Where are we?

- Threads, multithreading model

Two Types of Threads

- User threads
- Kernel threads

User and Kernel Threads

- **User threads** - Management done by user-level threads library
  - Kernel has no knowledge about user threads
  - Three primary thread libraries:
    - POSIX **Pthreads,** Windows threads, Java threads
- **Kernel threads** - Supported by the Kernel
  - Lightweight process (LWP)
  - Examples – virtually all general-purpose operating systems, including Windows, Linux, OSX, Android…

User and Kernel Threads

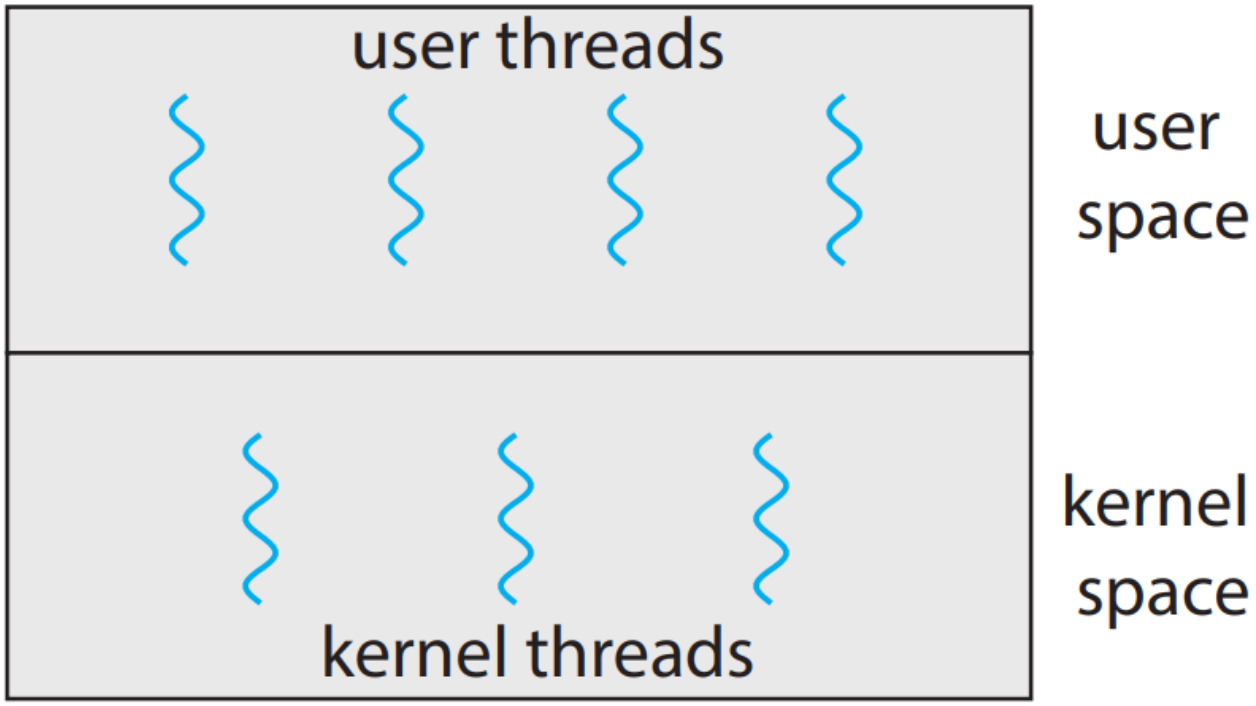
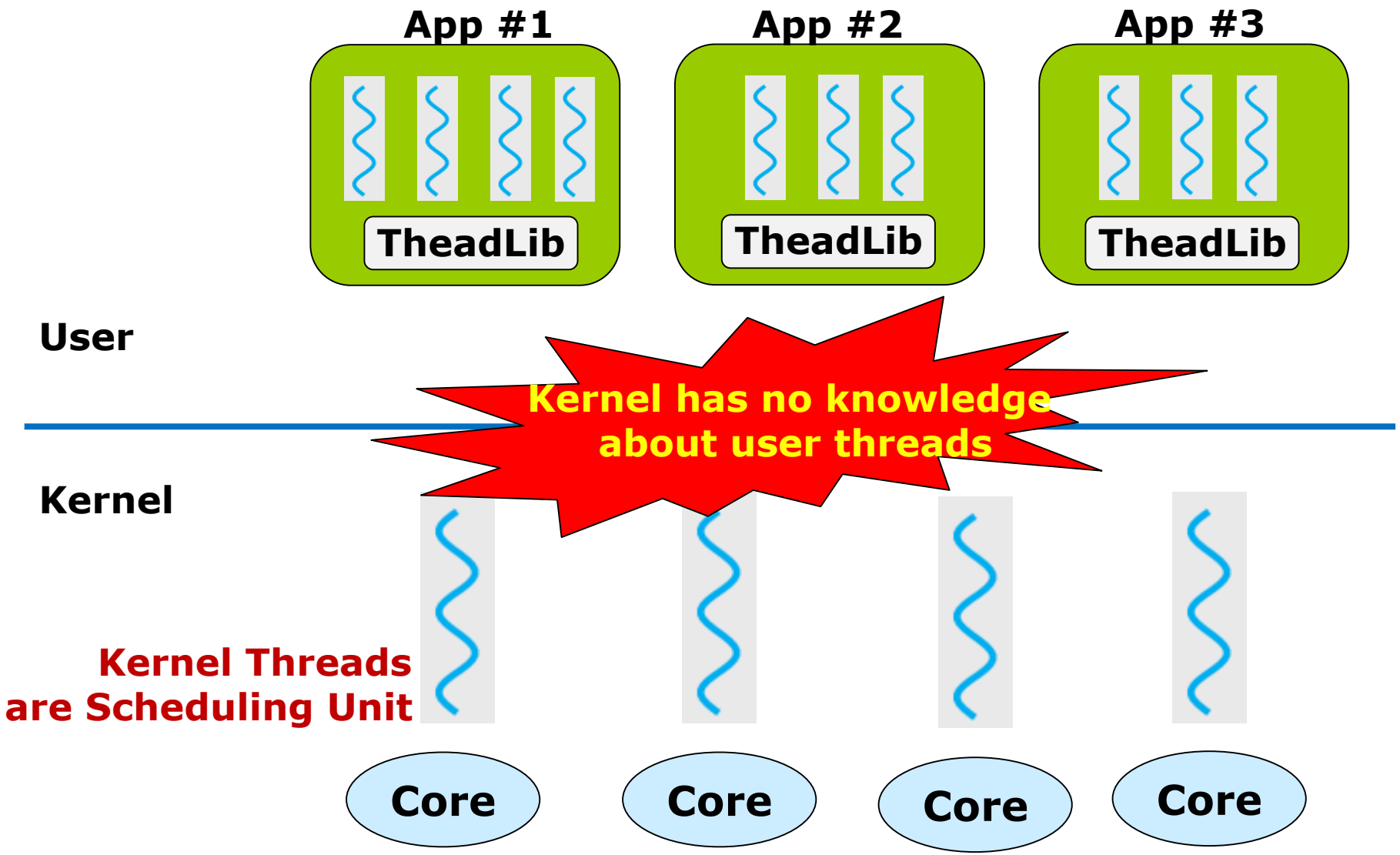
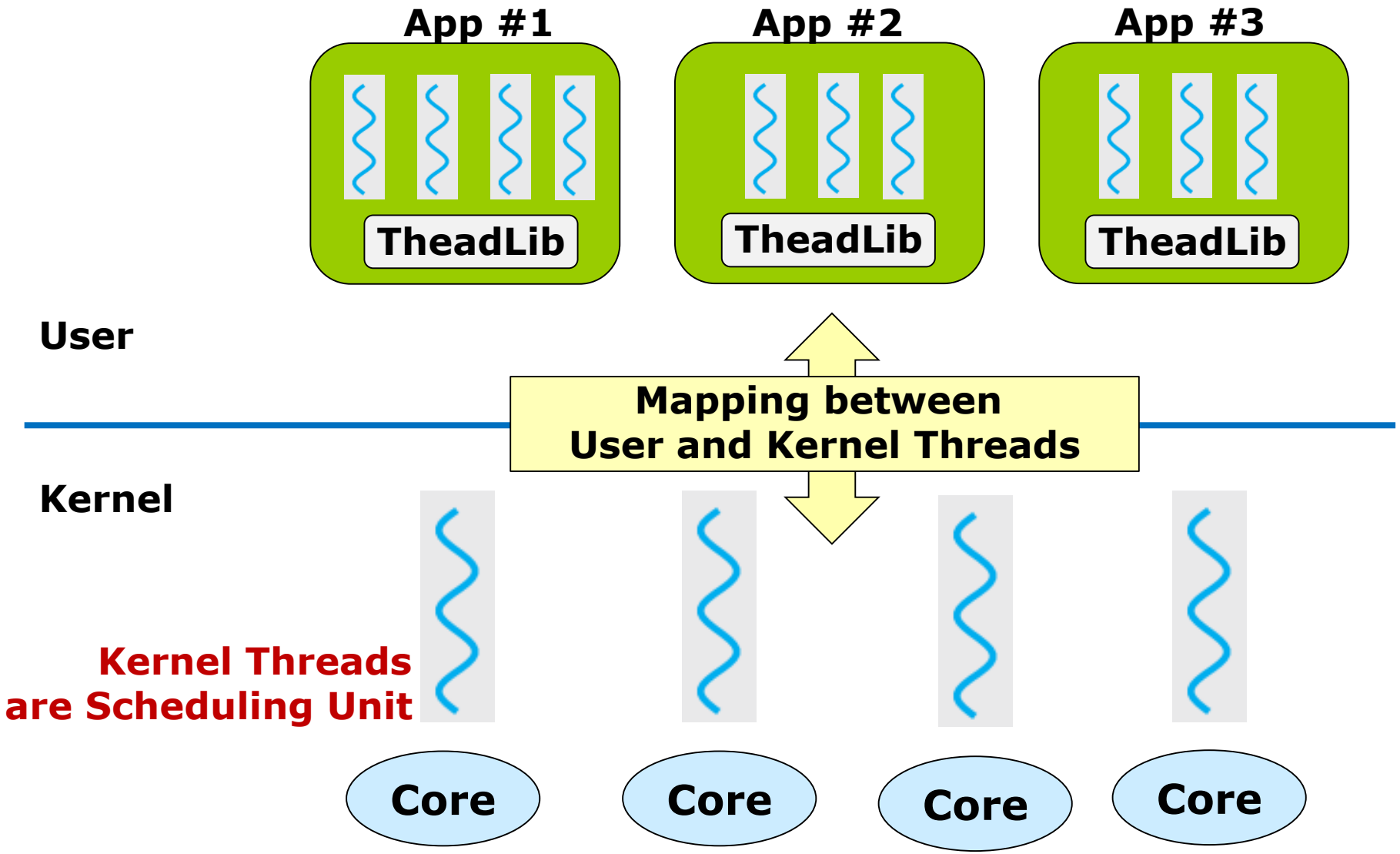


Figure 4.6 User and kernel threads.

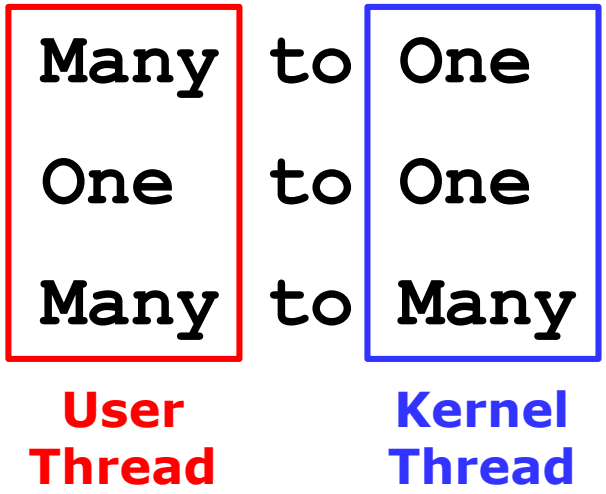
User and Kernel Threads



User and Kernel Threads

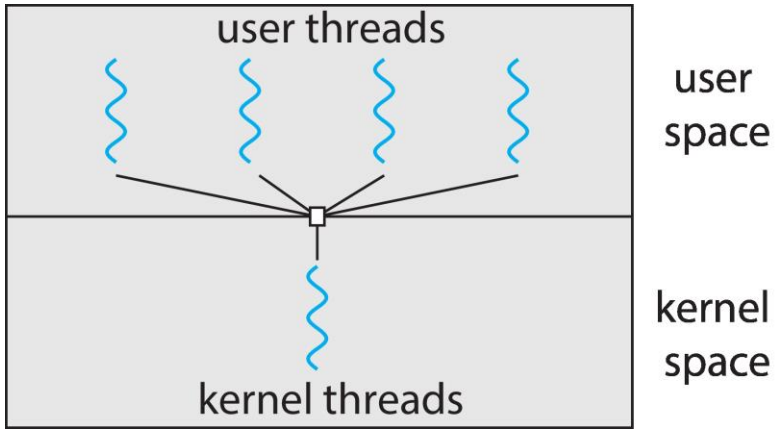


Multithreading Models



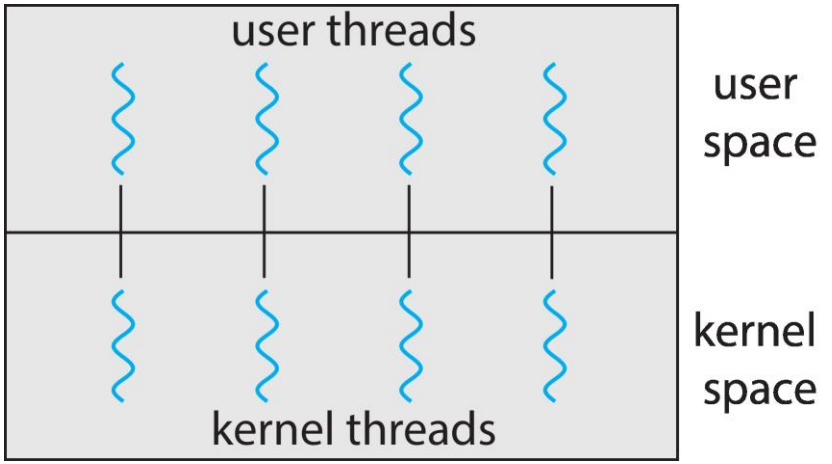
Many to One

- Many user-level threads mapped to single kernel thread
- Potential Issue?
  - **One thread blocking causes all to block**
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model. e.g., Solaris



One to One

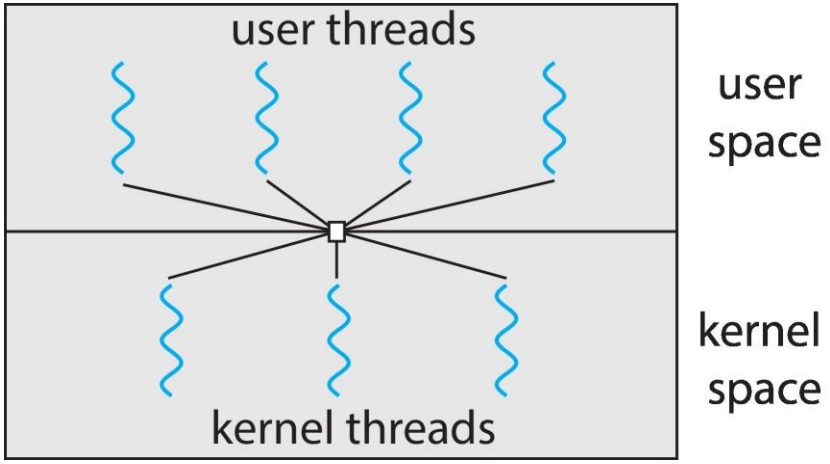
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread



- More concurrency than many-to-one
- Potential Issues?
  - **Number of threads per process sometimes restricted due to overhead**
- Examples – Windows, Linux

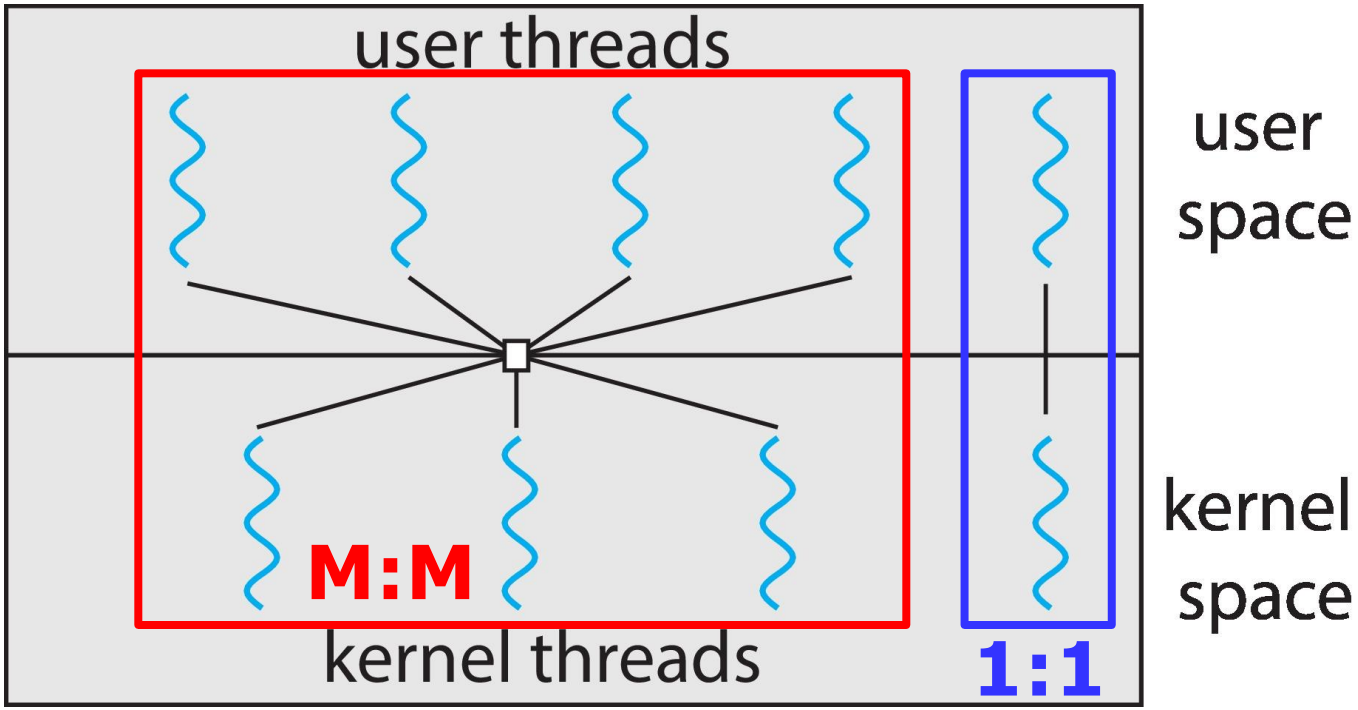
Many to Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
  - Otherwise not very common



(Hybrid) Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
    - No System Calls
  - Kernel-level library supported by the OS

Thread Libraries

- **Synchronous Threading**
  - Main (parent) thread creates children threads
  - Main thread must wait for all of its children to terminate before it resumes
    - Each child thread finishes → join with parent
  - Significant data sharing
- **Asynchronous Threading**
  - Main thread and children threads are independent
  - Little data sharing between them

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux & OS X)

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

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

What is output?

$$sum = \sum_{i=1}^{N} i$$

```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

```
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

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

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

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

  - Standard practice is to implement Runnable interface

Java Threads

Implementing Runnable interface:

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

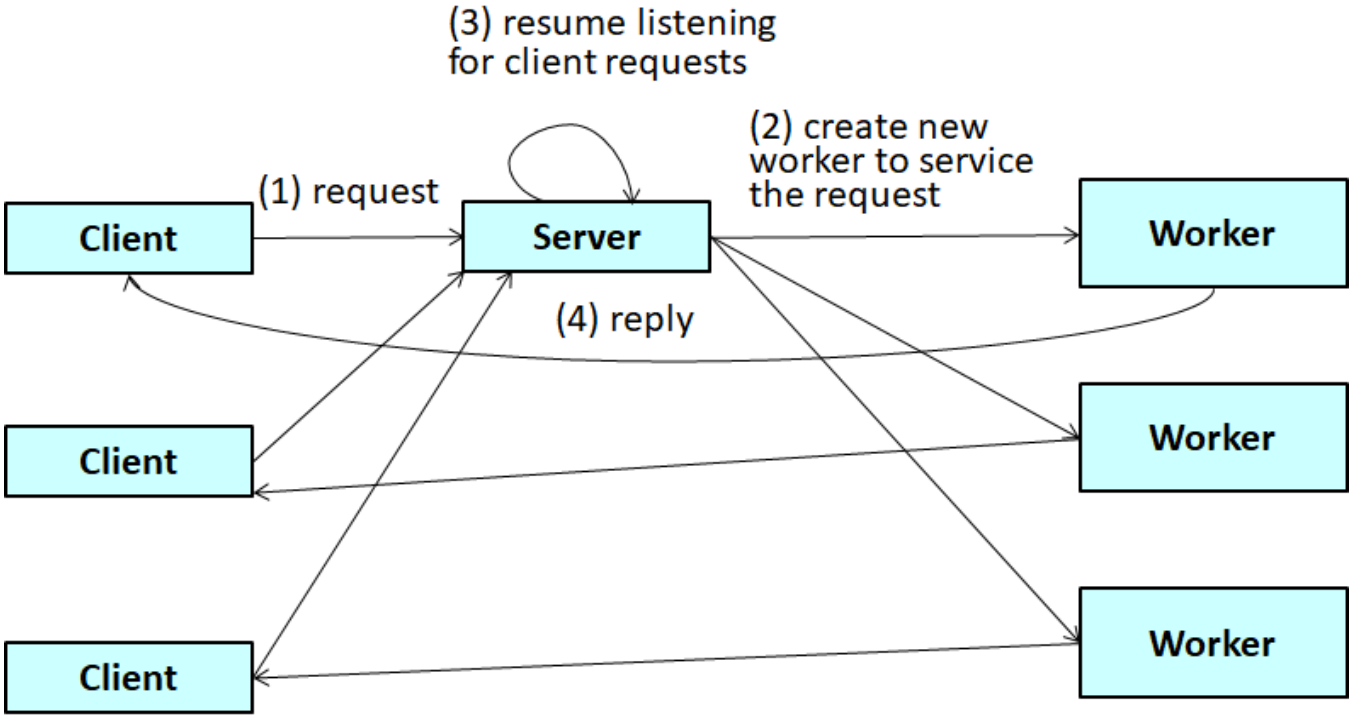
```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Implicit Threading

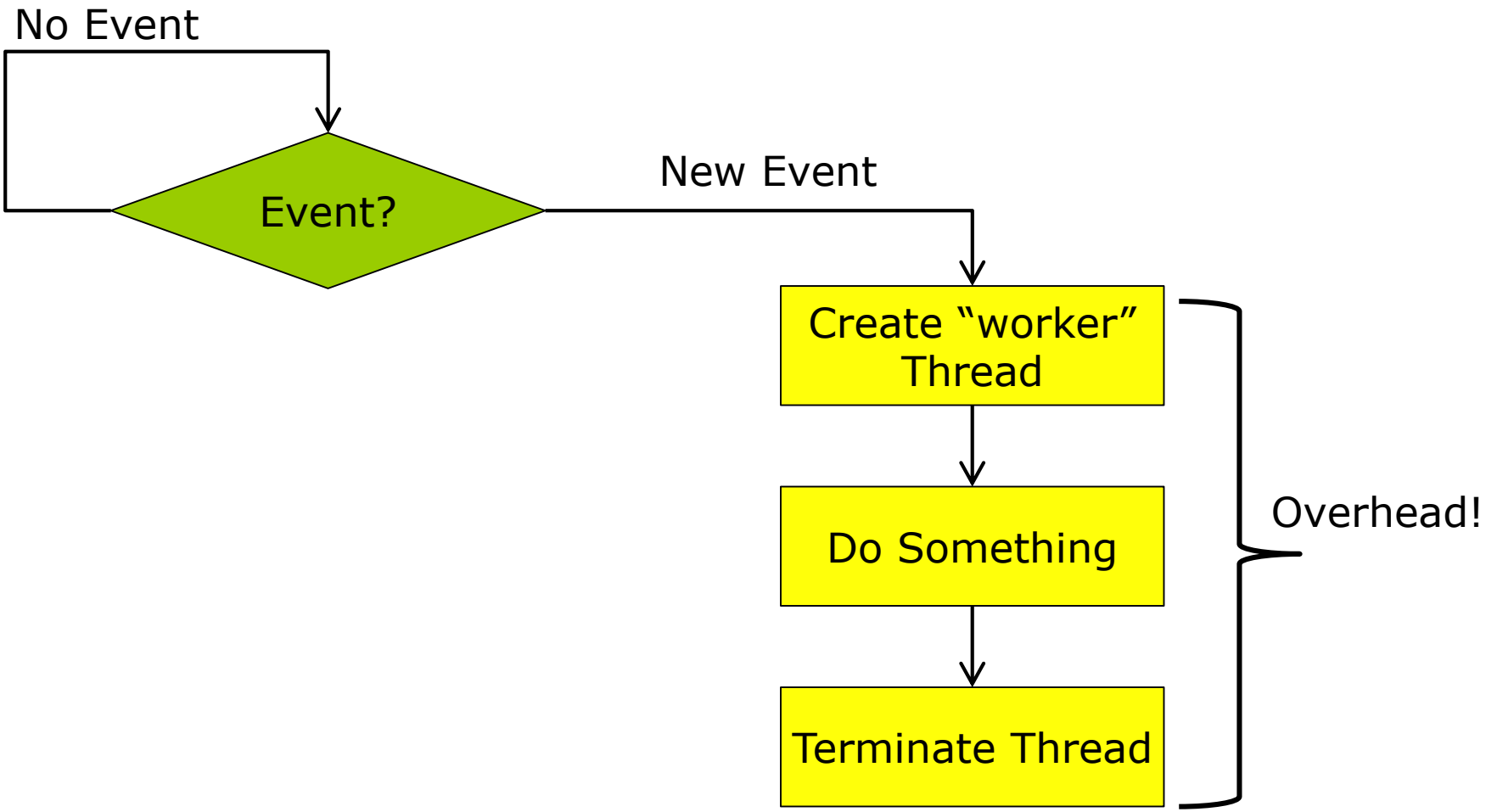
- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Five methods explored
  - **Thread Pools**, Fork-Join, OpenMP, Grand Central Dispatch, Intel Threading Building Blocks

Typical Thread Model in “Server”

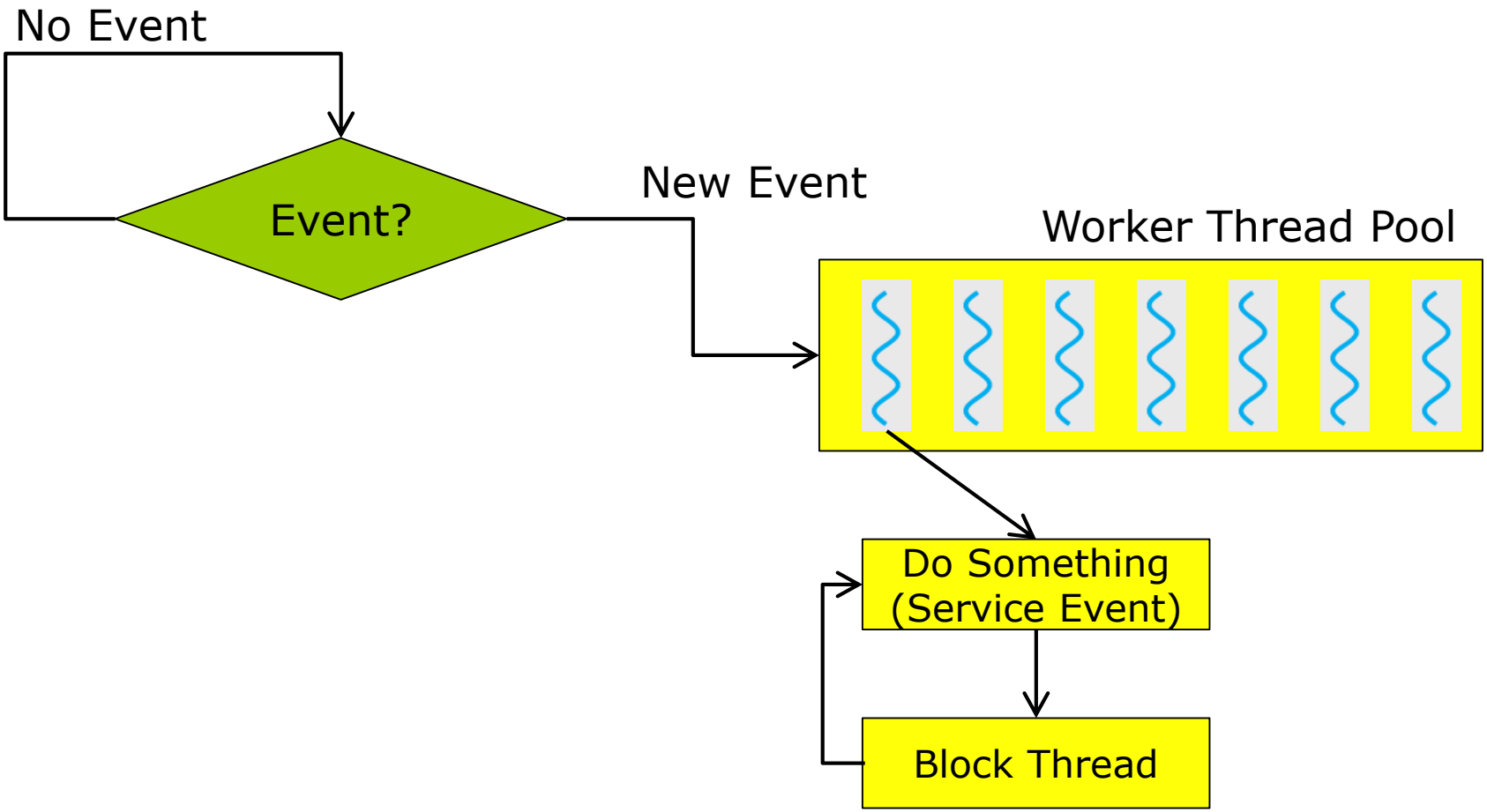
- Web server handles multiple client connections simultaneously



Typical Thread Model in “Server”



Thread Pools



Thread Pools

- Create a number of threads in a pool where they await work
- What are Advantages:
  - Usually *slightly* faster to service a request with an existing thread than create a new thread
    - No Thread Startup/Creation Delay
  - Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Pools

- How many threads an app can create?
  - `cat /proc/sys/kernel/threads-max`
- How many processes OS can create?
  - `cat /proc/sys/kernel/pid_max`

Thread Issues

- These are more like design and/or policy issues
  - Semantics of **fork()** and **exec()** system calls
  - Signal handling
    - Synchronous and asynchronous
  - Thread-local storage

Semantics of `fork()` and `exec()`

- When a thread calls **`fork()`**
  - Duplicate all threads
  - Duplicate one thread
  - Some UNIXes have two versions of **`fork()`**

- **`exec()`** usually works as normal
  - Replace the running process including all threads

Signal Handling

- A signal is an ***asynchronous*** event (software interrupt), which is delivered to a process.
  - man 7 signal

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     - default
     - user-defined

Signal Handling

- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override the default handler
  - For single-threaded, signal delivered to process

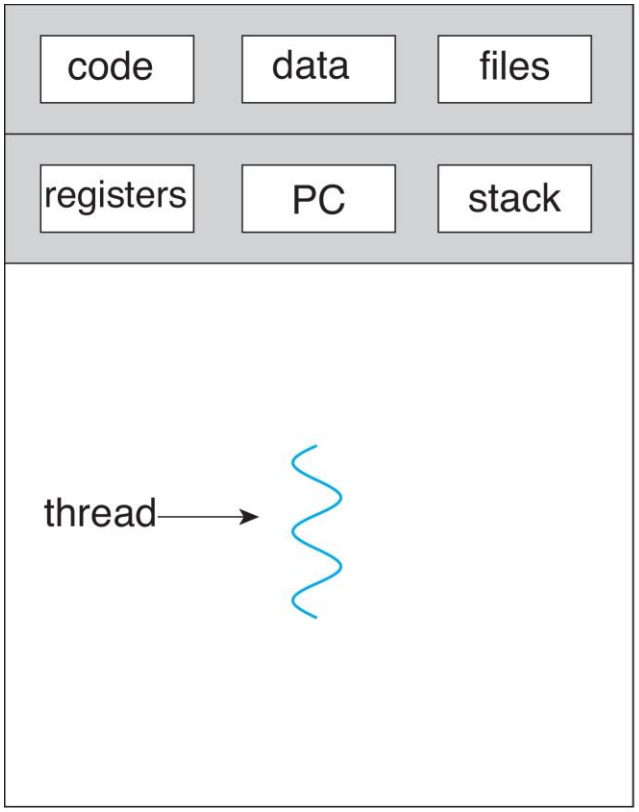
Signal Handling

- Where should a signal be delivered for multi-threaded?
  - 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 for the process

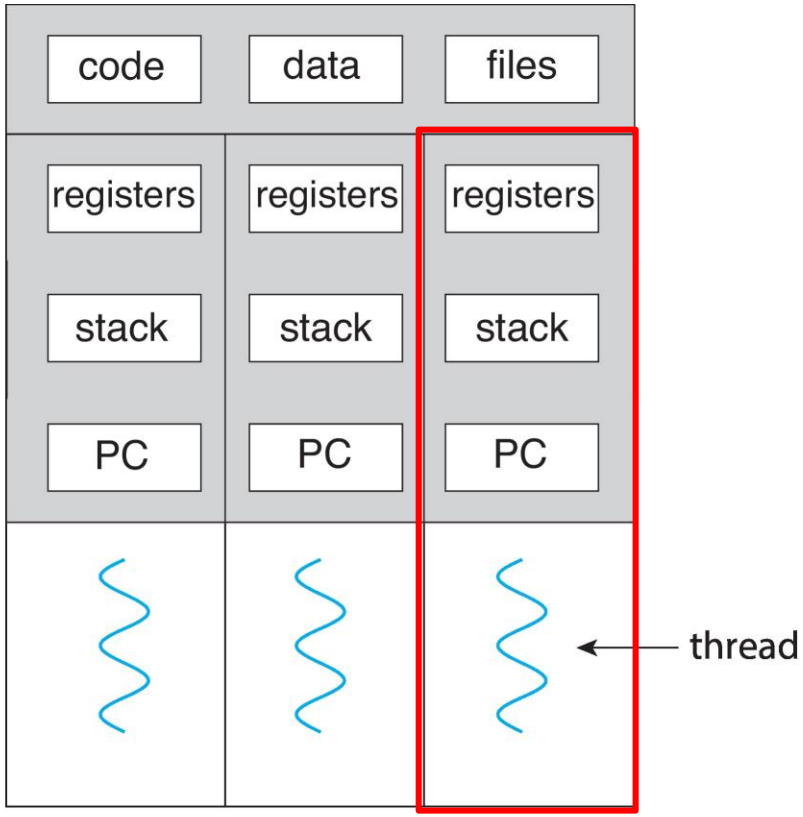
Similar Issue: Segmentation Fault

- Segmentation fault in a thread, should OS terminate only the thread or the entire process?

Thread-Local Storage



single-threaded process



multithreaded process

What if a thread needs its own copy of data?

Thread-Local Storage

- **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)
  - Why?
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- TLS is unique to each thread
  - It is something between local and global variable

TLS Example

nike.cs.uga.edu - PuTTY

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <syscall.h>

int global = 0;
__thread int tls = 0;

void *mythread(void *arg) {
    int local = 0;


    printf("[%d]: mythread: Address: global-%p, tls-%p, local-%p\n",
        syscall(186)-getpid(), &global, &tls, &local);
    printf("[%d]: mythread: Value: global-%d, tls-%d, local-%d\n\n",
        syscall(186)-getpid(), ++global, ++tls, ++local);

    return NULL;
}

int main() {
  pthread_t p1, p2;
  int rc;

  tls = 100;
  global = 100;

  printf("[%d] main value: begin, tls = %d, global = %d\n", getpid(), tls, global);
  printf("[%d] main addre: begin, tls = %p, global = %p\n\n", getpid(), &tls, &global);
  rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
  rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);

  rc = pthread_join(p1, NULL); assert(rc == 0);
  rc = pthread_join(p2, NULL); assert(rc == 0);
  printf("[%d] main: end, tls = %d, global = %d\n", getpid(), tls, global);
  return 0;
}
```

1,1 Top

TLS Example 2

- TLS visible across function invocations

nike.cs.uga.edu - PuTTY

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <syscall.h>

__thread int tls;
int global;
int *ptr;

int func_a()
{
    int local = 100;
    tls = rand() % 100;

    printf("[%d]: func_a: Addre: global-%p, tls-%p, local-%p\n",
        syscall(186)-getpid(), &global, &tls, &local);
    printf("[%d]: func_a: Value: global-%d, tls-%d, local-%d\n\n",
        syscall(186)-getpid(), global, tls, local);
}

void *mythread(void *arg) {
    int local = 0;

    func_a();

    printf("[%d]: mythread: Addre: global-%p, tls-%p, local-%p\n",
        syscall(186)-getpid(), &global, &tls, &local);
    printf("[%d]: mythread: Value: global-%d, tls-%d, local-%d\n\n",
        syscall(186)-getpid(), global, tls, local);

    return NULL;
}

int main() {
    pthread_t p1, p2;
    int rc;

    tls = 100;
    global = 100;

    printf("[%d] main: begin, tls = %d, global = %d\n\n", getpid(), tls, global);
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);

    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("[%d] main value: end, tls = %d, global = %d\n", getpid(), tls, global);
    printf("[%d] main addre: end, tls = %p, global = %p\n", getpid(), &tls, &global);
    return 0;
}

                                                    1,1           All
```

TLS Example 3

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <syscall.h>

__thread int tls;
int global;
int *ptr;

int func_a()
{
    int local = 100;
    tls = rand() % 100;

    printf("[%d]: func_a: Addre: global-%p, tls-%p, local-%p\n",
        syscall(186)-getpid(), &global, &tls, &local);
    printf("[%d]: func_a: Value: global-%d, tls-%d, local-%d\n\n",
        syscall(186)-getpid(), global, tls, local);
}

void *mythread(void *arg) {
    int local = 0;

    func_a();

    if(*(char*)arg  == 'A') {
        ptr = &tls;
        sleep(2);
    } else {
        sleep(1);
        *ptr = 1234;
        printf("[%d]: change A's tls value to 1234\n", syscall(186) - getpid());
    }
    printf("[%d]: mythread: Addre: global-%p, tls-%p, local-%p\n",
        syscall(186)-getpid(), &global, &tls, &local);
    printf("[%d]: mythread: Value: global-%d, tls-%d, local-%d\n\n",
        syscall(186)-getpid(), global, tls, local);

    return NULL;
}
```

```c
int main() {
    pthread_t p1, p2;
    int rc;

    tls = 100;
    global = 100;

    printf("[%d] main: begin, tls = %d, global = %d\n\n", getpid(), tls, global);
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);

    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("[%d] main value: end, tls = %d, global = %d\n", getpid(), tls, global);
    printf("[%d] main addre: end, tls = %p, global = %p\n", getpid(), &tls, &global);
    return 0;
}
```