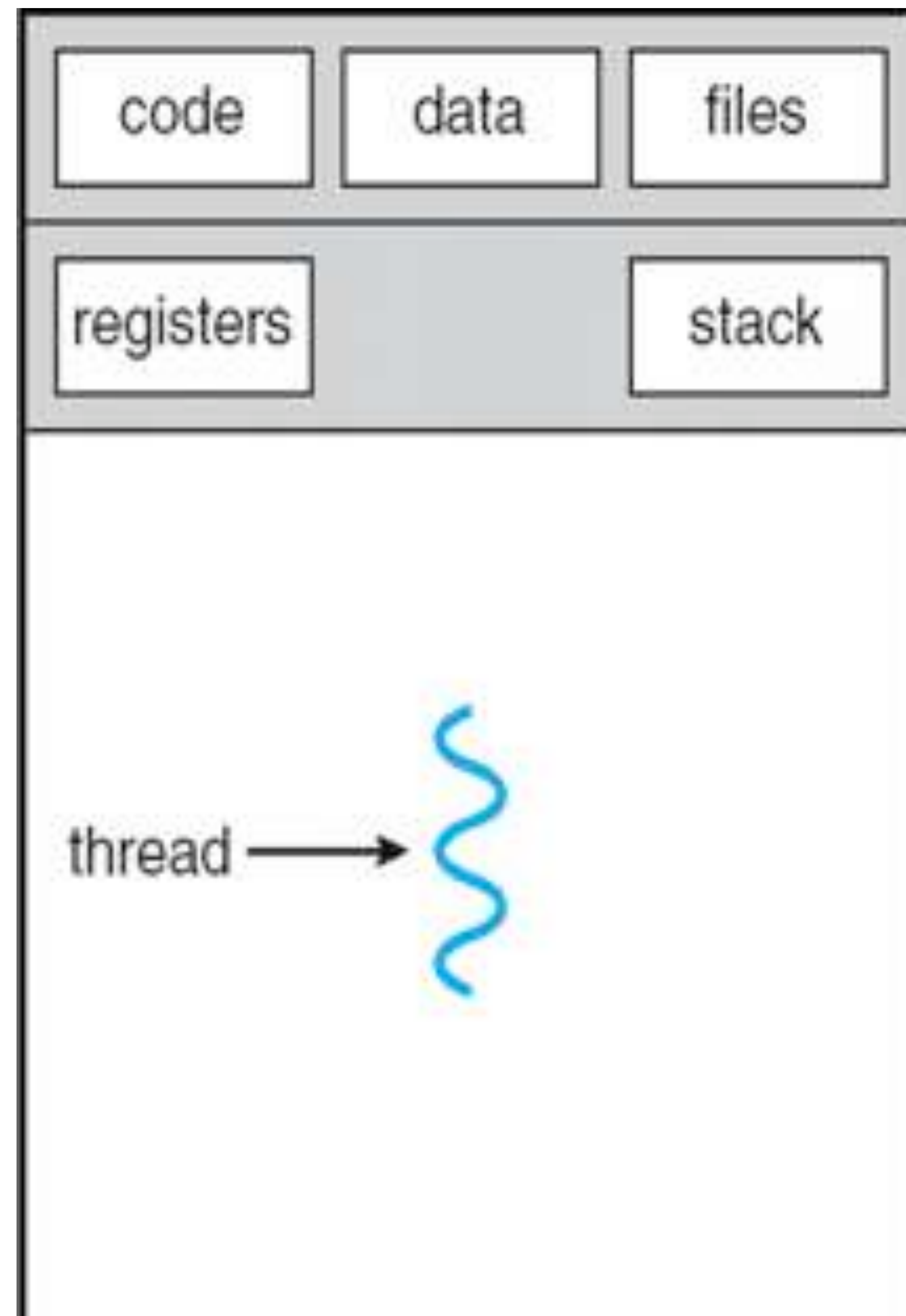
Threads

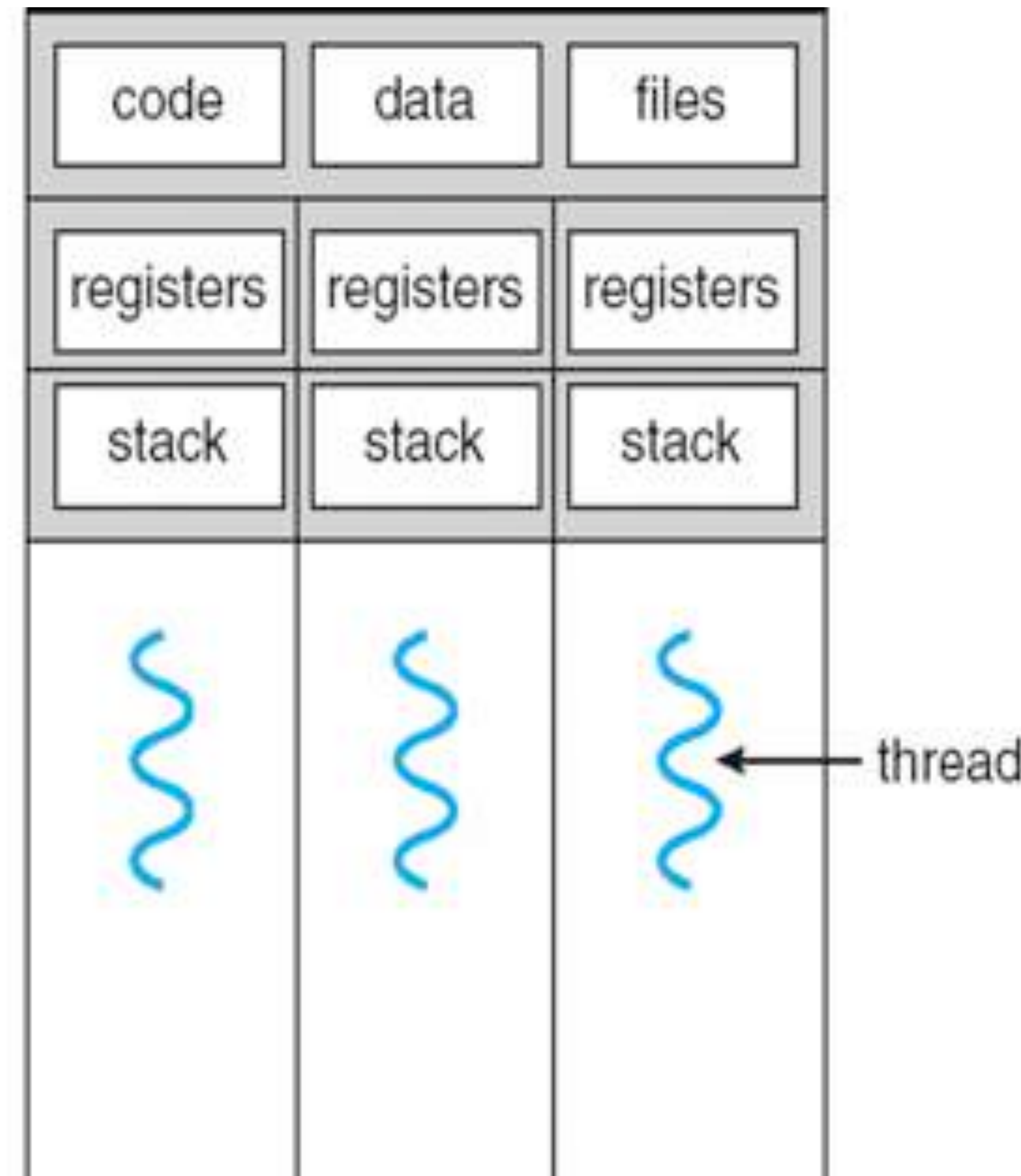
CSE4001 Operating Systems Concepts

E. Ribeiro

Single and Multithreaded Processes

- **Thread** – a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems





Threads

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

- Threads are discrete processing units that allow functions to execute concurrently (i.e., simultaneous execution of functions while taking turns in the CPU).
- Useful when functions take too long to complete their tasks as they should not block other functions.
- When an application is launched, it contains only one thread (i.e., executes the `main()` function). This type of application is called a *single-threaded application*.

Threads

» *Multi-threaded applications* create new threads to execute multiple functions.

» Modern computer architecture offers multiple processing cores by default. Threads allow programmers to use the available processing capacity.

» Having multi-core machines by default means that knowing how to develop multi-threaded programs has become a key skill in modern programming.

Single and Multithreaded Processes

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

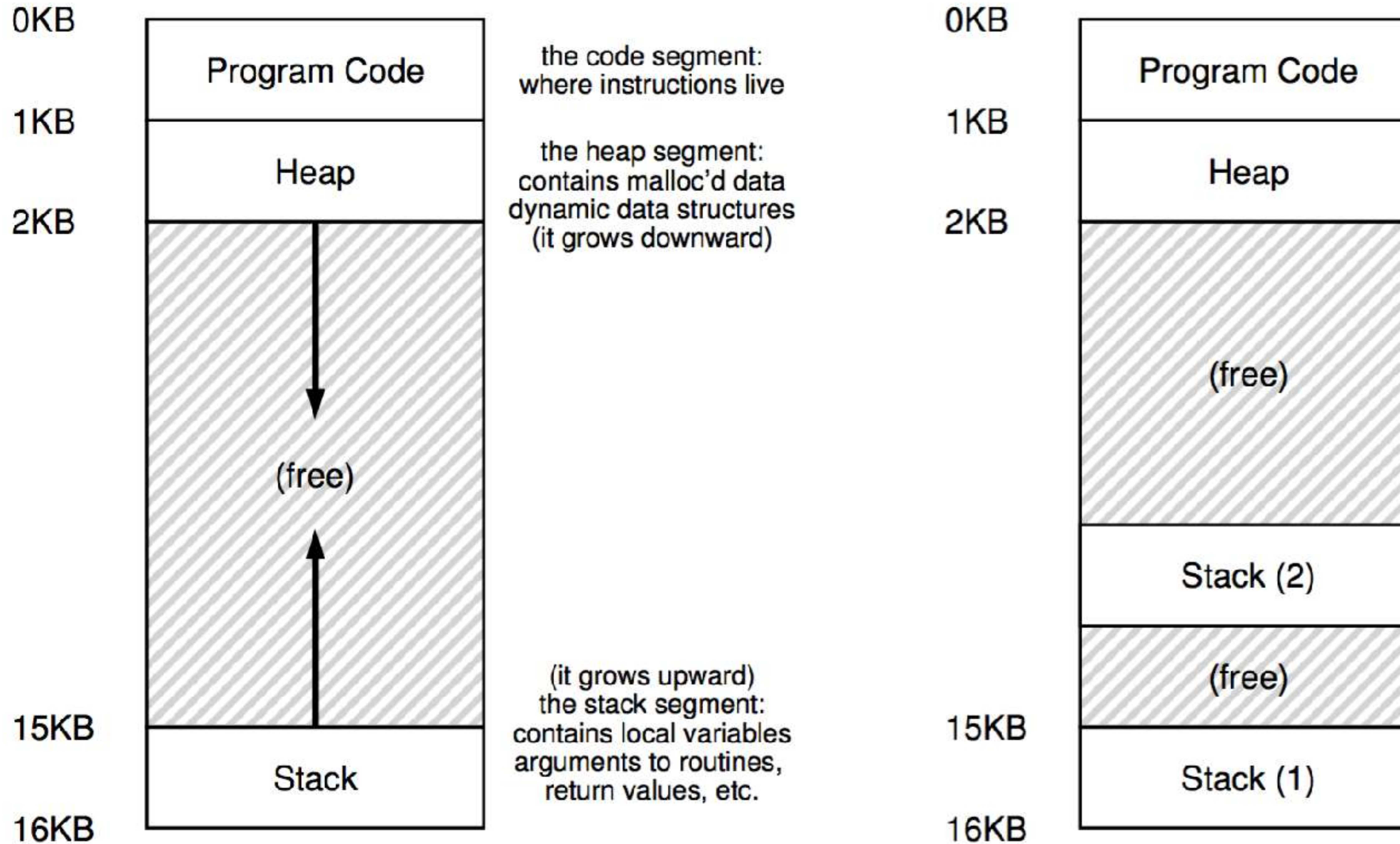
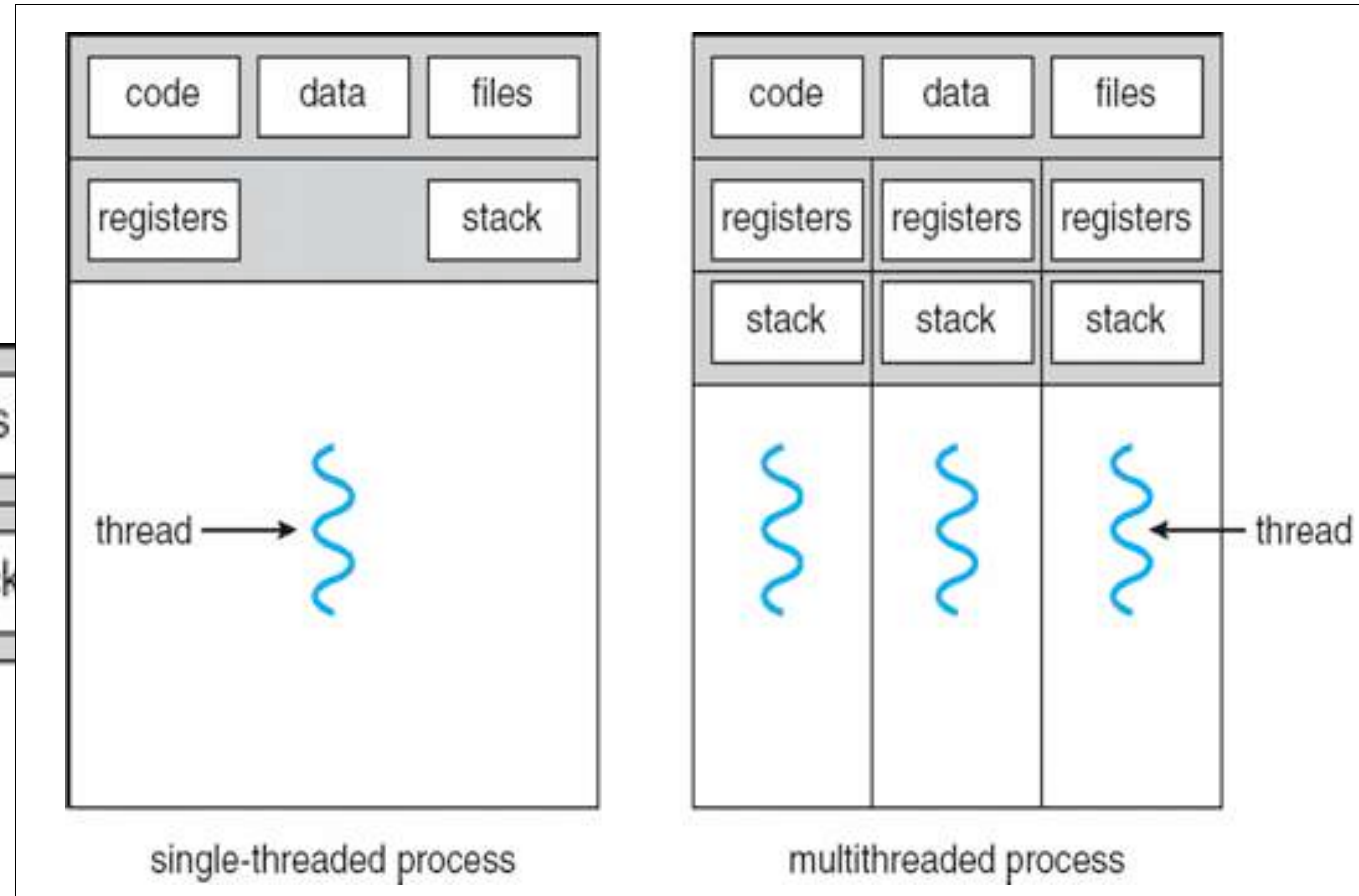


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

Figure from Arpaci-Dusseau

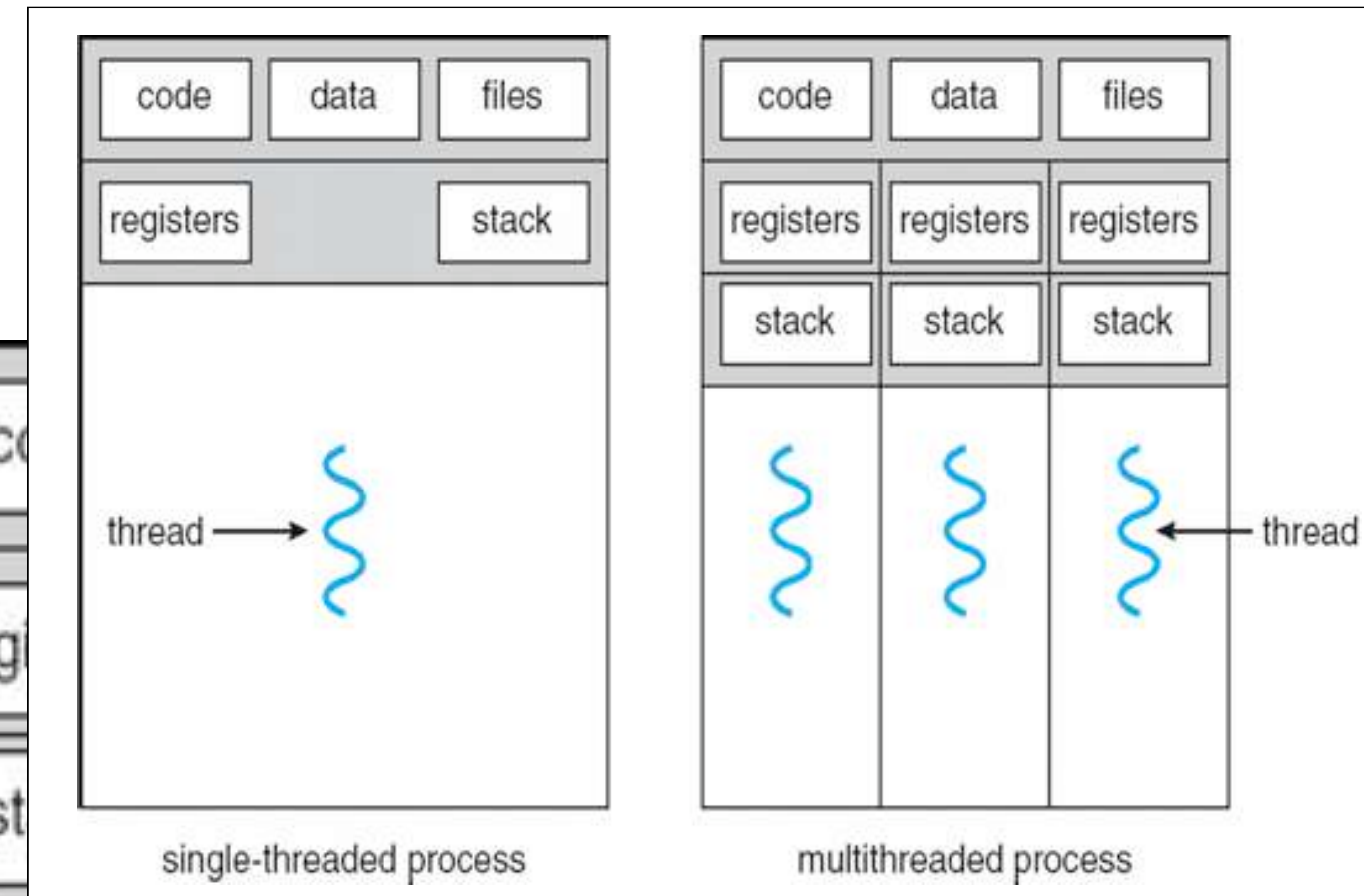
Single and Multithreaded Processes

- A thread is a basic unit of CPU utilization. It comprises:
  - a thread ID
  - a program counter
  - a register set
  - a stack



Single and Multithreaded Processes

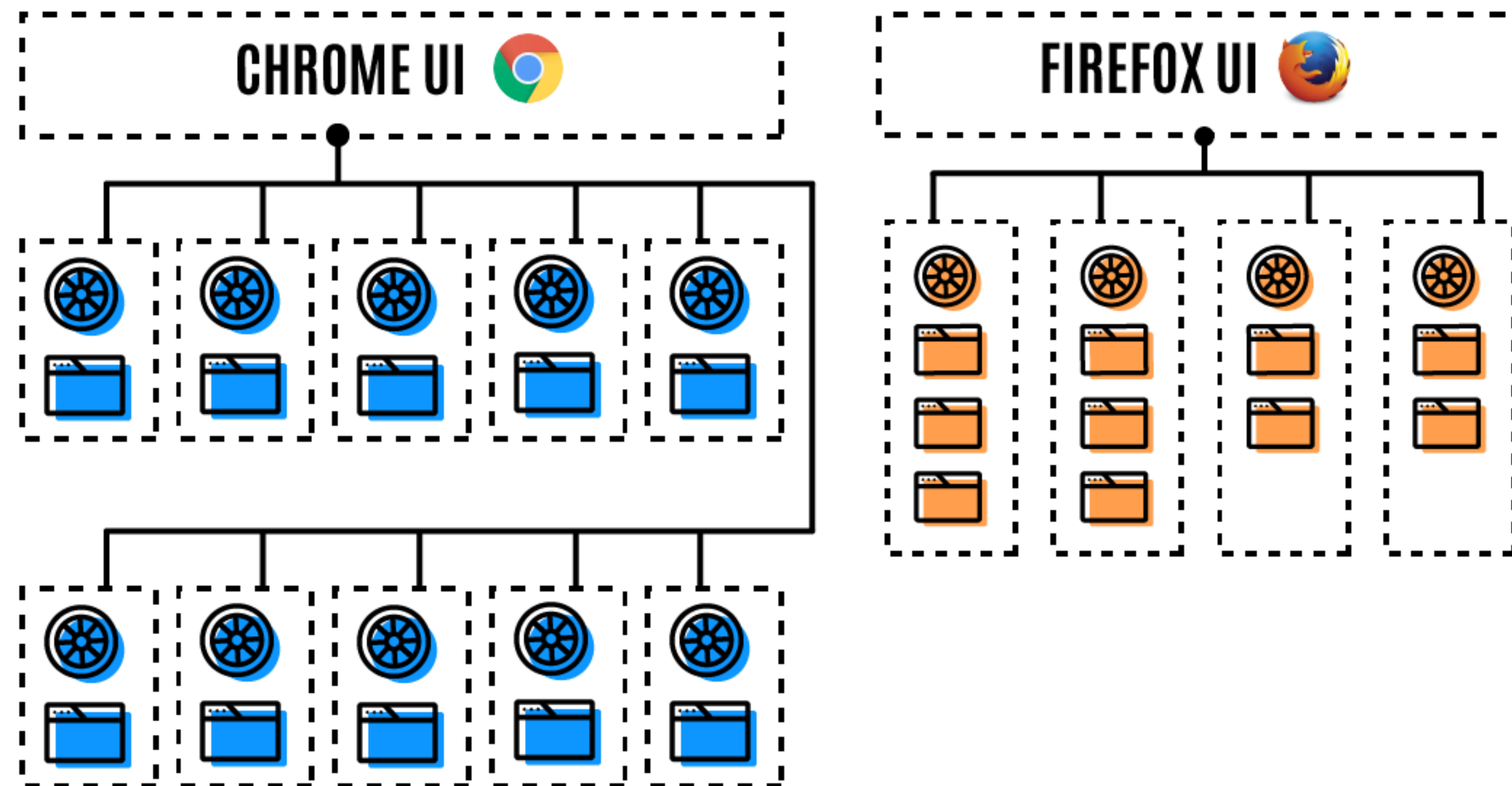
- A traditional process has a single thread of control.
- Processes that have multiple threads can perform multiple tasks concurrently.
- Software packages are usually multithreaded. They are implemented as a process with several threads of control.



Some multi-threaded packages

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

A web browser has threads for showing text, threads for showing images, threads to retrieve data from the network.

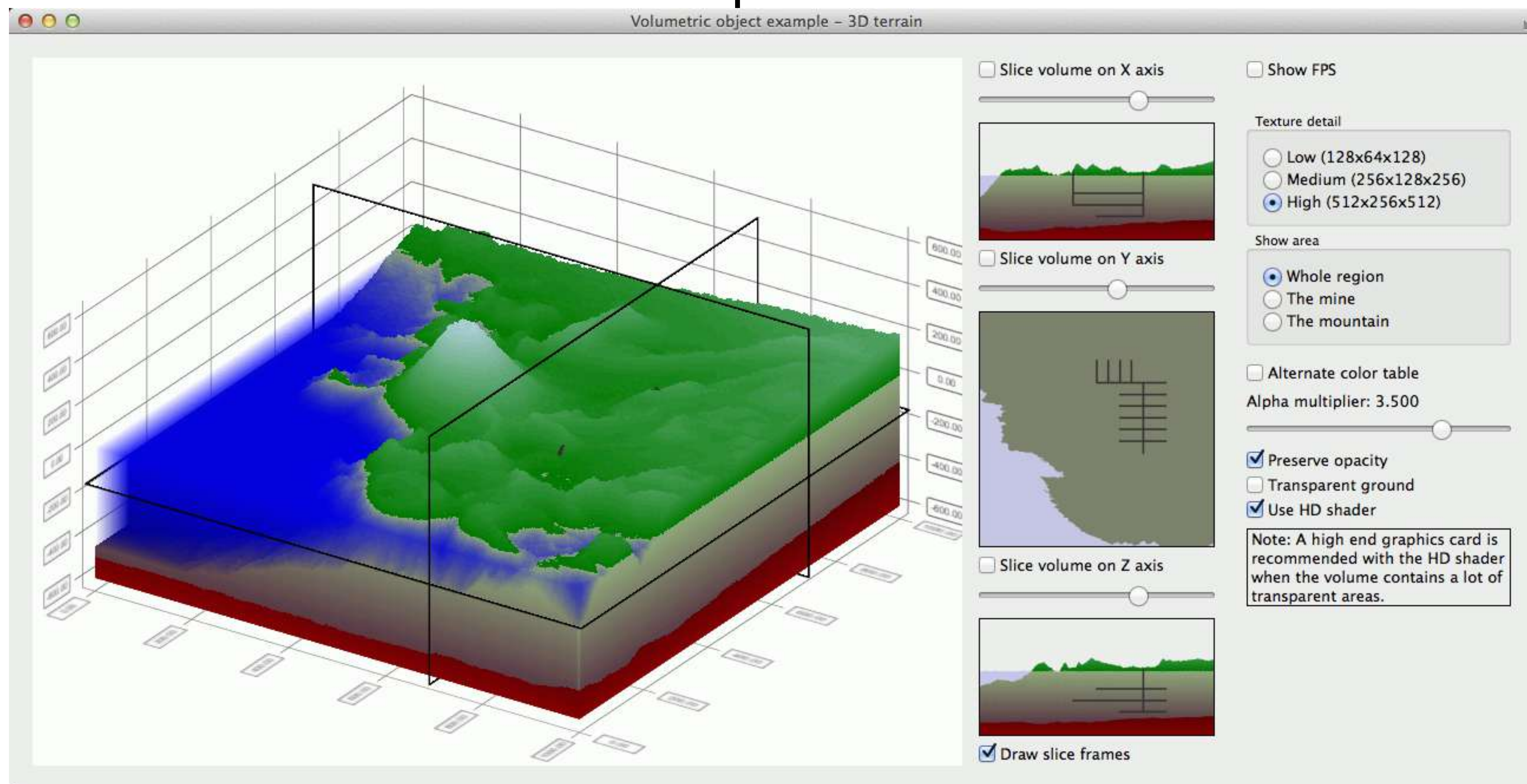


https://levelup.gitconnected.com/how-web-browsers-use-processes-and-threads-9f8f8fa23371

Some multi-threaded packages

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

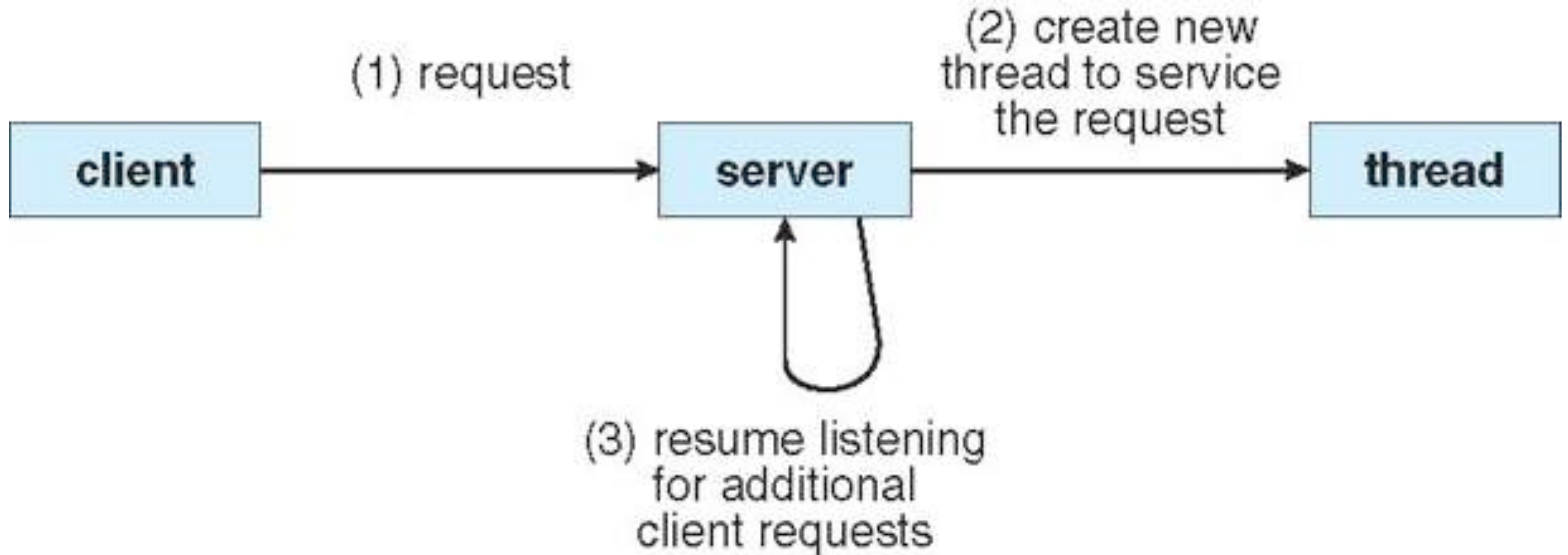
Most graphical user interfaces are multi-threaded programs.
Scientific software also uses multiple threads



https://doc.qt.io/qt-5.12/qtdatavisualization-volumetric-example.html

Multithreaded Server Architecture

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

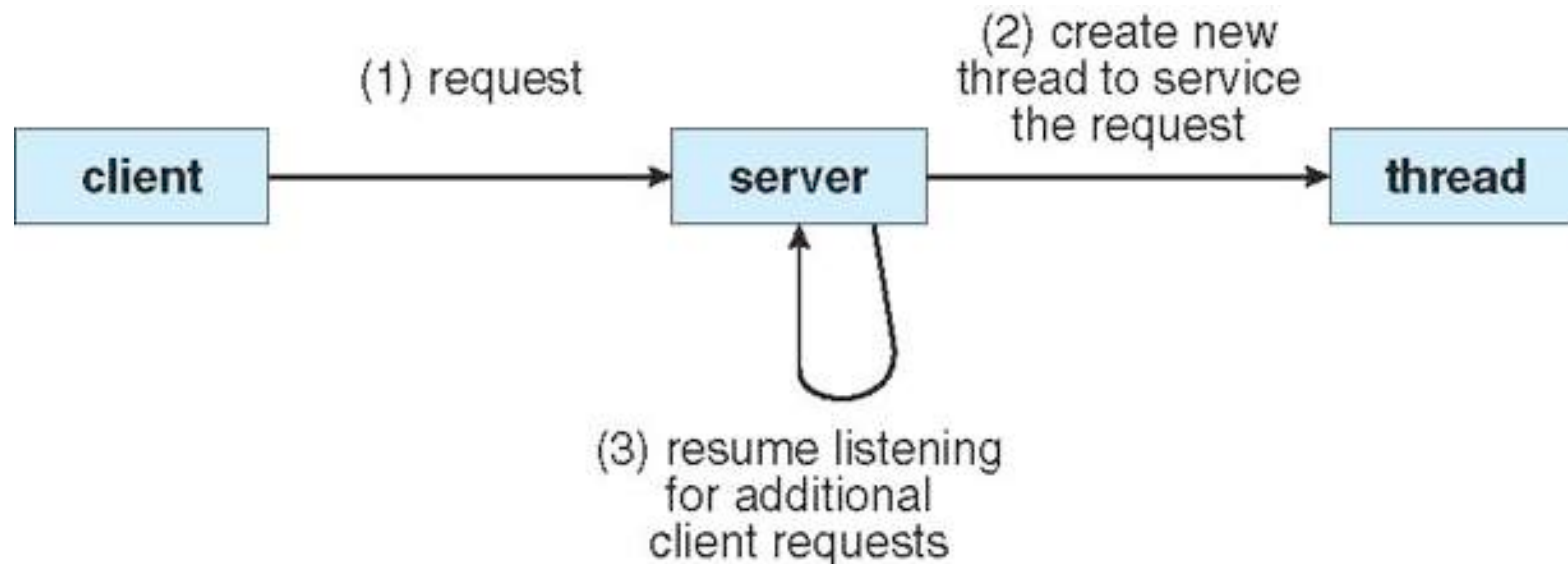


(3) resume listening
for additional

Multithreaded Server Architecture

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

- For a large number of clients, a single-threaded server implementation would take too long to respond.
- Processes were used often to solve this problem until threads became popular. Threads are known as light-weight processes.



Modern OSs are mult0-threaded

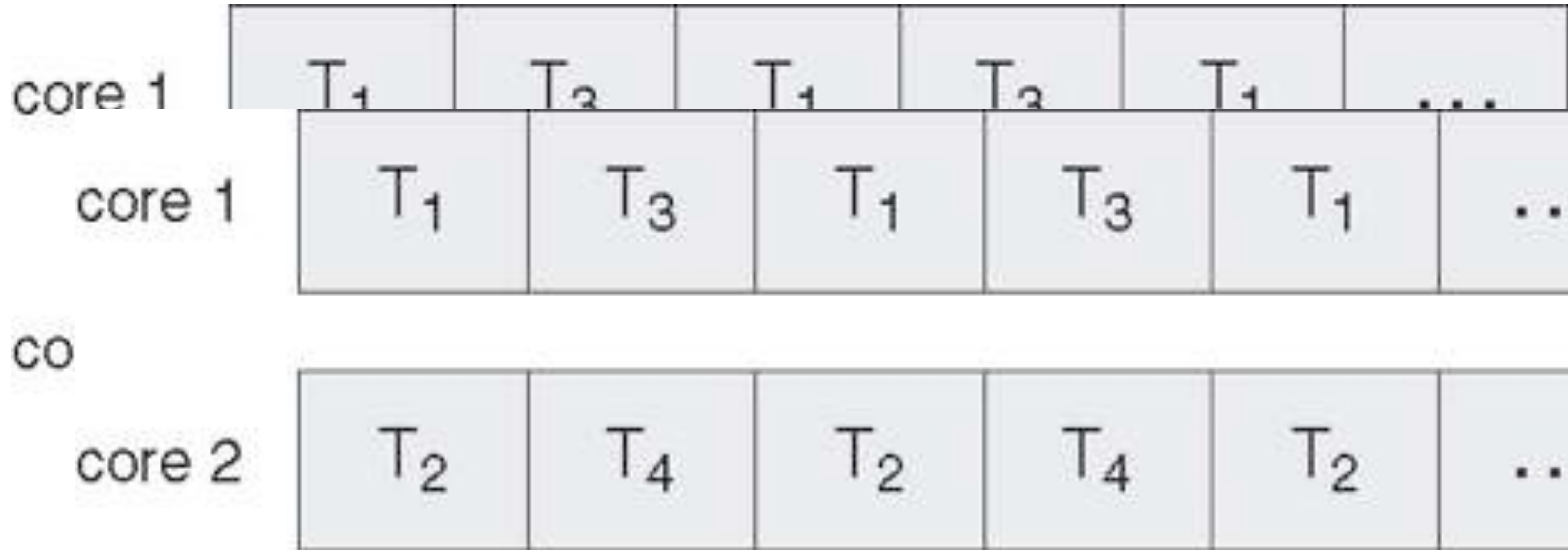
FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

- **Most OSs are now multithreaded**: several threads operate in the kernel managing devices and handling interrupts. For example, Linux uses a kernel thread for managing the amount of free memory in the system.

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

Concurrent Execution on a Single-core System

Parallel Execution on a Multicore System

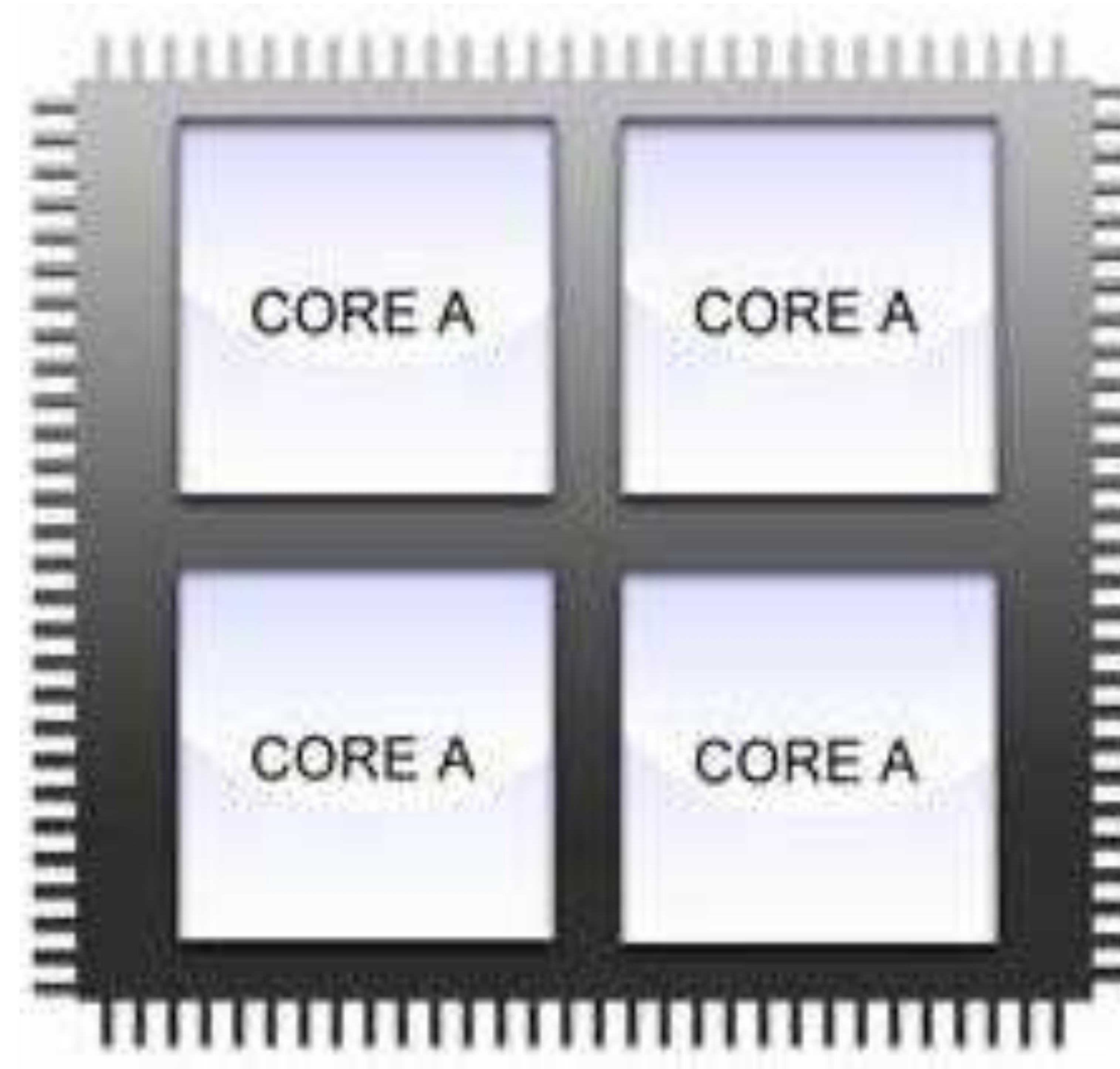


Multi-core Programming

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

Challenges:

- Dividing activities
- Balance
- Data splitting
- Data dependency
- Testing and debugging



The challenges of developing software for multi-core systems may require an entirely new approach to designing software systems.

User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

- Supported by the Kernel
- Examples:
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

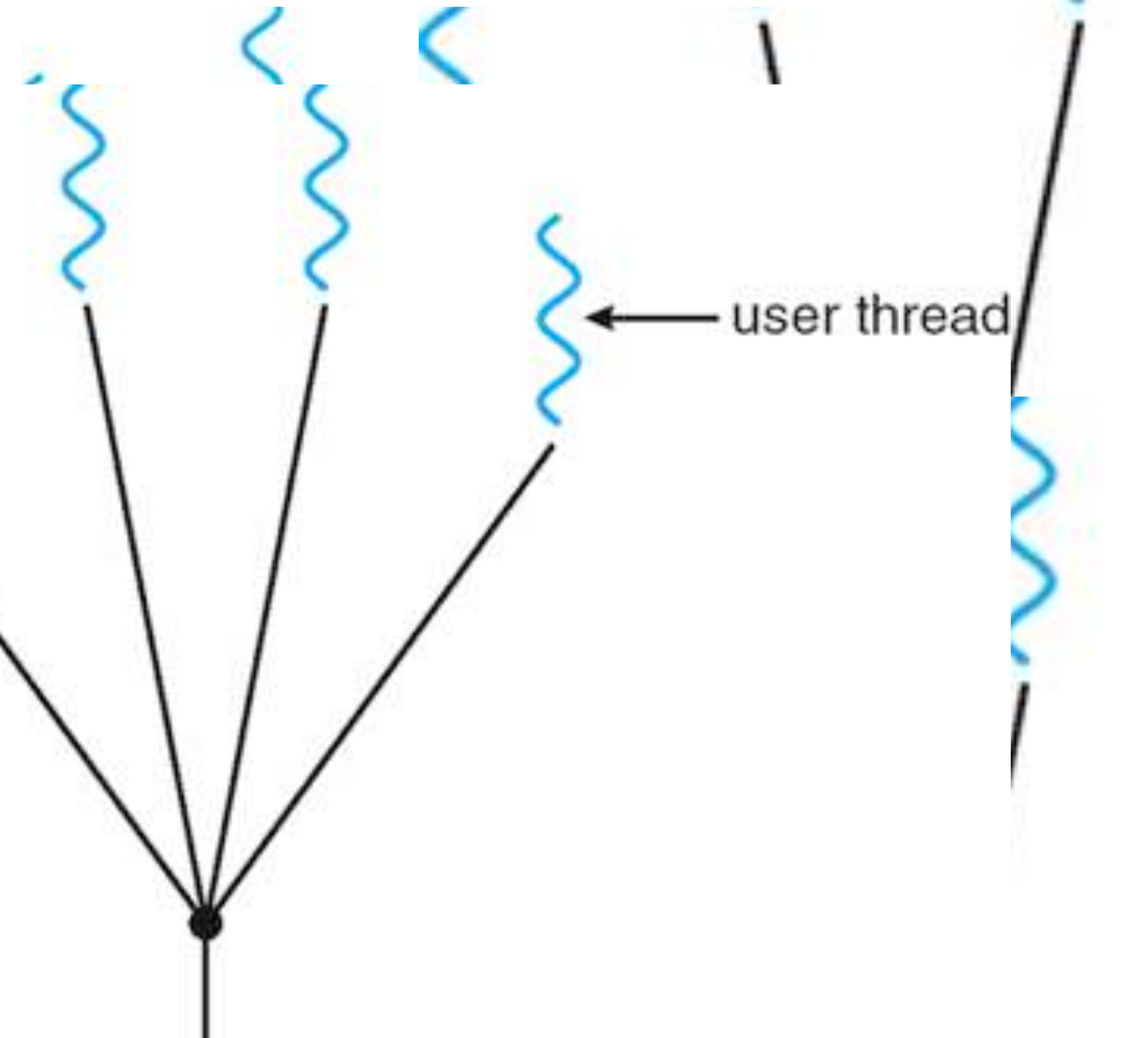
- Many-to-one
- One-to-one
- Many-to-many

Many-to-one model

Many user-level threads mapped to single kernel thread

Examples:

- Solaris Green Threads
- GNU Portable Threads



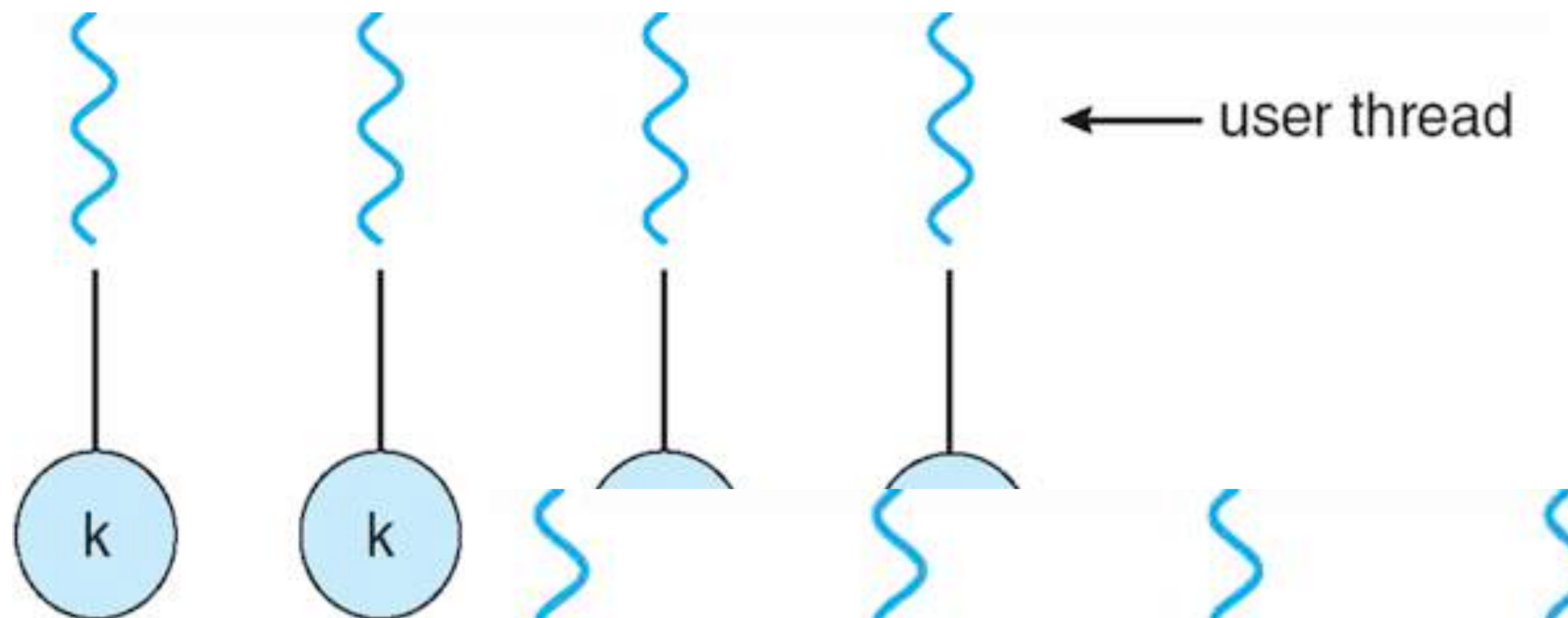
One-to-one model

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

Each user-level thread maps to kernel thread

Examples:

- Windows NT/XP/2000
- Linux
- Solaris 9 and later

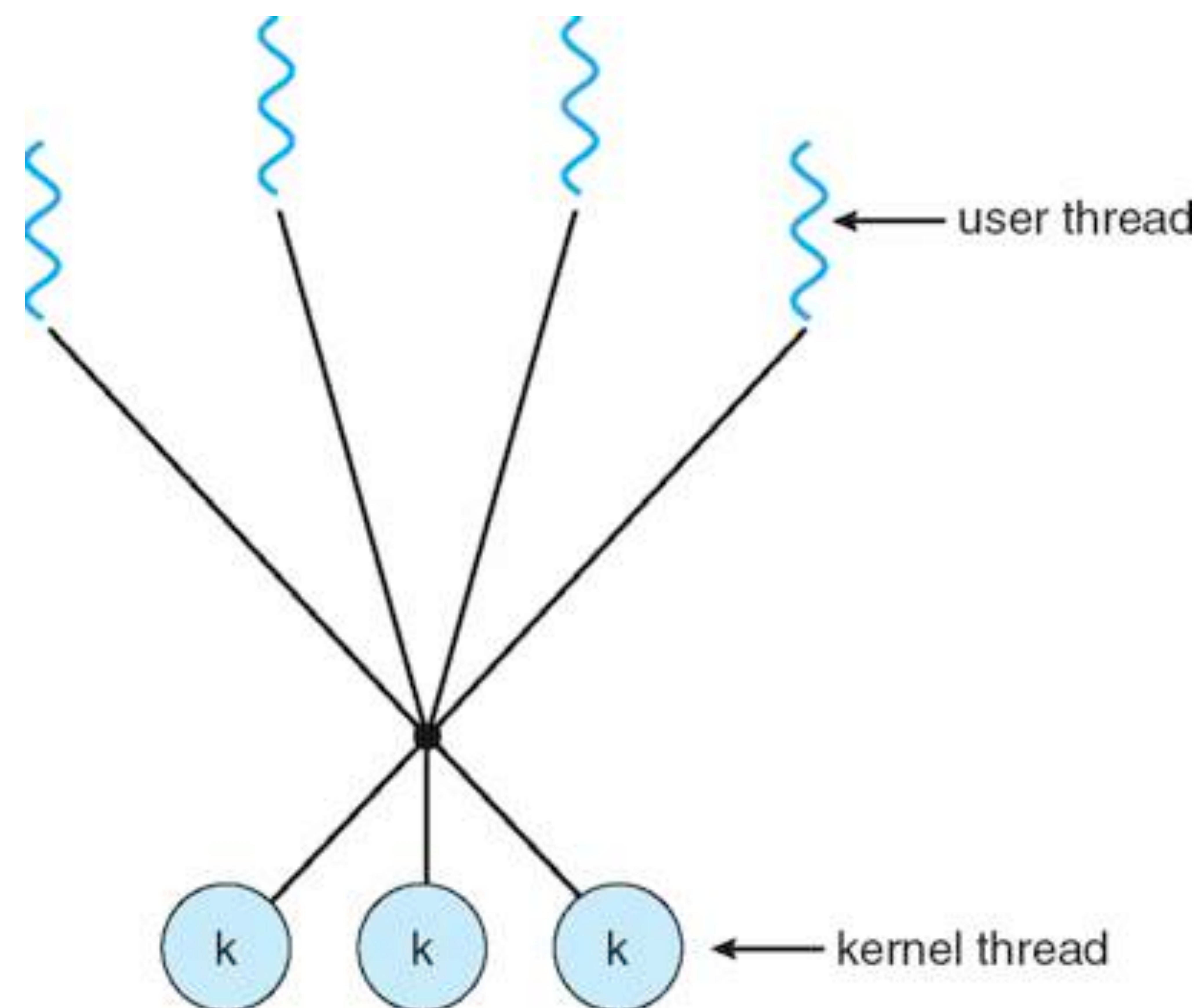


Many-to-many model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads

Examples:

- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



Thread Libraries: `pthreads`

Thread library provides programmer with API for creating and managing threads

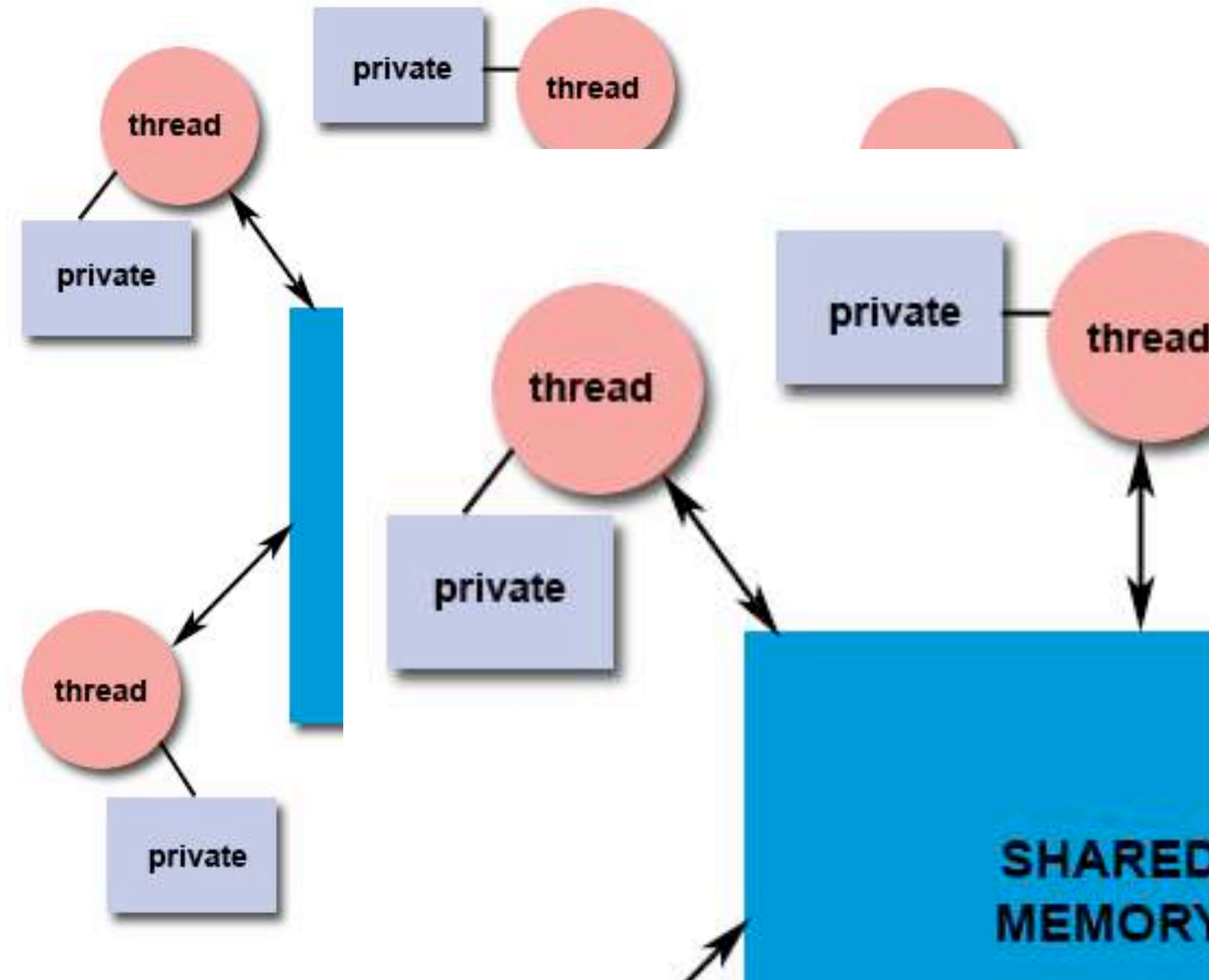
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to the developer
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Thread Programming

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

Shared Memory Model:

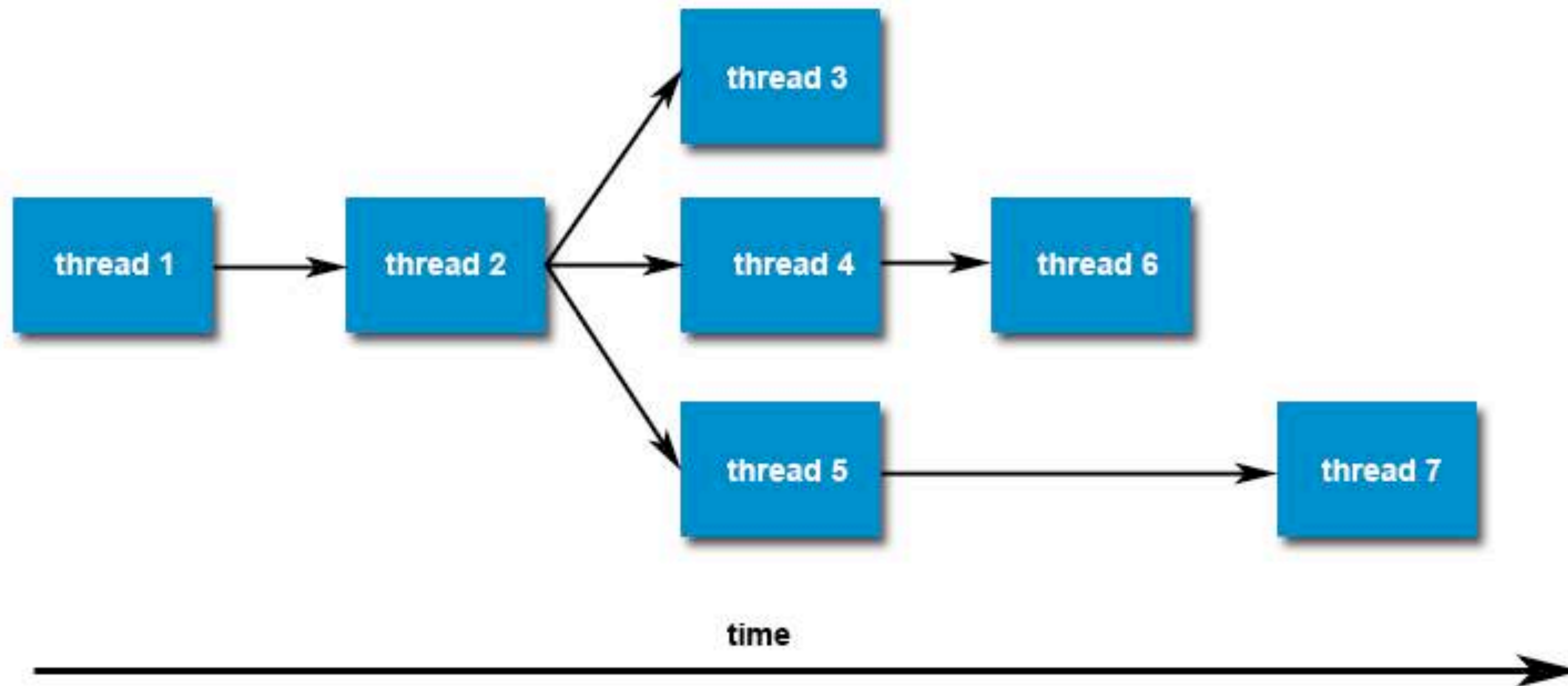
- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.



Creating and Terminating Threads

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

- The maximum number of threads that may be created by a process is implementation dependent.
- Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.



Creating and Terminating Threads

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid){
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}
```

Creating and Terminating Threads

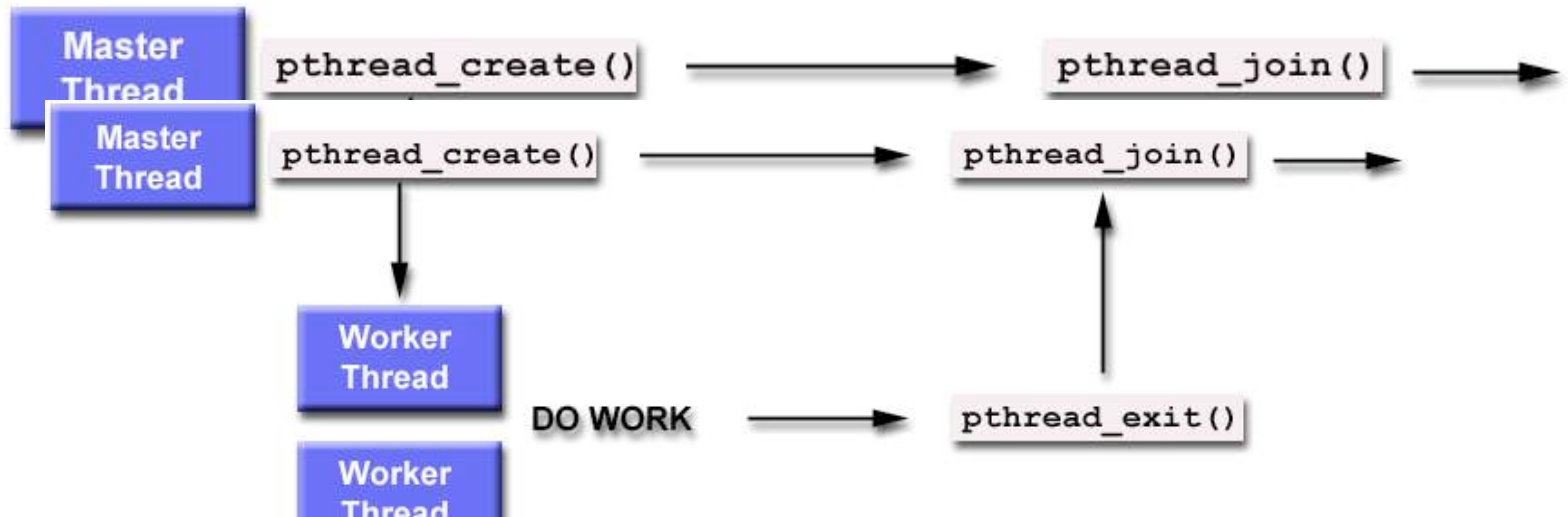
FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

```
int main (int argc, char *argv[]){
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create(): %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Thread Management - Joining and Detaching Threads

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

"Joining" is one way to accomplish synchronization between threads.



https://computing.llnl.gov/tutorials/pthreads/

Example – Joining and Detaching Threads

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

Worker Function

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4

void *BusyWork(void *t)
{
    int i;
    long tid;
    double result=0.0;
    tid = (long)t;
    printf("Thread %ld starting...\n",tid);
    for (i=0; i<1000000; i++)
        result = result + sin(i) * tan(i);

    printf("Thread %ld done. Result = %e\n",tid, result);
    pthread_exit((void*) t);
}
```

Main Function

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

```
int main (int argc, char *argv[]){
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc; long t; void *status;

  /* Initialize and set thread detached attribute */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  for(t=0; t<NUM_THREADS; t++) {
    printf("Main: creating thread %ld\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
    if (rc)  exit(-1);
  }

  /* Free attribute and wait for the other threads */
  pthread_attr_destroy(&attr);
  for(t=0; t<NUM_THREADS; t++) {
    rc = pthread_join(thread[t], &status);
    if (rc)  exit(-1);

    printf("Main: completed join with thread %ld having a status
          of %ld\n",t,(long)status);
  }

  printf("Main: program completed. Exiting.\n");
  pthread_exit(NULL);
}
```

Boost Libraries

Example 1

```
#include <iostream>
#include <boost/thread.hpp>
#include <boost/date_time.hpp>

void workerFunc()
{
    boost::posix_time::seconds workTime(3);
    std::cout << "Worker: running" << std::endl;

    // Pretend to do something useful...
    boost::this_thread::sleep(workTime);
    std::cout << "Worker: finished" << std::endl;
}

int main(int argc, char* argv[])
{
    std::cout << "main: startup" << std::endl;
    boost::thread workerThread(workerFunc);

    std::cout << "main: waiting for thread" << std::endl;
    workerThread.join();

    std::cout << "main: done" << std::endl;
    return 0;
}
```

```
main: startup
main: waiting for thread
Worker: running
Worker: finished
main: done
```

http://www.codeproject.com/Articles/279053/How-to-get-started-using-Boost-threads

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

Example 2

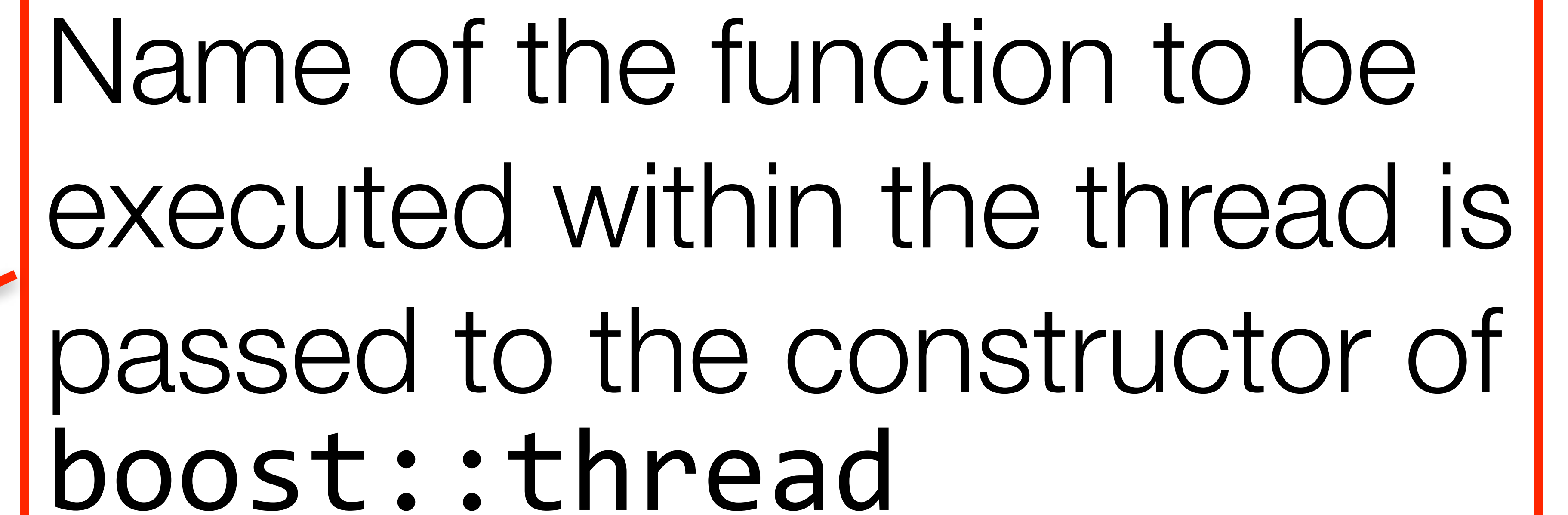
```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << std::endl;
  }
}

int main()
{
  boost::thread t(thread);
  t.join();
}
```

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << std::endl;
  }
}

int main()
{
  boost::thread t(thread);
  t.join();
}
```

This is the function we want to be executed within the thread.

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << std::endl;
  }
}

int main()
{
  boost::thread t(thread);
  t.join();
}
```



FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << std::endl;
  }
}

int main()
{
  boost::thread t(thread);
  t.join();
}
```

Upon creation, the thread function starts executing in its own thread *immediately*. Function **`main()`** is also executing in its own thread. Here, we say that these functions are executing *concurrently*.

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << std::endl;
  }
}

int main()
{
  boost::thread t(thread);
  t.join();
}
```

The method **`join()`** blocks the calling thread until thread t terminates. Basically, it forces **`main()`** to wait for **`t`**.

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

```cpp
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << std::endl;
  }
}

int main()
{
  boost::thread t(thread);
  t.join();
}
```

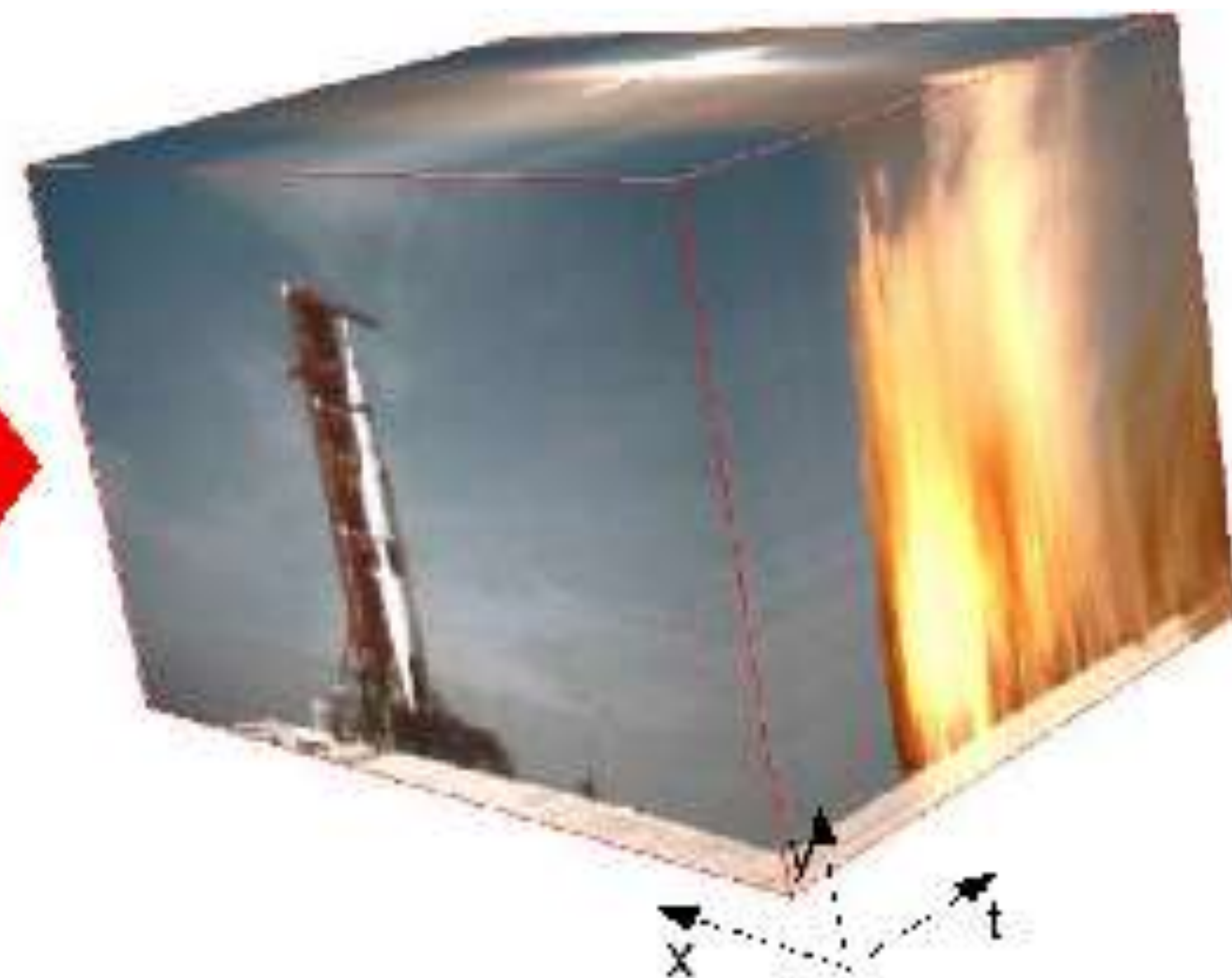
What happens if we don't call `join()`?

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

Example: Video processing

- Video as a 3-D array (volume)

Source: NASA



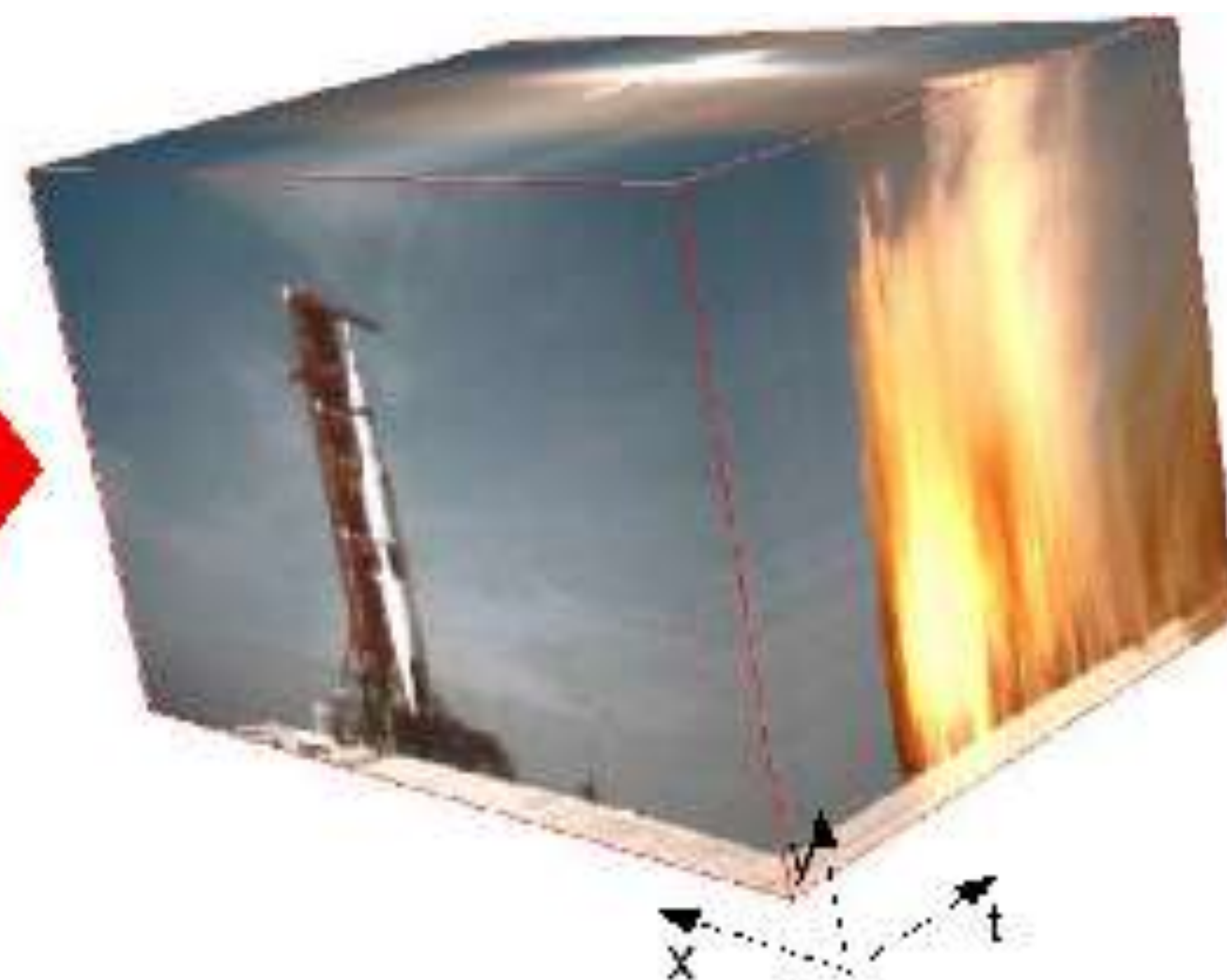
http://breckon.eu/toby/demos/videovolumes/

Example: Video processing

» Some tasks of a video-processing software can be done concurrently by separate threads.

- Calculate the average image
- Calculate the median image

Source: NASA



http://breckon.eu/toby/demos/videovolumes/

Example: Video processing

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

» We can also speed up each task by first dividing the video into sub volumes, and then assign each sub volume to be processed by a separate thread.

http://breckon.eu/toby/demos/videovolumes/

Synchronization

Synchronizing threads

- Multi-threaded programming can increase performance of applications. But, complexity is also increased.
- Access to shared resources must be controlled by trying to synchronize access.

```
void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    mutex.lock();
      std::cout << "Thread " << boost::this_thread::get_id() << ": " << i << std::endl;
    mutex.unlock();
  }
}

int main()
{
  boost::thread t1(thread);
  boost::thread t2(thread);
  t1.join();
  t2.join();
}
```

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

```
void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    mutex.lock();
      std::cout << "Thread " << boost::this_thread::get_id() << ": " << i << std::endl;
    mutex.unlock();
  }
}

int main()
{
  boost::thread t1(thread);
  boost::thread t2(thread);
  t1.join();
  t2.join();
}
```

Creates two threads, both execution the `thread()` function.

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

```
void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    mutex.lock();
      std::cout << "Thread " << boost::this_thread::get_id() << ": " << i << std::endl;
    mutex.unlock();
  }
}

int main()
{
  boost::thread t1(thread);
  boost::thread t2(thread);
  t1.join();
  t2.join();
}
```

The `thread()` function writes on the standard output stream (on the console). This stream is a *global object* shared by all threads.

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

```
void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    mutex.lock();
      std::cout << "Thread " << boost::this_thread::get_id() << ": " << i << std::endl;
    mutex.unlock();
  }
}

int main()
{
  boost::thread t1(thread);
  boost::thread t2(thread);
  t1.join();
  t2.join();
}
```

We need to synchronize access to this *shared resource* otherwise messages from multiple threads will overlap on the console.

```
void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    mutex.lock();
      std::cout << "Thread " << boost::this_thread::get_id() << ": " << i << std::endl;
    mutex.unlock();
  }
}

int main()
{
  boost::thread t1(thread);
  boost::thread t2(thread);
  t1.join();
  t2.join();
}
```

Here, we declare a global **mutex** (i.e., mutual-exclusion object)

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

```
void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    mutex.lock();
      std::cout << "Thread " << boost::this_thread::get_id() << ": " << i << std::endl;
    mutex.unlock();
  }
}

int main()
{
  boost::thread t1(thread);
  boost::thread t2(thread);
  t1.join();
  t2.join();
}
```

A `mutex` works like a "traffic semaphore" or lock.

Multiple threads will see it but only one thread can get hold of it. Once one thread locks the mutex, all other threads that "try it" will need to wait until the lock is released by the thread that was holding it.

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

Install the boost library (Ubuntu)

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

```
sudo apt-get install libboost-all-dev
```

To learn more about boost threads

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

» Tutorial:

http://theboostcpplibraries.com/boost.thread

OpenMP (Open Multi-Processing)

- OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or Fortran that provides support for parallel programming in shared-memory environments.

OpenMP (Open Multi-Processing)

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY

» OpenMP identifies **parallel regions** as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel.

OpenMP (Open Multi-Processing)

This program will print a message which will be getting executed by various threads.

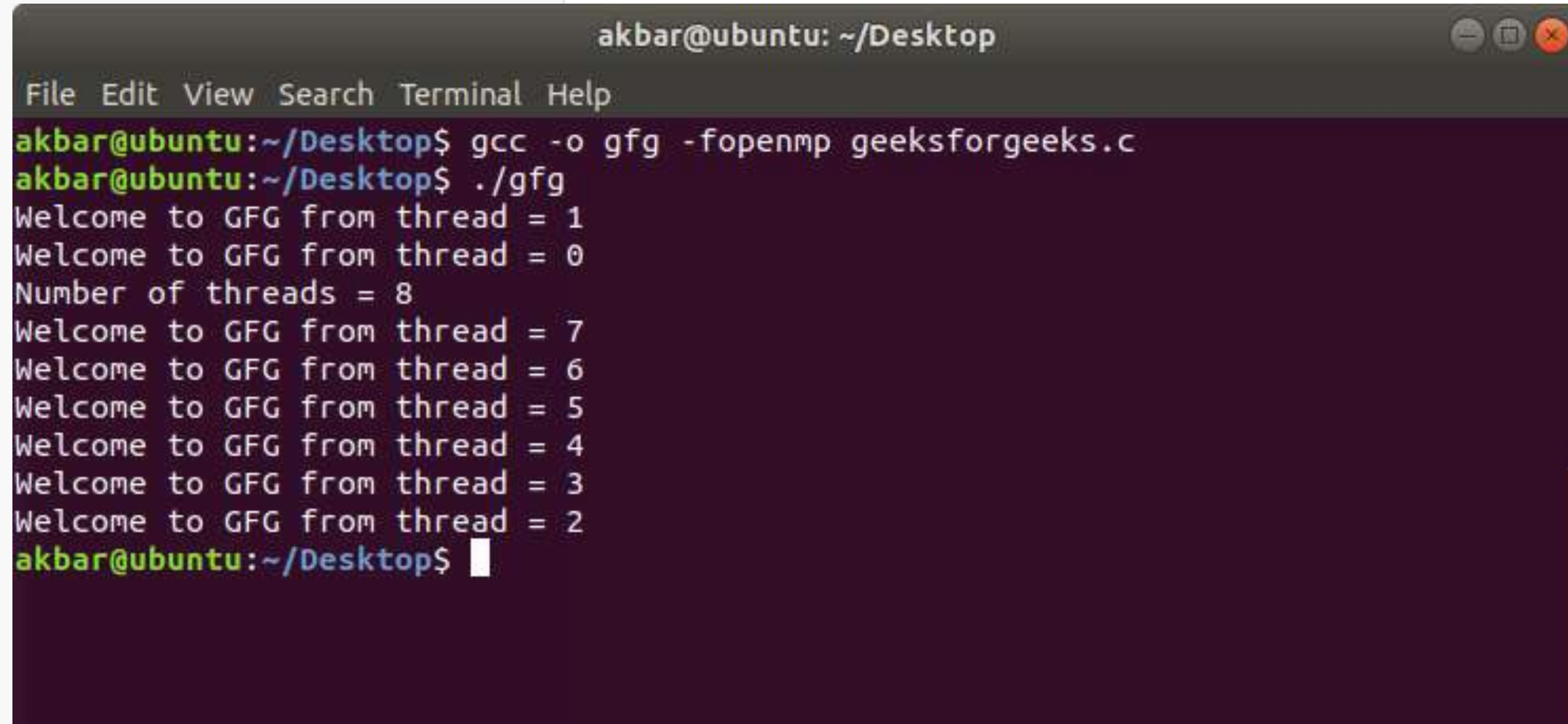
```c
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    int nthreads, tid;

    // Begin of parallel region
    #pragma omp parallel private(nthreads, tid)
    {
        // Getting thread number
        tid = omp_get_thread_num();
        printf("Welcome to GFG from thread = %d\n",tid);

        if (tid == 0) {
            // Only master thread does this
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
}
```

Compile:

`gcc -o gfg -fopenmp geeksforgeeks.c`

Execute:

`./gfg`



https://www.geeksforgeeks.org/openmp-introduction-with-installation-guide/

* To find out how many CPUs, type **lscpu** on the command line.

http://risewall.com/happy-halloween-new-wallpaper-for-android.html