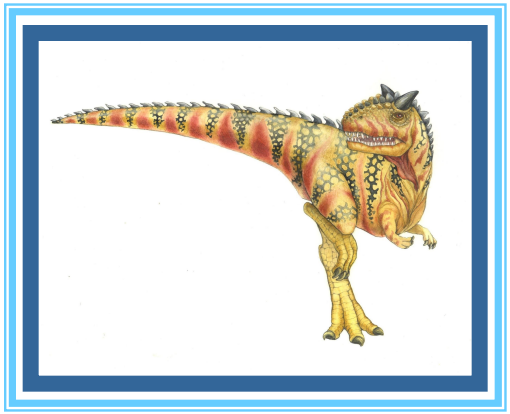
Chapter 4: Threads & Concurrency



Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

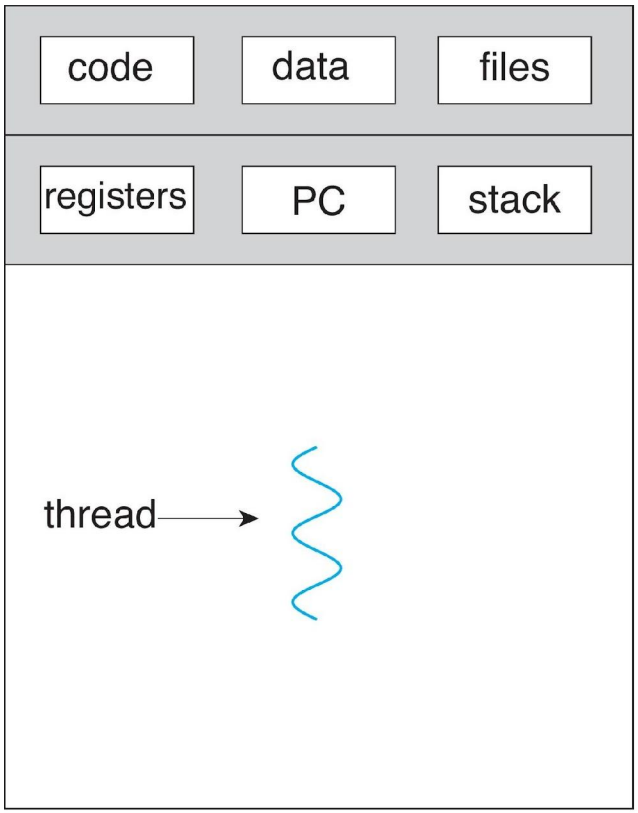
Objectives

- **Identify** the basic components of a thread, and **contrast** threads and processes
- Describe the **benefits and challenges** of designing multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux operating systems represent threads
- Designing multithreaded applications using the **Pthreads**, Java, and Windows threading APIs

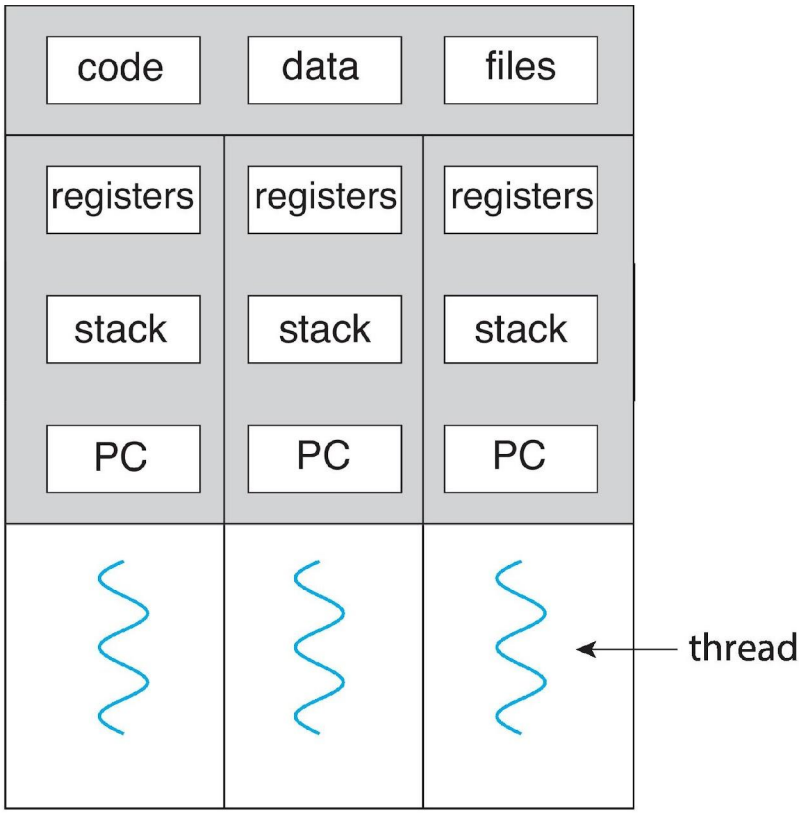
Motivation

- 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

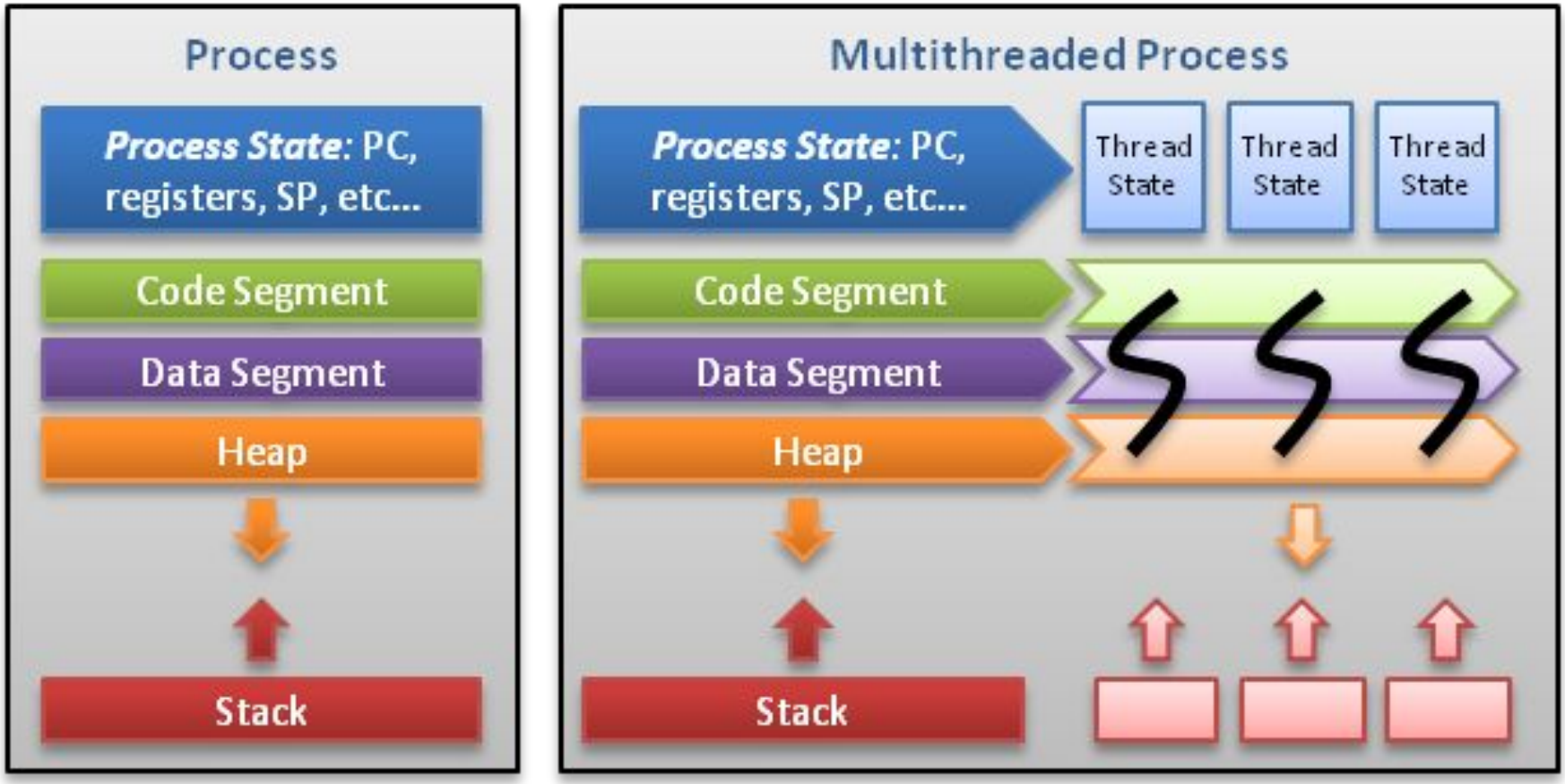


single-threaded process



multithreaded process

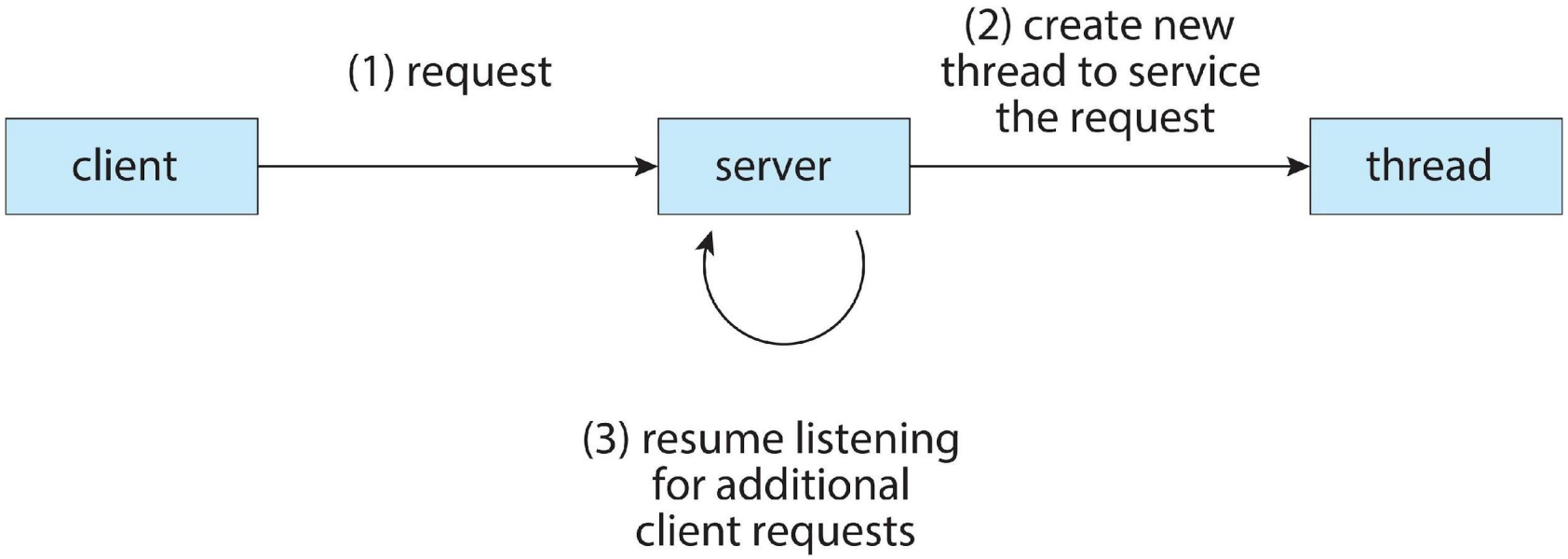
Single and Multithreaded Processes



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

https://randu.org/tutorials/threads/

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

Benefits

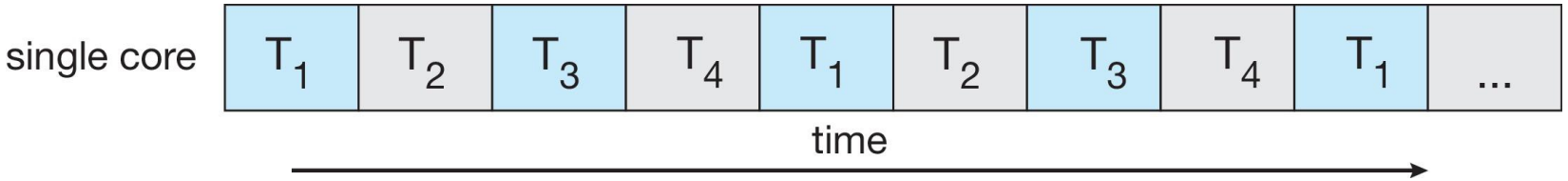
Aspect	Multiple Processes	Multiple Threads
Definition	Independent instances of a program	Lightweight units of execution within a process
Memory Usage	High (separate memory space for each process)	Low (shared memory space within a process)
Creation Overhead	High (OS resources required)	Low (less resource-intensive)
Isolation	Strong (processes are isolated)	Weak (threads share memory, risk of conflicts)
Communication	Complex (IPC mechanisms like pipes, and sockets)	Simple (shared memory, but requires synchronization)
Fault Tolerance	High (crash in one process doesn't affect others)	Low (a crash in one thread can crash the entire process)
Scalability	Limited by system resources (CPU, memory)	Limited by shared resources (e.g., GIL in Python)
Parallelism	True parallelism (on multi-core systems)	Limited parallelism (depends on language/runtime)
Synchronization	Not needed (independent memory)	Required (shared memory, risk of race conditions)
Use Case	CPU-bound tasks, independent workloads	I/O-bound tasks, tasks requiring shared data
Examples	Running separate programs, distributed systems	Web servers, GUI applications, async tasks

Multicore Programming

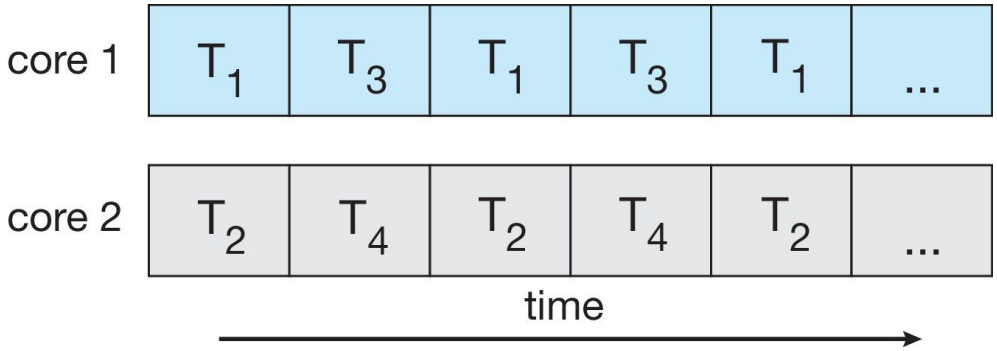
- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
-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

Concurrency vs. Parallelism

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



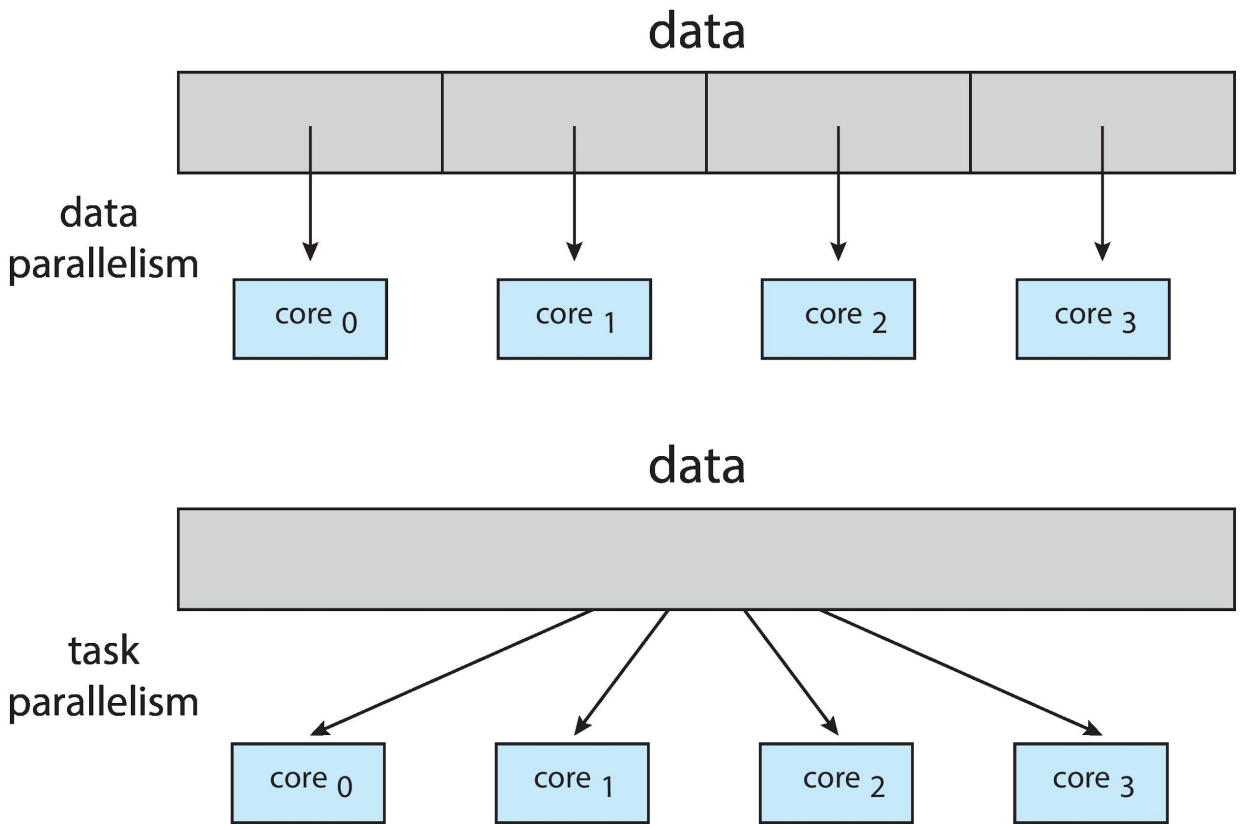
- **Parallelism on a multi-core system:**



Multicore Programming

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

Data and Task Parallelism



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

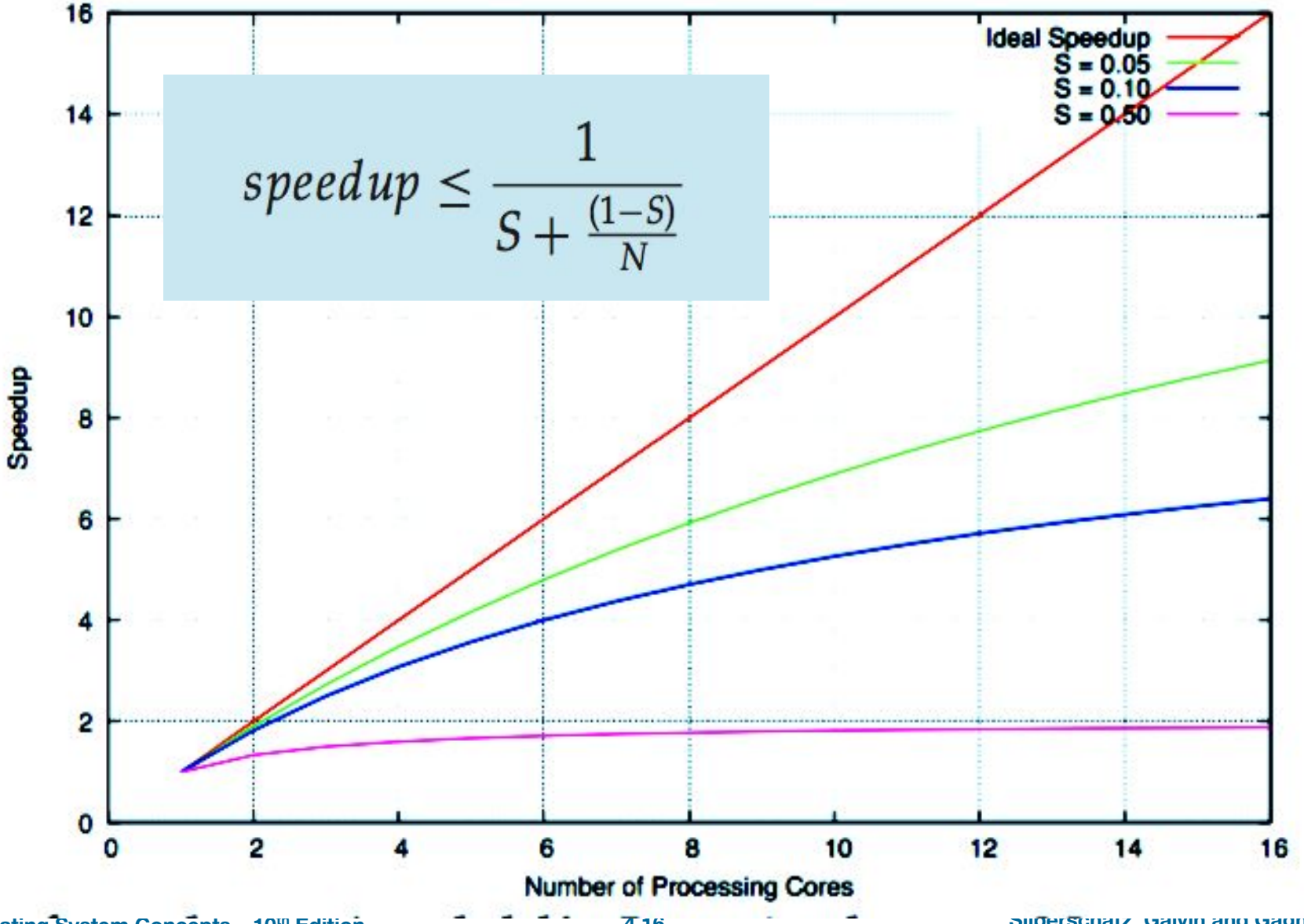
- But does the law take into account contemporary multicore systems?

Amdahl's Law (Scenarios)

1. **Parallelizing a Program:**
   - **Scenario:** A program spends 80% of its time in parallelizable code and 20% in serial code.
   - **Amdahl's Law:** Even with infinite parallel resources, speedup is limited to $\frac{1}{0.2} = 5x$. Serial portion caps performance.
2. **Database Query Optimization:**
   - **Scenario:** 90% of a query can be parallelized, but 10% remains sequential (e.g., final aggregation).
   - **Amdahl's Law:** Maximum speedup is $\frac{1}{0.1} = 10x$. No matter how many cores, 10x is the limit.
3. **Video Rendering:**
   - **Scenario:** 95% of rendering is parallel, but 5% is sequential (e.g., final frame composition).
   - **Amdahl's Law:** Speedup cannot exceed $\frac{1}{0.05} = 20x$. Serial bottleneck limits gains.

Amdahl's Law

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$



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

Performance Implications: User-level threads are managed entirely by the application and don't require intervention from the kernel. This can make thread creation, synchronization, and context switching faster compared to kernel-level threads, which involve system calls and kernel intervention. Understanding this distinction can help developers make informed decisions about which threading model to use based on performance requirements.

Concurrency Control: Kernel-level threads are managed by the operating system kernel, which provides better support for concurrency control and resource management. Understanding this distinction is crucial for designing applications that require precise control over concurrency and resource usage, such as server applications or real-time systems.

Thread Synchronization: Kernel-level threads typically offer better support for synchronization primitives such as mutexes, semaphores, and condition variables. Understanding this distinction helps developers choose the appropriate synchronization mechanisms for their application's requirements.

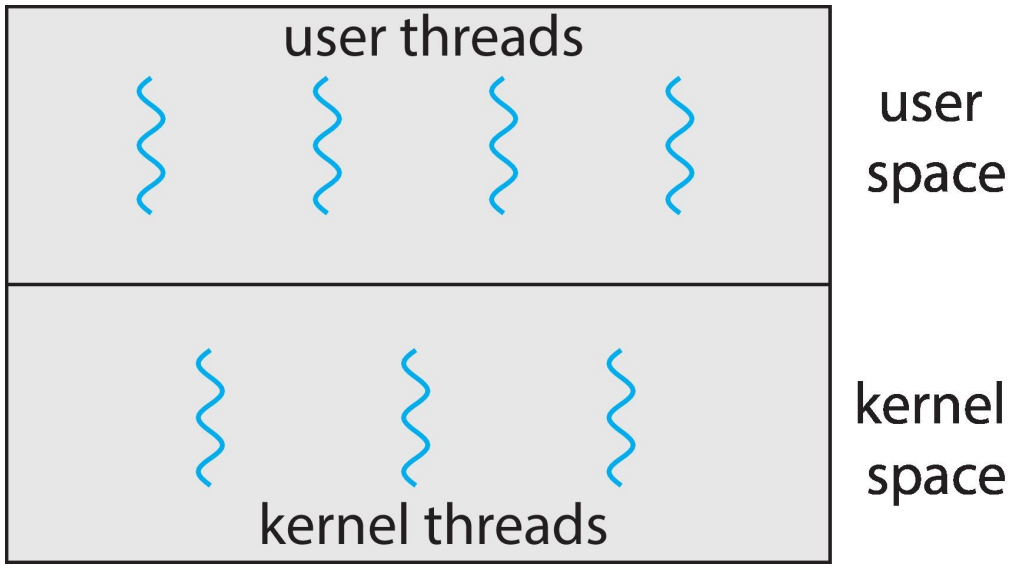
Portability: Different operating systems may have different implementations of user-level and kernel-level threading models. Understanding the differences between these models helps developers write portable code that can run efficiently across different platforms.

Resource Utilization: Kernel-level threads are generally heavier in terms of resource usage compared to user-level threads, as they involve kernel resources such as kernel stacks and scheduling structures. Understanding this distinction is important for designing scalable applications that can handle large numbers of threads efficiently.

User Threads and Kernel Threads

Aspect	User Threads	Kernel Threads
Management	Managed by user-level libraries (e.g., Pthreads)	Managed directly by the operating system
Creation Overhead	Low (no kernel involvement)	High (system calls and kernel resources)
Scheduling	Scheduled by user-level thread scheduler	Scheduled by OS scheduler
Blocking Behavior	One blocked thread blocks all threads in the process	One blocked thread doesn't affect others
Parallelism	Limited (runs as a single process in the kernel)	True parallelism (can utilize multiple cores)
Portability	High (OS-independent)	Low (OS-dependent)
Complexity	Easier to implement and manage	More complex (requires kernel support)
Performance	Faster context switching (no kernel mode)	Slower context switching (kernel mode)
Fault Tolerance	Low (crash in one thread affects all threads)	High (kernel isolates threads)
Use Case	Lightweight tasks, high-concurrency apps	CPU-bound tasks, real-time systems

User and Kernel Threads



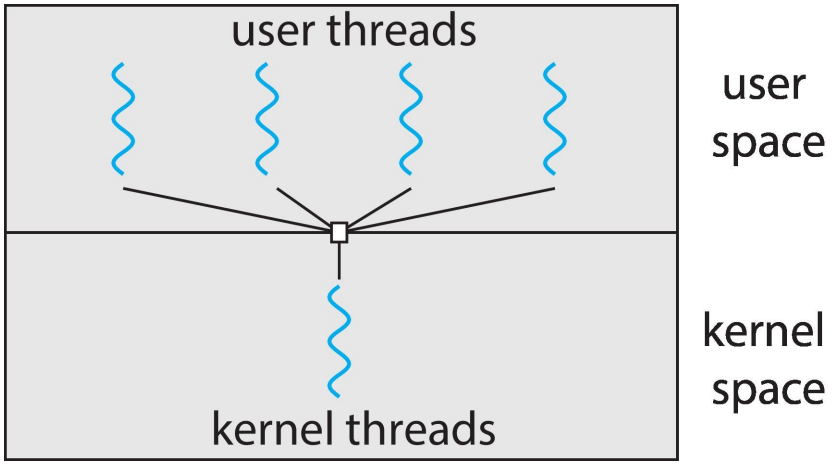
Multithreading Models

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

Model	Description	Pros	Cons	Scenario
Many-to-One	Maps many user threads to one kernel thread.	Efficient thread management in user space.	Blocks entire process on blocking system calls; no parallelism on multicore.	Early Java versions using Green threads on single-core systems.
One-to-One	Maps each user thread to a separate kernel thread.	Allows parallelism on multicore; no blocking issues.	High overhead due to many kernel threads; may burden system performance.	Modern OS like Linux and Windows for high-concurrency applications.
Many-to-Many	Maps many user threads to a smaller or equal number of kernel threads.	Balances concurrency and parallelism; flexible for multicore systems.	Complex to implement; less relevant with abundant cores.	Applications need high concurrency without overloading kernel resources.
Two-Level (Variation of Many-to-Many)	Allows binding some user threads to kernel threads while multiplexing others.	Combines the flexibility of many-to-many with direct kernel thread binding.	Adds implementation complexity.	Real-time systems require both flexibility and direct kernel control.

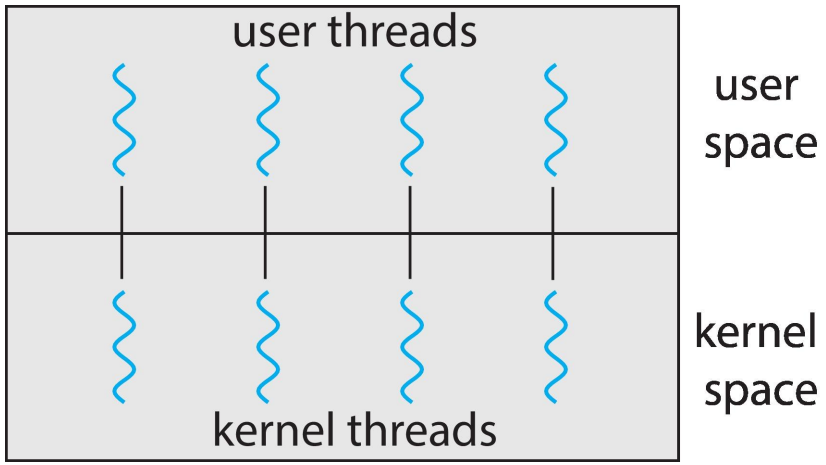
Many-to-One

- 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**



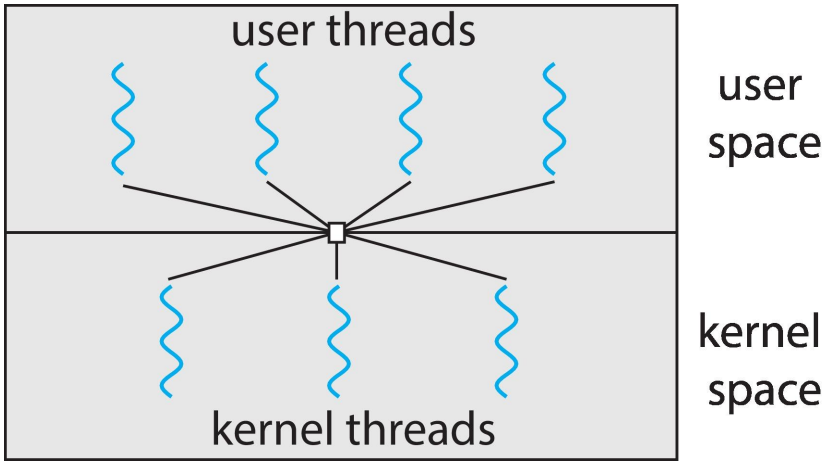
One-to-One

- 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



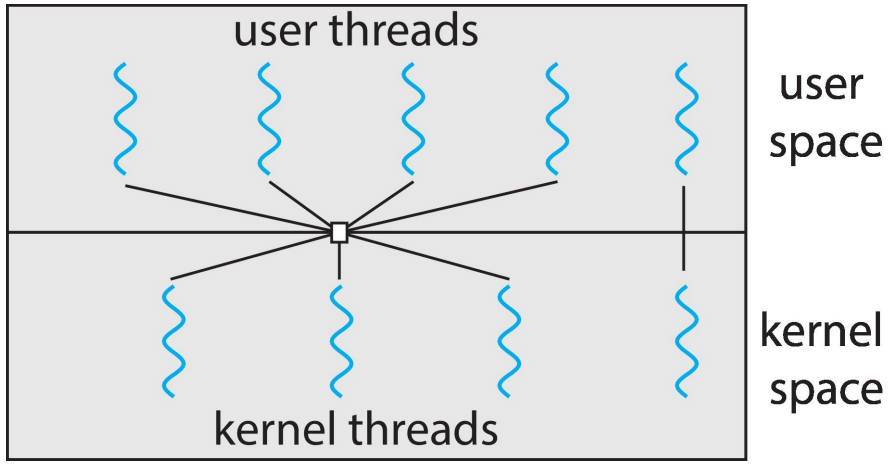
Many-to-Many Model

- 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)

Creating Threads

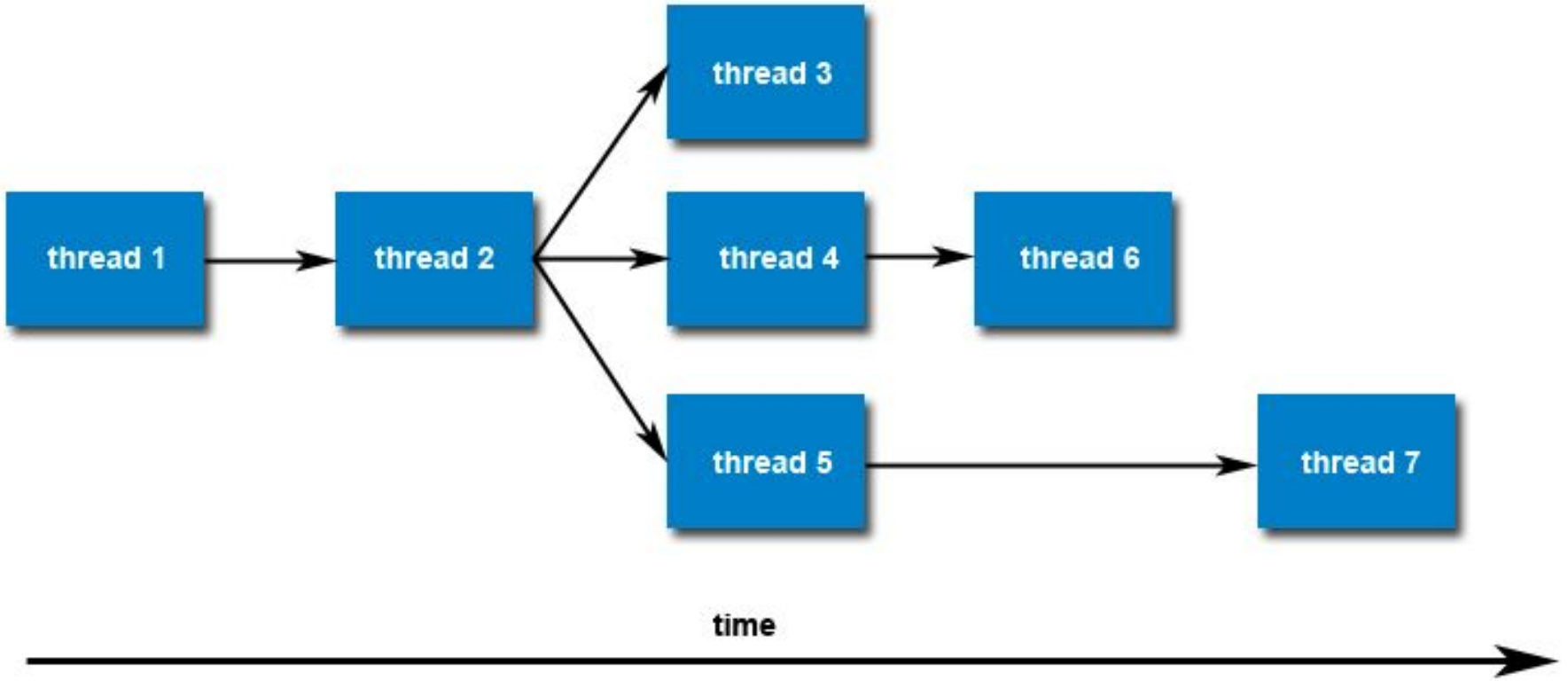
Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer. `pthread_create` creates a new thread and makes it executable. This method can be called any number of times from anywhere within your code.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start)(void *), void *arg);
```

- `thread` – An opaque, unique identifier for the new thread returned by the subroutine.
- `attr` – An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object or NULL for the default values.
- `start_routine` – the C function that the thread will execute once it is created.
- `arg` – A single argument that may be passed to `start_routine`. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

Upon successful completion pthread_create() returns code 0, a non-zero value signals an error.

A thread can create other threads.



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *helloWorld(void *vargp) {
    sleep(1);
    printf("Hello World \n");
    return NULL;
}
int main() {
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, helloWorld, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

Terminating Threads

There are several ways in which a thread may be terminated:

- The thread returns normally from its starting routine. Its work is done.
- The thread makes a call to the `pthread_exit` method - whether its work is done or not.
- The thread is cancelled by another thread via the `pthread_cancel` routine.
- The entire process is terminated due to making a call to either the `exec()` or `exit()` .
- If `main()` finishes first, without calling `pthread_exit` explicitly itself then all other threads will abruptly end.

For this section, we will take a look at the `pthread_exit` method.

```
void pthread_exit(void *retval);
```

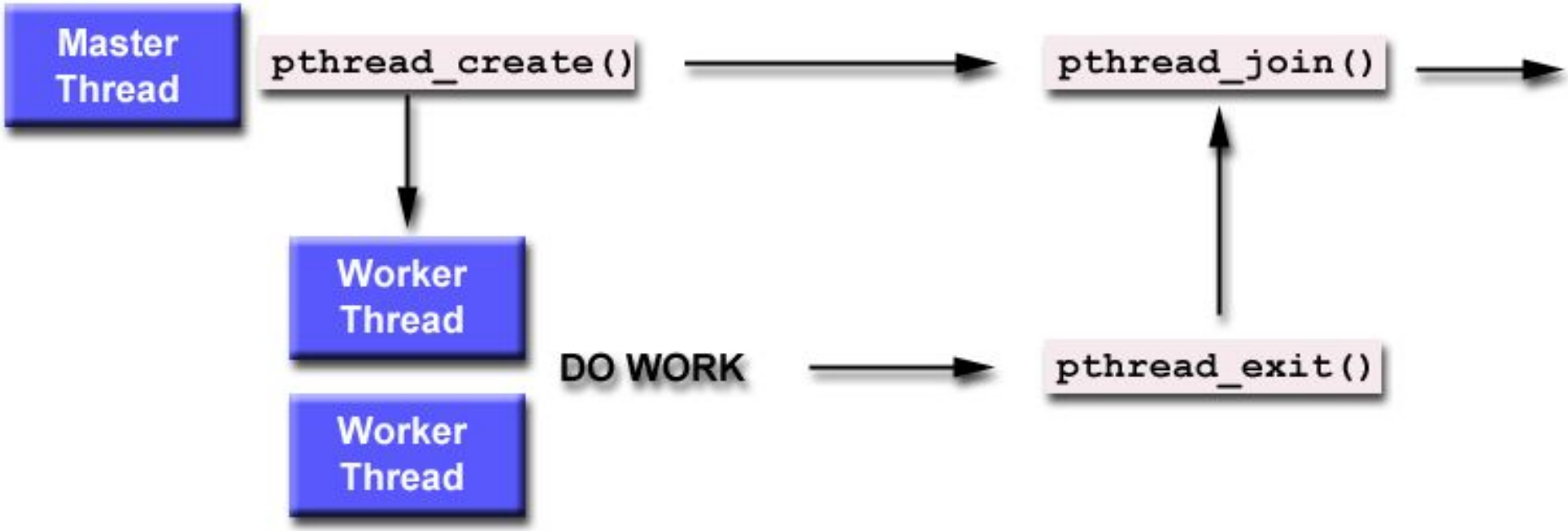
```c
#define NUM_THREADS 5
void *PrintHello(void *threadid) {
    long tid;
    tid = (long) threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;    long t;
    for(t=0; t<NUM_THREADS; 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() is %d\n", rc);
          exit(-1);
       }
    }
    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

Joining Threads

Joining is done by the pthread_join method.

What it essentially does is that it waits till the given thread ends.

If the thread was already completed before the pthread_join method was called then the method will immediately return a value.



Joining Example (1)

```
2   void *BusyWork(void *t) {
3       int i;
4       long tid;
5       double result=0.0;
6       tid = (long)t;
7       printf("Thread %ld starting...\n",tid);
8       for (i=0; i<1000000; i++)   {
9           result = result + sin(i) * tan(i);
10      }
11      printf("Thread %ld done. Result = %e\n",tid, result);
12      pthread_exit((void*) t);
13  }
```

Joining Example (2)

```
14  int main (int argc, char *argv[]){
15      pthread_t thread[NUM_THREADS];
16      pthread_attr_t attr;
17      int rc;   long t;   void *status;
18      /* Initialize and set thread detached attribute */
19      pthread_attr_init(&attr);
20      pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
21
22      for(t=0; t<NUM_THREADS; t++) {
23          printf("Main: creating thread %ld\n", t);
24          rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
25          if (rc) {
26              printf("ERROR; return code from pthread_create() is %d\n", rc);
27              exit(-1);
28              }
29          }
```

Joining Example (3)

```
30      /* Free attribute and wait for the other threads */
31      pthread_attr_destroy(&attr);
32      for(t=0; t<NUM_THREADS; t++) {
33          rc = pthread_join(thread[t], &status);
34          if (rc) {
35              printf("ERROR; return code from pthread_join()
36              exit(-1);
37              }
38          printf("Main: completed join with thread %ld havi
39          }
40  printf("Main: program completed. Exiting.\n");
41  pthread_exit(NULL);
42  }
```

Terminating Threads Early

Sometimes you just want to end a thread early.

You can do it using the `pthread_cancel` method.

```
int pthread_cancel (pthread_t THREAD_ID);
```

It is important to understand that although `pthread_cancel()` returns immediately and can terminate a thread prematurely, it cannot be called a means of forcing threads to terminate.

So essentially `pthread_cancel` sends a cancellation request but if you want to delete a thread, you must make sure to use `pthread_join` beforehand.

Here's a small code snippet on how `pthread_cancel` is used.

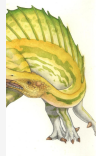
```
pthread_t tid;

pthread_create(&tid, 0, worker, NULL);

pthread_cancel(tid);
```

Detaching Threads (1)

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

// Function to be executed by the thread
Tabnine | Edit | Test | Explain | Document
void* threadFunction(void* arg) {
    printf("Thread is running...\n");
    // Simulate some work with sleep
    sleep(2);
    printf("Thread is finishing...\n");
    return NULL;
}
```

Detaching Threads (2)

```c
int main() {
    pthread_t thread;
    int result;
    // Create a new thread
    result = pthread_create(&thread, NULL, threadFunction, NULL);
    if (result != 0) {
        fprintf(stderr, "Error creating thread\n");
        return 1;
    }
    // Detach the thread
    result = pthread_detach(thread);
    if (result != 0) {
        fprintf(stderr, "Error detaching thread\n");
        return 1;
    }
    // Main thread continues to run
    printf("Main thread is continuing...\n");
    // Simulate some work in the main thread
    sleep(3);
    printf("Main thread is finishing...\n");
    return 0;
}
```



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);
}
```

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);
```

1. **Thread Creation:**
   - `pthread_create()` : Creates a new thread.
2. **Thread Termination:**
   - `pthread_exit()` : Terminates the calling thread.
   - `pthread_cancel()` : Cancels a specific thread.
3. **Thread Synchronization:**
   - **Mutexes:**
     - `pthread_mutex_init()` , `pthread_mutex_lock()` , `pthread_mutex_unlock()` , `pthread_mutex_destroy()` .
   - **Condition Variables:**
     - `pthread_cond_init()` , `pthread_cond_wait()` , `pthread_cond_signal()` , `pthread_cond_broadcast()` , `pthread_cond_destroy()` .
4. **Thread Joining:**
   - `pthread_join()` : Waits for a thread to terminate.
5. **Thread Attributes:**
   - `pthread_attr_init()` , `pthread_attr_setdetachstate()` , `pthread_attr_destroy()` : Manage thread attributes (e.g., detached state).

```
1   #include <stdio.h>
2   #include <assert.h>
3   #include <pthread.h>
4
5   void *mythread(void *arg) {
6       printf("%s\n", (char *) arg);
7       return NULL;
8   }
9
10  int
11  main(int argc, char *argv[]) {
12      pthread_t p1, p2;
13      br int rc;
14      printf("main: begin\n");
15      rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16      rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17      // join waits for the threads to finish
18      rc = pthread_join(p1, NULL); assert(rc == 0);
19      rc = pthread_join(p2, NULL); assert(rc == 0);
20      printf("main: end\n");
21      return 0;
22  }
```

Let us examine the possible execution ordering of this little program. In the execution diagram (Table 26.1), time increases in the downwards direction, and each column shows when a different thread (the main one, or Thread 1, or Thread 2) is running.

Note, however, that this ordering is not the only possible ordering. In fact, given a sequence of instructions, there are quite a few, depending on which thread the scheduler decides to run at a given point. For example, once a thread is created, it may run immediately, which would lead to the execution shown in Table 26.2.

We also could even see "B" printed before "A", if, say, the scheduler decided to run Thread 2 first even though Thread 1 was created earlier; there is no reason to assume that a thread that is created first will run first. Table 26.3 shows this final execution ordering, with Thread 2 getting to strut its stuff before Thread 1.

As you might be able to see, one way to think about thread creation is that it is a bit like making a function call; however, instead of first executing the function and then returning to the caller, the system instead creates a new thread of execution for the routine that is being called, and it runs independently of the caller, perhaps before returning from the create, but perhaps much later.

As you also might be able to tell from this example, threads make life complicated: it is already hard to tell what will run when! Computers are hard enough to understand without concurrency. Unfortunately, with concurrency, it gets worse. Much worse.

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Table 26.1: **Thread Trace (1)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Table 26.2: **Thread Trace (2)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Table 26.3: **Thread Trace (3)**

Volatile keyword: tells the compiler that variable might change at any moment due to some external influence (such as hardware or another thread). Therefore, the compiler should not optimize based on the assumption that the variable will remain constant.

Static keyword:

- When used inside a function to declare a variable, it creates a variable that retains its value between function calls. These variables are initialized only once before the program starts execution and they retain their value throughout the program's execution. Static local variables are not visible to functions in other files.
- When used outside of functions, it declares a variable that is accessible only within the file where it is declared. These variables are visible only within the file where they are declared, meaning they cannot be accessed by functions in other files. Static global variables have a file scope, which means they are accessible from the point of their declaration to the end of the file.

```
static volatile int counter = 0;
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

Global Variable

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

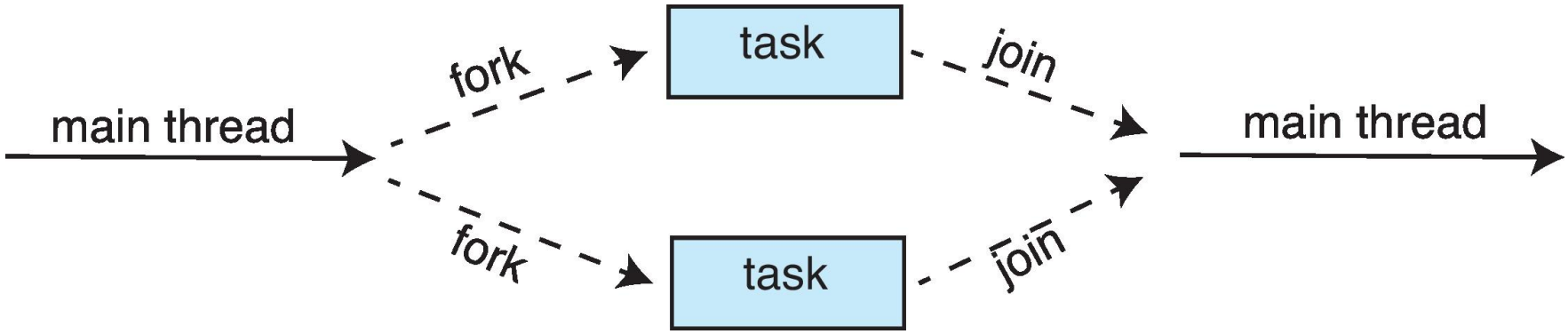
```
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

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~~

Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.



Fork-join parallelism is a programming model and parallel computing paradigm that involves splitting ("forking") a task into multiple subtasks, which are then executed concurrently across multiple processing units (e.g., CPU cores or threads). Once all subtasks have completed their execution, the results are combined ("joined") to produce the final result.

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

#define NUM_THREADS 4
#define ARRAY_SIZE 1000000

int global_array[ARRAY_SIZE]; // Shared array

// Function to initialize the array with random values
void initialize_array() {
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        global_array[i] = rand() % 1000;
    }
}

// Function to find the sum of elements in a portion of the array
void *sum_array(void *arg) {
    int thread_id = *((int *)arg);
    int start = thread_id * (ARRAY_SIZE / NUM_THREADS);
    int end = start + (ARRAY_SIZE / NUM_THREADS);
    int sum = 0;

    // Calculate the sum of elements in the assigned portion of the array
    for (int i = start; i < end; ++i) {
        sum += global_array[i];
    }

    return (void *)(long)sum; // Return the sum as a void pointer
}
```

```c
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    void *thread_results[NUM_THREADS];
    long total_sum = 0;

    // Initialize the array with random values
    initialize_array();

    // Create threads to compute the sum of array elements
    for (int i = 0; i < NUM_THREADS; ++i) {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, sum_array, (void *)&thread_args[i]);
    }

    // Join threads and collect results
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], &thread_results[i]);
        total_sum += (long)thread_results[i]; // Accumulate the partial sums
    }

    printf("Total sum of array elements: %ld\n", total_sum);

    return 0;
}
```

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

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 one thread in a program calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of `fork()`, one that duplicates all threads and another that duplicates only the thread that invoked the `fork()` system call.

The `exec()` system call typically works in the same way as described in Chapter 3. That is, if a thread invokes the `exec()` system call, the program specified in the parameter to `exec()` will replace the entire process—including all threads.

Which of the two versions of `fork()` to use depends on the application. 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

The standard UNIX function for delivering a signal is

```
kill(pid_t pid, int signal)
```

```
// Send SIGUSR1 signal to the thread
pthread_kill(thread_id, SIGUSR1);
```

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

```
pthread_kill(pthread_t tid, int signal)
```

Not part of syllabus

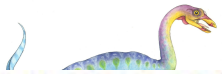
Signal Handling

1. `SIGHUP`: Hangup (POSIX).
2. `SIGINT`: Interrupt (ANSI).
3. `SIGQUIT`: Quit (POSIX).
4. `SIGILL`: Illegal instruction (ANSI).
5. `SIGTRAP`: Trace trap (POSIX).
6. `SIGABRT`: Abort (ANSI).
7. `SIGIOT`: IOT Trap (4.2 BSD).
8. `SIGBUS`: BUS error (4.2 BSD).
9. `SIGFPE`: Floating-point exception (ANSI).
10. `SIGKILL`: Kill, unblockable (POSIX).
11. `SIGUSR1`: User-defined signal 1 (POSIX).
12. `SIGSEGV`: Segmentation violation (ANSI).
13. `SIGUSR2`: User-defined signal 2 (POSIX).
14. `SIGPIPE`: Broken pipe (POSIX).
15. `SIGALRM`: Alarm clock (POSIX).
16. `SIGTERM`: Termination (ANSI).
17. `SIGSTKFLT`: Stack fault.
18. `SIGCHLD`: Child status has changed (POSIX).
19. `SIGCONT`: Continue (POSIX).

Symbolic constants for various signals defined in <signal.h> header file.

20. `SIGSTOP`: Stop, unblockable (POSIX).
21. `SIGTSTP`: Keyboard stop (POSIX).
22. `SIGTTIN`: Background read from control terminal (POSIX).
23. `SIGTTOU`: Background write to control terminal (POSIX).
24. `SIGURG`: Urgent condition on socket (4.2 BSD).
25. `SIGXCPU`: CPU limit exceeded (4.2 BSD).
26. `SIGXFSZ`: File size limit exceeded (4.2 BSD).
27. `SIGVTALRM`: Virtual alarm clock (4.2 BSD).
28. `SIGPROF`: Profiling alarm clock (4.2 BSD).
29. `SIGWINCH`: Window size change (4.3 BSD, Sun).
30. `SIGPOLL`: Pollable event occurred (System V).
31. `SIGIO`: I/O now possible (4.2 BSD).
32. `SIGPWR`: Power failure restart (System V).
33. `SIGSYS`: Bad system call.

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

// Signal handler function
void sigint_handler(int signum) {
    printf("Caught SIGINT signal (%d)\n", signum);
}

int main() {
    // Registering signal handler for SIGINT
    if (signal(SIGINT, sigint_handler) == SIG_ERR) {
        printf("Error setting up signal handler for SIGINT\n");
        return 1;
    }

    printf("Press Ctrl+C to send a SIGINT signal\n");

    // Infinite loop to keep the program running
    while(1) {
        sleep(1);
    }

    return 0;
}
```



4.6.2 Signal Handling

A **signal** is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled. All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as <control><C>) and

having a timer expire. Typically, an asynchronous signal is sent to another process.

A signal may be ***handled*** by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler

Every signal has a **default signal handler** that the kernel runs when handling that signal. This default action can be overridden by a **user-define signal handler** that is called to handle the signal. Signals are handled in different ways. Some signals may be ignored, while others (for example, an illegal memory access) are handled by terminating the program.

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

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:

```
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-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

```
static _thread int threadID;
```

Linux Thread Creation System call

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

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.

- The `clone` system call is used by the Linux kernel to create a new process or thread.
- It is a versatile system call that can be used to create threads with varying degrees of sharing of resources such as memory, file descriptors, and signal handlers.
- Threads created using `clone` share the same memory space (unless specified otherwise) and are commonly used by threading libraries like pthreads to implement threads.

5.4 Thread Scheduling

- On most modern operating systems, it is kernel-level threads—not processes—that are being scheduled by the operating system.

- User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).

5.4 Thread Scheduling

In the context of POSIX threads (pthreads), contention scope refers to how threads contend for shared resources, such as mutexes or condition variables. There are two contention scopes defined in pthreads:

1. **PTHREAD_SCOPE_SYSTEM**: In this contention scope, threads contend for resources across the entire system. This means that threads from different processes may contend for the same resources. The behavior of this scope is implementation-defined, and it may not be supported on all systems.
2. **PTHREAD_SCOPE_PROCESS**: In this contention scope, threads contend for resources within the same process. Threads from different processes do not contend for the same resources. This scope provides a more predictable behavior and is typically used when threads belong to the same process.

You can set the contention scope using the `**pthread_attr_setscope()**` function before creating threads. Here's an example:

```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
```

5.4 Thread Scheduling (LINUX Specific)

1. **Scheduling Classes**: Linux supports multiple scheduling classes, each designed for different types of workloads. The main scheduling classes in Linux include:
   - **Real-Time (RT)**: Provides deterministic scheduling for time-sensitive tasks.
   - **Fair Scheduler (CFS)**: Provides fair sharing of CPU time among processes.
   - **Deadline Scheduler (SCHED_DEADLINE)**: Ensures that tasks meet their deadlines.
2. **Priorities**: Threads and processes in Linux are assigned priorities that influence their scheduling. Lower priority threads are preempted by higher priority threads when they become runnable.
3. **Scheduling Policies**: Linux supports various scheduling policies that determine how threads are scheduled. Common scheduling policies include:
   - **SCHED_FIFO**: First-In-First-Out scheduling, where threads run until they voluntarily yield or are preempted by a higher priority thread.
   - **SCHED_RR**: Round-Robin scheduling, similar to FIFO but with a time quantum after which the thread is preempted.
   - **SCHED_OTHER**: Default scheduling policy used by non-real-time processes.
4. **Scheduler Parameters**: Linux provides parameters that can be adjusted to customize the behavior of the scheduler, such as time quantum for round-robin scheduling, scheduling priorities, and CPU affinity.
5. **Scheduler Classes**: Threads and processes in Linux are organized into scheduling classes, such as real-time and fair scheduling classes, each with its own scheduling policies and parameters.