ECE437/CS481

M02C: PROCESSES & THREADS THREADS

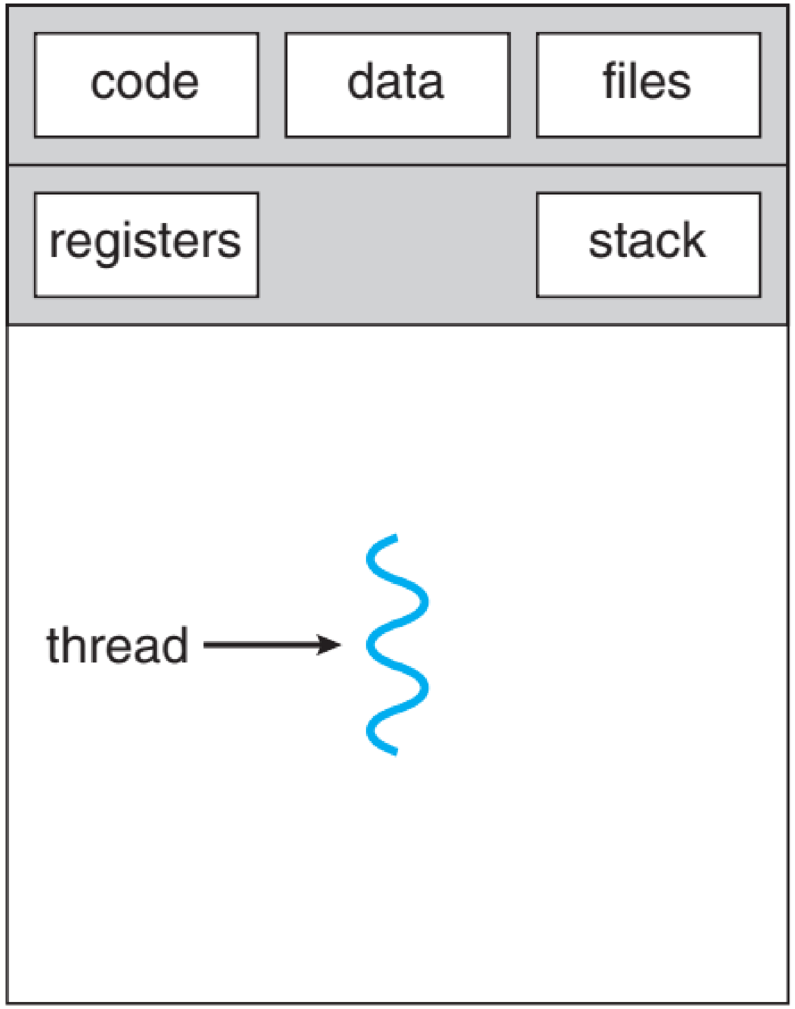
Chapter 4.1-4.7

Xiang Sun

The University of New Mexico

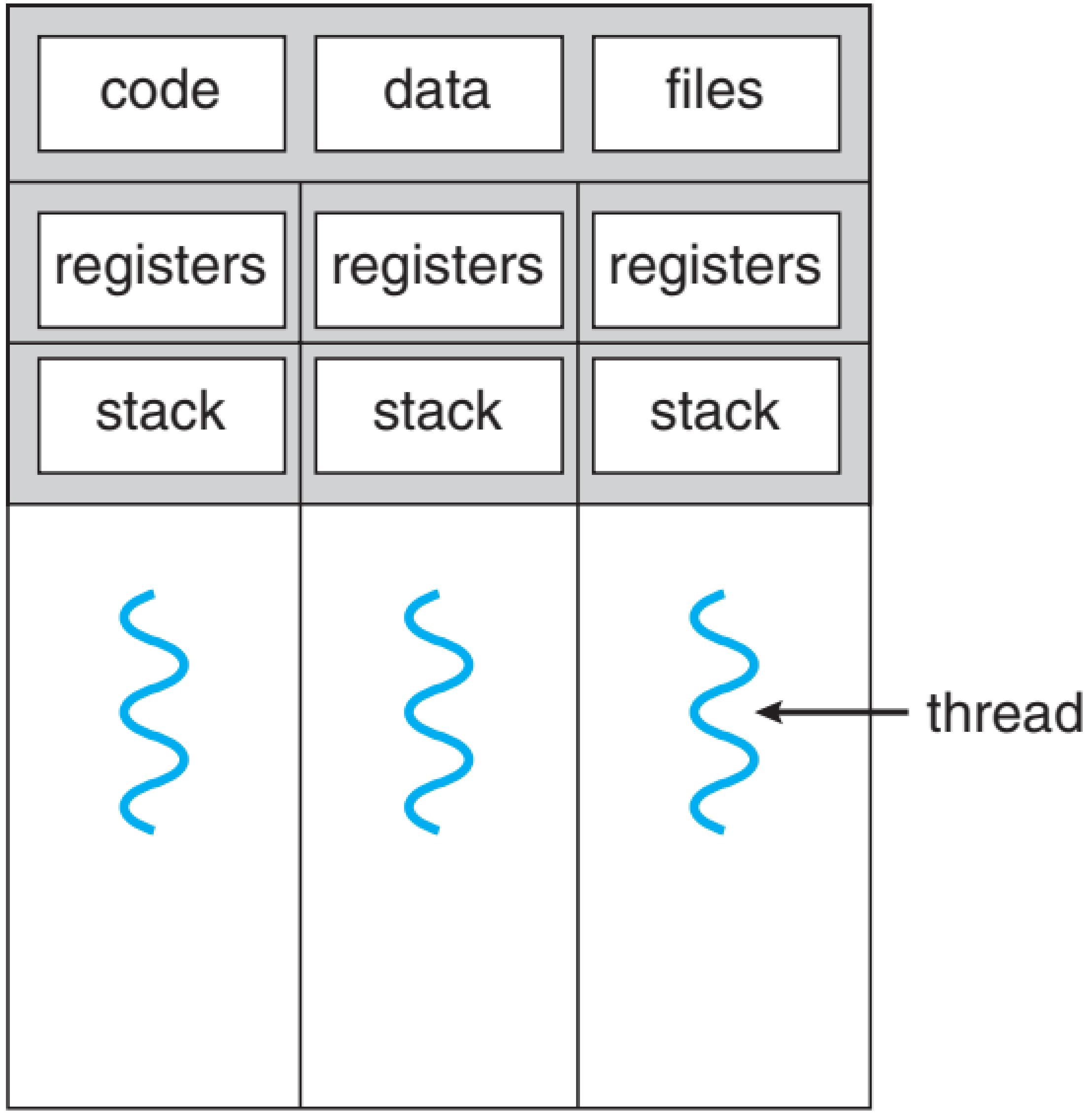
Threads

- What is a thread?
  - Process: Program in execution
    - ✓ A specific execution environment (memory space, I/O,...).
    - ✓ Includes more than one execution threads.
    - ✓ Multiple execution threads on multicores
  - Thread: a basic unit of CPU utilization, a lightweight process
    - ✓ An abstract data type representing flow of control within a process.



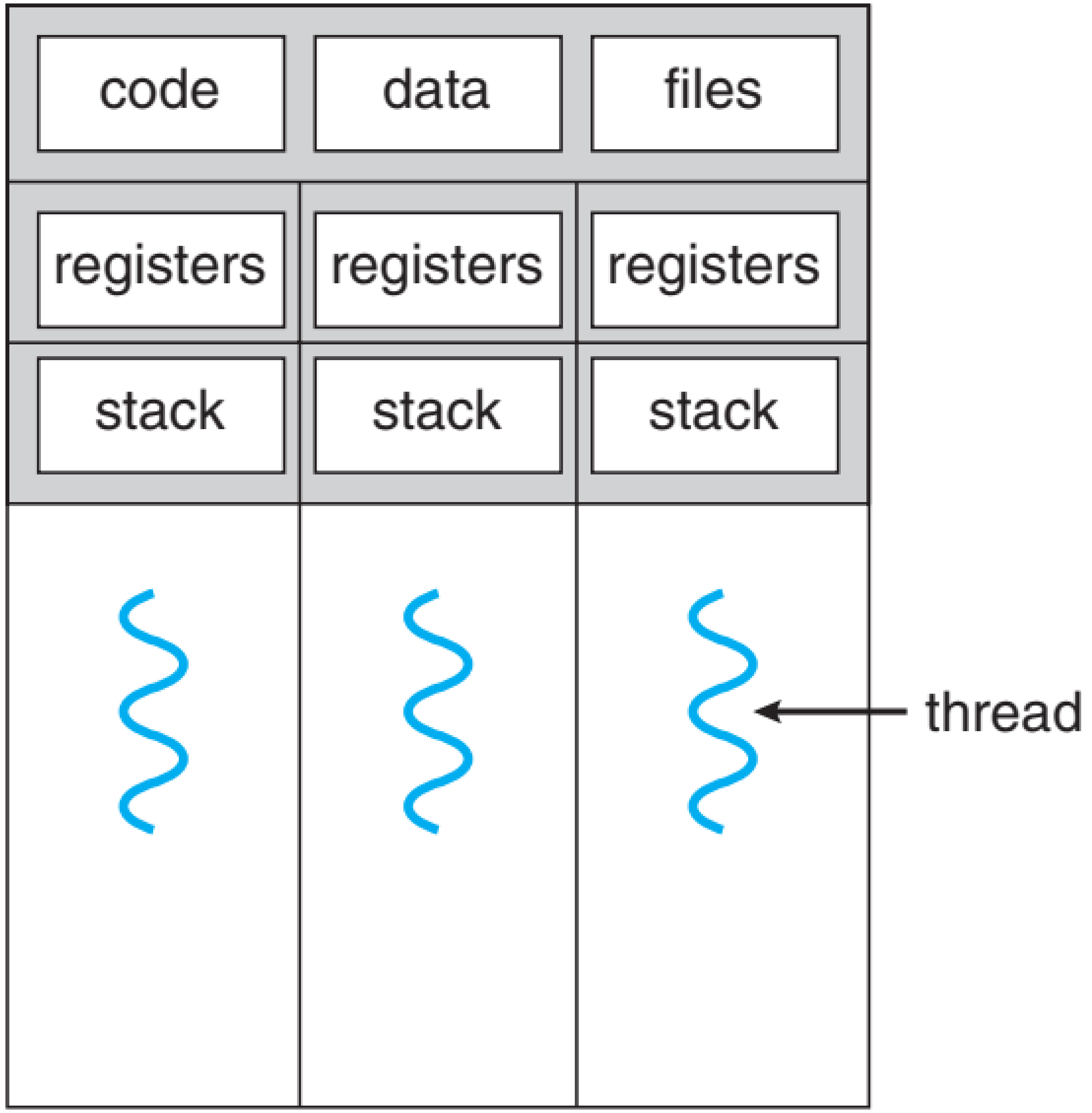
Threads

- Programs in execution with multithreads
  - Multithreading: Having multiple execution threads
    - ✓ Different threads have separate PCs, registers, and stack.
    - ✓ Different threads share the code and resource.
      - Shared memory includes the heap and global/static data
      - No memory protection among the threads (inter-thread communication via memory)



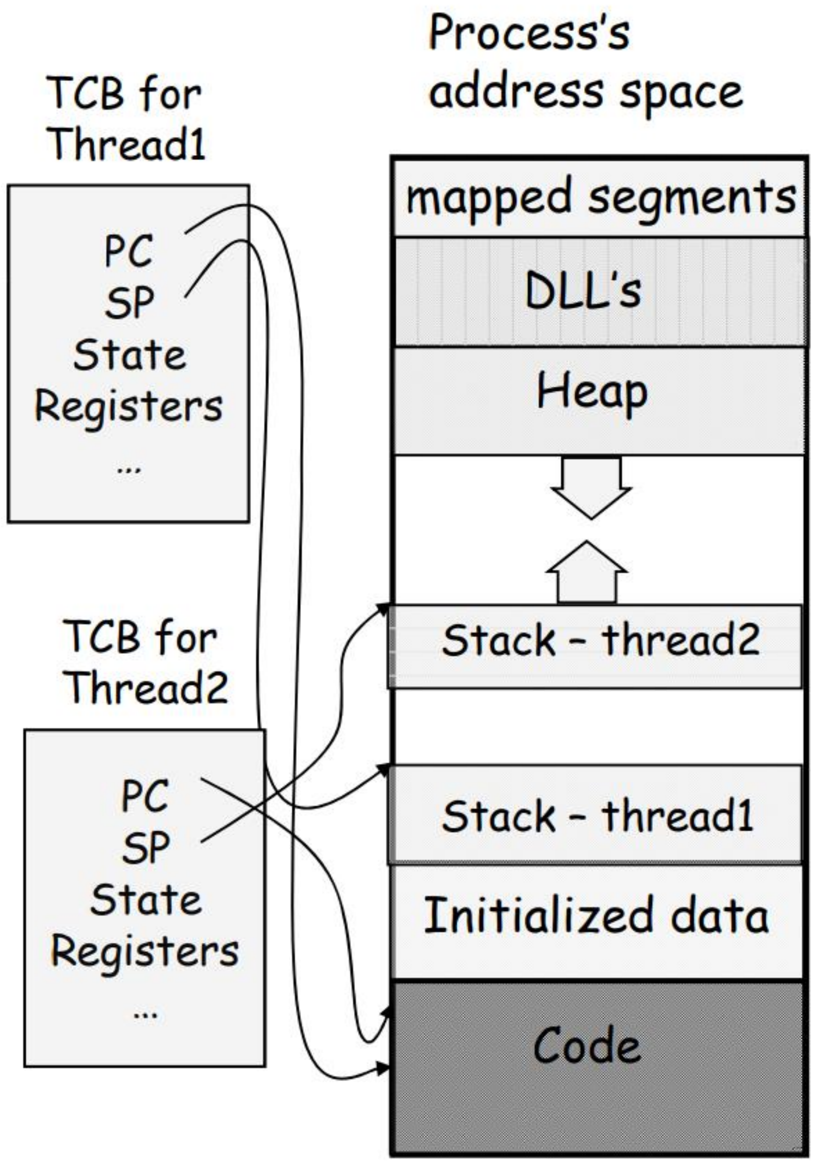
Threads

- Programs in execution with multithreads
  - What threads share:
    - ✓ Text segment (instructions)
    - ✓ Data segment (static and global data)
    - ✓ BSS segment (uninitialized data)
    - ✓ Open file descriptors
    - ✓ Signals
    - ✓ Current working directory
    - ✓ User and group IDs
  - What threads do NOT share:
    - ✓ Thread ID
    - ✓ Saved registers, SP, IP
    - ✓ Stack (local variables, temporary variables, return addresses)
    - ✓ Signal mask
    - ✓ Priority (scheduling information)



Threads

- A process define an address space; its threads share the address space
- Process Control Block (PCB) contains process-specific information
  - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- Thread Control Block (TCB) contains thread-specific information
  - Stack pointer, PC, thread state (running, …), register values, a pointer to PCB, …



Threads

- Benefits of multithreading
  - Less expensive for creation, since it is NOT necessary to
    - ✓ Setup new memory space & file descriptors
    - ✓ Create code segment & initialize data segment
  - Less expensive for context switching:
    - ✓ Don't have to switch virtual memory space
    - ✓ Smooth transition for cache

Process context switching

- Modify registers of the MMU
- Invalidate address translation cache in TLB
- Change PCB (file table, IPC data)

Thread context switching

- CPU registers
- Stack
- Program counter
- Change TCB

Threads

- Benefits of multithreading
  - More fine grained control
    - ✓ Achieve thread level scheduling
    - ✓ Fine control of multithreading with priority
  - Provide parallelism:
    - ✓ Be able to partition computation workloads
    - ✓ Utilize multiple cores for speedup

```
for(k = 0; k < n; k++)
a[k] = b[k] * c[k] + d[k] * e[k];
```

$\Rightarrow$

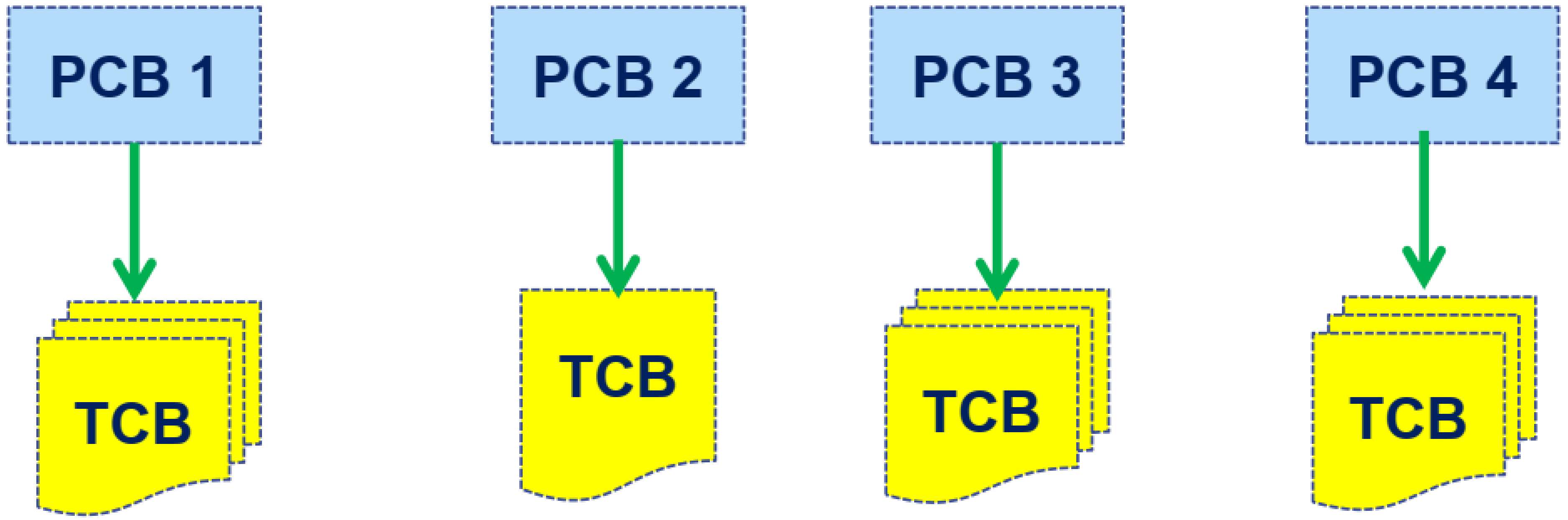
```
do_mult(l, m) {
  for(k = l; k < m; k++)
     a[k] = b[k] * c[k] + d[k] * e[k];
}

main() {
  CreateThread(do_mult, 0, n/2);
  CreateThread(do_mult, n/2, n);
}
```

Threads

- Drawbacks of multithreading
  - Need coordination for data sharing
    - ✓ Multiple threads may try to access the same resource or data
  - Lack protection among threads
    - ✓ Thread's stack and local variables can be accessible
  - Less robust against programming errors
    - ✓ Hard to debug multithreading programs

Threads

- Implement multithreads under processes
  - PCB contains one or more Thread Control Blocks (TCB):
    - ✓ Thread ID
    - ✓ Saved registers (Program counter, stack pointer,…)
    - ✓ Other per-thread info (signal mask, scheduling and priority parameters…)



Threads

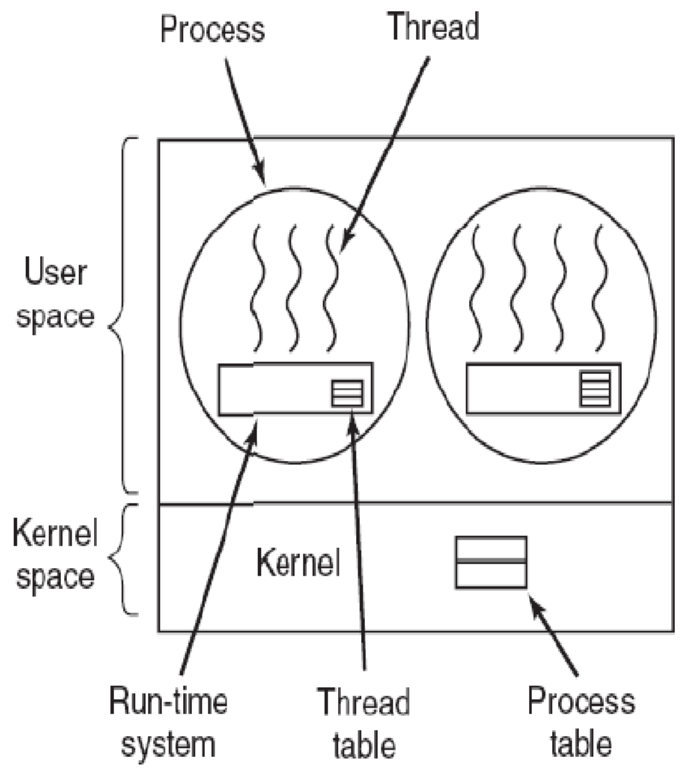
- Who manages threads: User Level Thread Management v.s. Kernel User Level Thread Management

  - User Level Thread Management
    - ✓ Part of user application.
    - ✓ ULT management is done by the application
    - ✓ The kernel is NOT aware of the existence of threads (i.e., kernel sees one execution context: process)

  - Kernel User Level Thread Management
    - ✓ Part of operating system, with Kernel-level library
    - ✓ OS manage kernel threads (e.g., scheduling, creation, synchronization…)

Threads

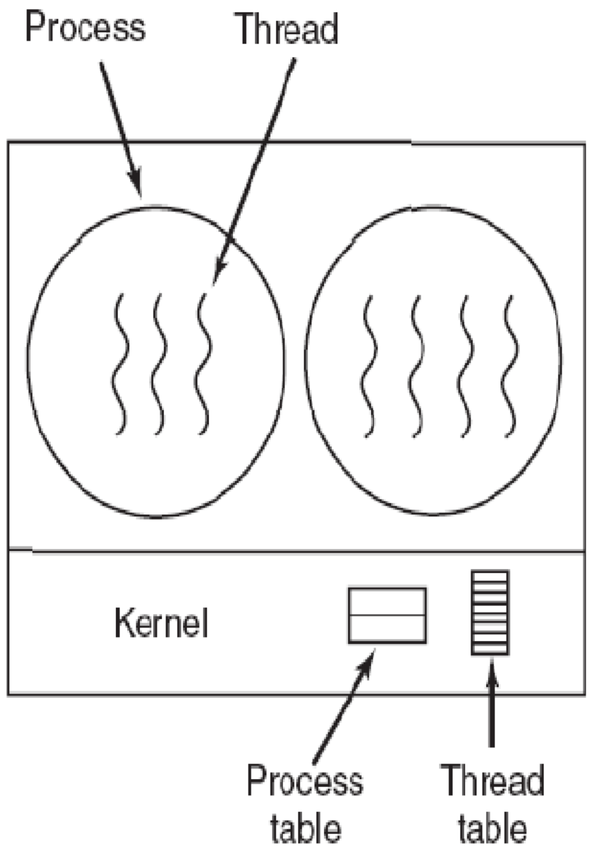
- User level thread management
  - **Pros**
    - ✓ Fast (lightweight--no system call to mange threads. The thread library does everything.
    - ✓ High compatible—can be implemented in an OS, which may/may not support threading.
    - ✓ Fast context switching—no switching from user to kernel mode.



  - **Cons**
    - ✓ Scheduling can be an issue. --consider that one thread is blocked on an I/O, and thus all the threads of the process are blocked.
    - ✓ Lack of coordination between kernel and threads.--- A process with 100 threads competes for a timeslice with a processing with just 1 thread

Threads

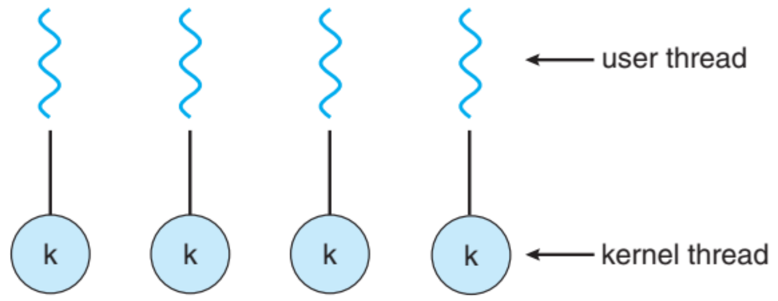
- Kernel level thread management
  - **Pros**
    - ✓ Scheduler in the kernel can optimize the scheduling--give more time to a process having larger number of threads than process having small number of threads.
    - ✓ More efficient--if a thread is blocked, the kernel can schedule another thread from the same process.
    - ✓ Parallel—the kernel can simultaneously schedule multiple threads on multiple processor.
  - **Cons**
    - ✓ The kernel-level thread management are slow—involve system calls and user-kernel mode switching.
    - ✓ Incur overheads in the kernel—the kernel has to maintain information about threads (e.g., thread table).



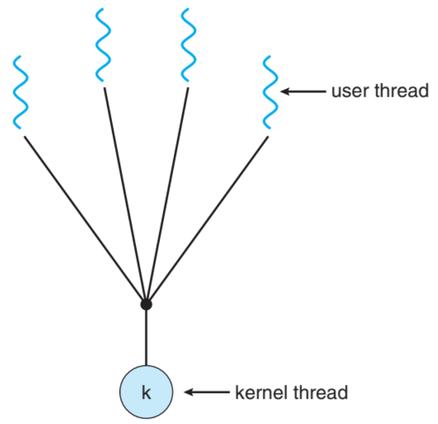
Threads

❑ User level thread (N) to kernel level thread (M) mapping

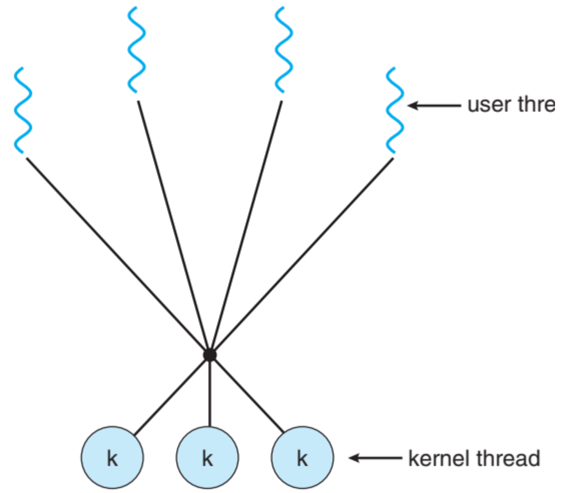
- ➢ One-to-one: N=1 & M=1 (each user-level thread maps to kernel thread)--User level thread management
  - ✓ Example: Windows NT/XP/2000, Linux, Solaris 9 and later
- ➢ Many-to-one: N>1 & M=1 (many user-level threads mapped to single kernel thread)-- Kernel level thread management
  - ✓ Example: Solaris Green Threads
- ➢ Many-to-Many: N>1, M>1 (allows many user level threads to be mapped to many kernel threads) -- Hybrid threading
  - ✓ Example: Windows NT/2000 with the ThreadFiber package



One-to-one model



Many-to-one model



Many-to-many model

Threads

❑ Hybrid threading

- TCBs are maintained by run-time system in the user space.
- The operating system creates a pool of threads in the kernel space.
- A user thread can be mapped into different kernel threads.
- It is better to handle blocking system calls (i.e., if a kernel thread is blocked, a user thread can map to another kernel thread).
- Improve the utilization of the multiple CPU.

Threads

❑ Standards: POSIX threads (Pthreads)

- ➢ POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995), defines an API for creating and manipulating threads.
  - ✓ Generally, it is a hybrid thread.
  - ✓ Flexible enough to support both user-level and kernel-level threads. Currently, implemented as a native kernel threads.

- ➢ Pthread programming
  - ✓ All calls are prefixed with pthread_. (return 0 if success >0 if error)
  - ✓ Every source file include:

```
#include <pthread.h>
#include <sys/types.h>
```

  - ✓ Compile with pthread lib

```
gcc files… -lpthread
e.g., gcc thread.c -o thread -lpthread
```

  - ✓ Make sure system calls used are MT-safe.

Threads

❑ pthread Management

API	Description
pthread_create	create a new thread and execute a function
pthread_exit	terminate itself by calling pthread_exit or just by returning from the function that was invoked
pthread_kill	terminate another thread
pthread_self	get own thread ID
pthread_join	wait for another thread’s termination
pthread_detach	let thread release resource upon its termination

Threads

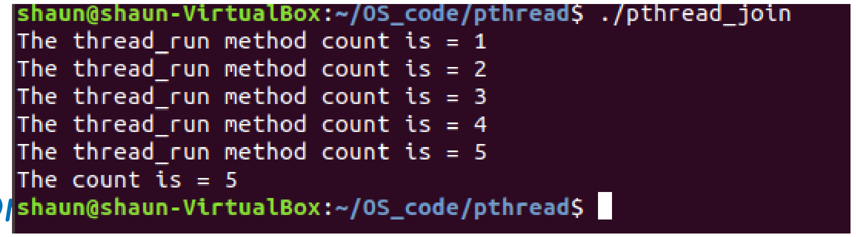
- ❑ Pthread_create
  - ➢ int pthread_create (pthread_t *tid, const pthread_attr_t *attr, void *(*func)(void *), void *arg)
    - ✓ tid: point to the ID of the newly created thread
    - ✓ attr: point to an attribute object (NULL if use default attributes)
    - ✓ func: thread code
    - ✓ arg: passing parameters
- ❑ Pthread_join
  - ➢ int pthread_join(pthread_t tid, void **exit_status);
    - ✓ tid: ID of the thread to be joined
    - ✓ exit_status: the calling thread retrieves point to exit status (pthread_exit())
    - ✓ Different from the wait system call used for processes since there is no parent-child relationship with threads. Any thread may join (wait on) any other thread.

Pthread_joint blocks the calling thread/process until the joined threads terminate

Threads

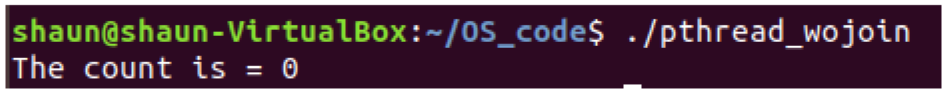
❑ pthread_create() and pthread_joint example

```c
#include <sys/types.h>
#include <pthread.h>
#include <stdio.h>
//#pragma comment(lib, "pthreadVC2.lib")

static int count = 0;
void* thread_run(void* parm)
{
        for (int i=0;i<5;i++)
        {
                count++;
                printf("The thread_run method count is = %d\n",count);
                sleep(5);
        }
        return NULL;
}

int main()
{
        pthread_t tid;
        pthread_create(&tid, NULL, thread_run,NULL);
        pthread_join(tid,NULL);
        // Main() is blocked
        printf("The count is = %d\n",count);
        return 0;
}
```



```
shaun@shaun-VirtualBox:~/OS_code/pthread$ ./pthread_join
The thread_run method count is = 1
The thread_run method count is = 2
The thread_run method count is = 3
The thread_run method count is = 4
The thread_run method count is = 5
The count is = 5
shaun@shaun-VirtualBox:~/OS_code/pthread$ 
```

```c
#include <sys/types.h>
#include <pthread.h>
#include <stdio.h>
//#pragma comment(lib, "pthreadVC2.lib")

static int count = 0;
void* thread_run(void* parm)
{
        for (int i=0;i<5;i++)
        {
                count++;
                printf("The thread_run method count is = %d\n",count);
                sleep(5);
        }
        return NULL;
}

int main()
{
        pthread_t tid;
        pthread_create(&tid, NULL, thread_run,NULL);
        //pthread_join(tid,NULL);
        printf("The count is = %d\n",count);
        return 0;
}
```



```
shaun@shaun-VirtualBox:~/OS_code$ ./pthread_wojoin
The count is = 0
```

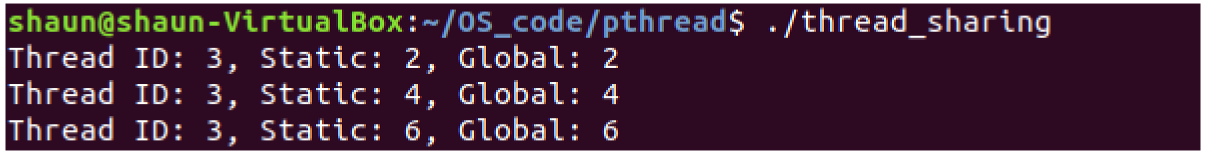
Threads

❑ Data sharing among threads example

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

//create a global variable to change it in threads
int g = 0;
// The function to be executed by all threads
void *myThreadFun(void *varp)
{   // Store the value argument passed to this thread
    int *myid = (int *)varp;
    // create a static variable to observe its changes
    static int s = 0;
    // Change static and global variables
    ++s; ++g;
    // Print the argument, static and global variables
    printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
    return NULL;
}

int main()
{
    int i;
    pthread_t tid[3];
    // Let us create three threads
    for (i = 0; i < 3; i++) {
        pthread_create(&tid[i], NULL, myThreadFun, (void *)&i);
        pthread_join(tid[i],NULL);}
    return 0;
}
```

Parallel thread creation

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

//create a global variable to change it in threads
int g = 0;
// The function to be executed by all threads
void *myThreadFun(void *varp)
{   // Store the value argument passed to this thread
    int *myid = (int *)varp;
    // create a static variable to observe its changes
    static int s = 0;
    // Change static and global variables
    ++s; ++g;
    // Print the argument, static and global variables
    printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
    return NULL;
}

int main()
{
    int i, j;
    pthread_t tid[3];
    // Let us create three threads
    for (i = 0; i < 3; i++) {
        pthread_create(&tid[i], NULL, myThreadFun, (void *)&i);}
    for (j = 0; j < 3; j++) {
        pthread_join(tid[j],NULL);}
    return 0;
}
```



```
shaun@shaun-VirtualBox:~/OS_code/pthread$ ./thread_sharing
Thread ID: 3, Static: 2, Global: 2
Thread ID: 3, Static: 4, Global: 4
Thread ID: 3, Static: 6, Global: 6
```

Threads

- pthread_detach

Every thread must be either joined or detached!

  - int pthread_detach(pthread_t tid);
    - ✓ The calling thread/process (e.g., main()) is not blocked if a new created thread is detached from the process, the new thread could be alive after the calling thread/process terminates.
    - ✓ Once a thread has been detached, it can't be joined with pthread_join() or be made joinable again.
    - ✓ When a detached thread terminates, all its resources are released.
    - ✓ A thread can detach itself.

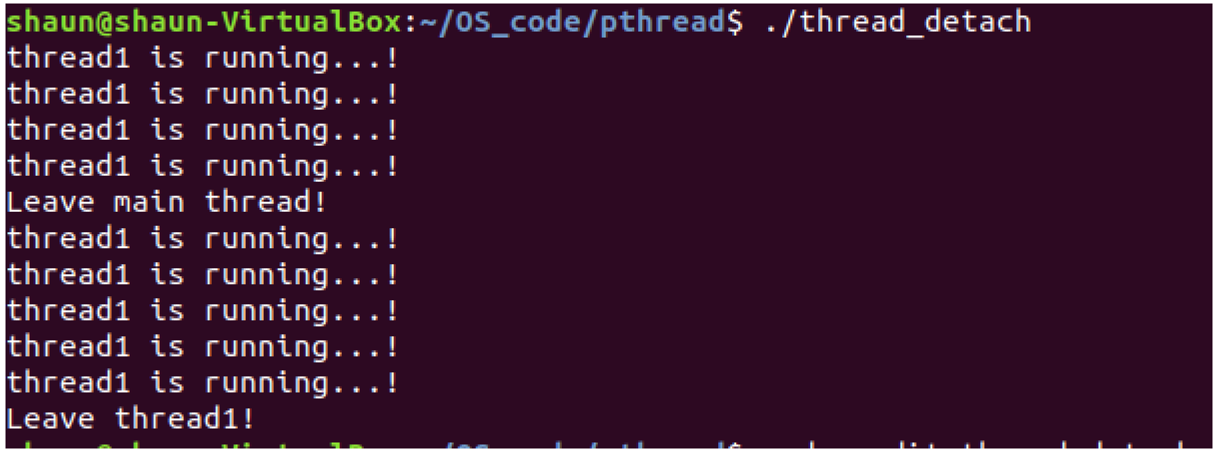
- pthread_exit

  - int pthread_exit(void *exit_status);
    - ✓ The exit_status is available to a successful thread_join
    - ✓ When a joined thread terminates, its thread ID and exit status are retained until another thread calls pthread_join.

Threads

❑ pthread_detach example

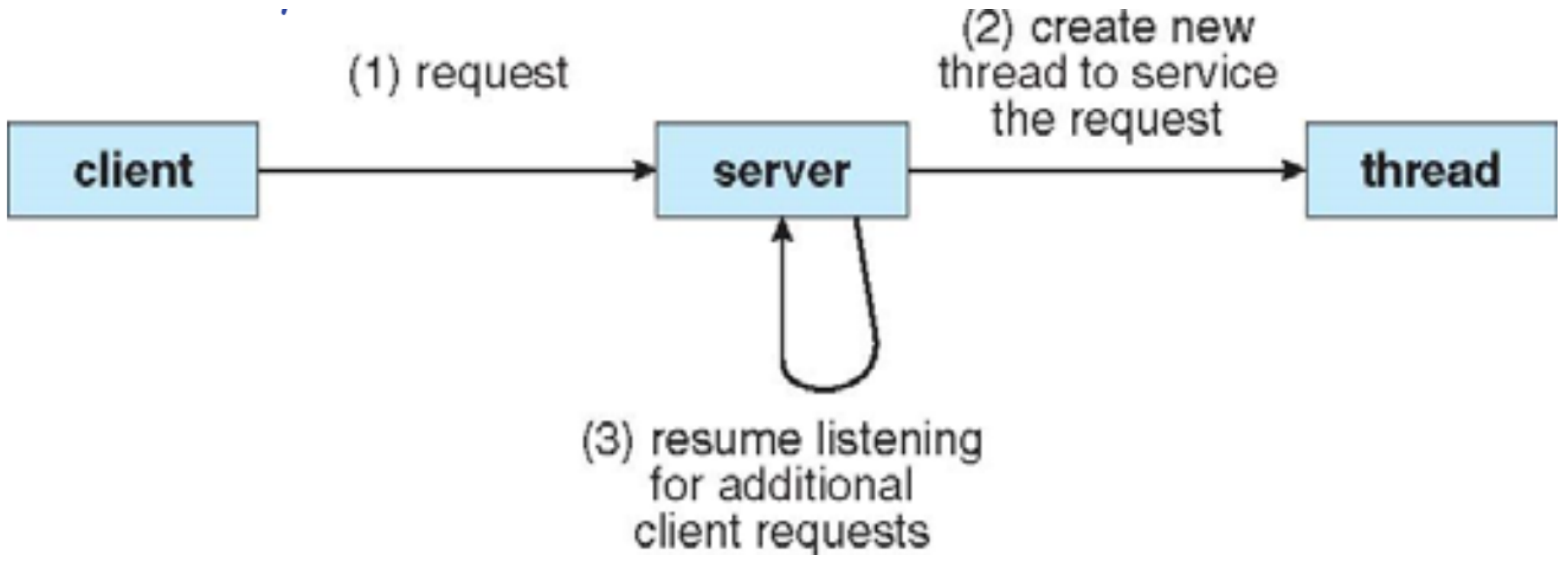
```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>
#include <time.h>

void* thread1(void *arg)
{
    int i;
    for (i=1;i<10;i++)
    {
        sleep(1);
        printf("thread1 is running...!\n");
    }
    printf("Leave thread1!\n");
    return NULL;
}

int main(int argc, char** argv)
{
    pthread_t tid;
    pthread_create(&tid, NULL, (void*)thread1, NULL);
    pthread_detach(tid);  // detach the thread from the main thread
    sleep(5);
    printf("Leave main thread!\n");
    pthread_exit(NULL);
}
```



```
shaun@shaun-VirtualBox:~/OS_code/pthread$ ./thread_detach
thread1 is running...!
thread1 is running...!
thread1 is running...!
thread1 is running...!
Leave main thread!
thread1 is running...!
thread1 is running...!
thread1 is running...!
thread1 is running...!
thread1 is running...!
Leave thread1!
```

Threads

- Multithreads Application Usage
  - Master-slave threads
    - ✓ A master dynamically create slave threads upon requests.
    - ✓ A worker thread executes a specific task.
    - ✓ May have a number of distinct tasks that could be performed concurrently with each other.

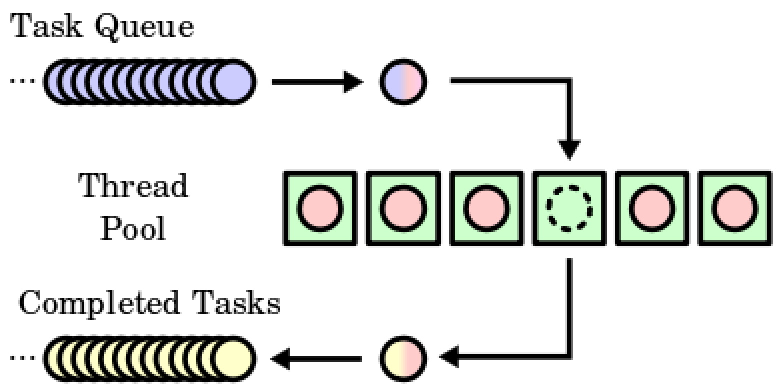

(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Threads

❑ Multithreads Application Usage

➢ Thread pool

- ✓ A number of threads are created upon start-up.
- ✓ All of these threads get work assigned from the same task queue.
- ✓ If a thread finishes the task, it returns back to the thread pool and ready to be assigned.


Task Queue
Thread Pool
Completed Tasks