BILKENT UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CS342 OPERATING SYSTEMS

LECTURE 4
THREADS

(CHAPTER 4)

Dr. İbrahim Körpeoğlu
http://www.cs.bilkent.edu.tr/~korpe

References

- *The slides here are adapted/modified from the textbook and its slides: Operating System Concepts, Silberschatz et al., 7th & 8th editions, Wiley.*

REFERENCES

- Operating System Concepts, 7th and 8th editions, Silberschatz et al. Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum, 3rd edition, 2009.

Outline

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

Objectives

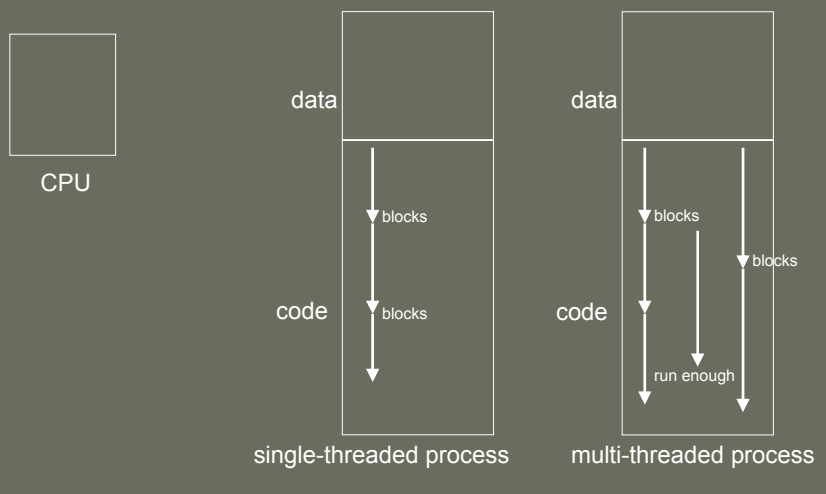
- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming

Threads and Thread Usage

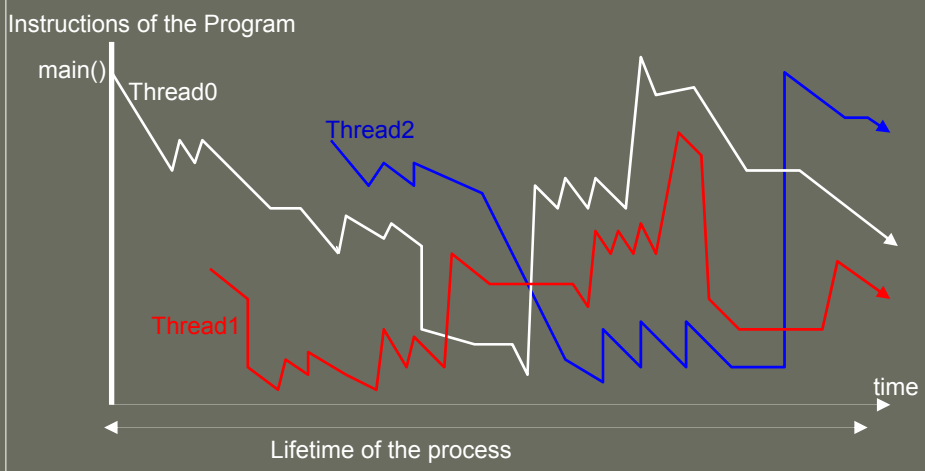
- A process has a single thread of control (execution sequence/flow).
- If it blocks on something, no other activity (task) can be done as part of the process during this time.
- Need for ability to concurrently run several tasks as part of the same process.
- Every process has at least one thread (although threading is not supported).

- A process now can have multiple threads of control. Threads run in pseudo-parallel manner (concurrently) as part of the same process.
- All threads share the same text and data segments.

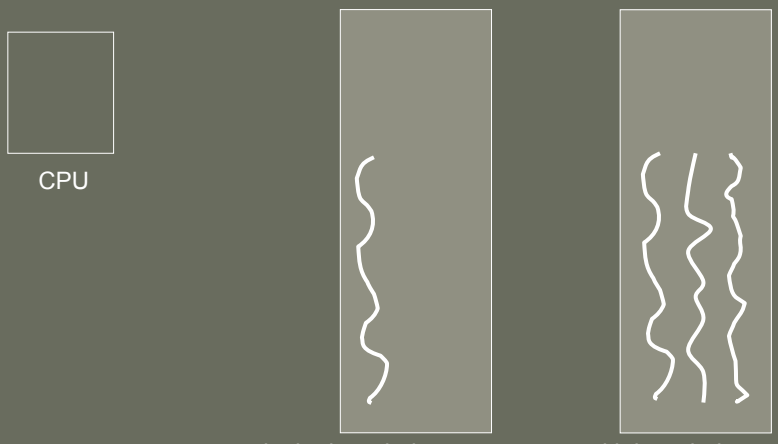
Threads and Thread Usage



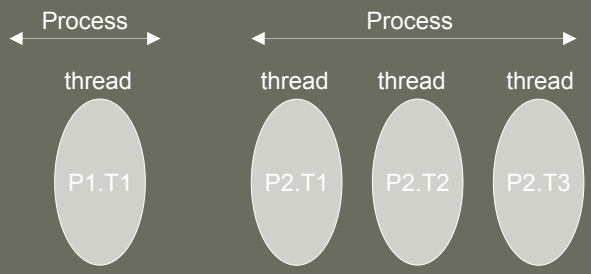
a multithreaded process' execution flows: threads



Multithreading Concept



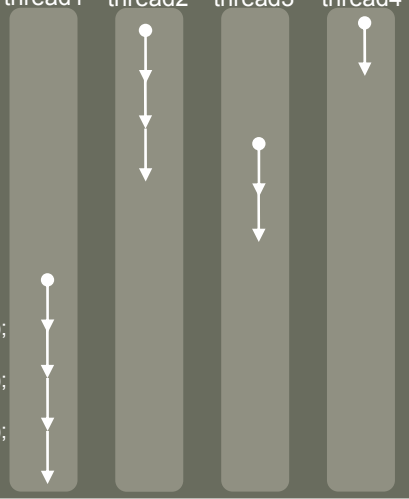
Multithreading Concept



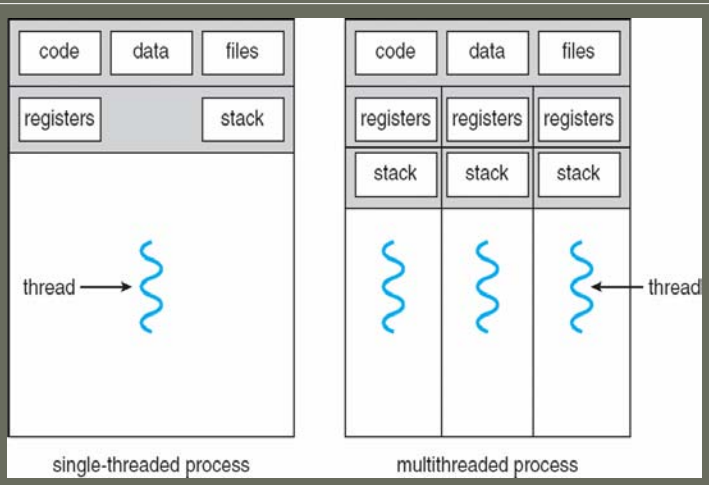
Schedulable Entities
We can select one of them and run

Multithreading Concept

```
function1(…)
{
        ….
{

function2(…)
{
        ….
{

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



Single and Multithreaded Processes



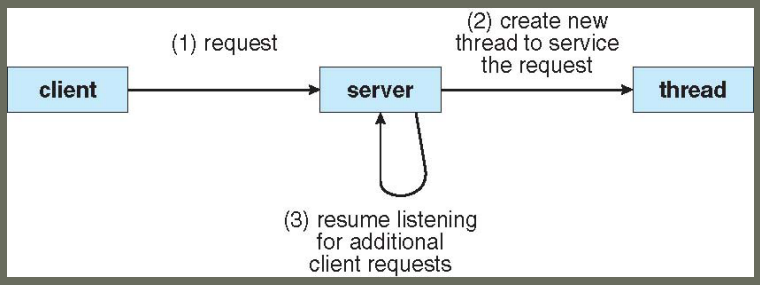
Benefits

- Responsiveness
  - One thread blocks, another one runs.
  - One thread may always wait for the user
- Resource Sharing
  - Threads can easily share resources
- Economy
  - Creating a thread is fast
  - Context switching among threads may be faster
- Scalability
  - Multiprocessors can be utilized better

Multicore Programming

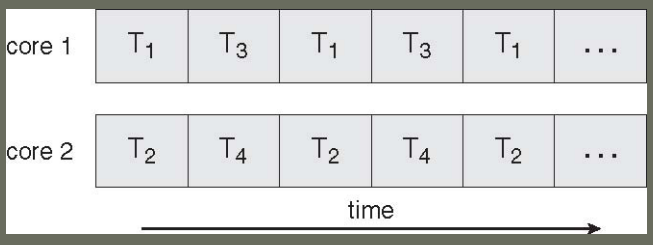
- Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

Multithreaded Server Architecture



Concurrent Execution on a Single-core System

Parallel Execution on a Multicore System



Threading Support

- Multithreading can be support by:
  - User level libraries (without Kernel being aware of it)
    - Library manages threads (user level implementation)
  - Kernel itself
    - Kernel manages threads (kernel space implementation)
- No matter which was is implemented, threads can be created, used, and terminated via a set of functions that are part of a Thread API (a thread library)

Thread Library

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Support of Threads

- Kernel may implement threading and manage threads.
- Kernel is aware of threads.

- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

- All these kernels have threading support. They can schedule processes and their threads (not only processes)

Multithreading Models

- A user process wants to create one or more threads.
- Kernel can create one (or more) thread(s) for the process.

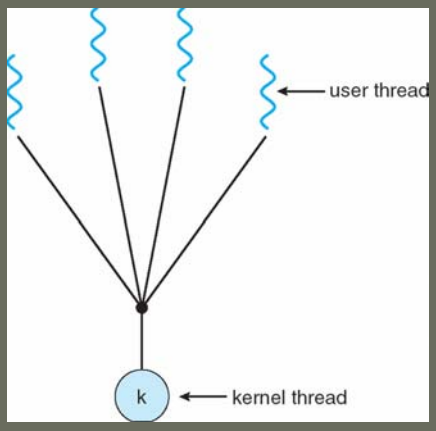
- How the threads created by the process are actually run (mapped into kernel threads)?

- What is the relationship between user level threads and kernel level threads?

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

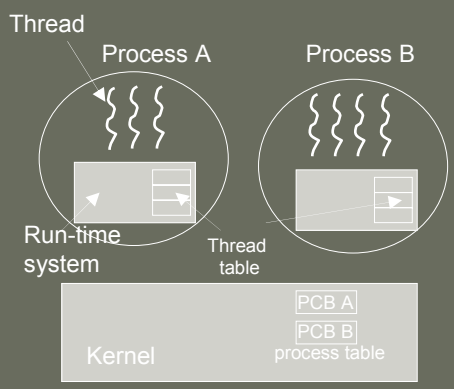
Many-to-One Model

- 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
- No need for kernel support for multithreading (+)
- Multiple threads will run on a single processor, not utilizing multi-processor machines. (-)



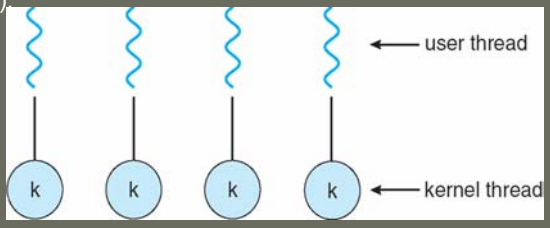
Many-to-One Model
Implementing Threads in User Space

- Kernel does not have to be aware of multithreading
- Library is used to create and manage, schedule threads.
- A thread has to call a function to voluntarily give the CPU (-)
  - example: thread_yield()
- Blocking systems calls defeats the purpose and have to be handled (-)
- Switching between threads is fast; efficient approach (+)
- Thread creation is fast (+)



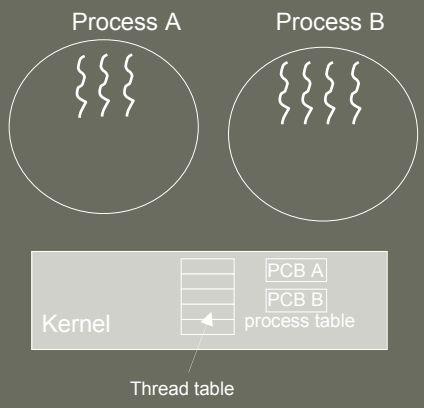
One-to-one Model

- Each user-level thread maps to a kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later
- Provides more concurrency; when a thread blocks, another can run (+).
- Multiple threads can run on several processors if available; utilizing multiprocessor machines (+).
- Creating a user thread requires creating a kernel thread (-)



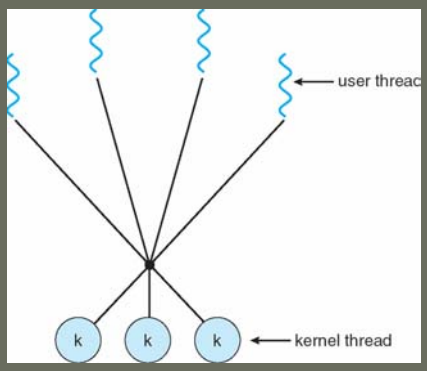
One-to-one Model
Implementing Threads in Kernel

- Kernel is aware of threads. It is managing threads, schedules threads.
- Need system calls to create threads
- Thread switching and creation is more costly (-)
- Blocking system calls are not problem anymore. (+)
- Kernel can stop a long running thread and run another thread. No need for explicit request from a thread to be stopped. (+)
- Any thread function requires a system call: (-)
  - ex: thread_wait



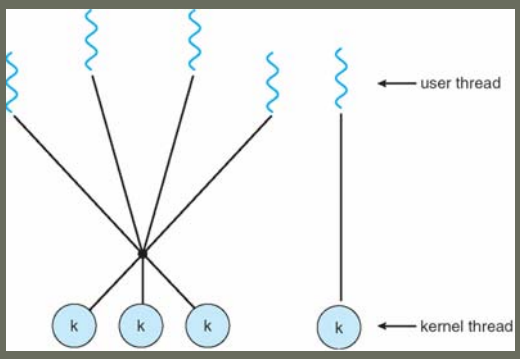
Many-to-Many Model

- 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 kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Programmer just have to know the thread library interface (API). Threads may be implemented in user space or kernel space.
  - Library may be entirely in user space
  - Kernel-level library supported by the OS

Pthreads Library

- 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 operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

- We will show a program that creates a new thread.
  - Hence a process will have two threads :
    - 1 - the initial/main thread that is created to execute the main() function (that thread is always created even there is no support for multithreading);
    - 2 - the new thread.

    (both threads have equal power)
- The program will just create a new thread to do a simple computation. The new thread will get a parameter, an integer value, and will sum all integers from 1 up to that value.
  - sum = 1+2+…+parameter_value $sum = \sum_{i=1}^{N} i$
- 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

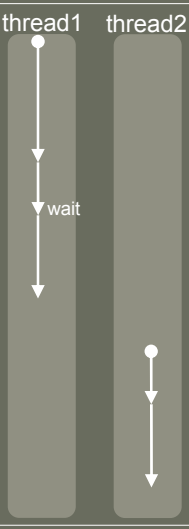
```
int  main(int argc, char *argv[]){
        pthread_t tid; /* id of the created thread */
        pthread_attr_t attr;  /* set of thread attributes */

        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

- You can put the above code into a .c file, say mysum.c
- In order to use the Pthreads functions, we need to include **pthread.h** header file in our program (as shown in previous slides)
- 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.
- Hence we can compile+link our program as follows:
  - **gcc -Wall -o mysum -lpthread mysum.c**
- Then we run it as (for example):
  - ./mysum 6
- It will print out 21

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 Thread class
  - Implementing the Runnable interface

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
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

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
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

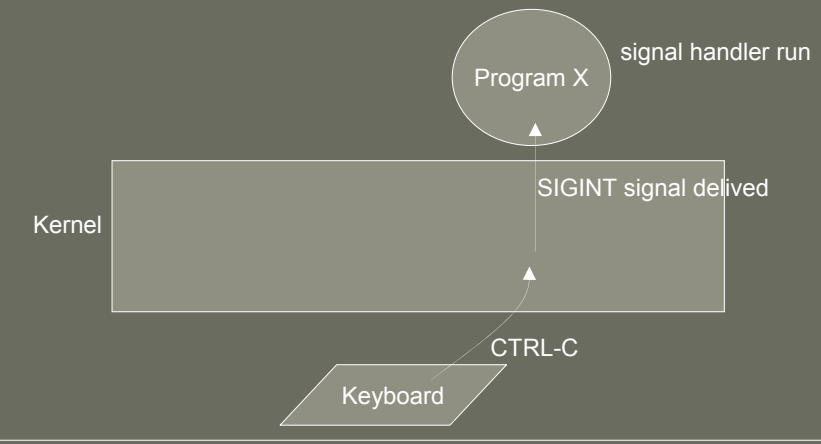
a C program using signals

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

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

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

        while (1)
                ;
}
```

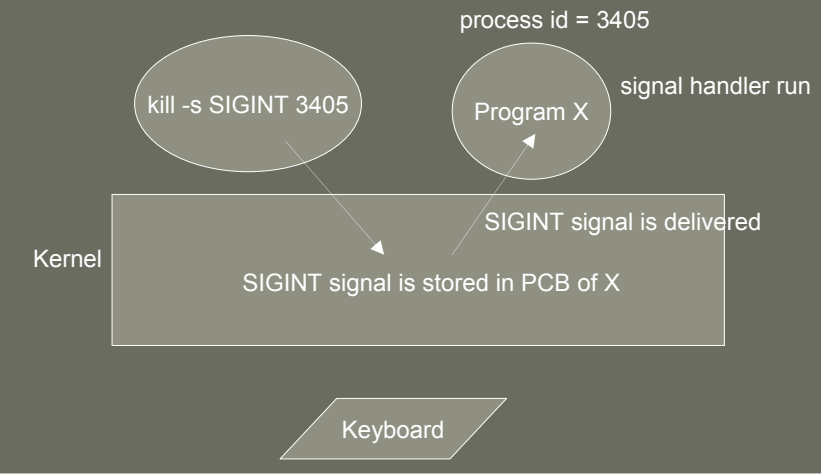
Program X

- While a program is running, if we press CTRL-C keys, the program will be terminated (killed). We are sending a SIGINT signal to the program
- By default, SIGINT is handled by kernel. By default, kernel terminates the program.
- But if we specify a handler function as here, then our program can handle it.
- Kernel will notify us when user presses the CTRL-C keys.

delivering signal (notifying)



kill program



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.
```

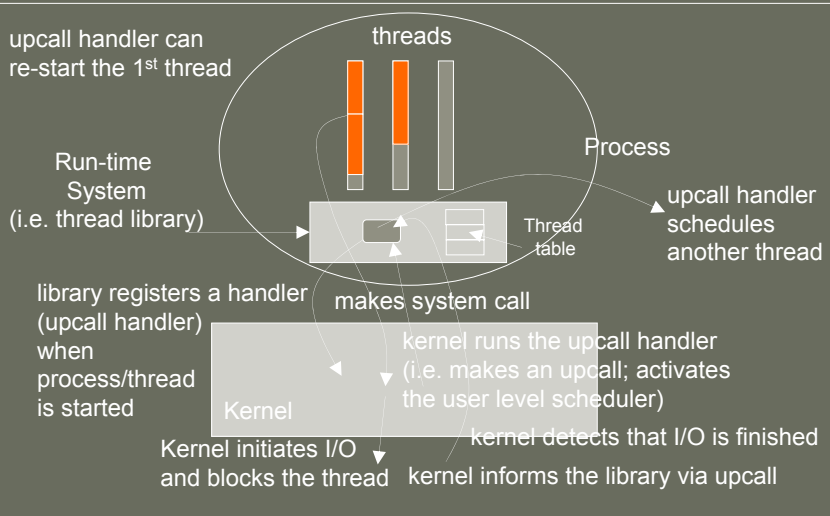

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

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 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



From Singlethread to Multithreaded

- Many programs are written as a single threaded process.
- If we try to convert a single-threaded process to multi-threaded process, we have to be careful about
  - the global variables
  - the library functions

From Singlethread to Multithreaded

```
int status;

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

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

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!

From Singlethread to Multithreaded

- Scope of variables:
  - Normally we have: global, local
  - With threads we want: global, local, **thread-local**
- **thread-local**: 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.
- But we don't have language support to define such variables.
  - C can not do that.
- Therefore thread API has special functions that can be used to create such variables – data.
  - This is called thread specific data.

Thread Specific Data

- Allows each thread to have its own copy of data
  - Each thread refers to the data with the same name.
- create_global (“bufptr”); // create pointer to such a variable
- set_global (“bufptr”, &buf); // set the pointer
- bufptr = read_global (“bufptr”); // get the pointer to access

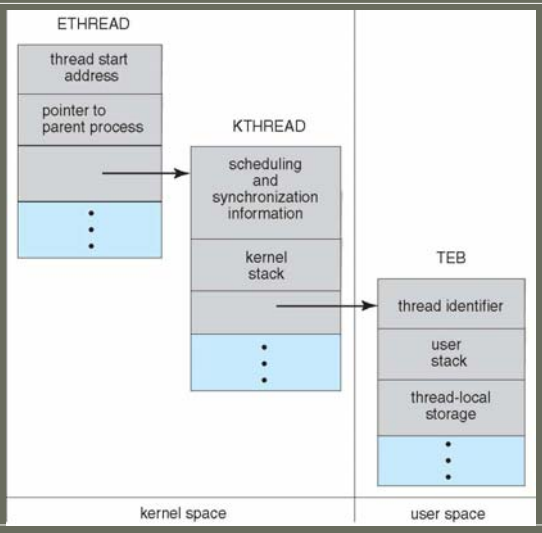
From Singlethread to Multithreaded

- Many library procedures that are used may not be **reentrant**.
- They are not designed to have a second call to the same procedure from the same process before the procedure is completed.
  - We are talking about non-recursive procedures.
- They may be using global variables. Then they are not thread-safe.
- We have to be sure that we use **thread-safe** library routines.

Operating System Examples

- Windows XP Threads
- Linux Thread

Windows XP Threads



Windows XP Threads

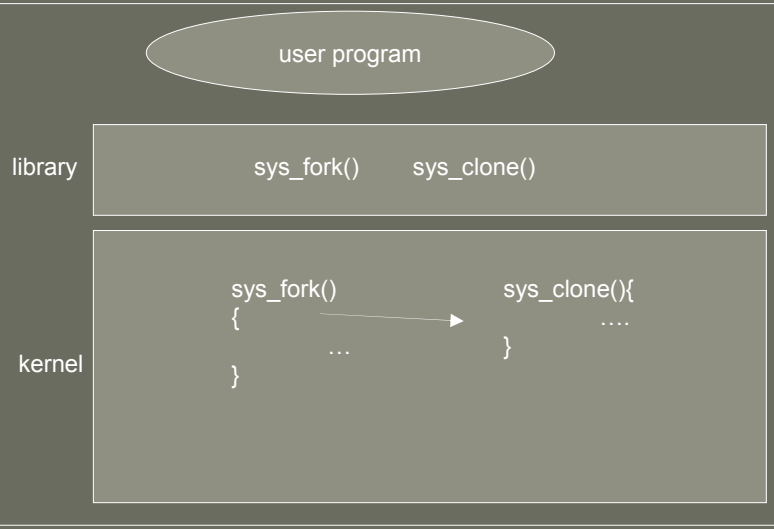
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

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()



End of lecture