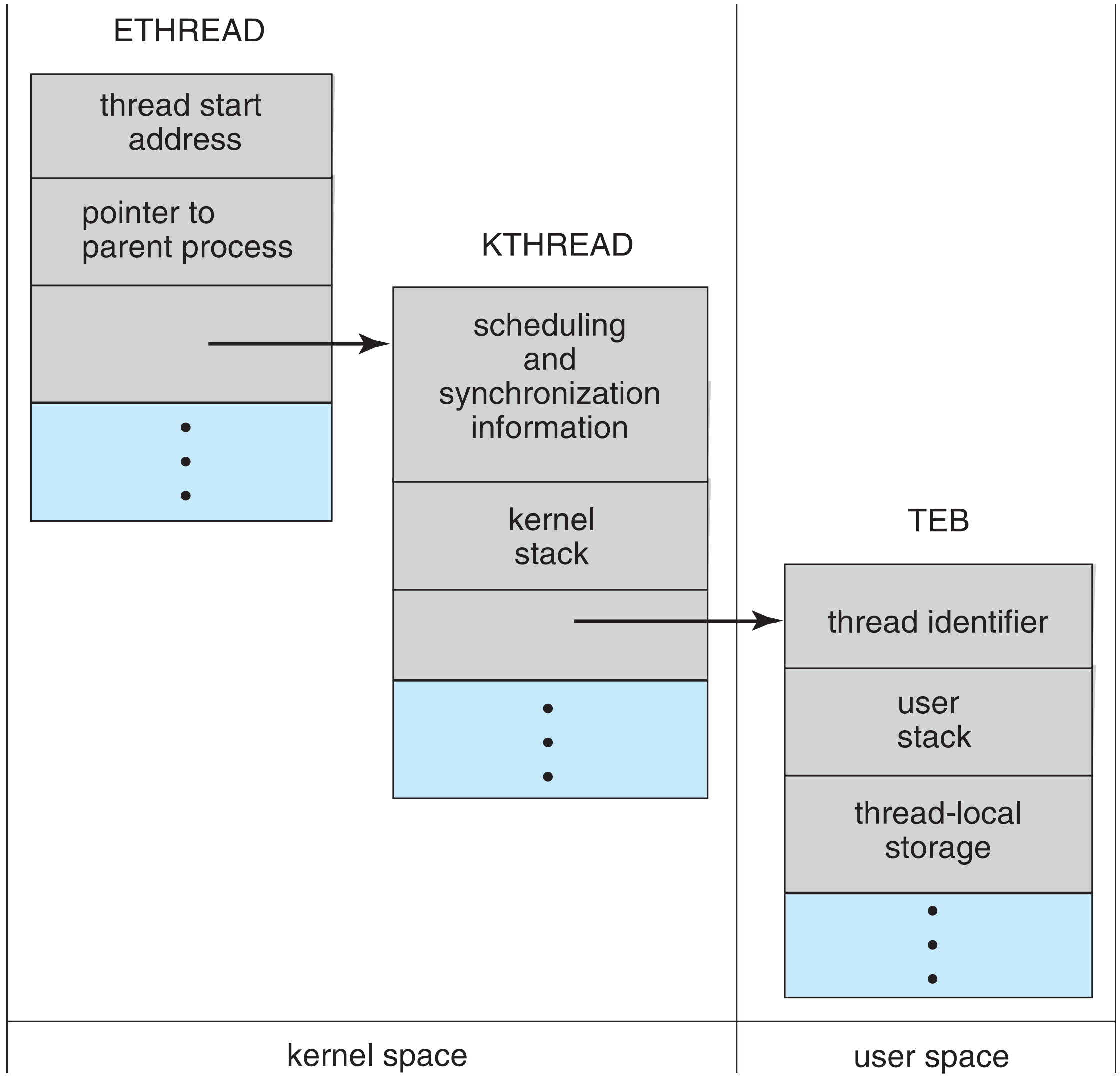
Signals

- We’ve talked about signals for processes
  - Signal handlers are either default or user-specified
  - signal() and kill() are the system calls
- In a multi-threaded program, what happens?
- Multiple options
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals
- Most UNIX versions: a thread can say which signals it accepts and which signals it does not accept
- On Linux: dealing with threads and signals is tricky but well understood with many tutorials on the matter and man pages
  - man pthread_sigmask
  - man sigemptyset
  - man sigaction

Win XP Threads

- Win XP uses one-to-one mapping
  - Many-to-Many via a separate library
- A thread is defined by its context
  - An ID
  - A register set
  - A user stack and a kernel stack
    - For user mode and kernel mode
  - A private storage area for convenience
- The OS keeps track of threads in data structures, as seen in the following figure

Win XP Threads



Linux Threads

- Linux does not distinguish between processes and threads: they're called **tasks**
  - Kernel data structure: task_struct
- The clone() syscall is used to create a task
  - Allows to specify what the new task shares with its parent
  - Different flags lead to something like fork() or like pthread_create()

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Conclusion

- Threads are something you cannot ignore today
  - Multi-core programming
- Programming with threads is known to be difficult, and a lot of techniques/tools are available
- In this course we focus more on how the OS implements threads than how the user uses threads

- We will use threads in a Programming Assignment soon, probably not #2 maybe #3
- We skip over Chapter 5 for now “Process Synchronization” and come back to it.
- Read Chapter 6 “CPU Scheduling”