


# Threads

Last Update: Oct 3, 2022

# Outline and Objectives

**Outline**

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Windows XP Threads
- Linux Threads
- Re-entrancy
- Thread specific data

Objectives

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming

# Threading Concept

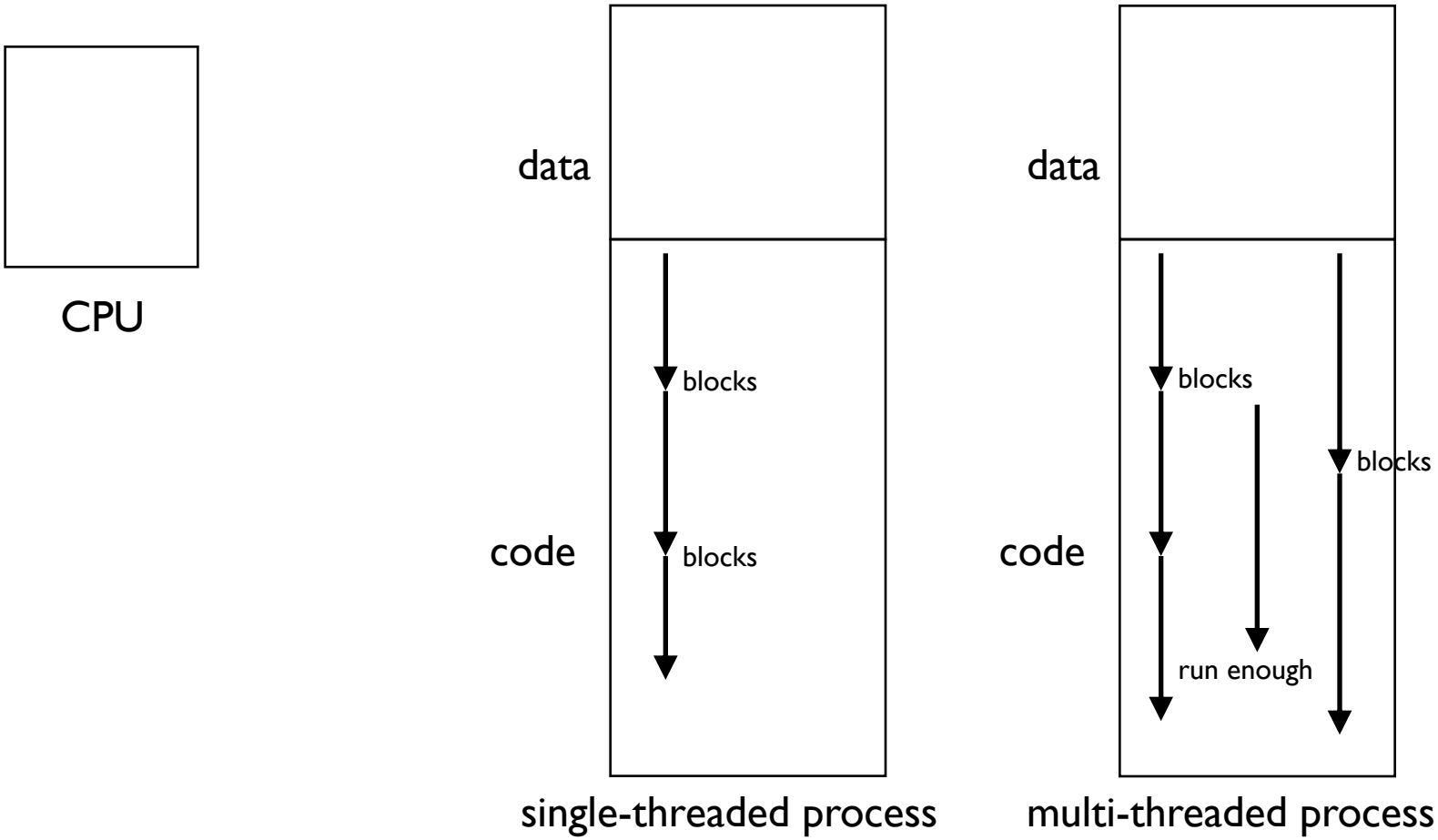
# Threads and thread usage

- A process has normally a single thread of control; that means a single stream (flow) of instructions executed sequentially.
- A process always has at least one thread (default thread or main thread).
- In a single-threaded process, if the thread (process) blocks, no activity can be done as part of the process.
  - Better would be being able to run multiple concurrent threads (tasks) as part of the same process.
- With multithreading we have that.
  - A process can have multiple threads running concurrently.
  - Each thread runs in a *sequential* manner.
  - Threads share code (instructions) and data (global variables) of the process.

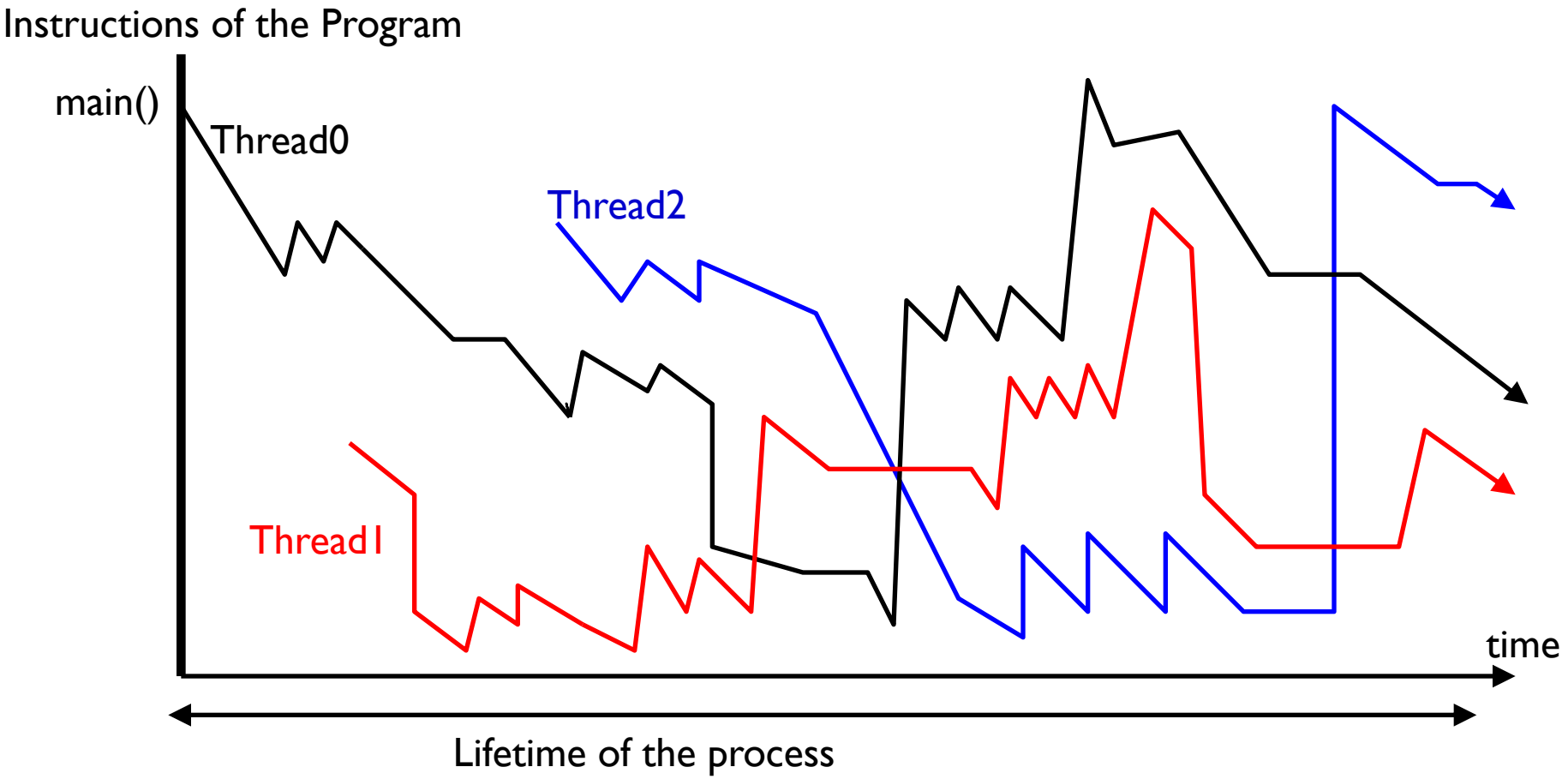
# Benefits of using multiple threads in an application (process)

- Responsiveness
  - If one thread blocks, another can run.
  - One thread may always wait for the user, for example.
- Easy resource/data sharing
  - Threads can easily share resources, global variables.
- Economy
  - Creating a thread is faster than creating a process.
  - Context switching among threads may be faster.
- Scalability
  - Multiple CPUs in a computer can be utilized by a process. Each thread running on another available CPU.

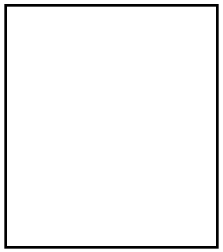
# Threads and thread usage



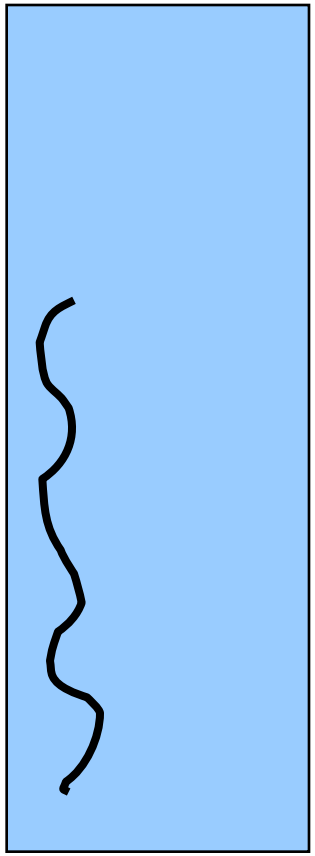
# a multithreaded process' execution flows: threads



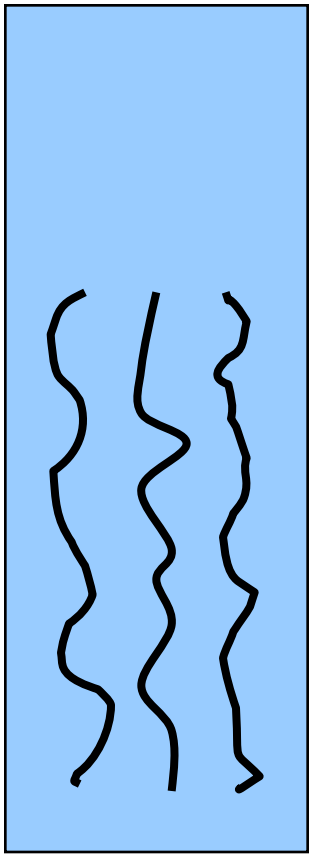
# Multithreading concept



CPU

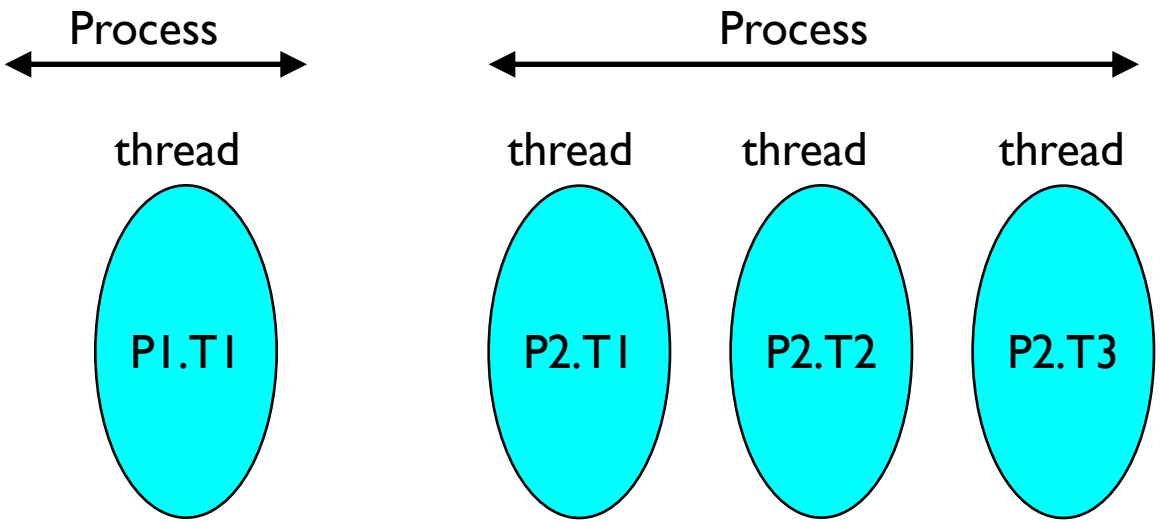


single-threaded process



multi-threaded process

# Multithreading concept

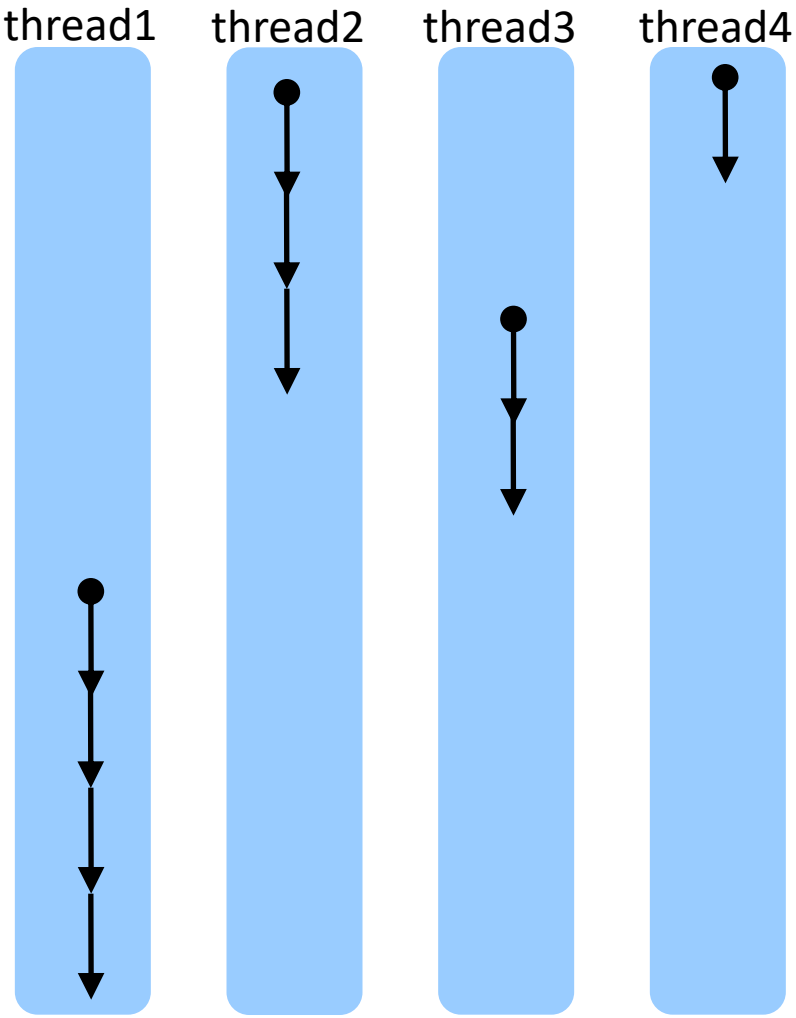



Schedulable Entities

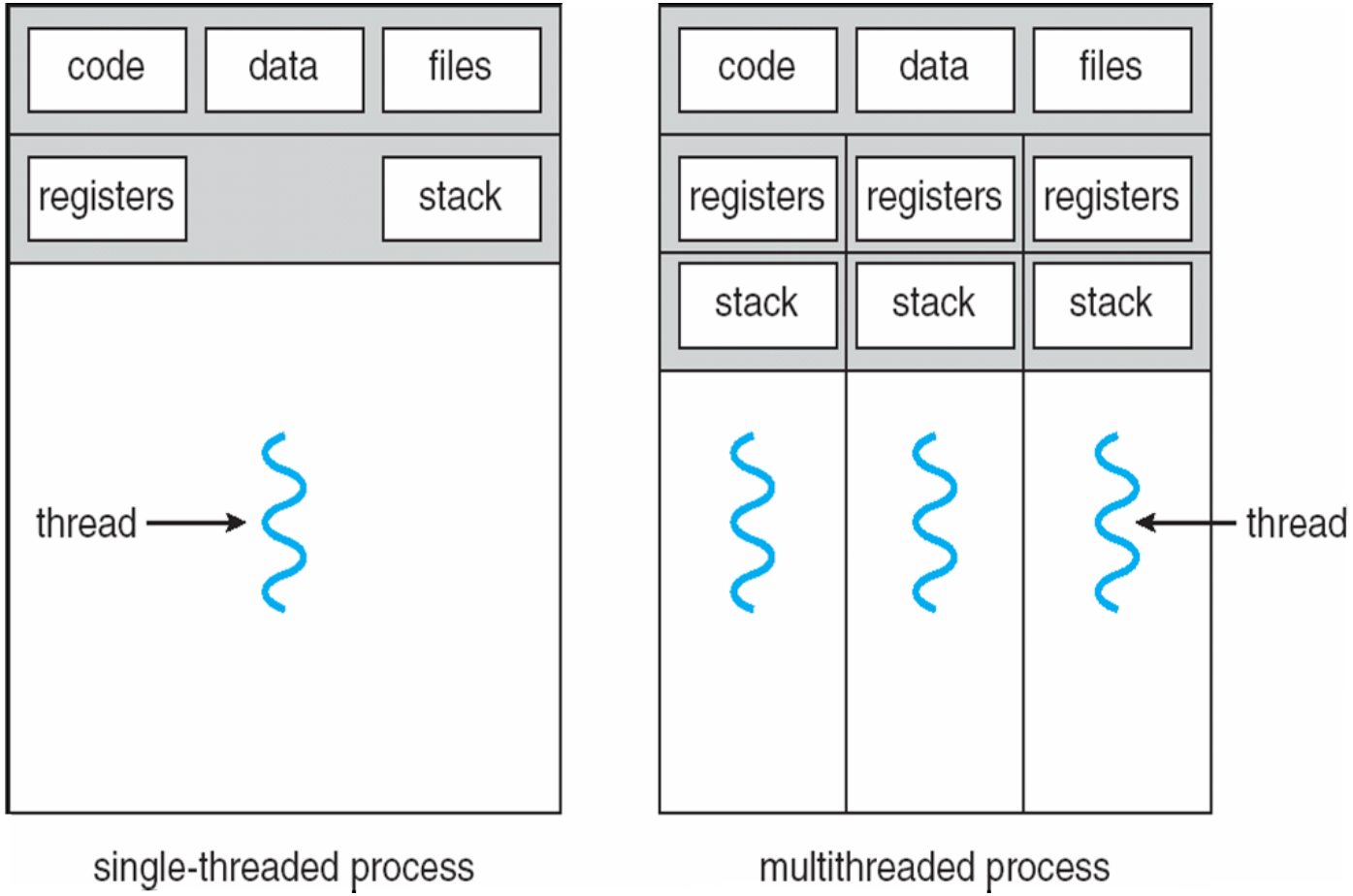
We can select one of them and run

# Multithreading concept

```
function1(…)
{
        ….
}

function2(…)
{
        ….
}
main()
{       ….
        thread_create (function1 ,…);
        ….
        thread_create (function2, …);
        ….
        thread_create (function1, …);
        ….
}
```



# Single and multithreaded processes

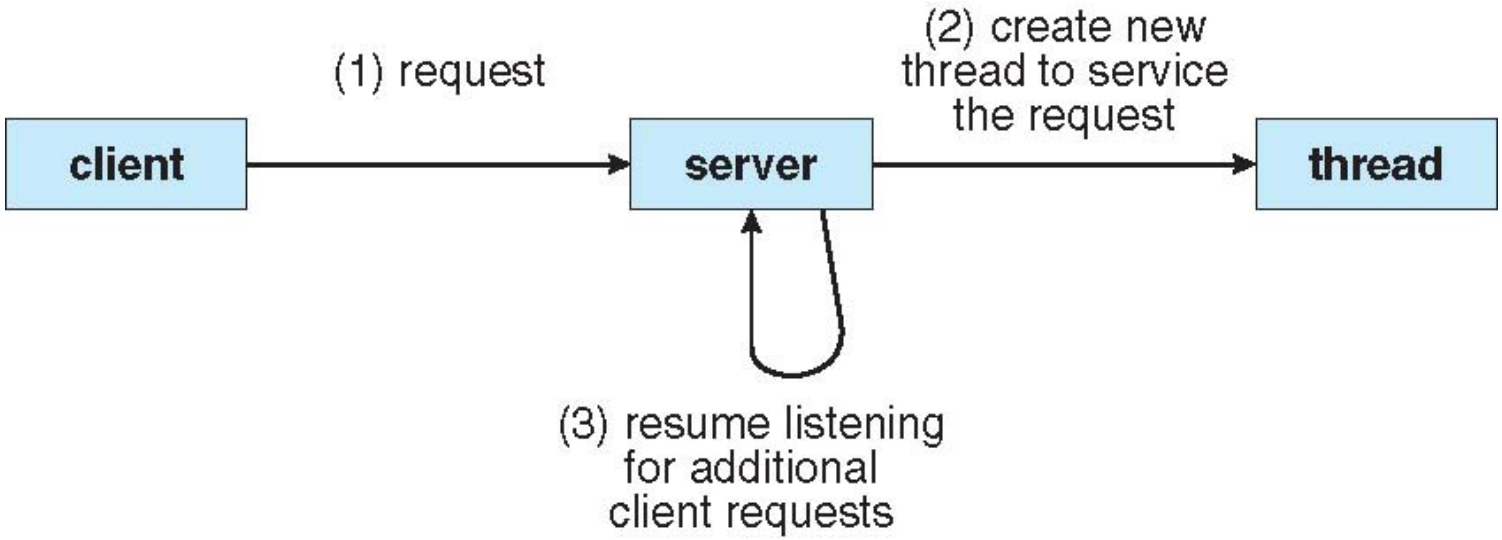


# Multicore programming and multithreading challenges

Threading challenges include

- Dividing activities: come up with concurrent tasks
- Balance: tasks should be of similar importance and load
- Data splitting: data may need to be split as well
- Data dependency: data dependencies should be considered; need synchronization of activities.
- Testing and debugging: Debugging a multithreaded program is more difficult.
  - Programmer should explicitly care about coordinated access to shared variables.

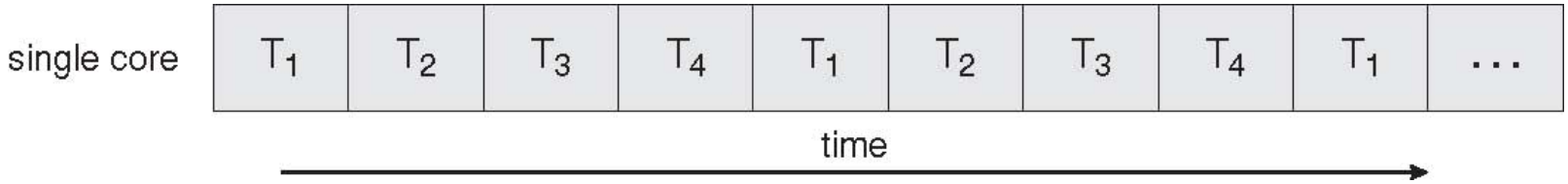
# Multithreaded Server Architecture




Using threads to implement a server is common.

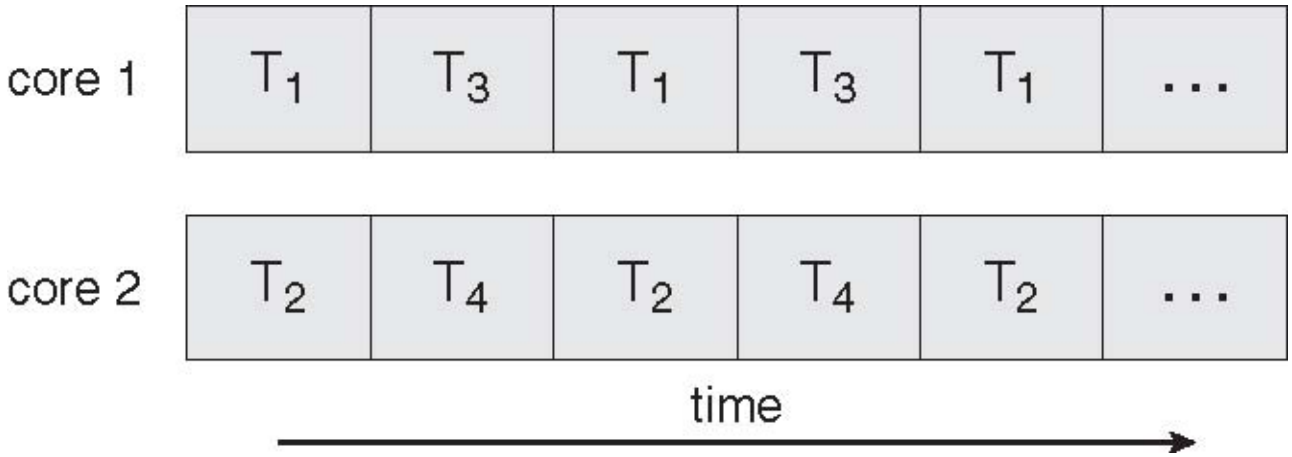
- Provides better performance (multicore machines).
- Natural structuring of the server.
- Ease of programming. A thread executes sequentially.

# Concurrent Execution on a Single-core System




- In a single CPU computer, multiple threads will share the same CPU.
- CPU switched among the threads frequently.
- We don't have physical parallelism, but we have concurrency (pseudo-parallelism).

# Parallel Execution on a Multicore System




- If there are multiple CPUs, some threads can run in parallel.
- All threads will run concurrently.

# Speedup: Amdahl's Law

- What is the potential performance gain (speed up) with multiple CPUs in system?
- A program may have both serial and parallelizable parts.
  - Serial part may not be parallelized.
- Assume the program executes and finishes in 1 time unit in a single CPU system. Then:
  - $S$: fraction of time to execute the serial portion ($0 \leq S \leq 1$) in a single-CPU system.
  - $1 - S$ : is the fraction of time to execute the parallelizable portion in a single-CPU system.

# Speedup: Amdahl's Law

- $N$ : number of processing cores.
  - $N$ tasks/threads can run in parallel.
  - Parallelizable portion of the program can finish $N$ times faster (at most).
- We define speed-up as the ratio of program completion time in a single-CPU system to the completion time in an $N$-CPU system.
- Then we have the following formula for the speed-up (called Amdahl's law).

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

# Threading Support

# Threading Support

- Multithreading can be supported by:
  - User level libraries (without Kernel being aware of it): library creates and manages threads (user-space implementation)
  - Kernel itself: kernel creates and manages threads (kernel-space implementation).
- No matter which method is used, threads can be created, used, and terminated by a program via a set of functions that are part of a Thread API (a thread library)
- Three are primary thread libraries: POSIX threads, Java threads, Win32 threads (for Windows 32 bit systems).

# Multithreading Models

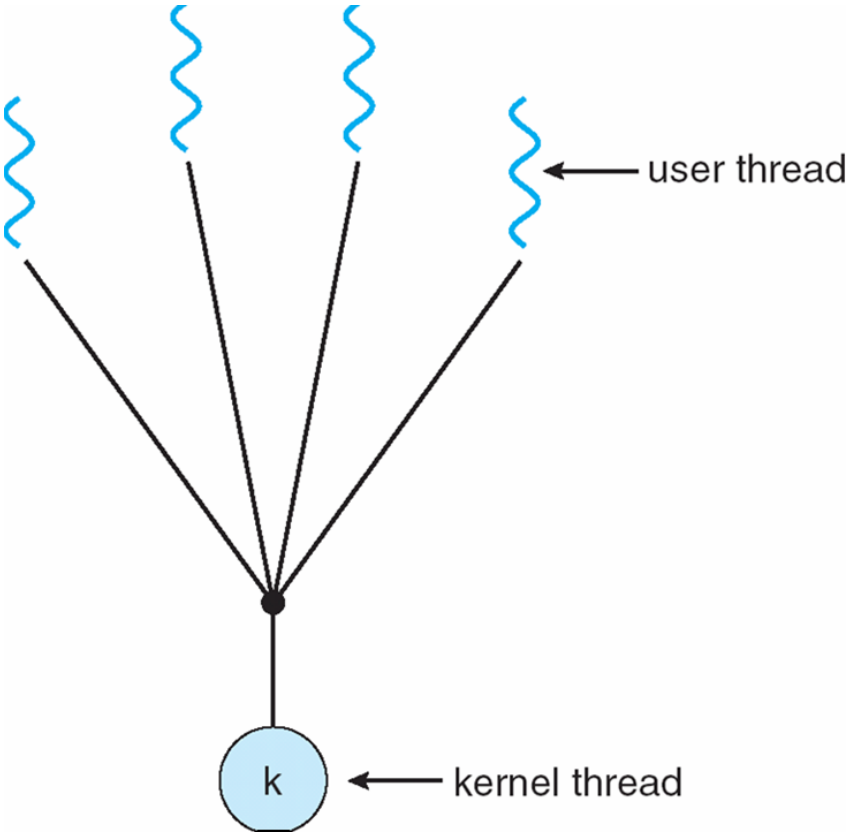
- A user process creates one or more threads ($N$ threads) by making thread API calls.
- Kernel can create one or more threads ($M$ threads) for the user process. Kernel creates at least one one thread (the process itself).
- A relationship must exist between user threads and kernel thread(s).
  - Mapping user-level threads to kernel-level threads.
- Three common ways of establishing such a relationship.
  - Many-to-One model.
  - One-to-One model.
  - Many-to-Many model.

# Many-to-One Model: Implementing Threads in User Space

- Many user-level threads

mapped to a single kernel thread

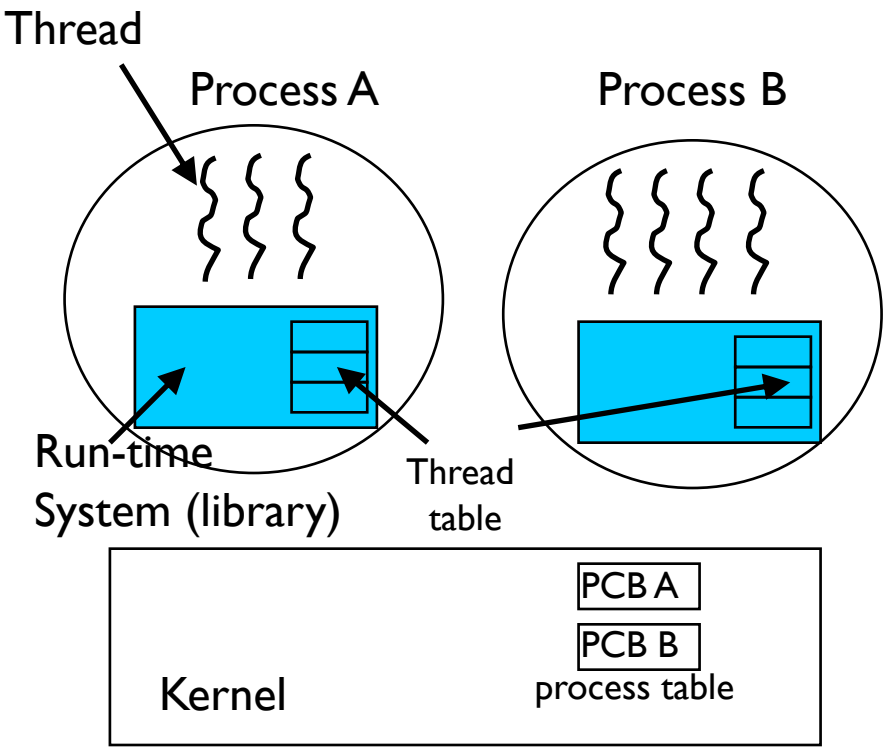
- Examples:
  - Solaris Green Threads.
  - GNU Portable Threads.
- Thread management done at user space, by a thread library.




Kernel supports process concept; not threading concept

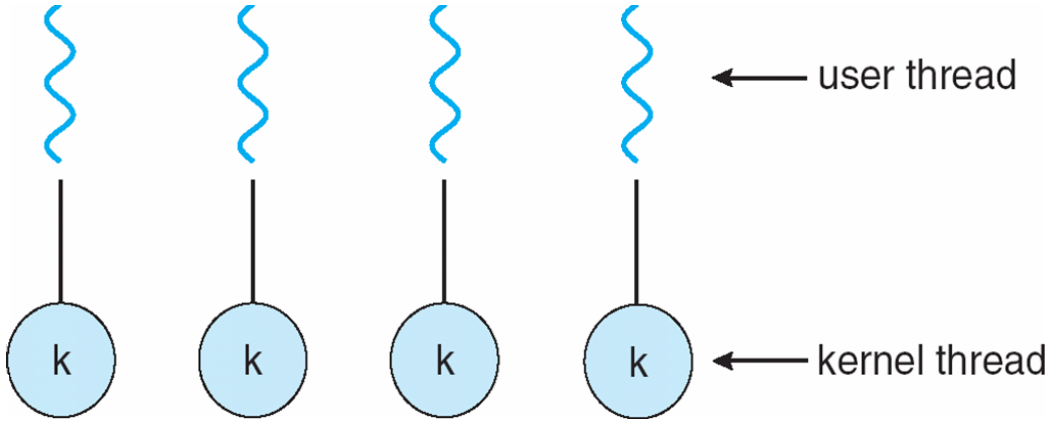
# Many-to-One Model: Implementing Threads in User Space

- No need for kernel support for multithreading (+)
- Thread creation is fast (+)
- Switching between threads is fast (+)
- A blocking system call blocks the whole process **(-)**
- A thread has to explicitly call a function to give the CPU to some other thread. Example: thread_yield().
- All threads will run in a single CPU. Will not utilize multiple CPUs. (-)



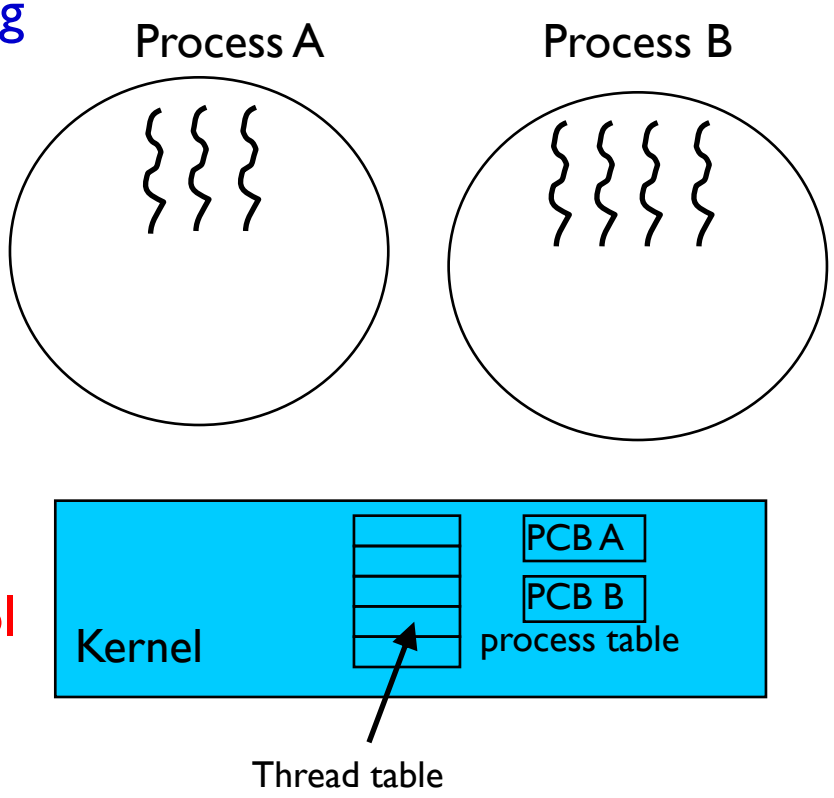
# One-to-One Model: Implementing Threads in Kernel Space

- Kernel implements threading and can manage threads, schedule threads.
- Examples (nearly all modern OSs): Windows, Linux, …
- All these kernels have threading support. They schedule processes and also threads.
- Each user-level thread maps to a single kernel thread



# One-to-One Model: Implementing Threads in Kernel Space

- Provides more concurrency; when a thread blocks, another can run. Blocking system calls are not problem anymore. (+)
- Multiple processors can be utilized (+).
- Kernel can stop a long running thread and run another thread. No need for explicit request from a thread to be suspended. (+)
- Need system calls to create and control threads. Making system call takes more time than a library call. (-)
- Thread switching is costly; (-)



# Threading API

# Thread Libraries

- Thread library provides programmers with an API for creating and controlling threads
- Programmer just have to know the thread library interface (API).
- Threads may be implemented in user space or kernel space.
- Thread API library may be entirely in user space or may get kernel support for threading.

# Pthreads Library

- Pthreads: POSIX (Portable Operating System Interface) threads
- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX-like operating systems
  - Solaris, Linux, Mac OS X.

# Pthreads Example

- We will show a program that creates a new thread, and in this way will have two threads:
  - the initial/main thread (i.e, the process)
  - the new thread (both threads have equal power)
- The program will just create the new thread to do a simple computation: will sum all integers up to a given parameter value $N$

$$sum = 1 + 2 + \cdots + N = \sum_{i=1}^{N} i$$

- The main thread will wait until sum is computed into a global variable.
- Then the main thread will print the result.

# Pthreads Example

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

int sum;      /* shared sum by threads - global variable */
void *runner (void *param);   /* thread start function */
```

# Pthreads Example

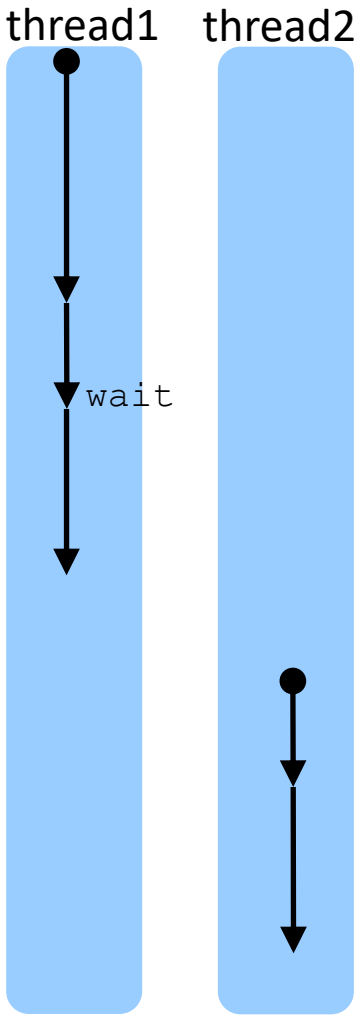
```
int  main(int argc, char *argv[]){
       pthread_t tid;          /* id of the created thread */
       pthread_attr_t attr;  /* set of thread attributes */

       if (argc != 2) {
              fprintf (stderr, “usage: a.out <value>\n”);
              return -1;
       }
       if (atoi(argv[1]) < 0) {
              fprintf (stderr, “%d must be >= 0\n”, atoi(argv[1]);
              return -1;
       }

       pthread_attr_init (&attr);
       pthread_create (&tid, &attr, runner, argv[1]);
       pthread_join (tid, NULL);
       printf (“sum = %d\n”, sum);
}
```

# Pthreads Example

```
void *runner (void *param)
{
        int i;

        int upper;

        upper = atoi(param);
        sum = 0;

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

        pthread_exit(0);
}
```

# Pthreads Example

```
int main(…)
{

        …
        ….
        pthread_create(&tid,…,runner,..);

        pthread_join(tid);

.       printf (…, sum, …);
}

runner  (…)
{
        ….
        sum = …
        pthread_exit();
{
```



# Compiling and running the program

- We can put the above code into a `.c` file, say `mysum.c`
- To use Pthreads functions, we need to include the header file `pthread.h` in our program.
- We also need to link with the pthread library (the Pthreads API functions are not implemented in the standard C library). The way to do that is using the `-l` option of the C compiler. After `-l` you can provide a library name like `pthread`.
- We can compile+link our program as follows:
```
gcc -Wall -o mysum -lpthread  mysum.c
```
- Then we run it as:
```
./mysum 6
```
- It will print out `21`

# Windows Threads

```
/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

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

# Java Threads

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

# From Single-threaded to Multithreaded

- Many programs are written as a single threaded process.
- If we want to convert a single-threaded program to a multi-threaded program, we need to be careful about the following:
  - the global variables
  - the library functions we use

# From Singlethread to Multithreaded

```
int status;

func1(…) {
        ….
        status = …
        do_something_based_on(status);
}

func2(…) {
        …
        status = …
        do_something_based_on(status);
}
main() {
        ….
        thread_create(…, func1, …);
        thread_create(…, func2, …);
}
```

- We can have problem here.
- Just after func1 of thread 1 updated status, a thread switch may occur and 2nd thread can run and update status.
- Then thread 1 will run again, but will work with a different status value.

  Wrong result!

# Thread-safe / Reentrant libraries

- Many library procedures may not be reentrant.
  - They are not designed to have a second call to itself from the same process before it is completed (not re-entrant).
  - They may be using global variables. Hence may not be thread-safe.
- We have to be sure that we use thread-safe (reentrant) library routines in multi-threaded programs we are developing.

# From Single-threaded to Multithreaded

- Scope of variables:
  - Normally we have: global, local
  - With threads, we want: global, local, and thread-specific (thread wide)
- thread-specific: global inside the thread (thread-wide global), but not global for the whole process. Other threads can not access it. *But all functions of the thread can access.*
- But we may not have language support to define such variables.
  - C can not do that. Normally in C we have just global variables and local variables. We do not have thread-wide (thread-local or thread-global) variables.
- Therefore thread API has special functions that can be used to create such variables – data. This is called thread specific data.

# Thread Specific Data (TSD)

- Allows each thread to have its own copy of data.
- Each thread refers to the data with the same name (key).

# Thread Specific Data

- In POSIX Pthreads Library we have the following functions to create thread specific data that has an associated name (key).

pthread_key_create()
pthread_key_delete()

pthread_setspecific()
pthread_getspecific()

- key is seen by all threads.
- in each thread, key is associated with thread specific data.

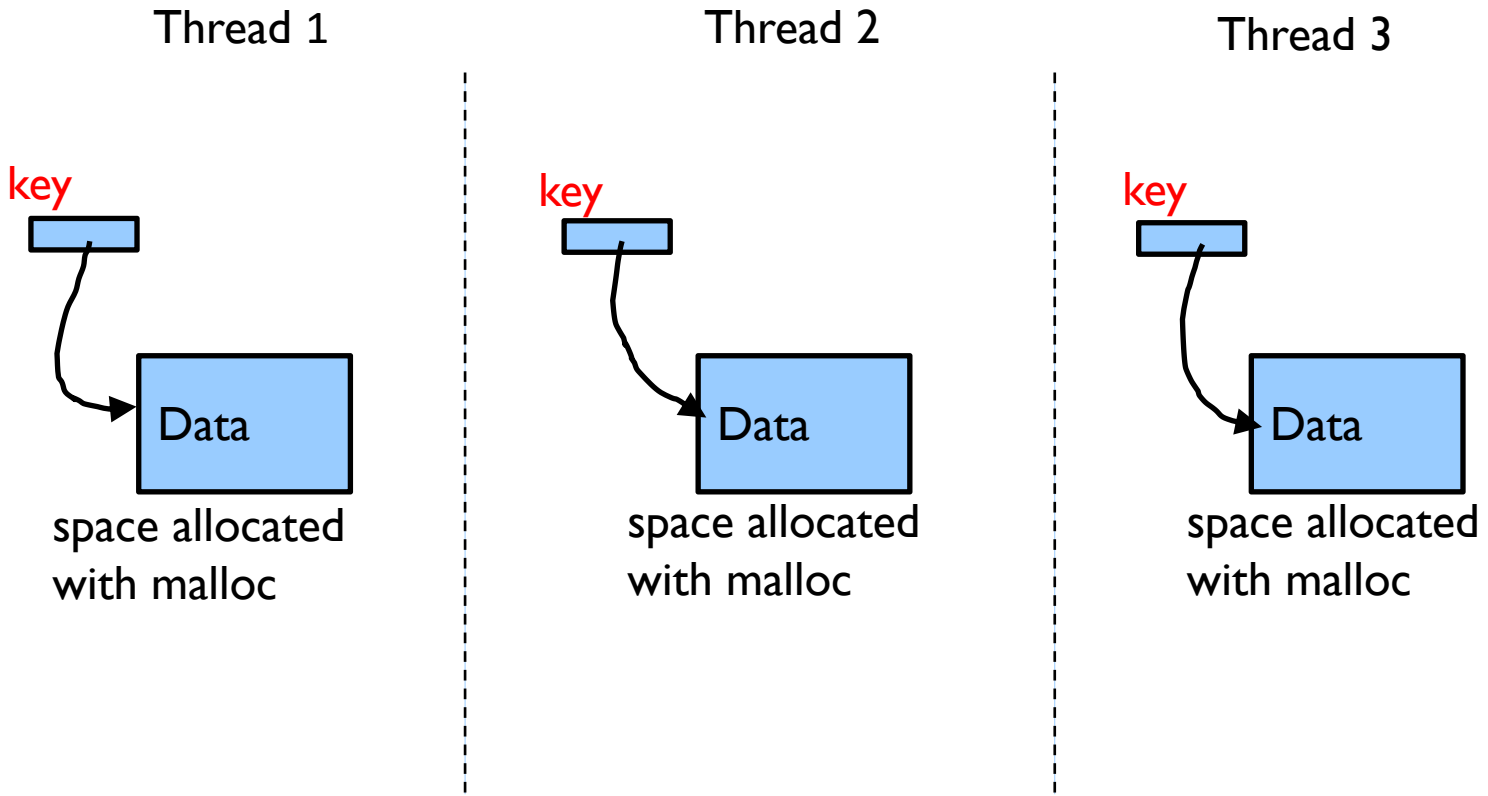
Example:

```
pthread_key_t  key1;
pthread_key_create (&key1,  …);
```

do this once

```
// In each thread we can do the following:
char *buf = malloc(1024);
pthread_set_specific (key1, buf); // associate buf (buffer pointer) with key1
buf = pthread_get_specific (key1);  // get pointer to access data
```

# Thread Specific Data




Each thread is using the same variable name (i.e., key) to get access to *its own specific data*.

# Examples from Operating Systems

# Operating System Examples

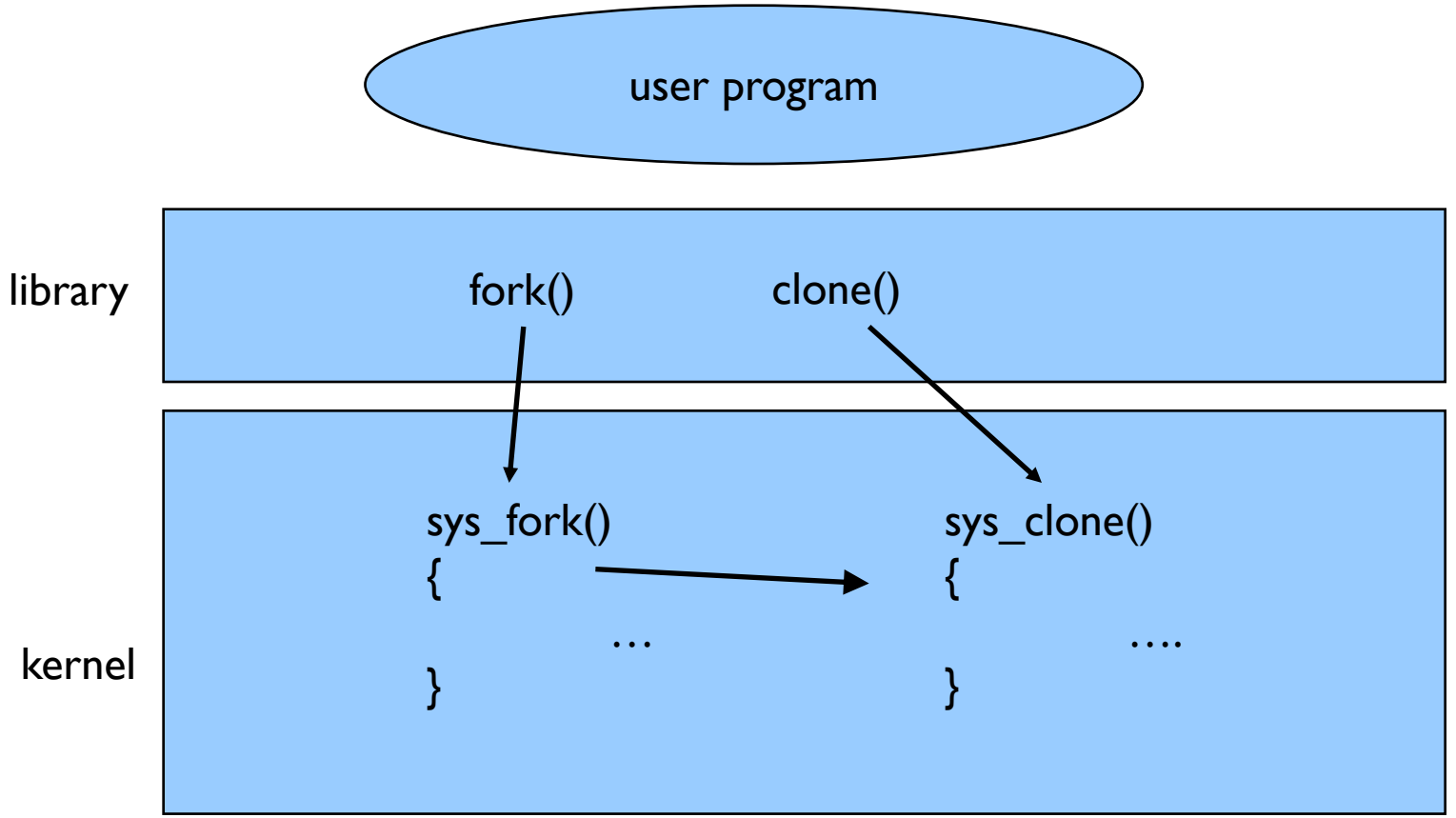
- Windows XP Threads
- Linux Threads

# Linux Threads

- 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).

| 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. |

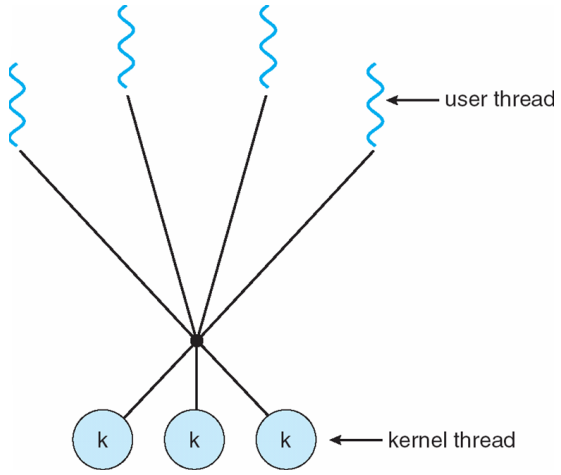
# Clone() and fork()



# Many-to-Many Model & Two-level Model

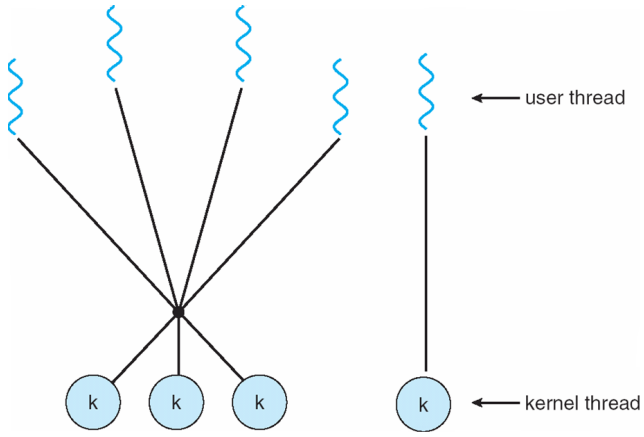
**Many-to-Many Model**

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package




**Two-level Model**

- Similar to M:M, except that it allows a user thread to be **bound** to a kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



# Implicit Threading

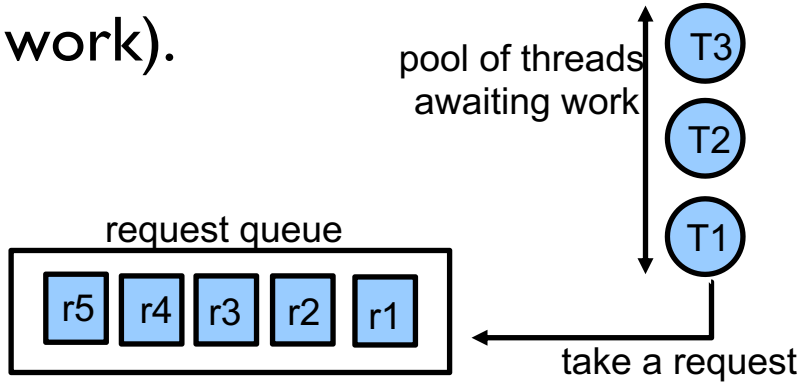
# Implicit Threading

- Applications can create hundreds or thousands of threads.
  - Can be creating *explicitly*. (App programmer worries)
  - Can be created by compiler, library, platform, etc. (easier for App programmer) - called *Implicit* Threading.

- Some alternatives for implicit threading
  - Thread Pools
  - OpenMP (Compiler Supported)
  - Grand Central Dispatch (in Mac OS X)

# Thread Pools

- Create a number of threads in a pool where they wait for work.
- Advantages: 1) faster to start a thread. 2) enables limiting the number of threads application is using – pool size)
- Incoming task requests are queued.
- An available thread takes a request and processes it.
- When done, the busy thread becomes availabla again (waits for work).



# OpenMP

- Is a set of compiler directives as well as an API for programs in C and C++.
- Parallel programming in shared memory environments.
- Parallel regions identified by programmer.
- Examples:

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

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

Number of threads == Number of cores

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
  c[i] = a[i] + b[i];
}
```

# Grand Central Dispatch (GCD)

- A combination of extensions to the C language, and API, and a run-time library that allows app developers to identify sections of code to run in parallel.
- Programmer identifies the region to be parallel:

```
^{ printf("I am a block"); }
```

- GCD schedules such blocks  for run-time execution by placing them in a dispatch queue (serial or concurrent).
- Blocks can run in parallel in different cores.

# Other Threading Issues

# Threading Issues

- Semantics of fork() and exec() system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

# Semantics of fork() and exec()

- Does fork() duplicate only the calling thread or all threads?
- How should we implement fork?
- logical thing to do is:
  - 1) If exec() will be called after fork(), there is no need to duplicate the threads. They will be replaced anyway.
  - 2) If exec() will not be called, then it is logical to duplicate the threads as well; so that the child will have as many threads as the parent has.
- So we may implement two system calls: like fork1 and fork2!

# Thread Cancellation

- Terminating a thread before it has finished
  - Can be asynchronous or deferred.
- Asynchronous cancellation terminates the target thread immediately
- Deferred cancellation allows the *target thread* to periodically check if it should be cancelled
  - Canceller thread indicates a target thread to be cancelled.
  - Target threads performs checks at cancellation points and if it is safe gets terminated.

# Thread Cancellation

Cancelling thread

```
pthread_t  tid;

pthread_create (&tid, 0, worker, NULL);
…
/* cancel the thread */
pthread_cancel(tid);  /* requesting cancellation */
```

cancellation modes

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

## Deferred cancellation

Target thread

```
while (1) {
        /* do some work */
        …
        /* check if there is cancel request */
}       pthread_testcancel();
```

mode can be set with pthread_setcancelstate()

# Signal Handling

- If a signal is sent to a Multithreaded Process, who will receive and handle that?
- In a single threaded process, it is obvious.
- In a multi-threaded process, the options are:
  - deliver signal thread to which signal applies
  - deliver signal to all threads of the process
  - deliver signal a specific thread responsible for handling signals

# Signal Handling

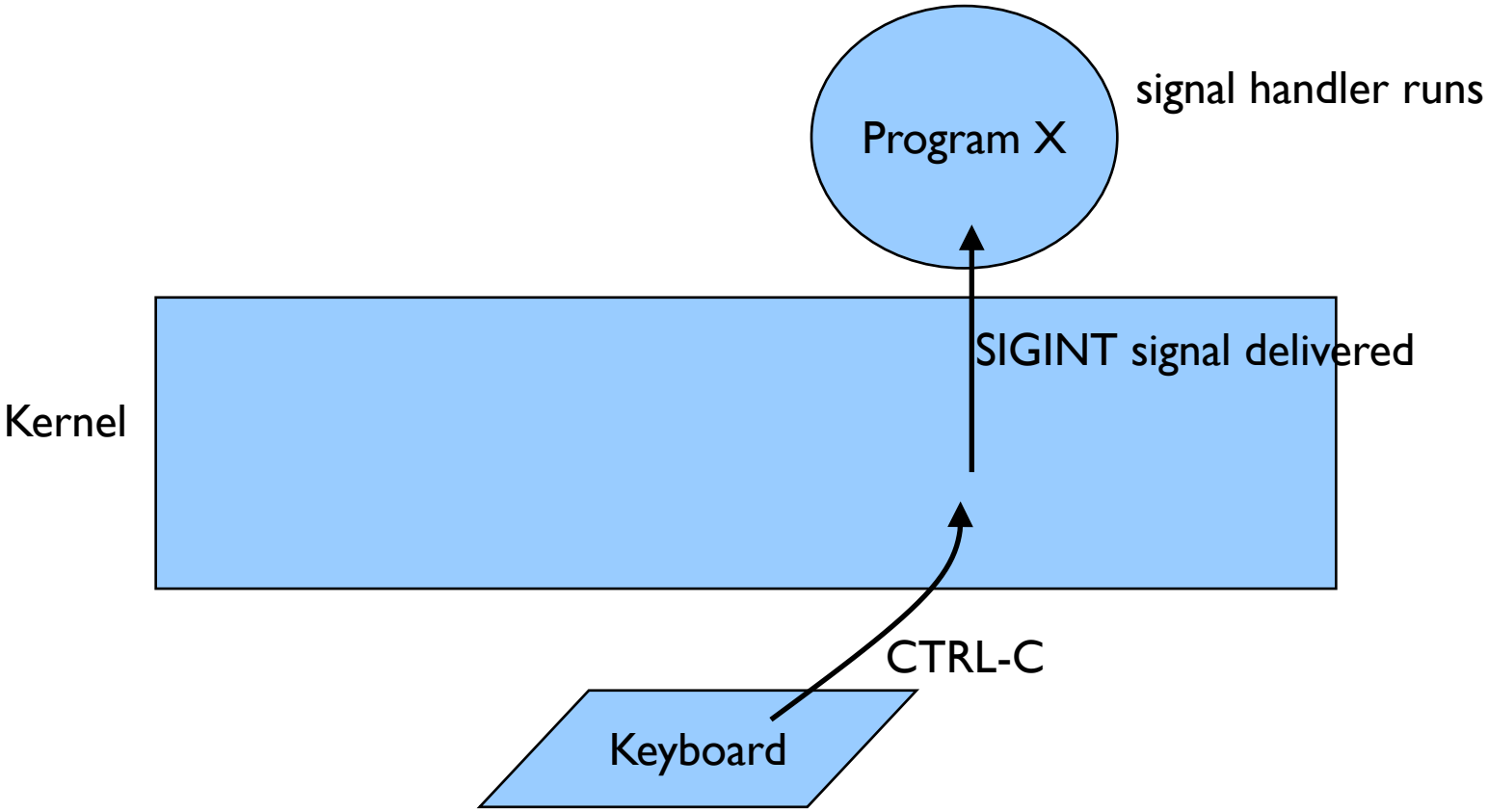
- Signals are used in Unix systems to notify a process that a particular event has occurred
  - a signal is generated by a particular event or a process
  - A signal is delivered to a process (same or different process
  - a signal is handled
- A signal handler is used to process signals
  - In kernel (default handler)
  - In process (if process specified a handler)
- Handled asynchronously

# a C program using signals

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

static void sig_int_handler() {
     printf("received SIGINT signal.\n");
     fflush(stdout);
     exit(0);
}

int main() {
     signal (SIGINT, sig_int_handler);

     while (1)
         ;
}
```

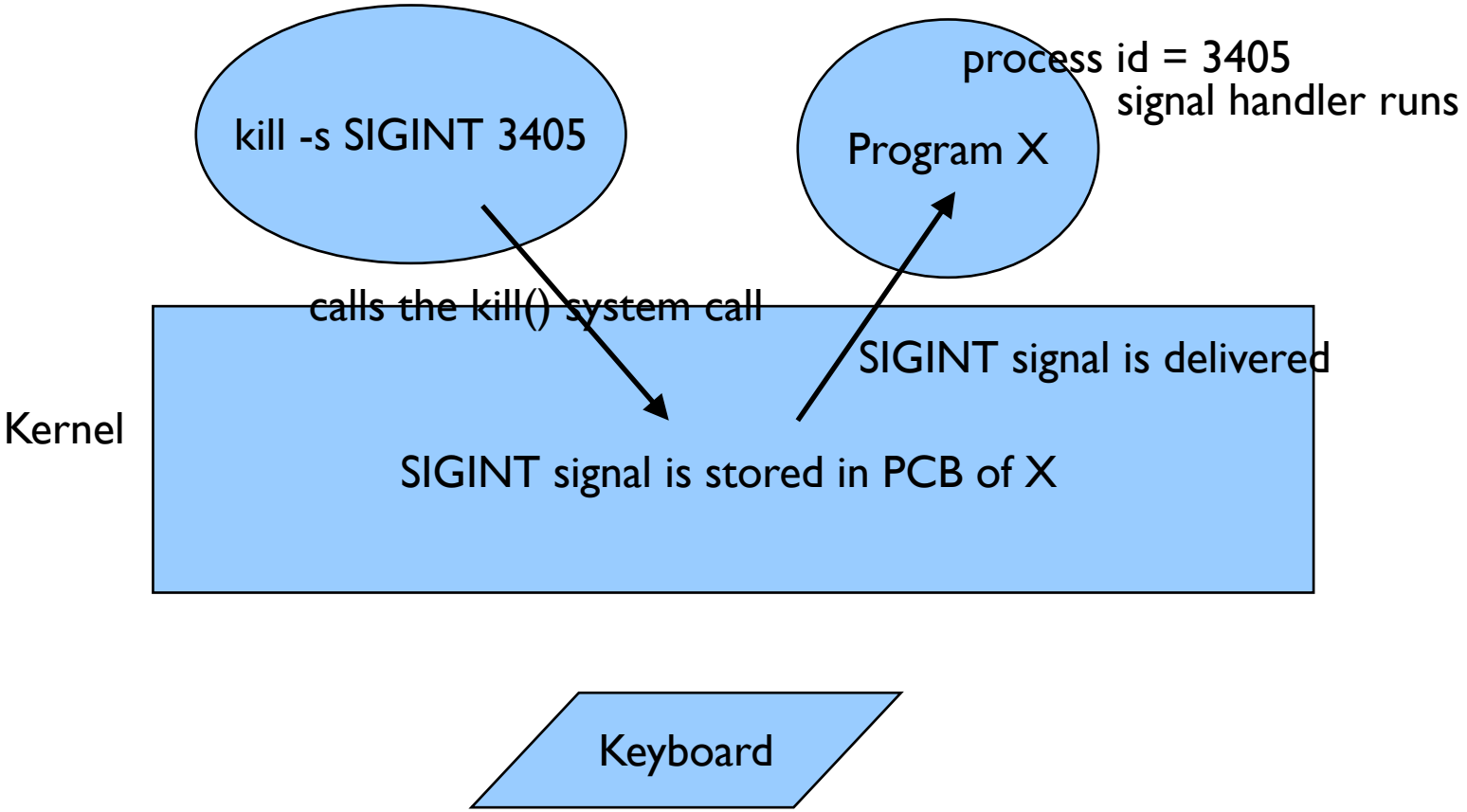
- While a program is running, if we press CTRL-C, we are sending a SIGINT signal to the program.
- By default, SIGINT is handled by kernel. Kernel terminates the program.
- But if we specify a handler function (shown left), then our program can handle it.
- Kernel will notify our process with a signal when user presses CTRL-C.

# delivering signal (notifying)



# Sending a signal

In Unix, we have the kill program that can be used to send a signal to a process (uses the kill() system call).



# Some Signals

| | |
|---|---|
| SIGABRT | Process abort signal. |
| **SIGALRM** | Alarm clock. |
| SIGBUS | Access to an undefined portion of a memory object. |
| SIGCHLD | Child process terminated, stopped, or continued. |
| SIGCONT | Continue executing, if stopped. |
| SIGFPE | Erroneous arithmetic operation. |
| SIGHUP | Hangup. |
| SIGILL | Illegal instruction. |
| **SIGINT** | Terminal interrupt signal. |
| SIGKILL | Kill (cannot be caught or ignored). |
| SIGPIPE | Write on a pipe with no one to read it. |
| SIGQUIT | Terminal quit signal. |
| SIGSEGV | Invalid memory reference. |
| SIGSTOP | Stop executing (cannot be caught or ignored). |
| SIGTERM | Termination signal. |

# Additional Study Material

# Scheduler Activations

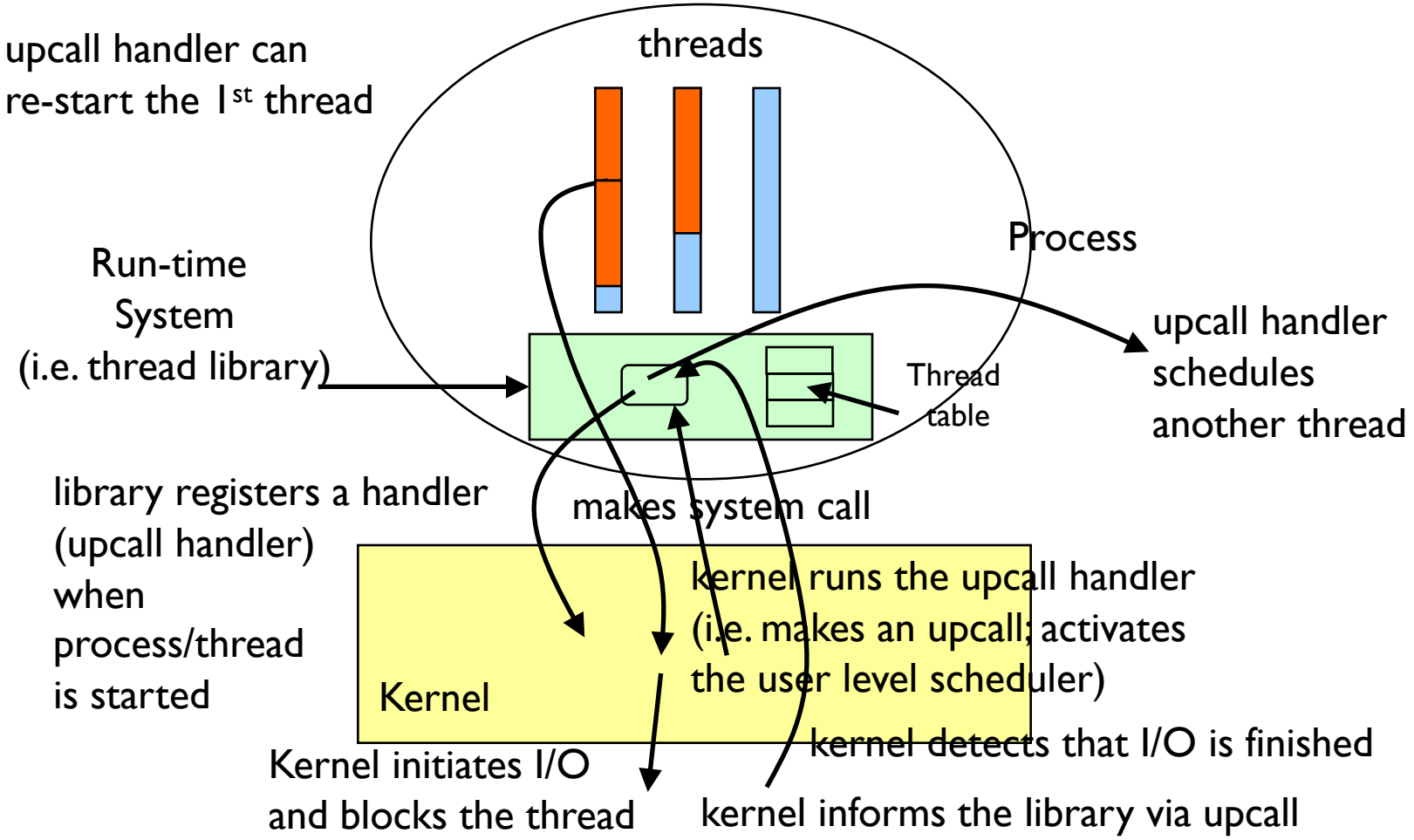
- Kernel threads are good, but they are slower if we create short threads too frequently, or threads wait for each other too frequently.

- Is there a middle way?
  - Schedule Activation

- Goal is to mimic kernel threads at user level with some more kernel support. But kernel will not create another thread for each user thread (M:1 or M:M model).

- Avoid unnecessary transitions between user and kernel space.

# Scheduler Activations: Upcall mechanism



# References


- Operating System Concepts, Silberschatz et al. Modern Operating Systems, Andrew S. Tanenbaum et al.