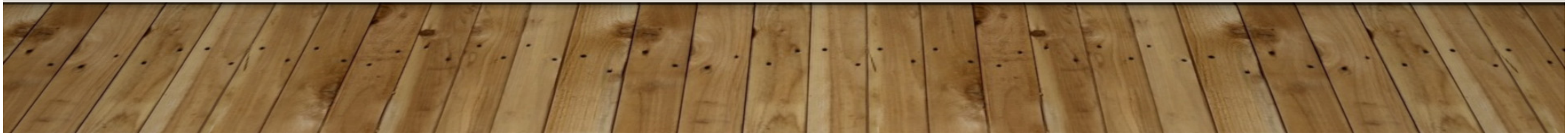
OPERATING SYSTEMS
CS3500

PROF. SUKHENDU DAS;

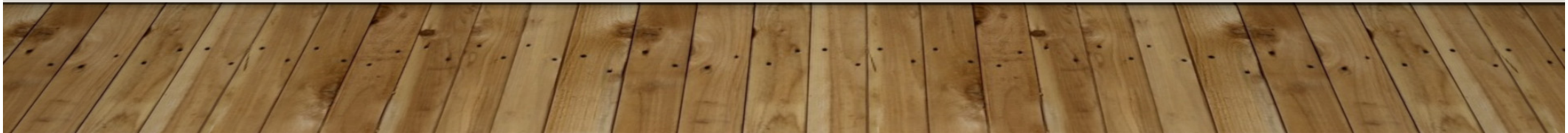
DEPTT. OF COMPUTER SCIENCE AND ENGG., IIT MADRAS, CHENNAI – 600036.

Email: sdas@cse.iitm.ac.in
URL: http://www.cse.iitm.ac.in/~vplab/os.html

Aug. – 2022.



THREADS & CONCURRENCY

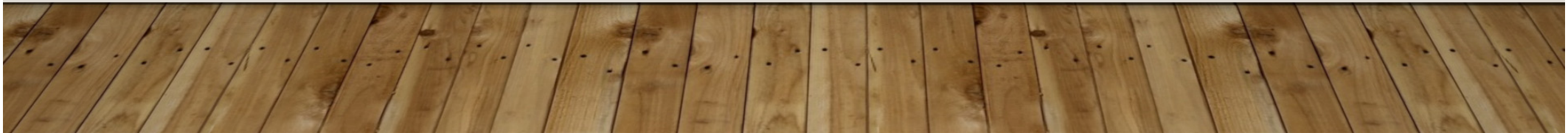


OUTLINE

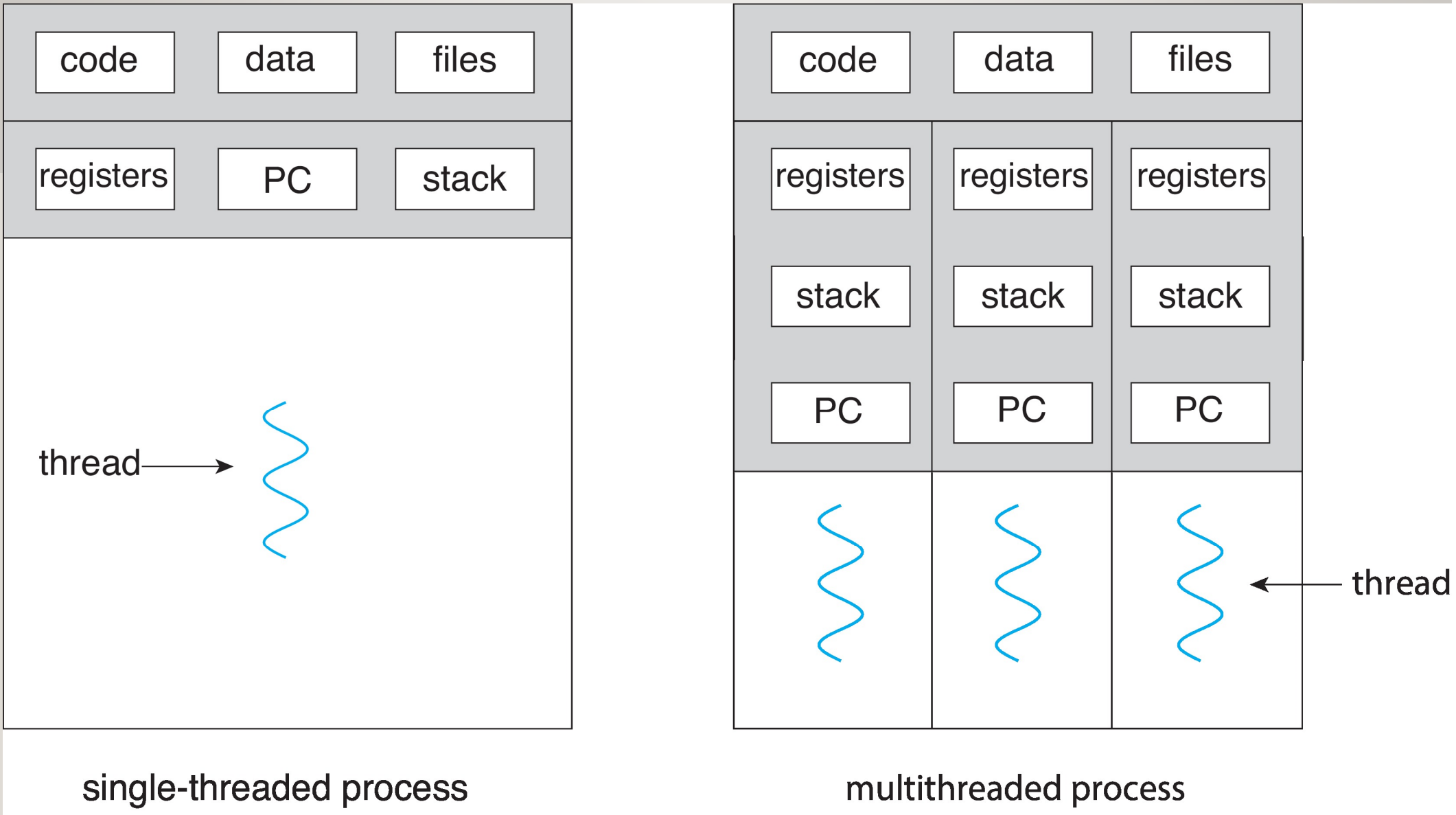
- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Threading Issues

MOTIVATION

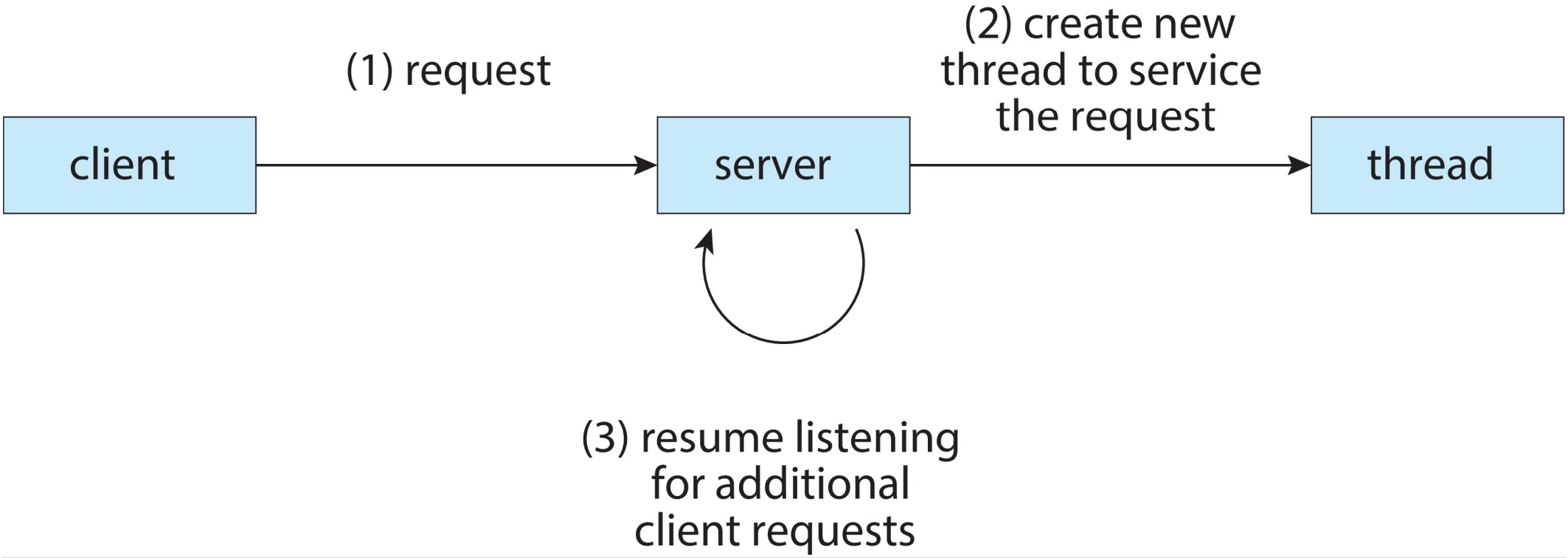
- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded



SINGLE AND MULTITHREADED PROCESSES

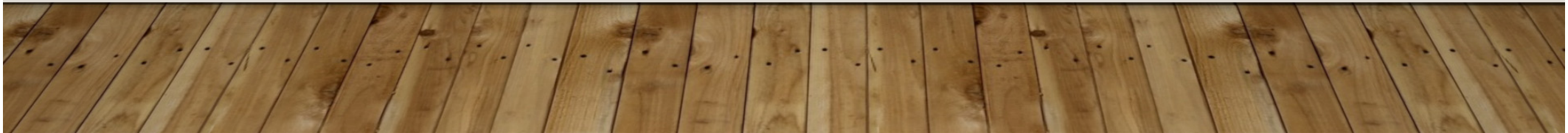


MULTITHREADED SERVER ARCHITECTURE



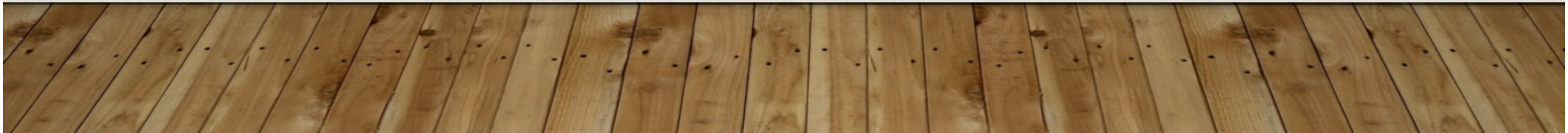
BENEFITS

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multicore architectures



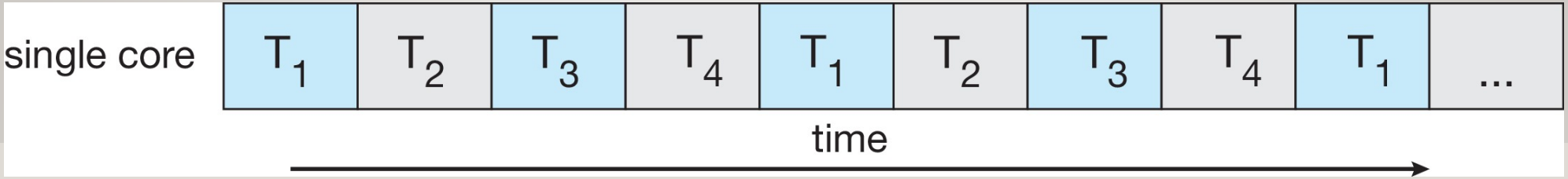
MULTICORE PROGRAMMING

- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

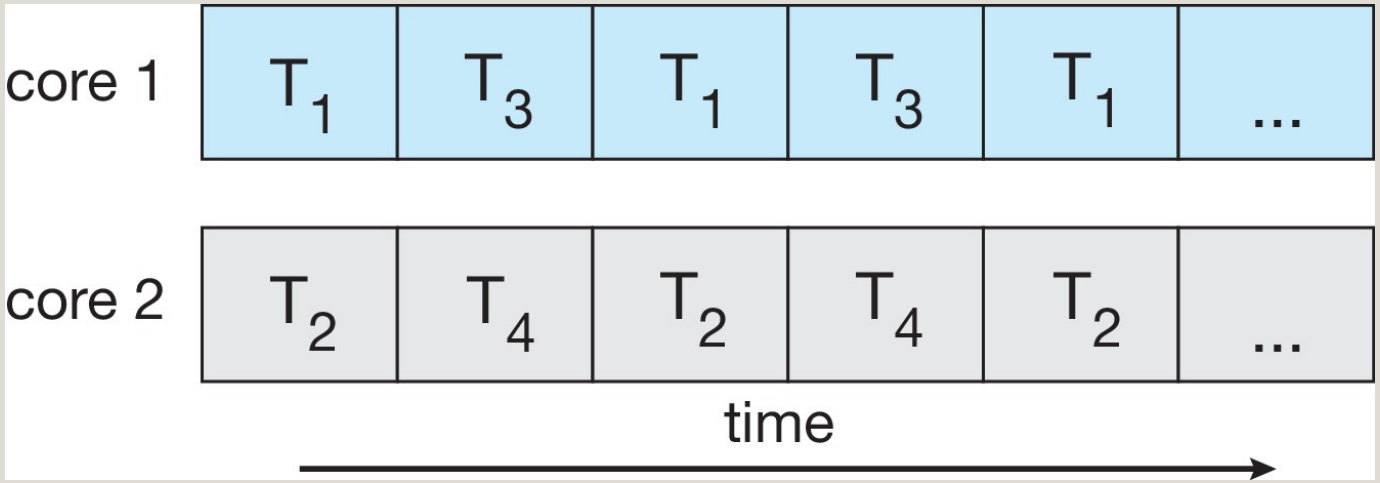


CONCURRENCY VS. PARALLELISM

- **Concurrent execution on single-core system:**

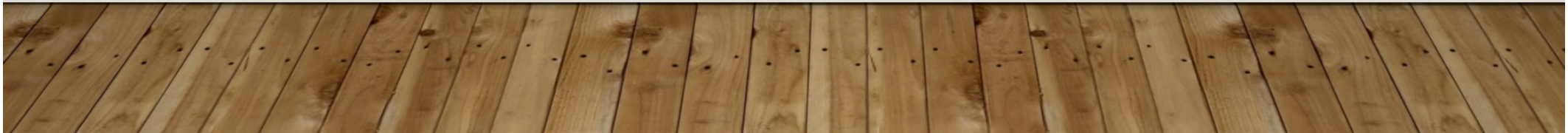


- **Parallelism on a multi-core system:**

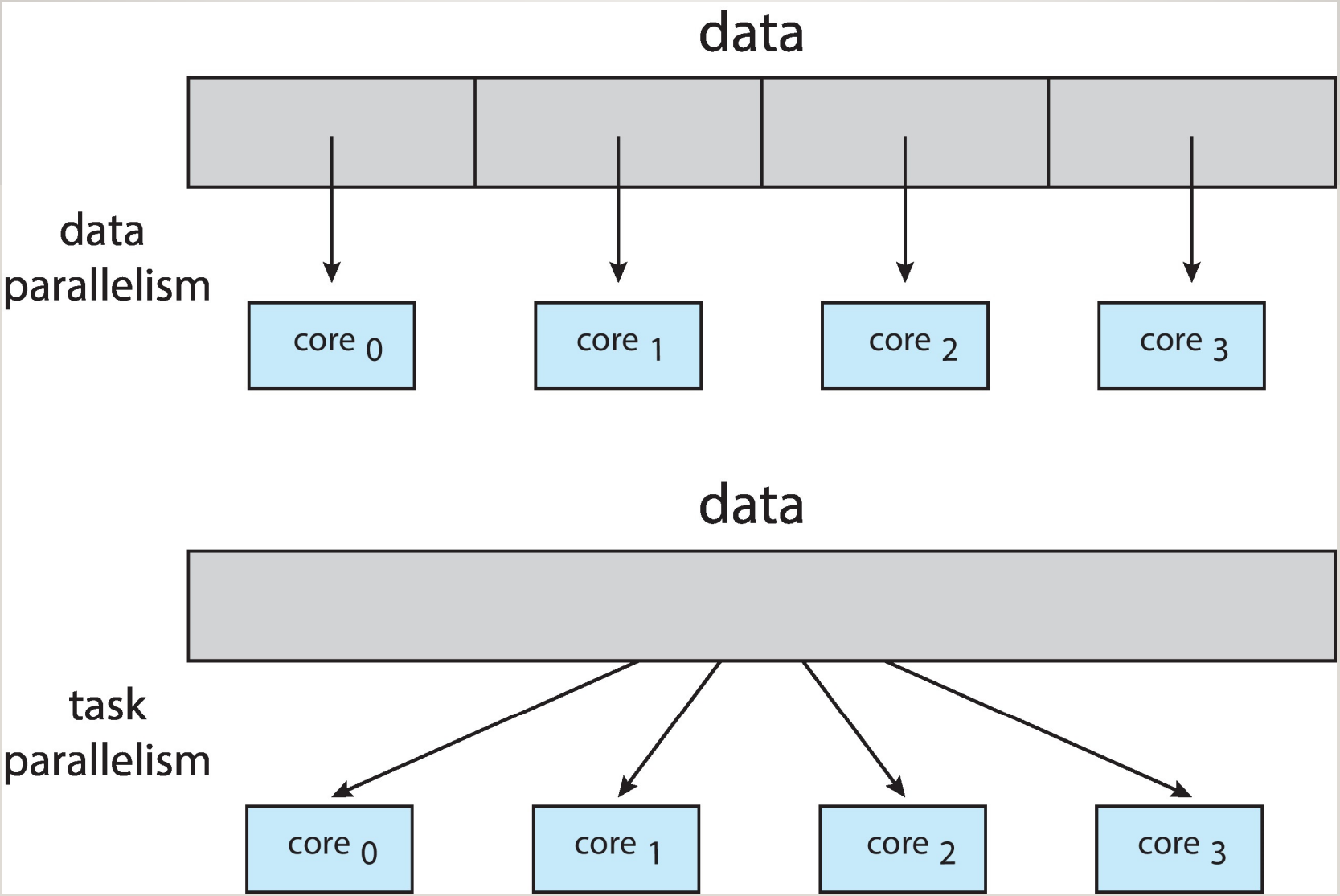


MULTICORE PROGRAMMING

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

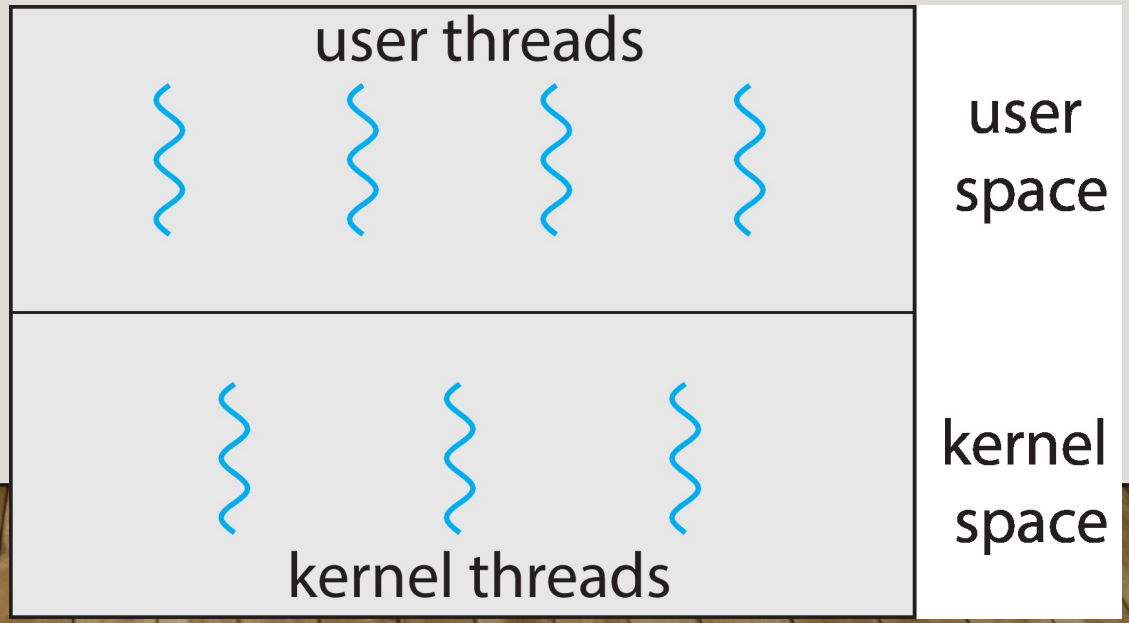


DATA AND TASK PARALLELISM



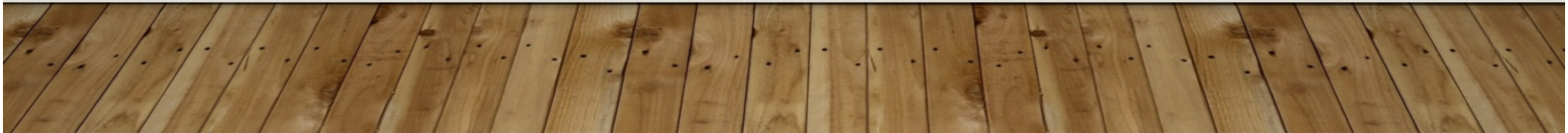
USER THREADS AND KERNEL THREADS

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android



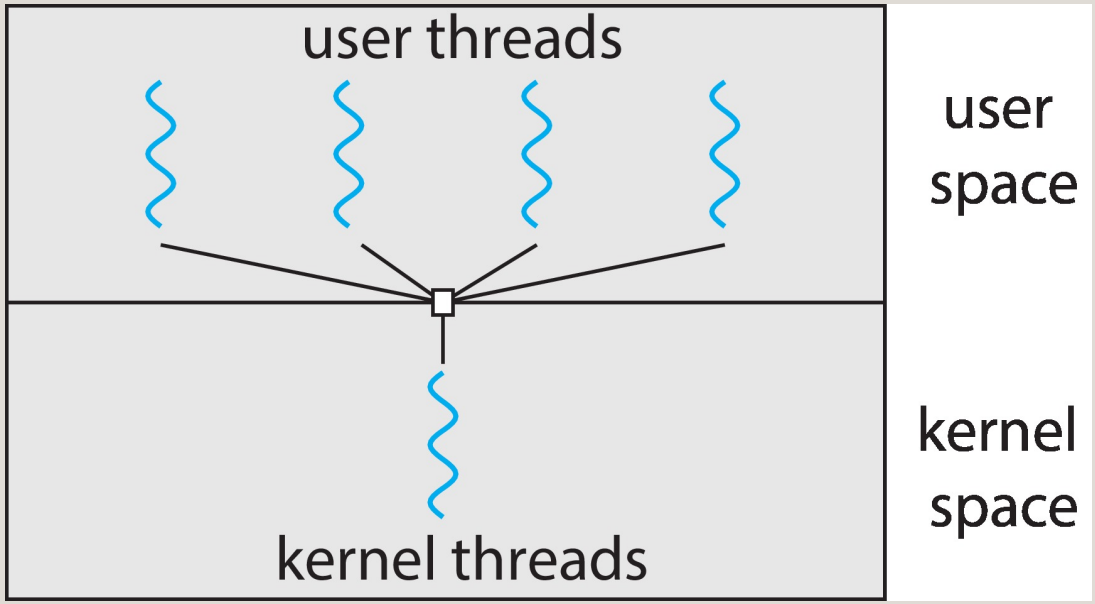
MULTITHREADING MODELS

- Many-to-One
- One-to-One
- Many-to-Many



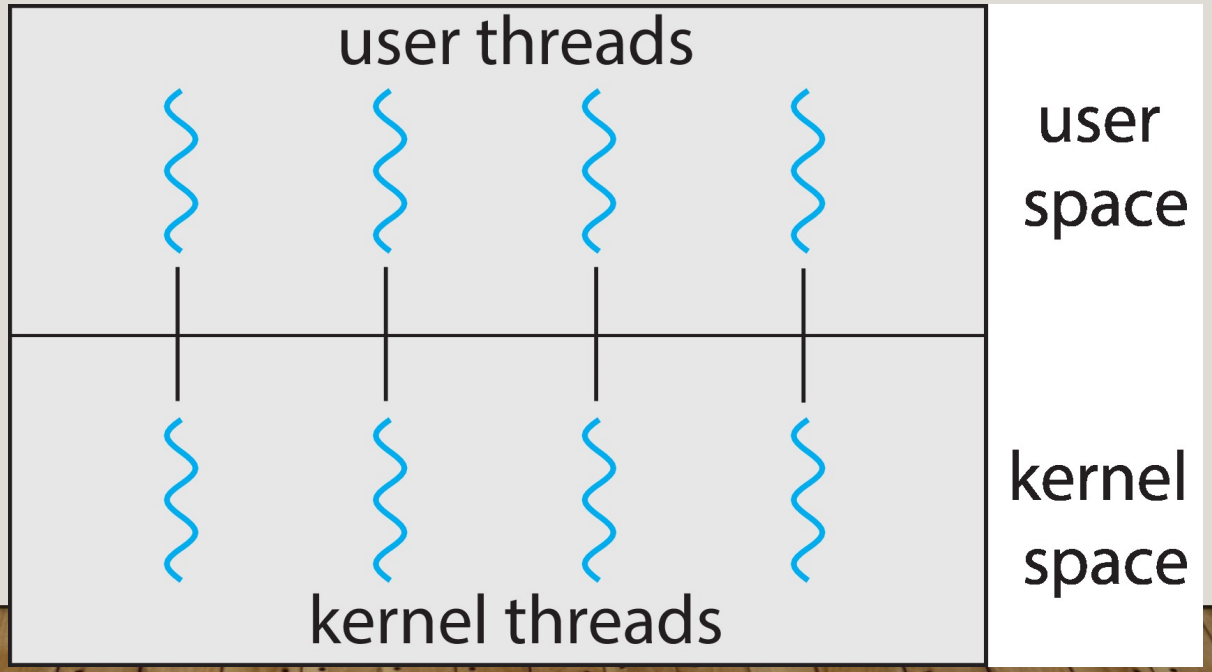
MANY-TO-ONE

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



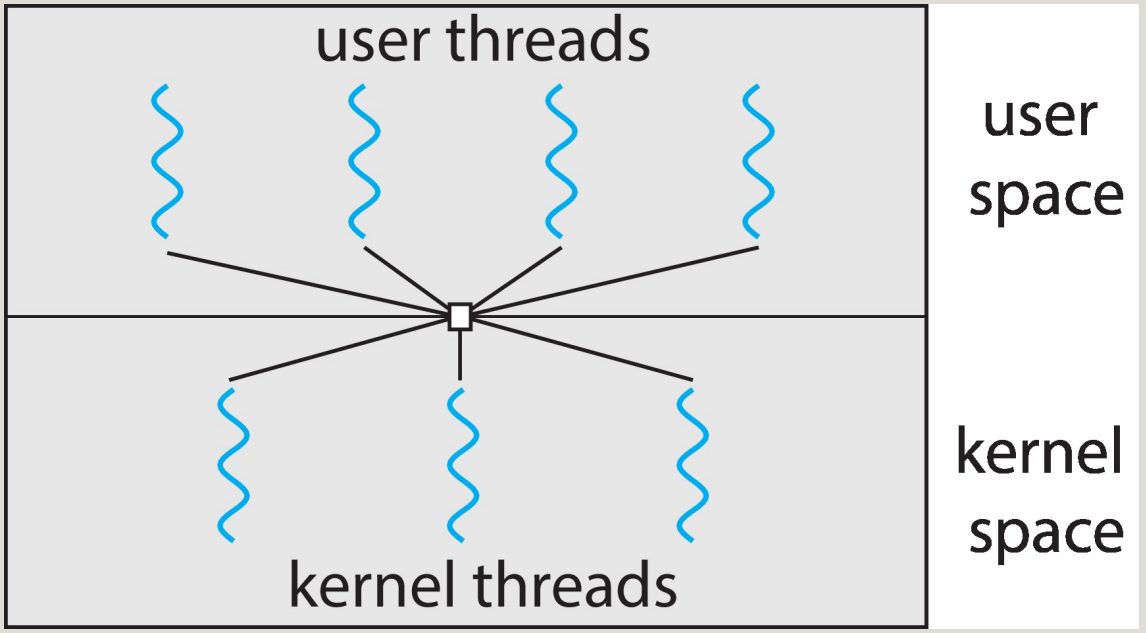
ONE-TO-ONE

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



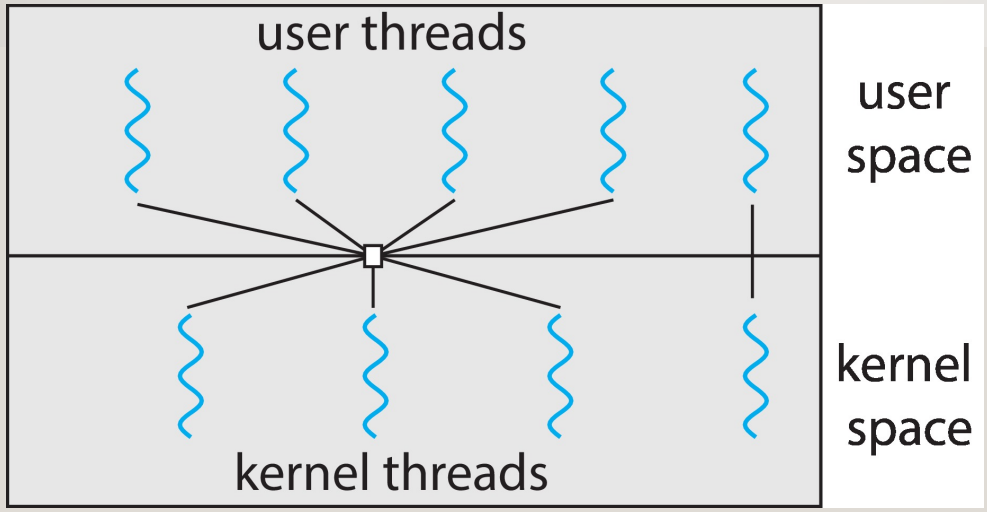
MANY-TO-MANY MODEL

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



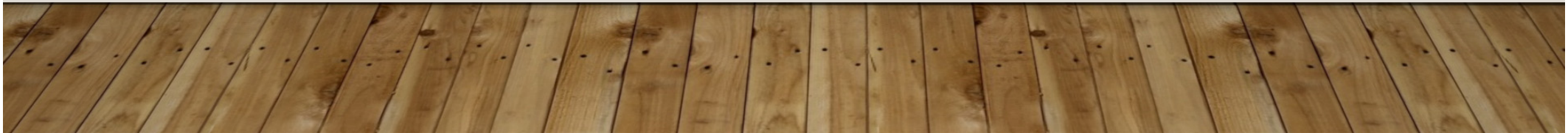
TWO-LEVEL MODEL

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



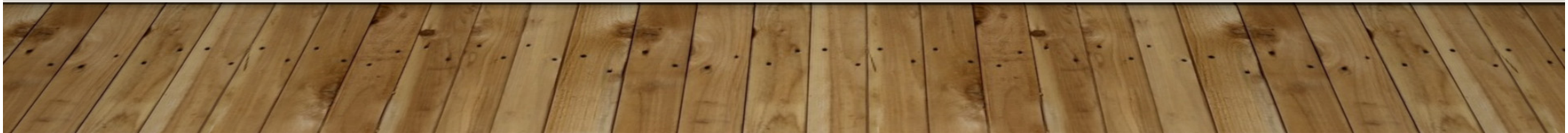
THREAD LIBRARIES

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS



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 & Mac 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);
}
```

PTHREADS EXAMPLE (CONT.)

```
/* 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);
}
```

- Try Pthreads Code for Joining 10 Threads!! (/** See fig. 4.12)

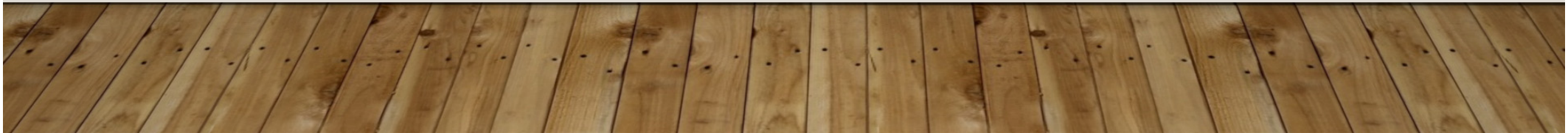
All Pthreads programs must include the *pthread.h* header file. The statement *pthread_t tid* declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The *pthread attr_t attr* declaration represents the attributes for the thread.

We set the attributes in the function call *pthread_attr_init(&attr*).

A separate thread is created with the *pthread_create()* function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution – the *runner* function (in this case).

This program follows the thread create/join strategy, whereby after creating the summation thread, the parent thread will wait for it to terminate by calling the function pthread_join() function.

The summation thread will terminate when it calls the function *pthread _exit().* Once the summation thread has returned, the parent thread will output the value of the shared data *sum*.



Implicit Threading

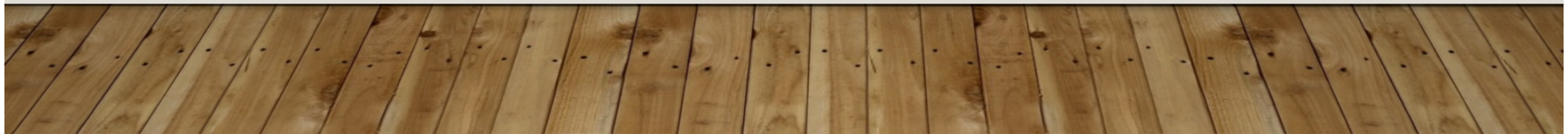
With the continued growth of multicore processing, applications containing hundreds—or even thousands—of threads are looming on the horizon.

A better support for the design of concurrent and parallel applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy is termed implicit threading, is an increasingly popular trend.

These strategies generally require application developers to identify tasks—not threads—that can run in parallel. A task is usually written as a function, which the run-time library then maps to a separate thread, typically using the many-to-many model.

The general idea behind a **thread pool** is to create a number of threads at start-up and place them into a pool, where they sit and wait for work.

The strategy for thread creation as the **fork-join** model - the main parent thread creates (forks) one or more child threads and then waits for the children to terminate and join with it, at which point it can retrieve and combine their results.



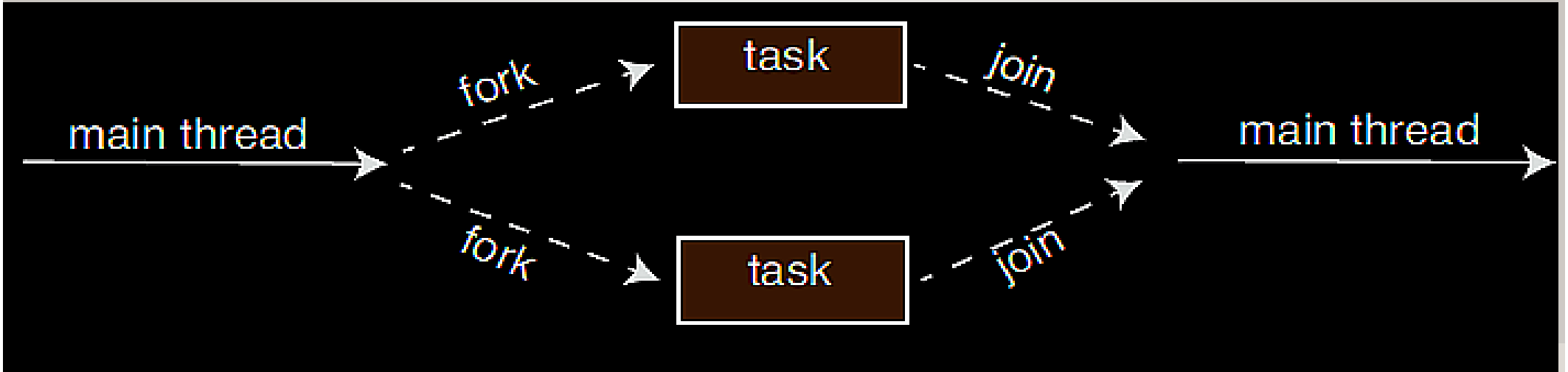


Figure 4.16 Fork-join parallelism.

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 identifies parallel regions as blocks of code that may run in parallel.

Like OpenMP, GCD (Grand Central Dispatch - a technology developed by Apple for its macOS and iOS) manages most of the details of threading. GCD schedules tasks for run-time execution by placing them on a *dispatch queue.* GCD identifies two types of dispatch queues: *serial and concurrent*.

Tasks placed on a serial queue are removed in FIFO order. Once a task has been removed from the queue, it must complete execution before another task is removed. Each process has its own serial queue (known as its main queue), serial Qs are called ***private dispatch queues.***

THREADING ISSUES

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

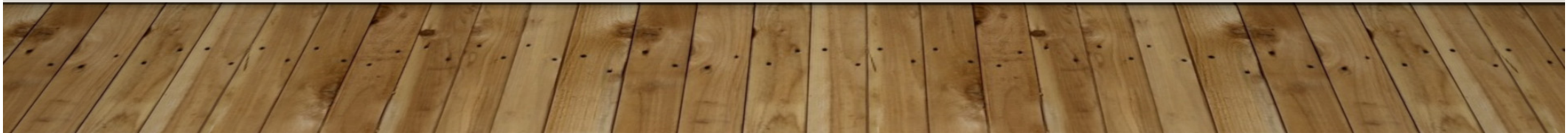
SEMANTICS OF FORK() AND EXEC()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- `exec()` usually works as normal – replace the running process including all threads

If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process.

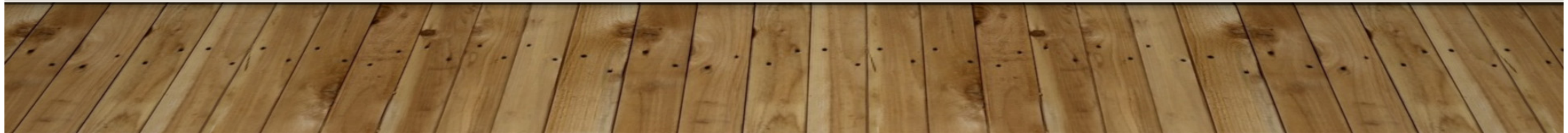
In this instance, duplicating only the calling thread is appropriate.

If, however, the separate process does not call exec() after forking, the separate process should duplicate all threads.



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:
        1. default
        2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
    - **User-defined signal handler** can override default
    - For single-threaded, signal delivered to process



SIGNAL HANDLING (CONT.)

- 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

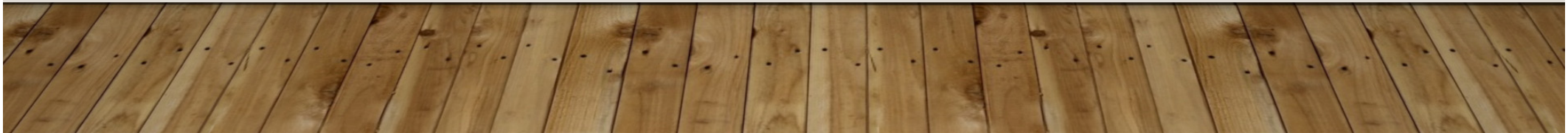
Synchronous signals are delivered to the same process that performed the operation that caused the signal;

When a signal is generated by an event external to a running process, that process receives the signal asynchronously.

kill(pid_t pid, int signal); Signals may be blocked by a process/thread.

Pthreads provides the following function, which allows a signal to be delivered to a specified thread (tid):

pthread_kill(pthread_t tid, int signal)



THREAD CANCELLATION

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

Difficulty with cancellation occurs in situations where resources have been allocated to a cancelled thread or where a thread is cancelled while in the midst of updating data it is sharing with other threads.

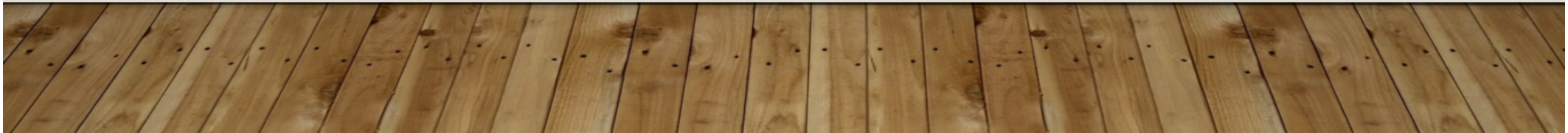
```
pthread_t tid;

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

  . . .

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

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

THREAD CANCELLATION (CONT.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

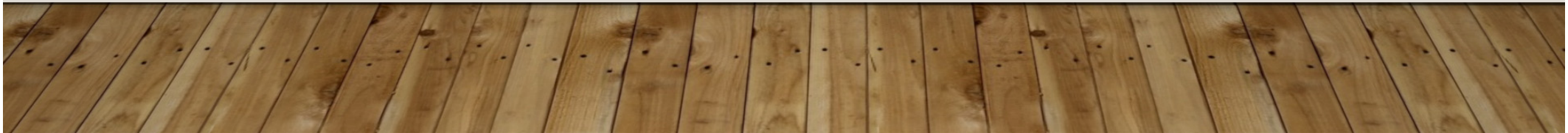
Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- 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



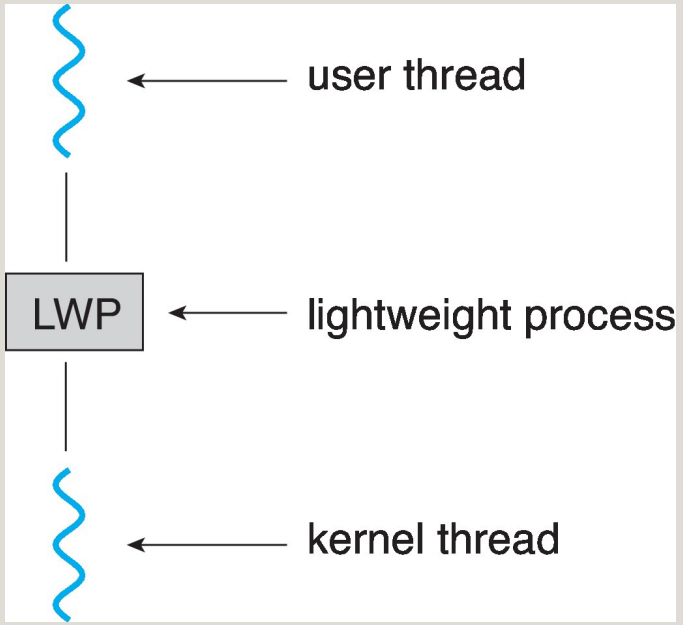
THREAD-LOCAL STORAGE

- **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)
- 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



PCB VS TCB

PCB vs TCB

S.No	PCB	TCB
1	Process Control Block	Thread Control Block
2	The PCB stores information about the kernel process. A process can include different kernel threads	The TCB includes thread specific information.
3	Some notable fields that the PCB could contain are the process id, process group id, the parent process and child processes, the heap pointer, program counter, scheduling state (running, ready, blocked), permissions (what system resources the process is allowed to access), content of the general purpose registers, and open files.	TCB has a few of the same fields as the PCB (register values, stack pointer, program counter, scheduling state), in addition to a few specific values like the **thread id and a pointer to the process that contains that thread**.
4	PCB describes an environment context (eg. memory segments and permissions)	TCB describes an execution context, (eg. stack pointer)