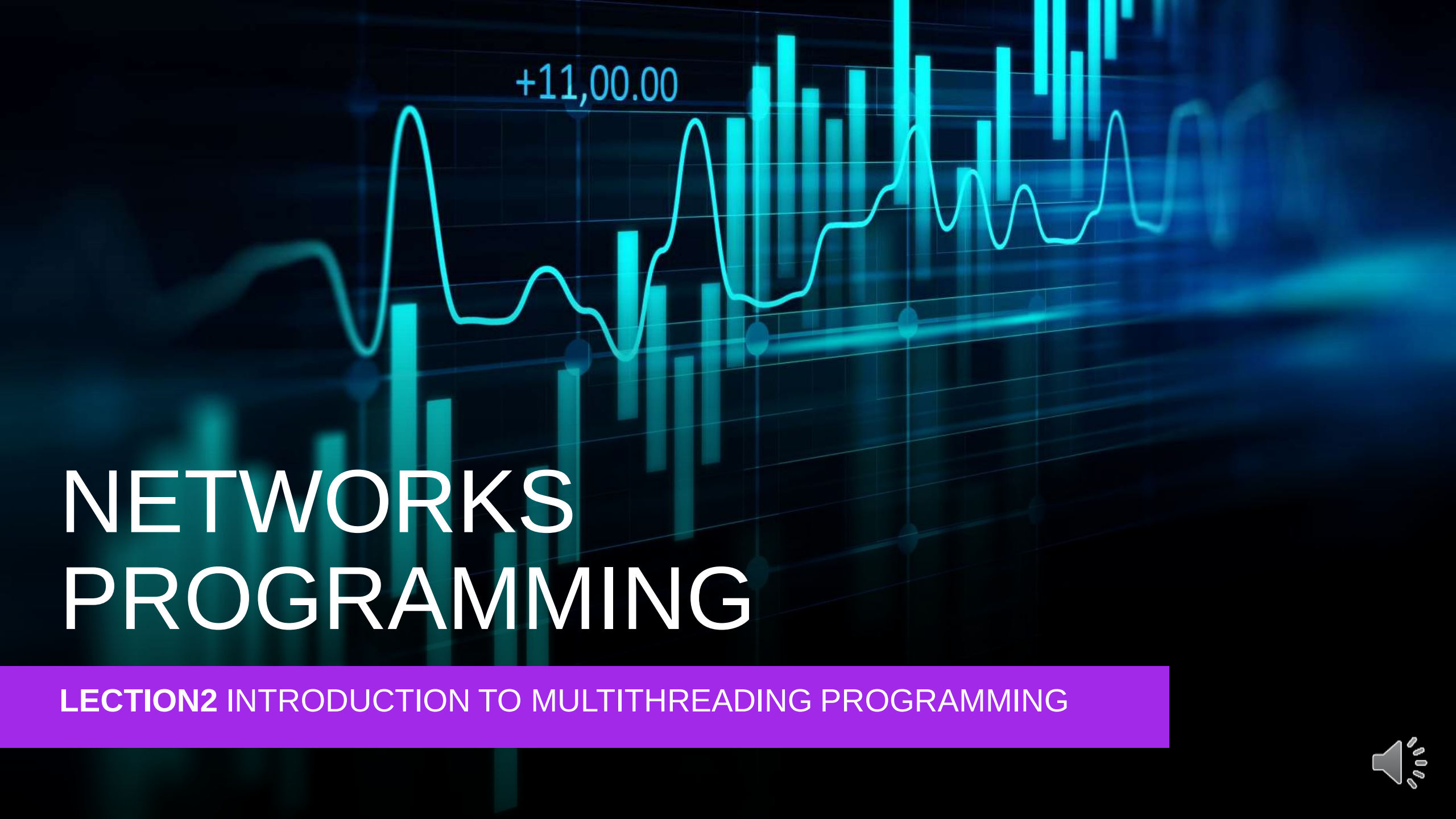


NETWORKS PROGRAMMING

LECTION2 INTRODUCTION TO MULTITHREADING PROGRAMMING

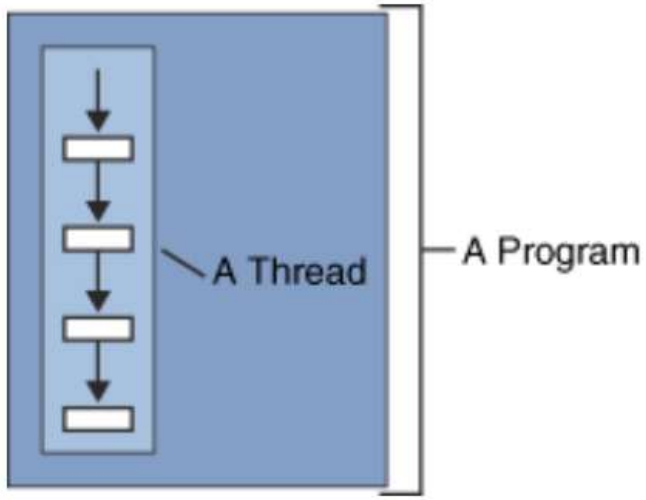
LECTION OBJECTIVES

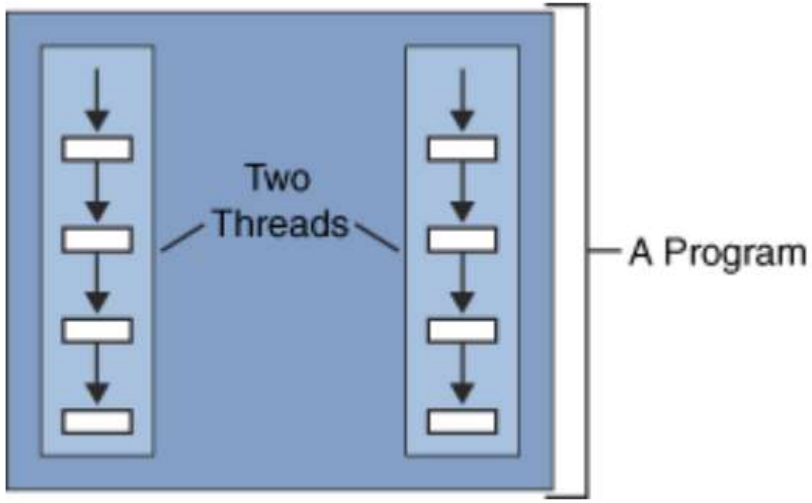
- What are Threads in Network Programs
- Threading Models
- Threading Programming Patterns
- Common Errors
  - Race Conditions
  - Deadlocks
  - Synchronization
- POSIX pthread API
- Practice

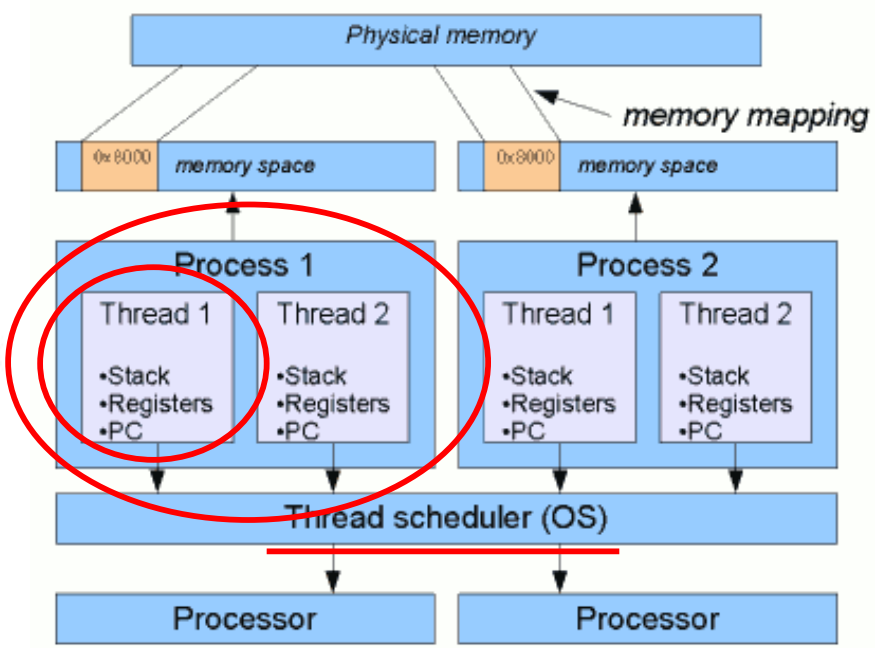
Thread is a single sequential flow of control within a program.

What Is a Thread?

A *thread* is a single sequential flow of control within a program.





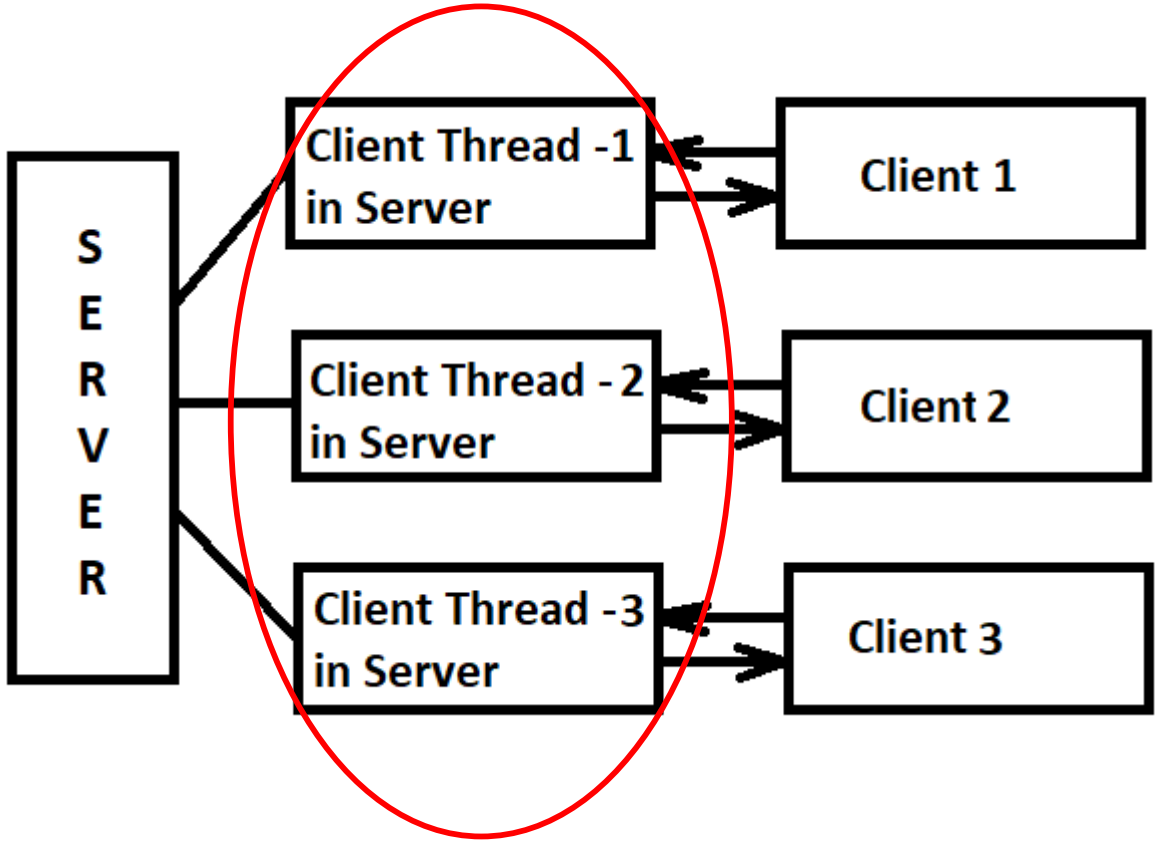


A single thread has a beginning, a sequence, and an end. At any given time during the runtime of the thread, there is a single point of execution. However, a thread itself is not a program; a thread cannot run on its own. Rather, it runs within a program.

Multithreading is about the use of multiple threads running at the same time and performing different tasks in a single program.

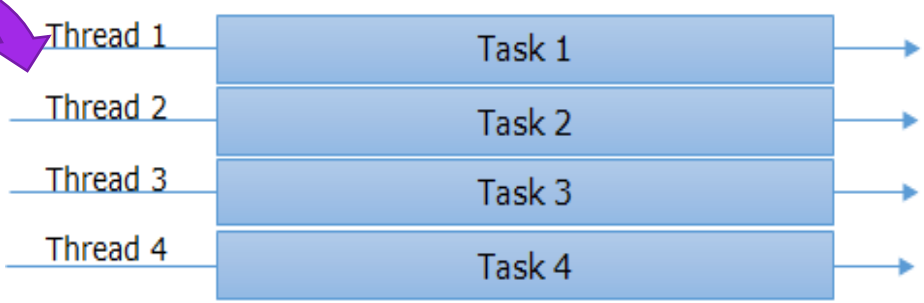
Why to use threads in network programming?

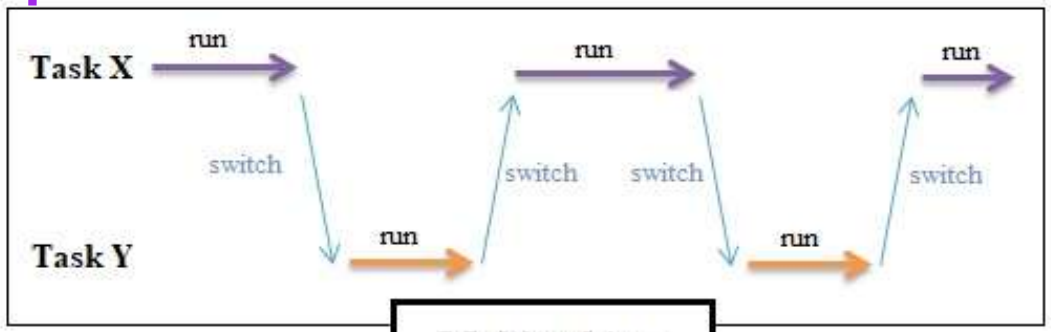
The reason is simple, we don’t want only a single client to connect to server at a particular time but many clients simultaneously. We want our architecture to **support multiple clients at the same time**. For this reason, we must use threads on server side so that whenever a client request comes, a separate thread can be assigned for handling each request.

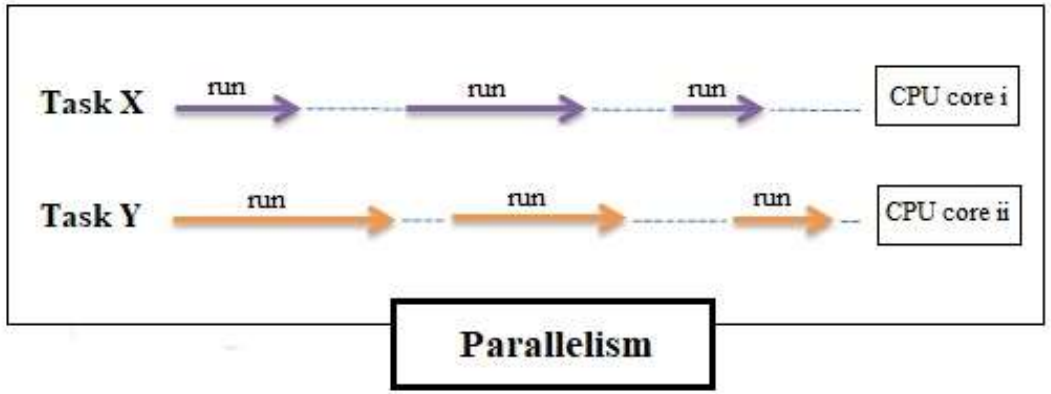


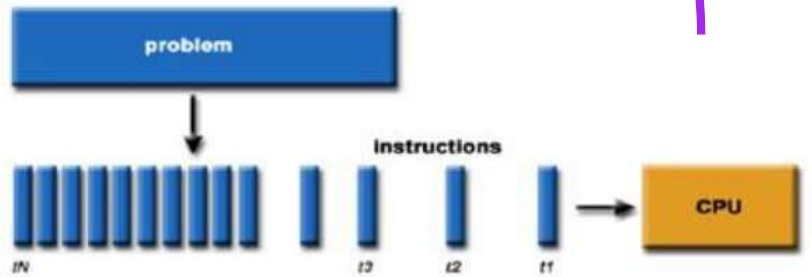
Pros and Cons of Multithreading

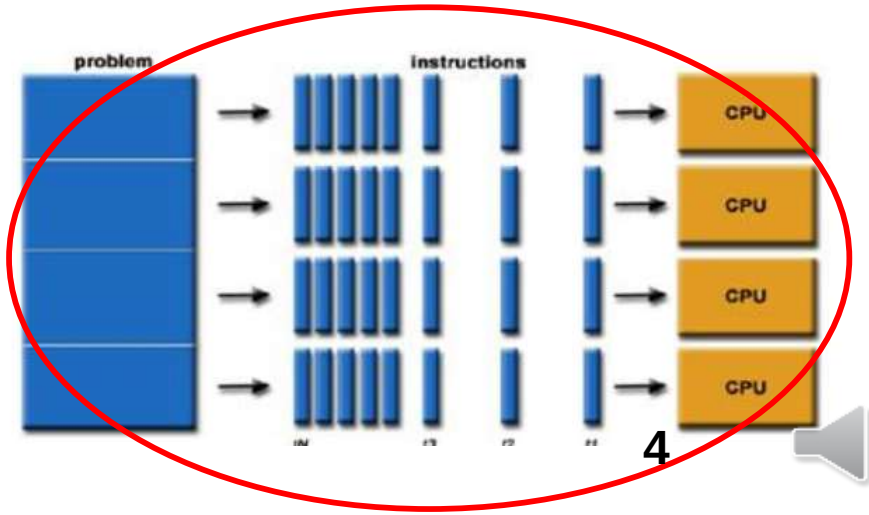
- Programming abstraction
- Parallelism
- Improving responsiveness





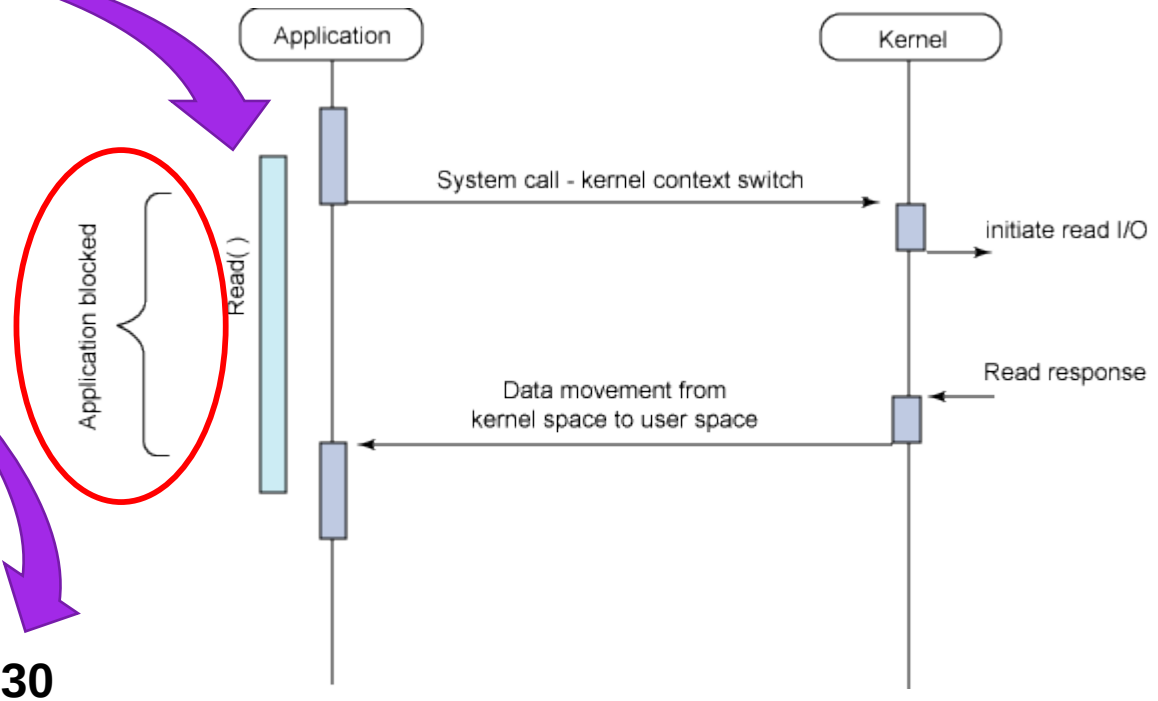


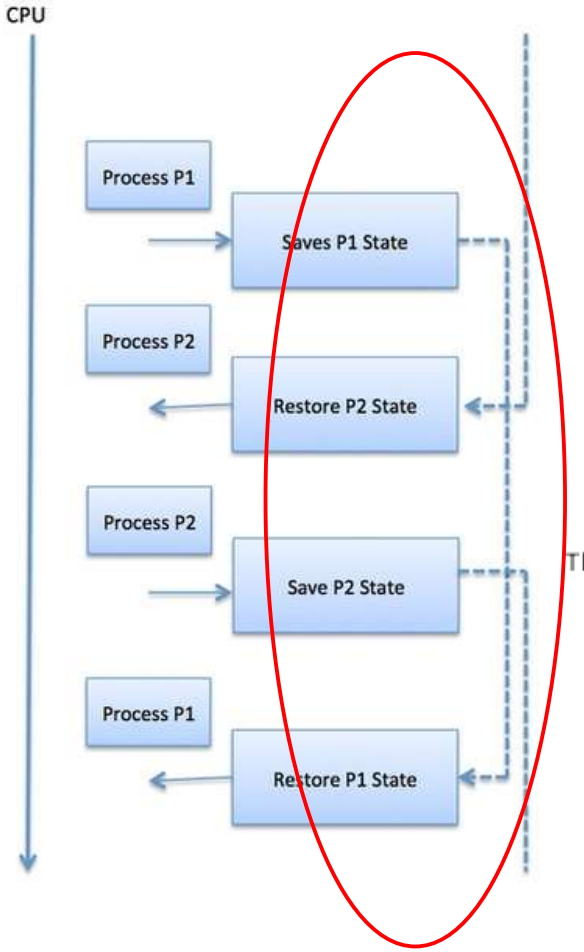




Pros and Cons of Multi reading

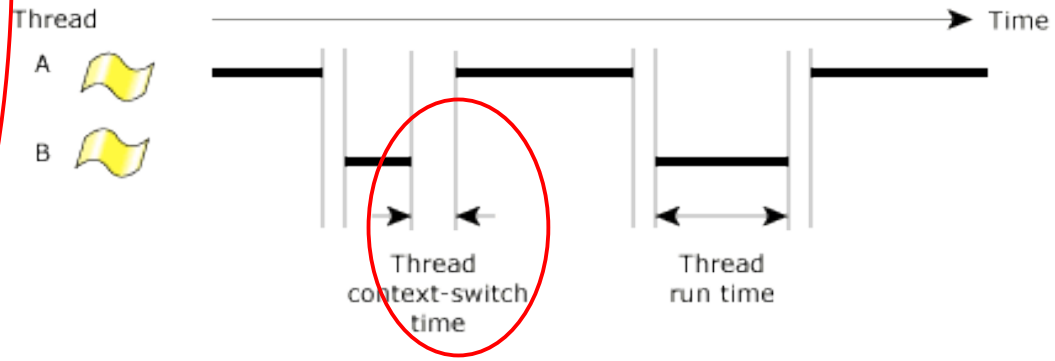
- Blocking I/O
- Context switchin
- Memory savings

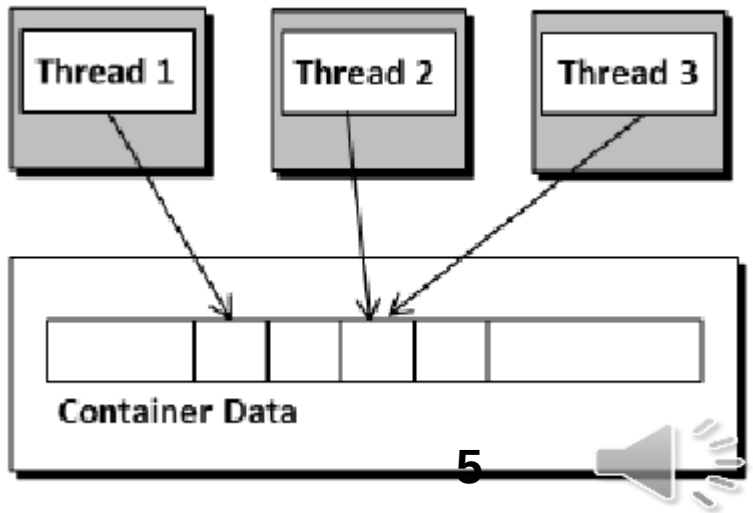




- Creating a process 30 times slower
- Switching process context – 5 times slower

!





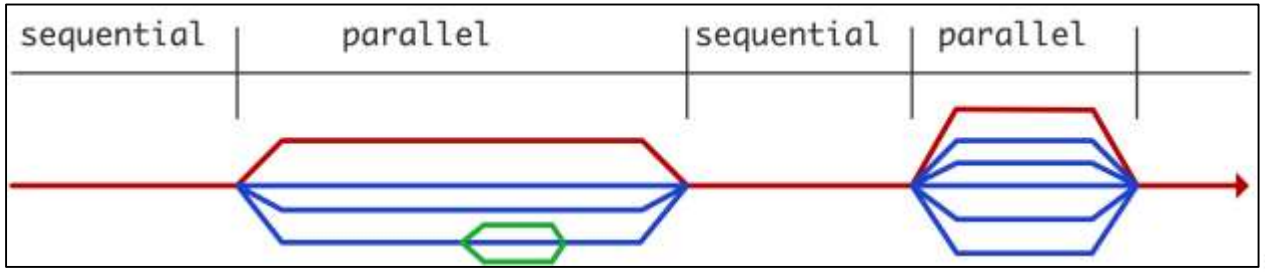
Costs of Multithreading

Amdahl's law illustrates the limitation of the performance growth (S_p) of a computing system with an increase in the number of processors (p).

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

$S_p = Speedup$

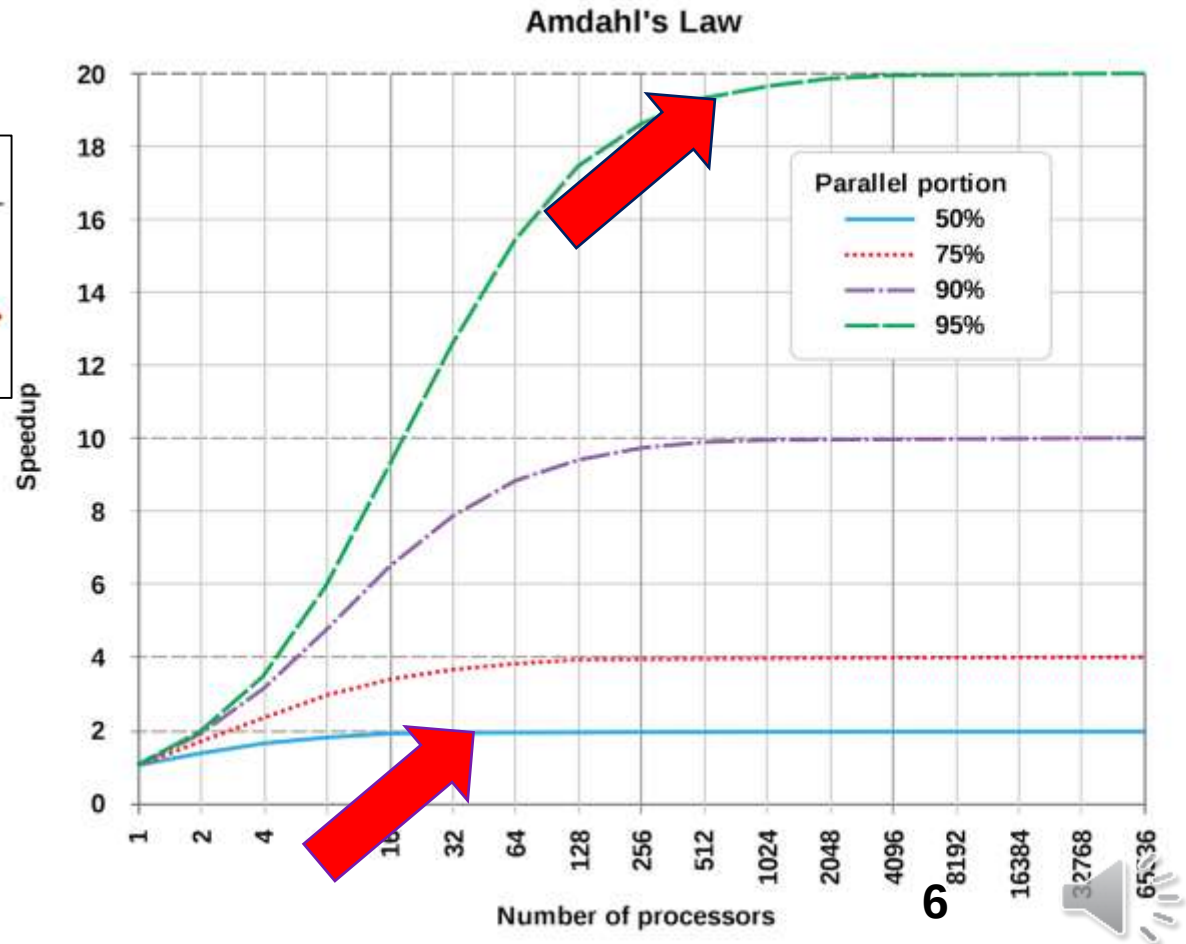
Program



Multithreading Multithreading

$$\alpha = \frac{sequential}{sequential + parallel}$$

$$p = number\ of\ processors$$



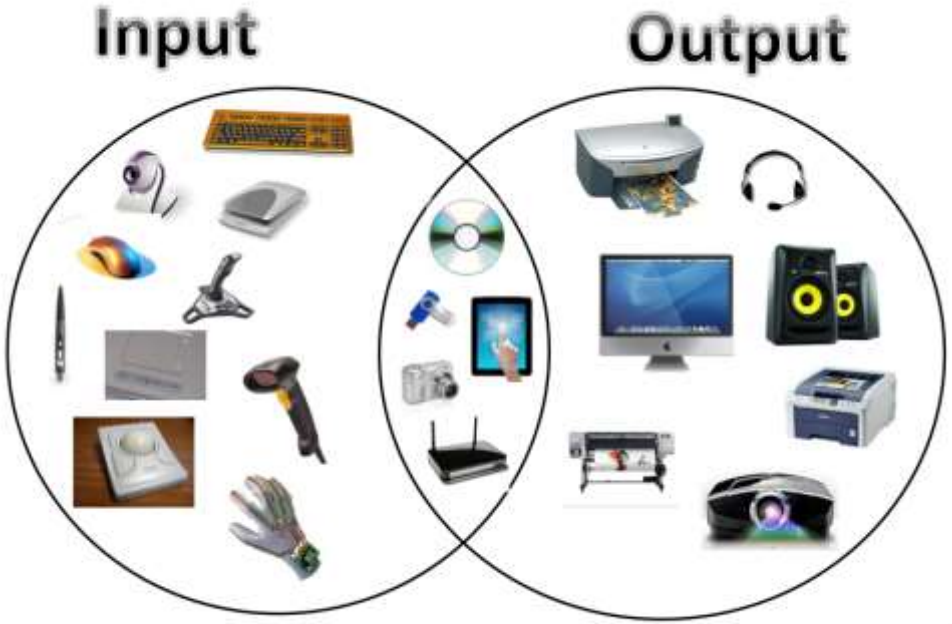
Alternative to Multithreading for I/O

- **Multiplexed I/O**
- **Nonblocking Reads**
- **Asynchronous I/O**

With *I/O multiplexing*, we call *select* or *poll* and block in one of these two system calls, instead of blocking in the actual I/O system call.

When an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead

Asynchronous I/O is defined by the POSIX specification, and various differences in the *real-time* functions that appeared in the various standards which came together to form the current POSIX specification have been reconciled. In general, these functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete.

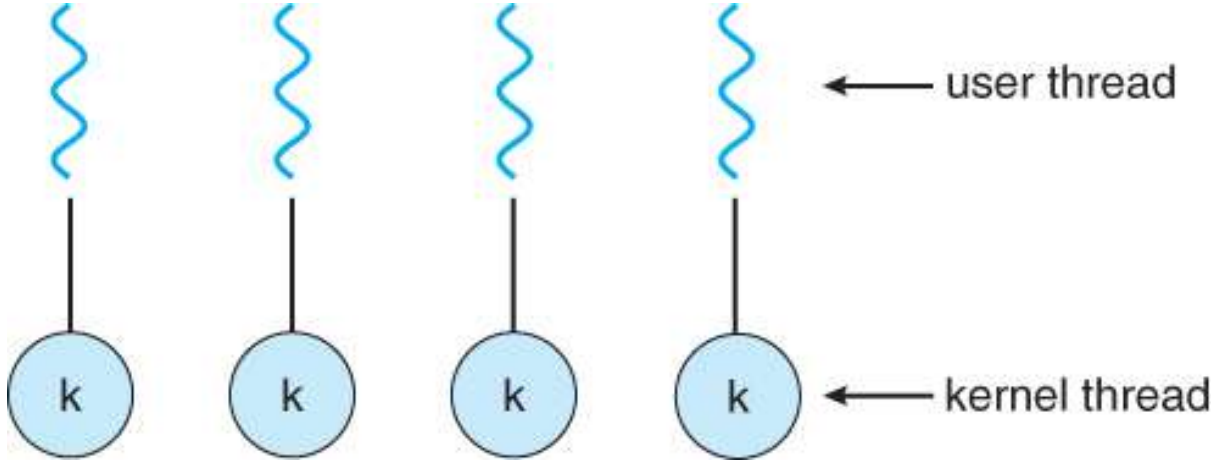


Threading Models

There are several approaches to the implementation of multithreading in a system with varying degrees of functionality provided by the kernel in user space.

- User Threads: managed without kernel support
- Kernel threads: managed directly by OS
- Relationship b/w user and kernel threads
  - Many-to-one model
  - One-to-One model
  - Many-to-Many model

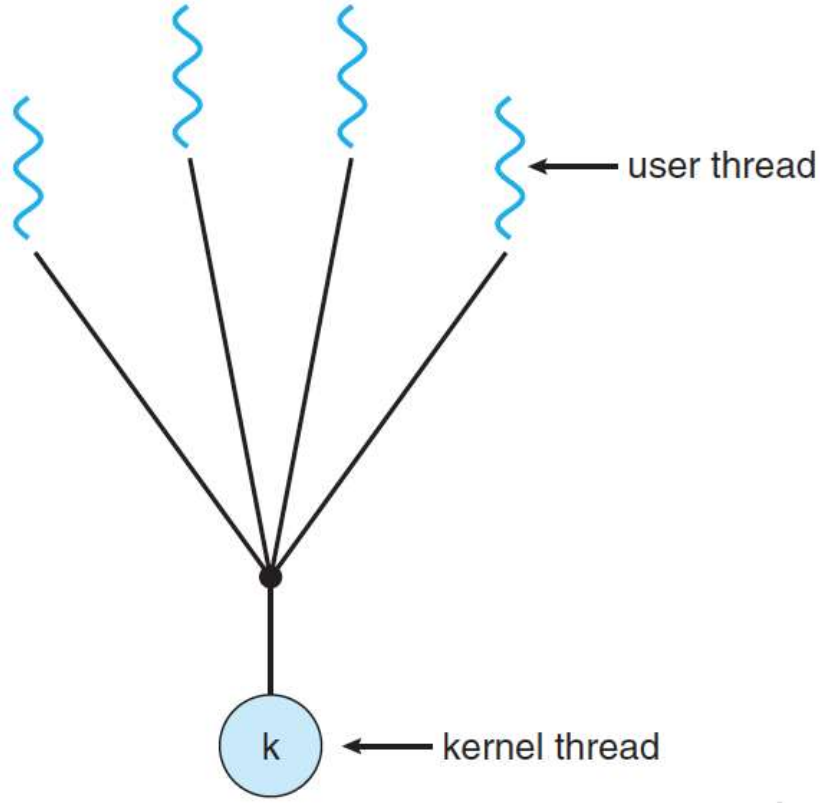
- **Kernel-Level Threading** - The simplest approach. The kernel provides its own thread support and each thread directly sends information to the user space. This model is called **one-to-one** multithreading since it has a one-to-one relationship between what the kernel represents and what the user receives. This model also known as multithreading at the kernel level because the kernel is the foundation of the multithreaded model system.



Threading Models

- **User-Level Threading ()** - contra kernel-level threading (**many-to-one model**). In this approach user space is the key to the system's threading support, as it implements the concept of a thread.

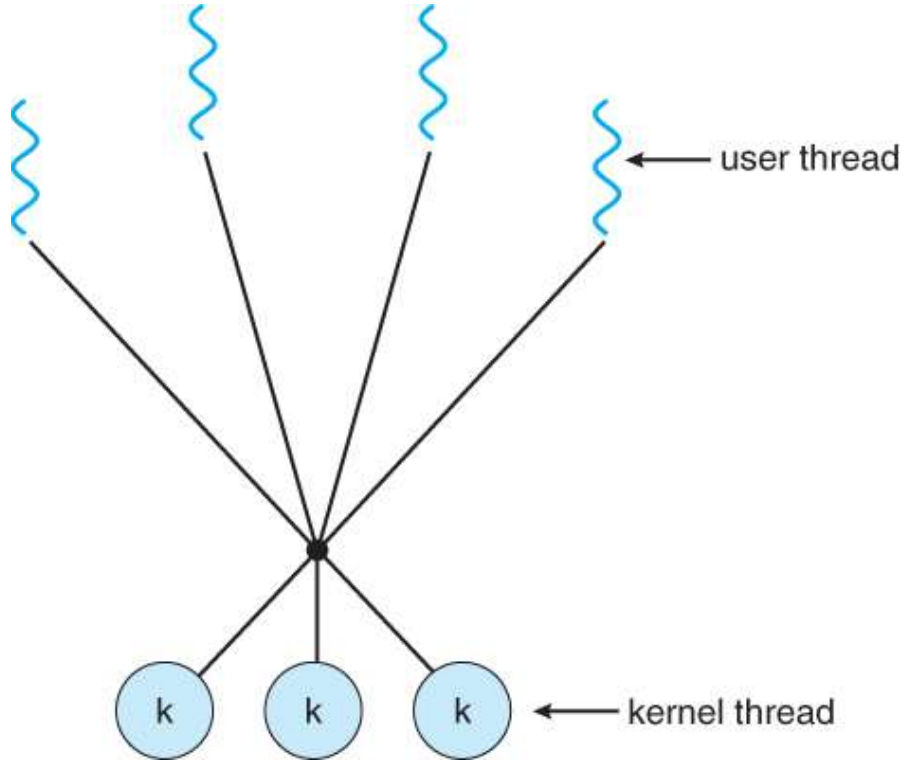
Advantages	Disadvantages
Does not require kernel support	Needs a lot of code in user space (threads scheduler, capturing and processing I / O input without blocking, etc.)
Context switching almost does not require any resources	Only one element in the kernel supports all the N threads. The model cannot use N processes and therefore true parallelism.



Threading Models

- **Hybrid Threading** - a process might contain hundreds of user threads but only a small number of kernel threads (**many-to-many**), with that small number a function of processors (with at least one kernel thread for each processor enabling the full utilization of the system) and blocking I/O.

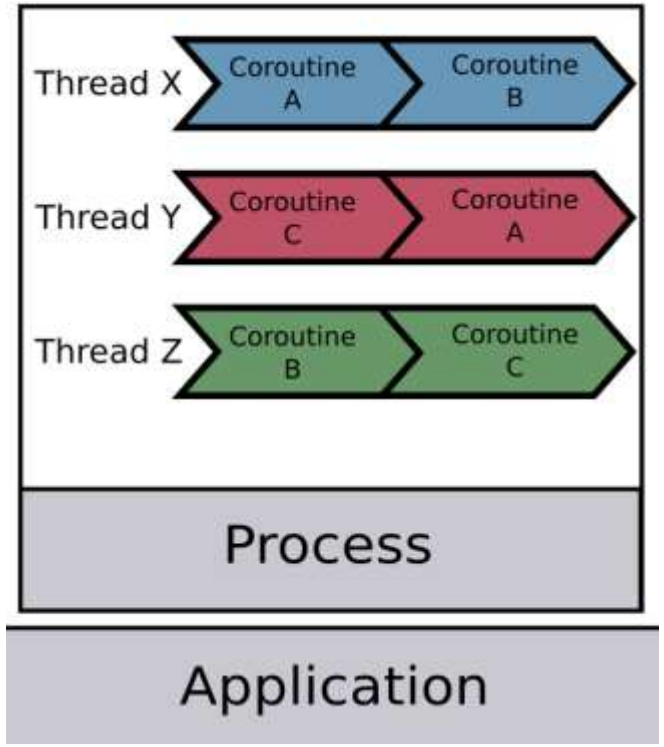
This approach combines the advantages of previous approaches (**one-to-one** and **many-to-one**). the kernel provides native threading support while user space implements user threads. User space, possibly with the participation of the kernel, then decides how to correlate N user threads with M kernels, where N> = M.

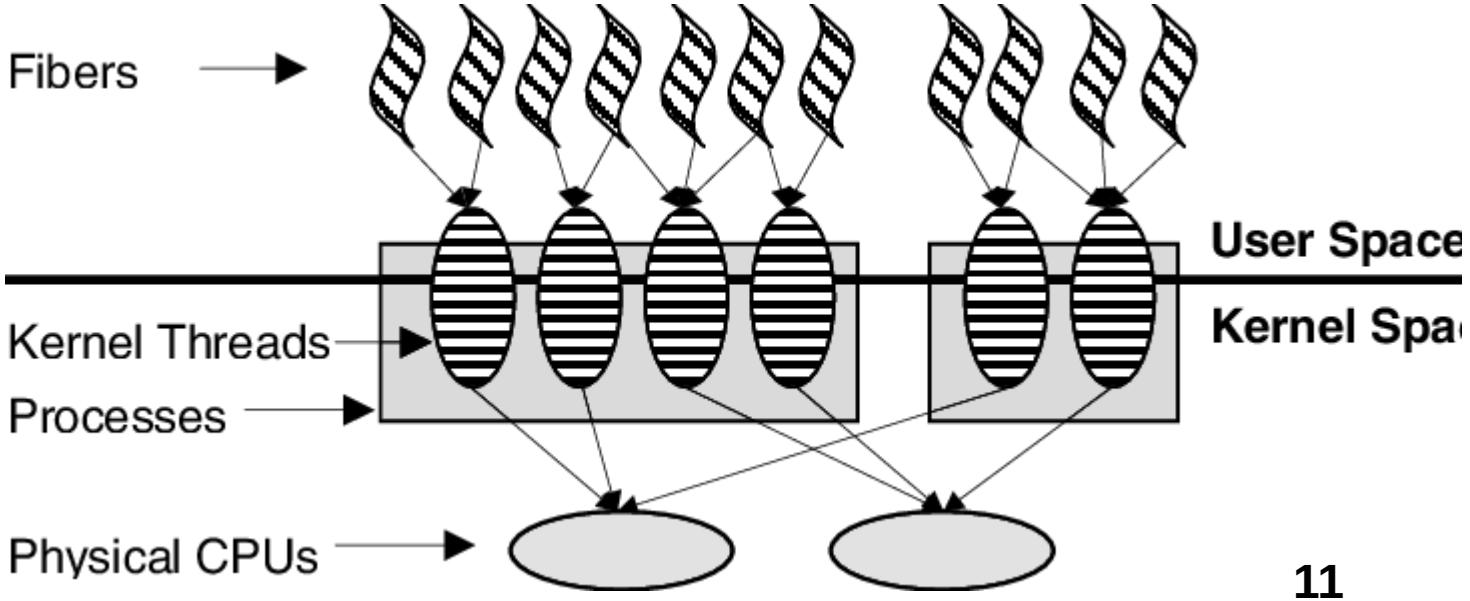


Threading Models

- **Coroutines and Fibers** - coroutines and fibers provide a unit of execution even lighter in weight than the thread. For them scheduling and execution almost does not require user space support. Instead, they are chedulled together, requiring only a clear transition from one to another.

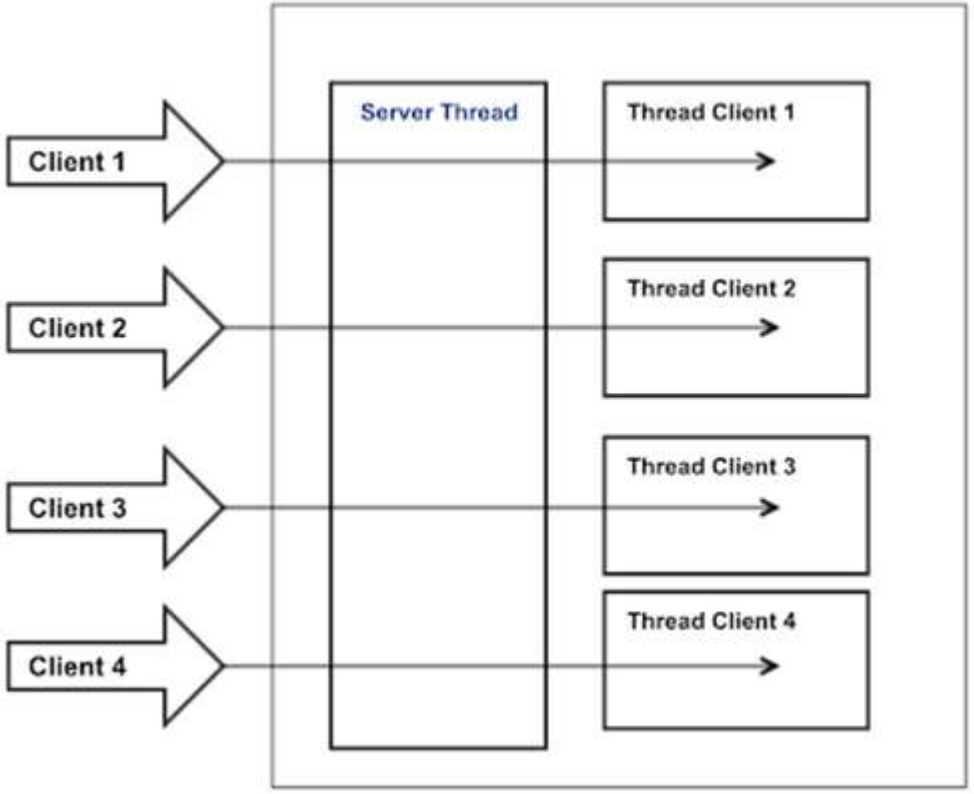
- **Coroutines and fibers** are more needed to control program execution than to execute parallelism.





Threading Patterns

- **Thread-per-Connection**



A thread-per-connection is a programming pattern in which one thread is assigned to one execution item. This thread is assigned to no more than one work item throughout all the work.

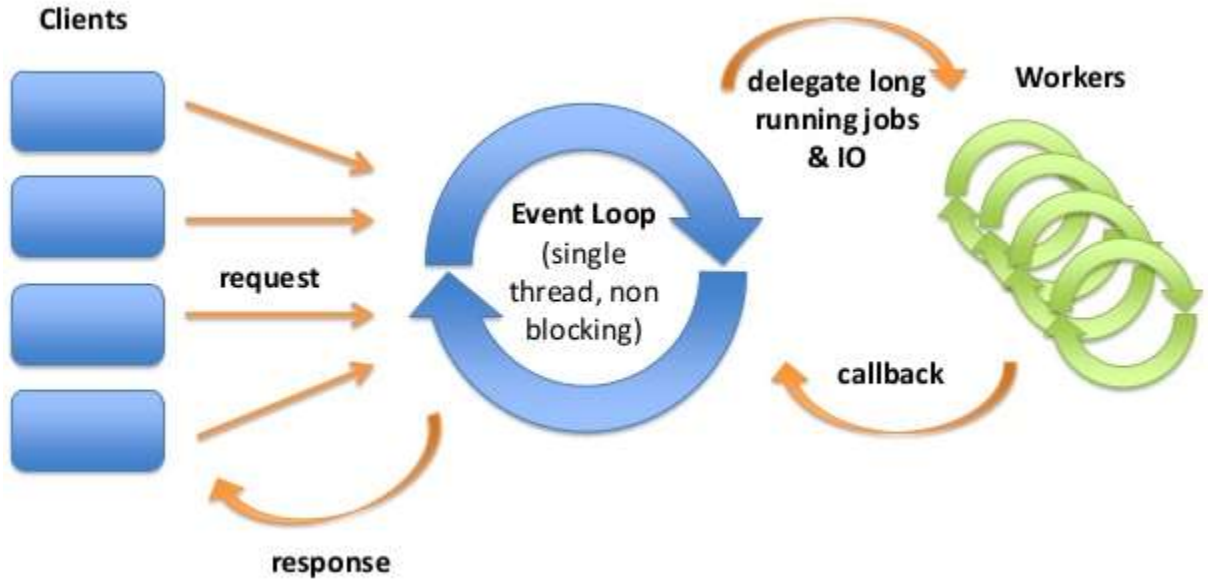
Work item - everything that can decompose the work of your application: a connection, request and so on.

Example of wrong choice of a threading pattern. Imagine a web server needs to handle a significant number of requests at the same time. With a thread-per-connection template, this means a huge number of threads that cause certain costs primarily of kernel resources and user space stack. These constant costs impose scaling restrictions on the number of threads in one process, especially in 32-bit architecture.

Threading Patterns

- **Event-Driven Threading**

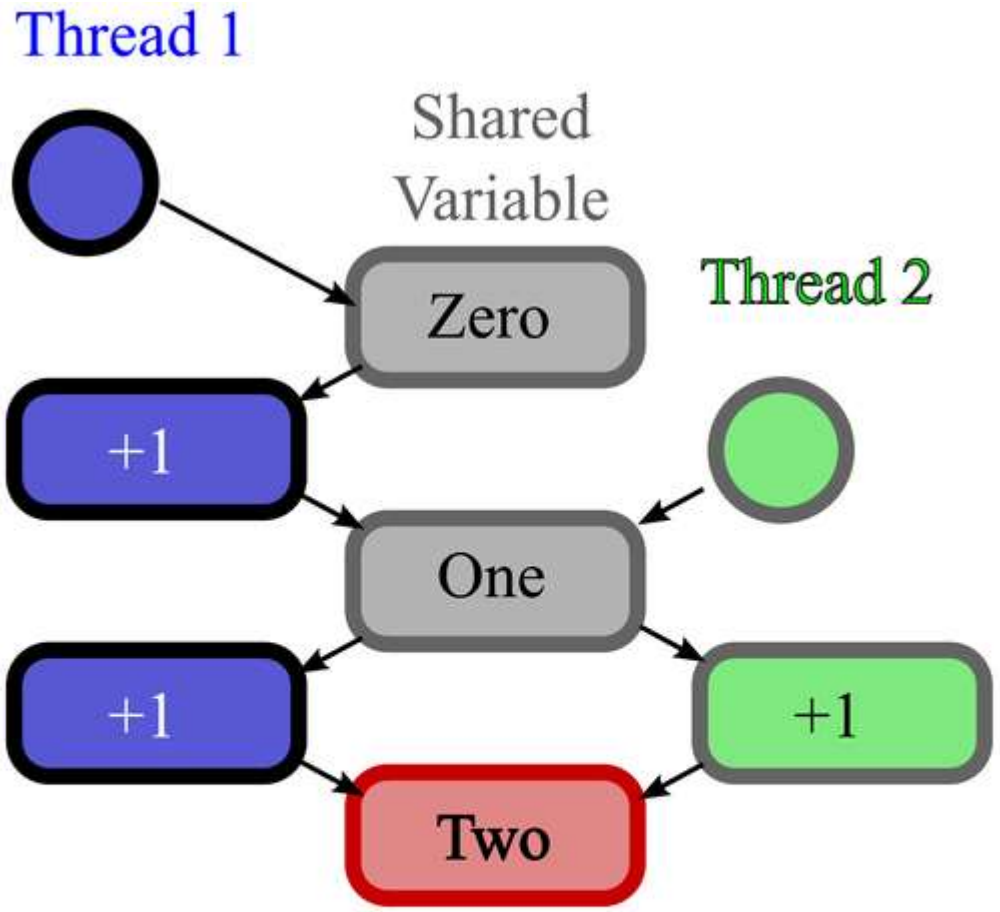
Event-driven approach can separate threads from connections, which only use threads for events on specific callbacks or handlers.
An event-driven architecture consists of event creators and event consumers.
The creator, which is the source of the event, only knows that the event has occurred.
Consumers are entities that need to know the event has occurred. They may be involved in processing the event or they may simply be affected by the event.



Race Conditions

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

```
1
2  int withdraw (struct account *account, int amount)
3  {
4      const int balance = account->balance;
5      if (balance < amount)
6          return -1;
7      account->balance = balance - amount;
8      disburse_money (amount);
9      return 0;
10 }
```

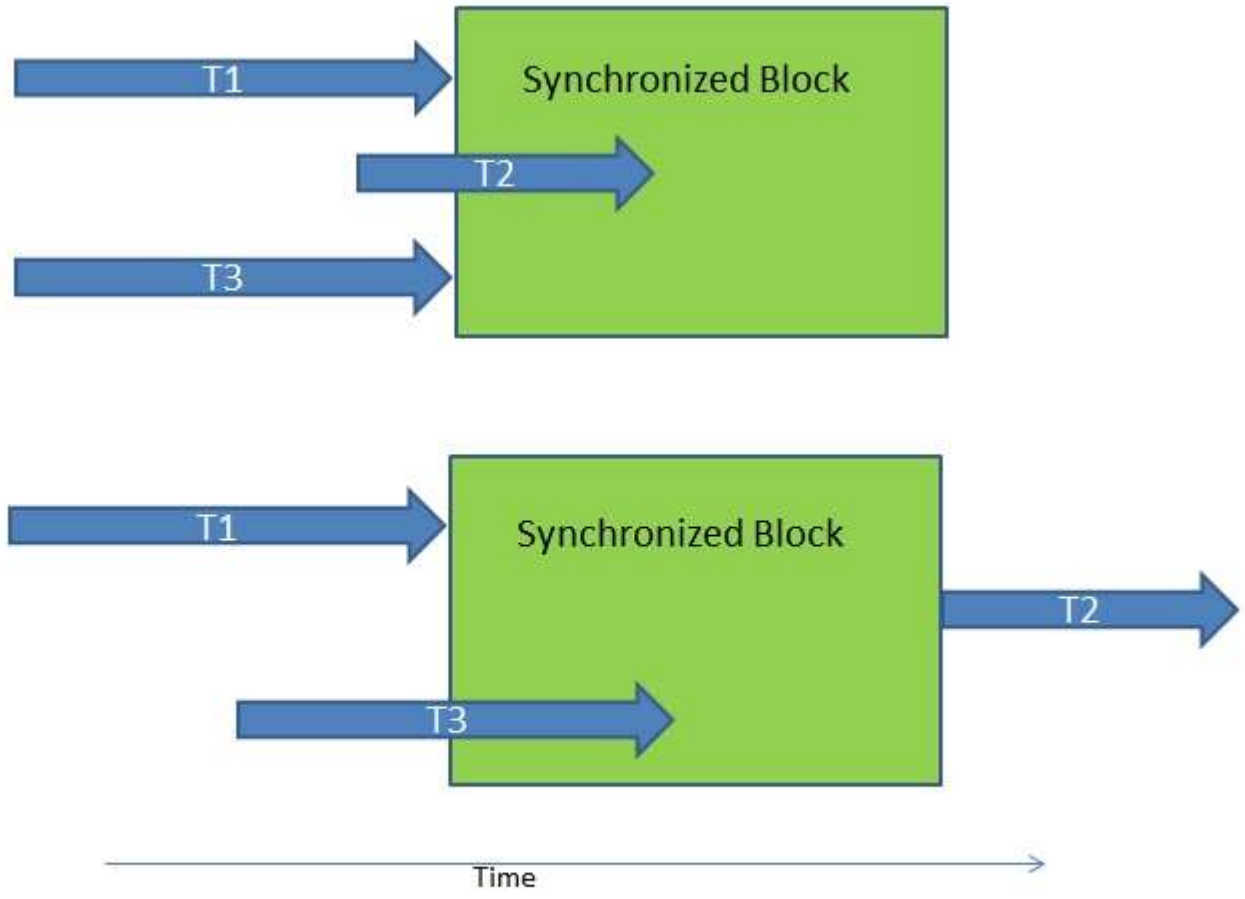


Race Condition!

Synchronization

The fundamental source of races is that critical regions are a window during which correct program behavior requires that threads do not interleave execution. To prevent race conditions, then, the programmer needs to synchronize access to that window, ensuring ***mutually exclusive access*** to the critical region.

In computer science, we say that an operation (or set of operations) is **atomic** if it is indivisible, unable to be interleaved with other operations. To the rest of the system, an atomic operation (or operations) appears to occur instantaneously. And that's the problem with **critical regions**: they are not indivisible, they don't occur instantaneously, they **aren't atomic**.



In Synchronized Block, other threads will have to wait when one thread is in

Mutexes - mutually exclusive access

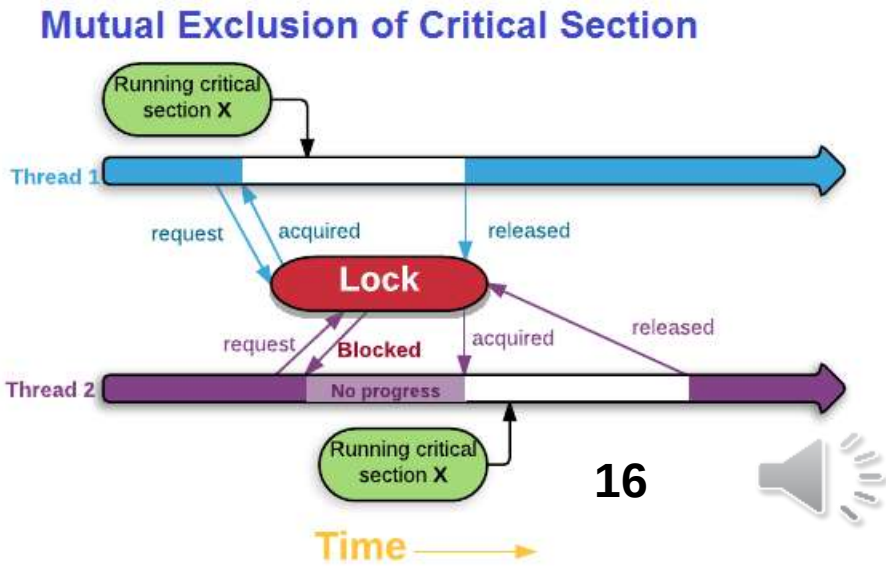
Only one person (thread) can enter the room (resource) if he has a key (lock).
There is only one key.
So at the same time only one thread can get the resource.

```
while ( Lock != 0 ) ;
Lock = 1
```
Entry Section

Critical Section

```
Lock = 0
```
Exit Section

```
1  int withdraw (struct account *account, int amount) {
2      lock ();
3      const int balance = account->balance;
4      if (balance < amount) {
5          unlock ();
6          return -1;
7      }
8      account->balance = balance - amount; unlock ();
9      disburse_money (amount); return 0;
10 }
```

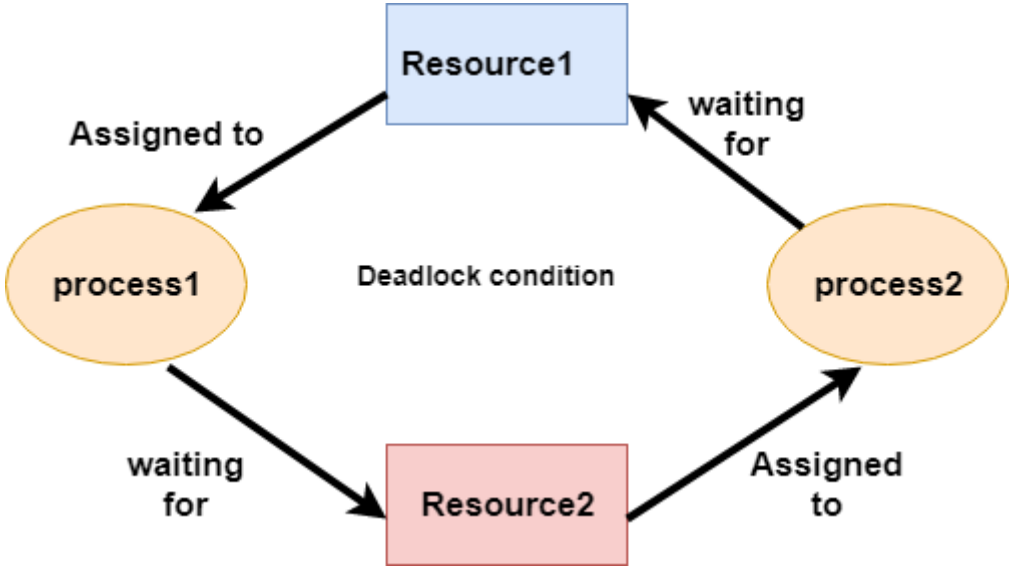


Deadlocks

A **deadlock** is a situation in which two threads are waiting for each other to finish, and thus neither does. In the case of mutexes, a deadlock occurs if two threads are both waiting for a different mutex, which the other thread holds.

A degenerate case is when a single thread is blocked, waiting for a mutex that it already holds.

Debugging deadlocks is often tricky, as your program needn't crash. Instead, it simply stops making forward progress, as ever more of your threads wait for a day that will never come.



POSIX Threads

- **pthread** POSIX standard tells us how threads should work on UNIX OS
- ***important calls***:
  - **pthread_create** : Create a new thread; analogous to fork
  - **pthread_exit** : Terminating the calling thread. Analogous to exit , has return value
  - **pthread_join** : Wait for another thread to exit. Analogous to wait - the caller cannot proceed other thread exists
  - **pthread_yield** : Release the CPU, let another thread run.
  - **pthread_attr_init** : Create and initialize a thread's attributes. Contains info like priority of thread
  - **pthread_attr_destroy** : Remove a thread's attributes. Free up the memory holding the attribute info.
  - **pthread_cancel** : Signal cancellation to a thread. Can be asynchronous or deferred, depending on thread attributes

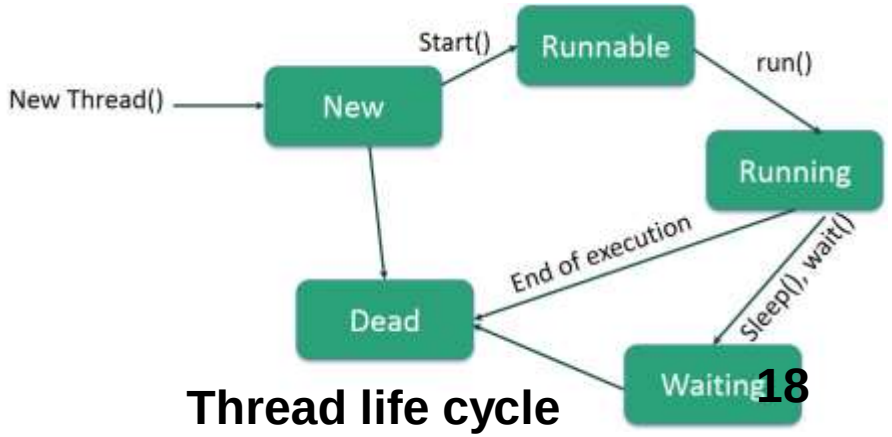
The **pthread** API is defined in <pthread.h>. Every function in the API is prefixed by pthread_. For example, the function to create a thread is called pthread_create(). **Pthread functions** may be broken into two large groupings:

Thread management

Functions to create, destroy, join, and detach threads.

Synchronization

Functions to manage the synchronization of threads, including mutexes, condition variables, and barriers.



Thread life cycle

Creating Threads

When your program is first run and executes the main() function, it is single threaded. Indeed, other than the compiler enabling some thread safety options and the linker linking in the Pthreads library, your process isn’t any different from any other. From this initial thread, sometimes called the default or master thread, you must create one or more additional threads to become multithreaded.

Pthreads provides a single function to define and launch a new thread, pthread_ create():

```
1  #include <pthread.h>
2  int pthread_create (pthread_t *thread,
3                     const pthread_attr_t *attr,
4                     void *(*start_routine) (void *),
5                     void *arg);
```


Terminating Threads

Threads may terminate under several circumstances, all of which have analogues to process termination:

- *If a thread returns from its start routine, it terminates. This is akin to "falling off the end" of main().*

- *If a thread invokes the pthread_exit() function (discussed subsequently), it ter- minates. This is akin to calling exit().*

- *If the thread is canceled by another thread via the pthread_cancel() function, it terminates. This is akin to being sent the SIGKILL signal via kill().*

Terminating yourself

The easiest way for a thread to terminate itself is to “fall off the end” of its start routine. Often you want to terminate a thread deep in a function call stack, far from your start routine. For those cases, Pthreads provides pthread_exit(), the thread equivalent of exit():

```
1  #include <pthread.h>
2  void pthread_exit (void *retval);
```

Upon invocation, the calling thread is terminated. retval is provided to any thread waiting on the terminating thread’s death, again similar to exit(). There is no chance of error.

```
1  /* Goodbye, cruel world! */
2  pthread_exit (NULL);
```

Terminating others

Pthreads calls the termination of threads by other threads *cancellation*. It provides the pthread_cancel() function.

On success, pthread_ cancel() returns zero.

On error, pthread_cancel() returns ESRCH, indicating that thread was invalid.

```
1  #include <pthread.h>
2
3  int pthread_cancel (pthread_t thread);
```

Threads can change their state via pthread_set cancelstate():

```
1  #include <pthread.h>
2  int pthread_setcancelstate (int state, int *oldstate);
```

Threads can change their type via pthread_setcanceltype():

```
1  #include <pthread.h>
2  int pthread_setcanceltype (int type, int *oldtype);
```


Terminating others

Let’s consider an example of one thread terminating another. First, the to-terminate thread enables cancellation and sets the type to deferred.

```
int unused;
int ret;
ret = pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, &unused);
if (ret) {
    errno = ret;
    perror ("pthread_setcancelstate");
    return -1;
}
ret = pthread_setcanceltype (PTHREAD_CANCEL_DEFERRED, &unused);
if (ret) {
    errno = ret;
    perror ("pthread_setcanceltype");
    return -1;
}
/*--------------------*/
int ret;
/* `thread' is the thread ID of the to-terminate thread */
ret = pthread_cancel (thread); if (ret) {
    errno = ret;
    perror ("pthread_cancel");
    return -1;
}
```


Joining threads

Joining allows one thread to block while waiting for the termination of another:

```c
#include <pthread.h>

int pthread_join (pthread_t thread, void **retval);
```

```c
int ret;
/* join with `thread' and we don't care about its return value */
ret = pthread_join (thread, NULL);
if (ret) {
    errno = ret;
    perror ("pthread_join");
    return -1;
}
```


Detaching threads

By default, threads are created as *joinable*. Threads may, however, *detach*, rendering them no longer joinable. Because threads consume system resources until joined, just as processes consume system resources until their parent calls wait(), threads that you do not intend to join should be detached.

```
1  #include <pthread.h>
2
3  int pthread_detach (pthread_t thread);
```


Initializing mutexes

Mutexes are represented by the pthread_mutex_t object. Like most of the objects in the Pthread API, it is meant to be an opaque structure provided to the various mutex interfaces. Although you can dynamically create mutexes, most uses are static:

```
1  #include <pthread.h>
2
3  /* define and initialize a mutex named `mutex' */
4  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
5
```

This snippet defines and initializes a mutex named mutex. That is all we have to do to start using it.

Locking mutexes

Locking (also called acquiring) a Pthreads mutex is accomplished via the pthread_mutex_lock() function:

```
1  #include <pthread.h>
2
3  int pthread_mutex_lock (pthread_mutex_t *mutex);
4
```

A successful call to pthread_mutex_lock() will block the calling thread until the mutex pointed at by mutex becomes available. Once available, the calling thread will wake up and this function will return zero. If the mutex is available on invocation, the function will return immediately.

Unlocking mutexes

The counterpart to locking is unlocking, or releasing, the mutex.

```
1  #include <pthread.h>
2
3  int pthread_mutex_unlock (pthread_mutex_t *mutex);
4
```

A successful call to pthread_mutex_unlock() releases the mutex pointed at by mutex and returns zero. The call does not block; the mutex is released immediately.

Conditional variable

Synchronization mechanisms need more than just mutual exclusion; also need a way to wait for another thread to do something (e.g., wait for a character to be added to the buffer)

- `wait(condition, lock)`: release lock, put thread to sleep until `condition` is signaled; when thread wakes up again, re-acquire lock before returning.
- `signal(condition, lock)`: if any threads are waiting on `condition`, wake up one of them. Caller must hold `lock`, which must be the same as the `lock` used in the `wait` call.
- `broadcast(condition, lock)`: same as `signal`, except wake up all waiting threads.
- Note: after `signal`, signaling thread keeps lock, waking thread goes on the queue waiting for the lock.
- Warning: when a thread wakes up after `cond_wait` there is no guarantee that the desired condition still exists: another thread might have snuck in.

Wait

The **pthread_cond_wait()** routine always returns with the mutex locked and owned by the calling thread, even when returning an error.

This function blocks until the condition is signaled. It atomically releases the associated mutex lock before blocking, and atomically acquires it again before returning.

```
1  // Prototype:
2  // int  pthread_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mutex);
3
4  #include <pthread.h>
5
6  pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
7  pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
8  while (!expr) {
9      int ret = pthread_cond_wait(&cv, &mp);
10 }
11
```


Signal

The *pthread_cond_signal*() function shall unblock at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).

```
1  // Prototype:
2  // int  pthread_cond_signal(pthread_cond_t *cv);
3
4  #include <pthread.h>
5
6  pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
7  int ret = pthread_cond_signal(&cv);
8
9
```


Broadcast

The *pthread_cond_broadcast*() function shall unblock all threads currently blocked on the specified condition variable *cond*.

```
1  // Prototype:
2  // int  pthread_cond_broadcast(pthread_cond_t *cv);
3
4  #include <pthread.h>
5
6  pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
7  int ret = pthread_cond_broadcast(&cv);
8
```


CAS

compare-and-swap (CAS) - it compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This is done as a single atomic operation

In the x86 (since 80486) and Itanium architectures this is implemented as the compare and exchange (CMPXCHG) instruction (on a multiprocessor the LOCK prefix must be used).

```
1
2  function cas(p : pointer to int, old : int, new : int) returns bool {
3      if *p ≠ old {
4          return false
5      }
6      *p ← new
7      return true
8  }
9
```


CAS

These functions enable a compare and swap operation to occur atomically. The value stored in target is compared with cmp. If these values are equal, the value stored in target is replaced with newval. The old value stored in target is returned by the function whether or not the replacement occurred.

```
1  #include <atomic.h>
2
3  uint8_t atomic_cas_8(volatile uint8_t *target, uint8_t cmp, uint8_t newval);
4  uchar_t atomic_cas_uchar(volatile uchar_t *target, uchar_t cmp, uchar_t newval);
5  uint16_t atomic_cas_16(volatile uint16_t *target, uint16_t cmp, uint16_t newval);
6  // ...
7  uint32_t atomic_cas_32(volatile uint32_t *target, uint32_t cmp, uint32_t newval);
8  // ...
9  void *atomic_cas_ptr(volatile void *target, void *cmp, void *newval);
```


CAS (example)

1. Reading old value
2. Performing a local increment
3. Trying to execute CAS
4. If we fail to perform CAS, we repeat

```
1
2  void atomic_inc(volatile uint32_t *target) {
3      uint32_t old = *target;
4      while (atomic_cas_32(target, old, old + 1) != old) {
5          old = *target;
6      }
7  }
8
```