ECE437/CS481

M02D: PROCESSES & THREADS
THREADS

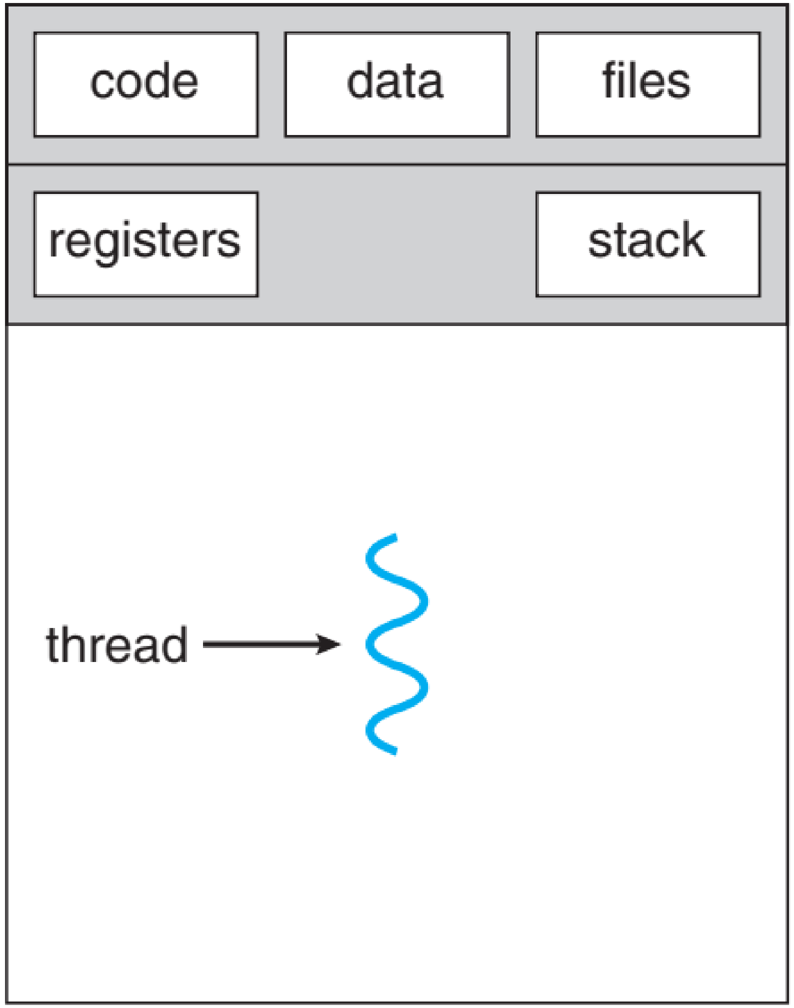
Chapter 4.1-4.7

Xiang Sun

The University of New Mexico

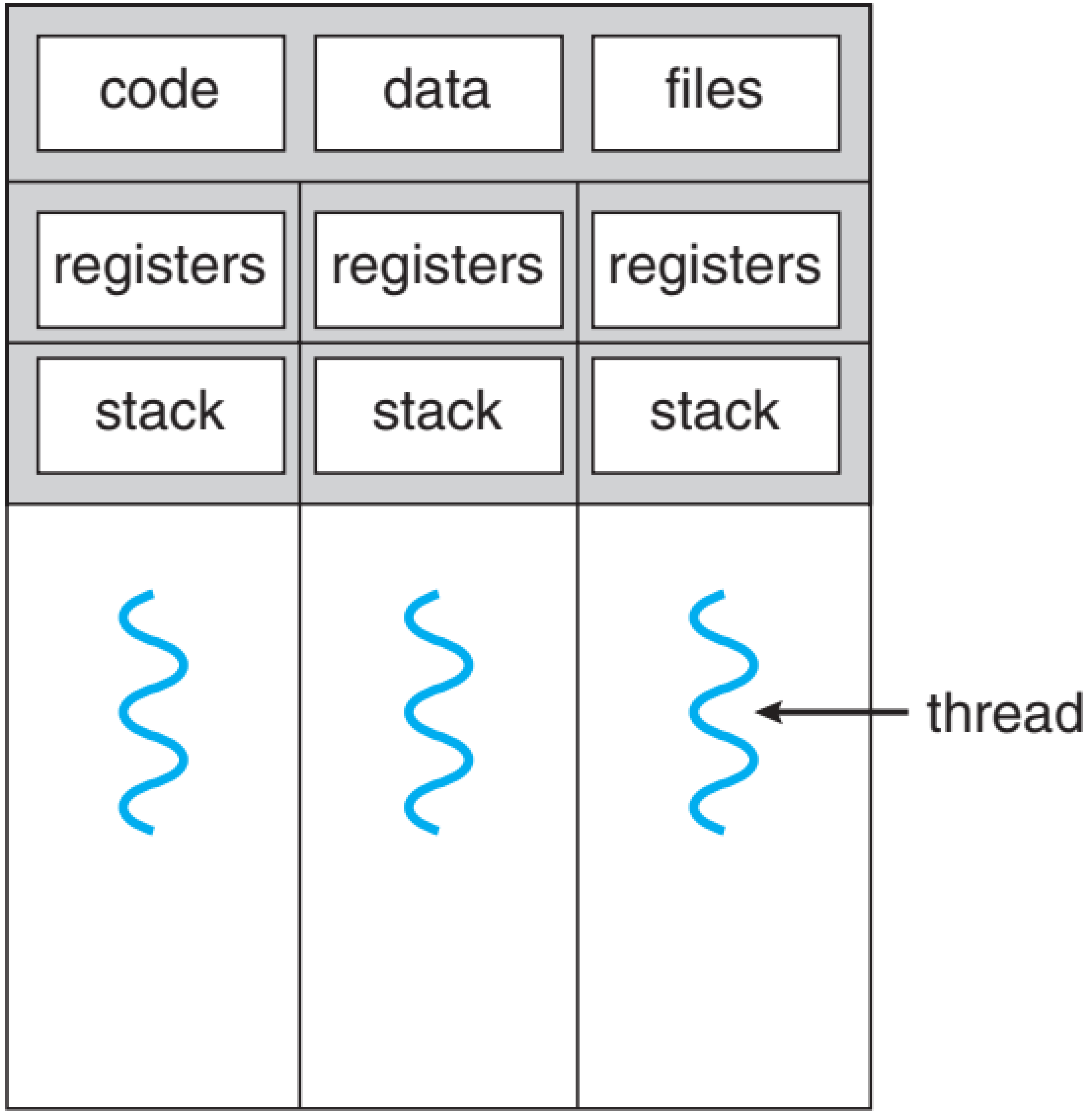
Threads

- What is a thread?
  - Process: Program in execution
    - ✓ A specific execution environment (memory space, I/O,…).
    - ✓ Different processes don't share execution environment.
  - Thread: a path of execution within a process
    - ✓ A process can be consisted of many threads.
    - ✓ Different threads in a process may share some resources.
    - ✓ A lightweight process.



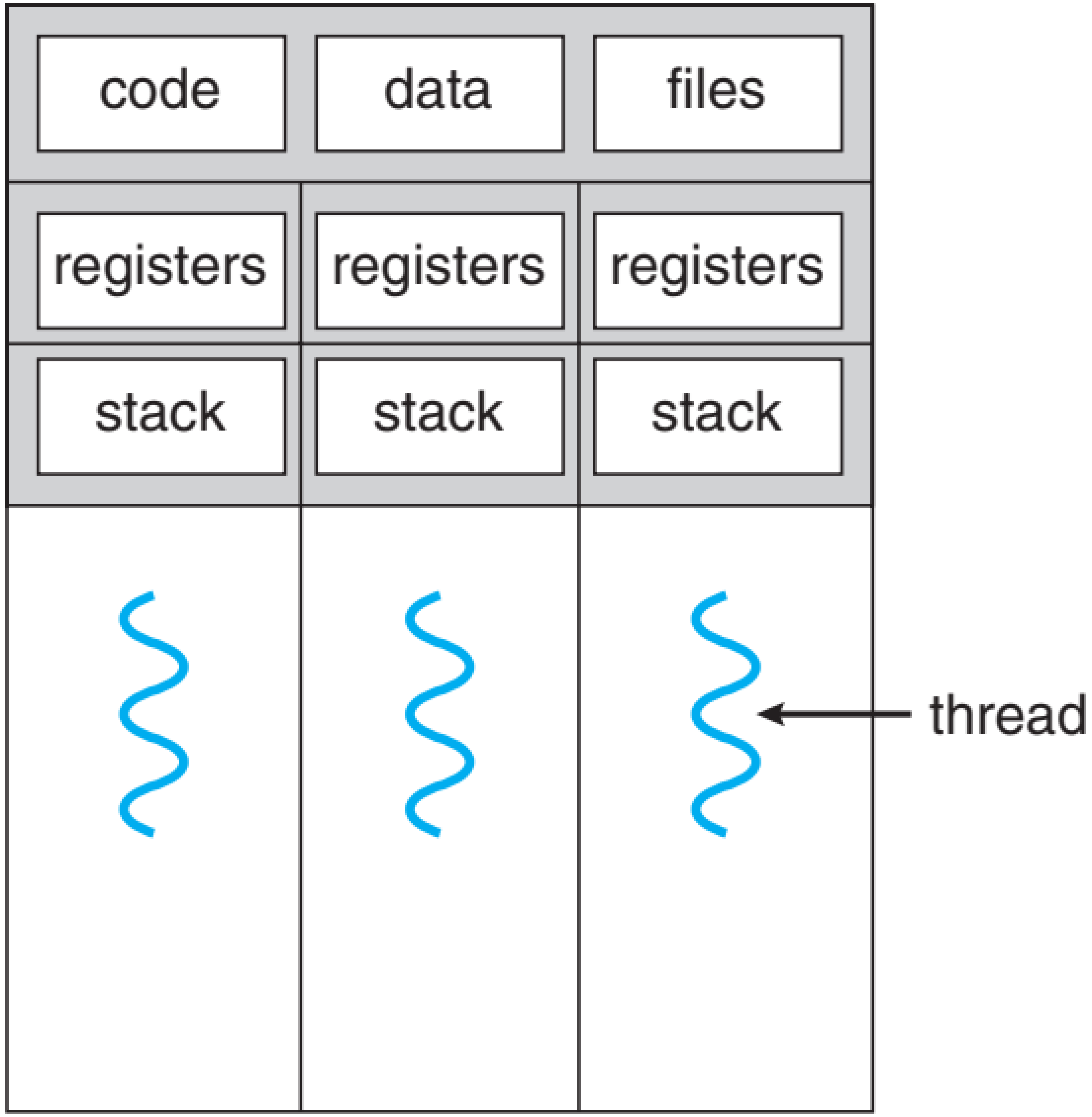
Threads

- Programs in execution with multithreads
  - Multithreading: Having multiple execution threads
    - ✓ Different threads have separated Program Counters (PCs), registers, and stacks.
    - ✓ Different threads share the code, data, and heap.
      - No memory protection among the threads (inter-thread communication via shared memory)



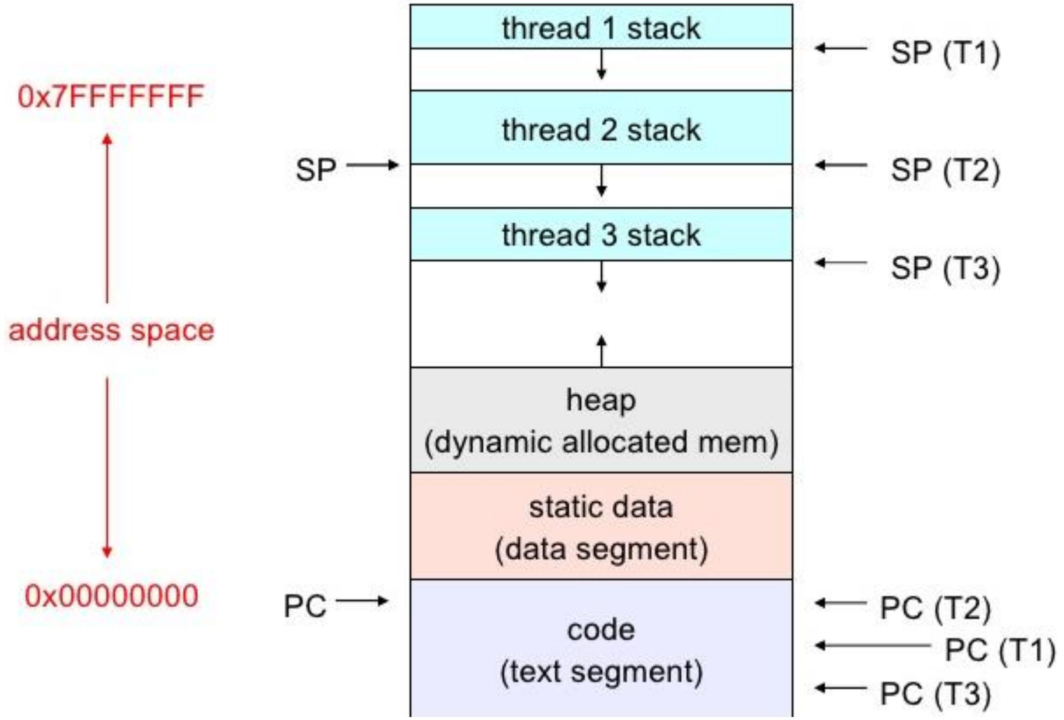
Threads

- 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
    - ✓ Registers, SP, PC
    - ✓ Stack (local variables, return addresses, etc.)
    - ✓ Signal mask
    - ✓ Priority (scheduling information)



Threads

- A process defines 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 context information
  - Stack pointer, PC, thread state (running, …), register values, a pointer to PCB, …



Threads

- Benefits of multithreading as compared to multiprocessing

  - 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 between two different memory spaces
    - ✓ Don't have to flush TLB
    - ✓ Note that TCBs are used in the thread context switching

Threads

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

```
for(k = 1; k < m; k++)
a[k] = b[k] * c[k] + d[k] * e[k];
```

→

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

main() {
  CreateThread(do_mult, 1, m/2);
  CreateThread(do_mult, m/2, m);
}
```

Threads

- Drawbacks of multithreading
  - Need coordination for data sharing
    - ✓ If multiple threads, try to access the same shared resource or data
  - Lack memory protection among threads
    - ✓ Thread's stack (i.e., local variables) can be accessible
  - Less robust against programming errors
    - ✓ Hard to debug multithreading programs

Threads

❑ User Level Thread Management v.s. Kernel Level Thread Management

- Thread management is used to manipulate (e.g., create, delete, schedule, synchronization…) threads.

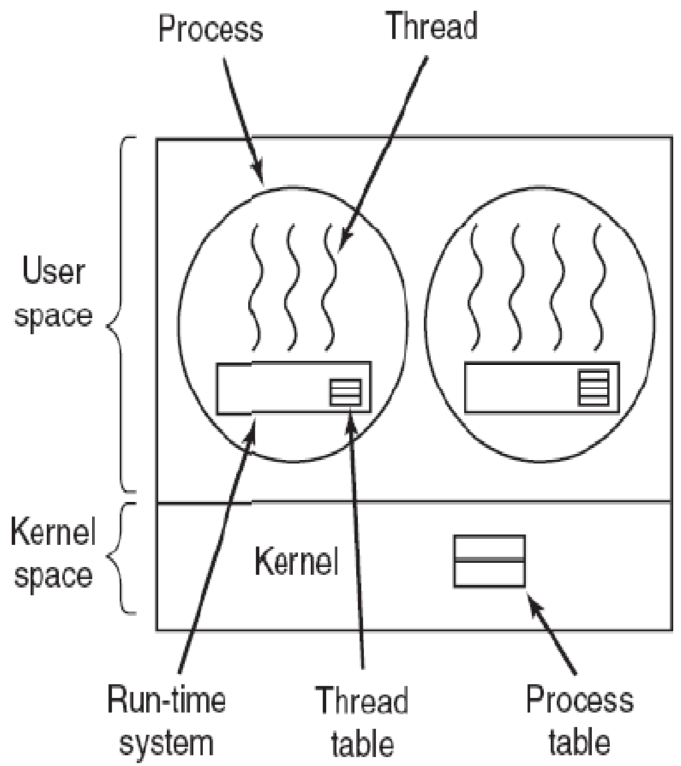
- Two types of threads/threads managements
  - ✓ User Level Thread Management
    - User level threads are managed entirely by the run-time system.
    - The kernel is NOT aware of the existence of threads (i.e., kernel sees one execution context: process).

  - ✓ Kernel Level Thread Management
    - Kernel level threads are managed by the operating system (Kernel-level library)
    - The kernel is aware of the existence of threads.

Threads

- User level thread management

  - Pros
    - ✓ Fast (lightweight)--No system call to manage threads (the thread library does everything). No switching from user to kernel mode.
    - ✓ High compatible—can be implemented in an OS, which may/may not support threading.



  - 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.
      - ❖ A process with 100 threads competes for a timeslot with a process with just 1 thread. The timeslot will be allocated to which process?

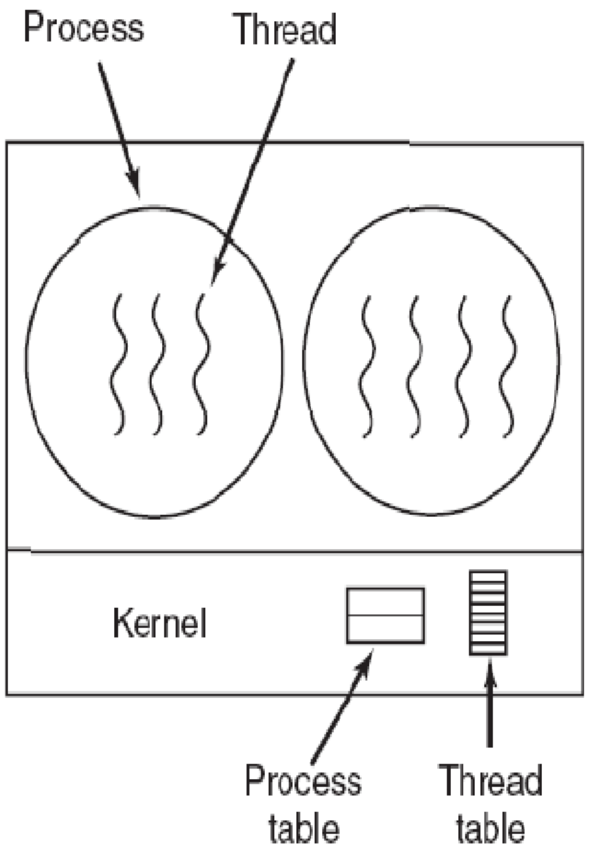
❑ Kernel level thread management

➢ Pros—efficient scheduling

✓ 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

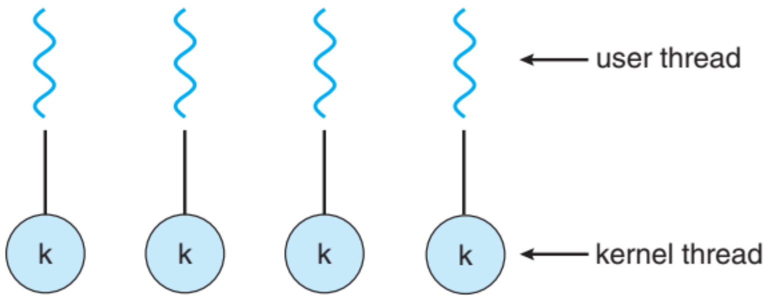
✓ The kernel-level thread management is slow—involve system calls and user-kernel mode switching.

✓ Incur overheads in the kernel—the kernel has to maintain information about threads (e.g., thread table).

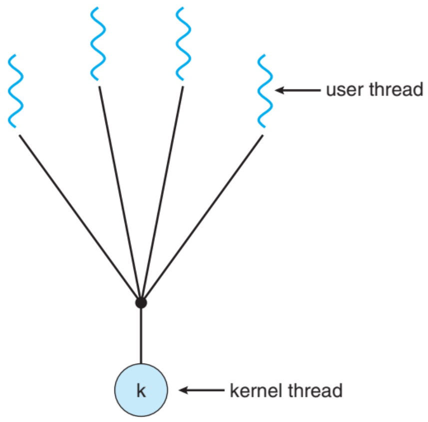
Threads

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

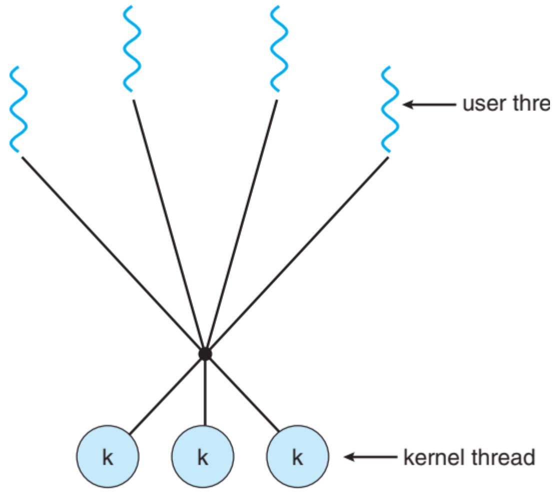
- One-to-one: N=1 & M=1 (each user-level thread maps to kernel thread)--Kernel level thread management
- Many-to-one: N>1 & M=1 (many user-level threads mapped to single kernel thread)-- User level thread management
- Many-to-Many: N>1, M>1 (allows many user level threads to be mapped to many kernel threads) --Hybrid threading



One-to-one model



Many-to-one model



Many-to-many model

Threads

❑ Standards: POSIX threads (Pthreads)

- POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995), defines an API for creating and manipulating threads.
  - ✓ It is implemented as a native kernel thread.

- Pthread programming
  - ✓ All system 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 –o thread thread.c -lpthread
```

❑ 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 *(*start_routine)(void *), void *arg)
  - ✓ *tid: point to where the thread ID is stored
  - ✓ attr: an attribute structure (NULL if use default attributes)
  - ✓ start_routine: the function/routine where the thread begins
  - ✓ arg: passing parameters

❑ Pthread_join

- int pthread_join (pthread_t tid, void **exit_status);

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

  - ✓ tid: ID of the thread to be joined
  - ✓ exit_status: the exit status of the target thread
  - ✓ 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_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;
}
```

```
shaun@shaun-VirtualBox:~/OS_code/pthread$ ./thread_sharing_1
Thread ID: 0, Static: 2, Global: 2
Thread ID: 1, Static: 4, Global: 4
Thread ID: 2, Static: 6, Global: 6
```

```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;
}
```

Parallel thread creation

```
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
```

pthread_detach

- int pthread_detach(pthread_t tid);
  - ✓ If a new created thread is detached from the calling process, the detached thread could be alive after the calling 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.

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

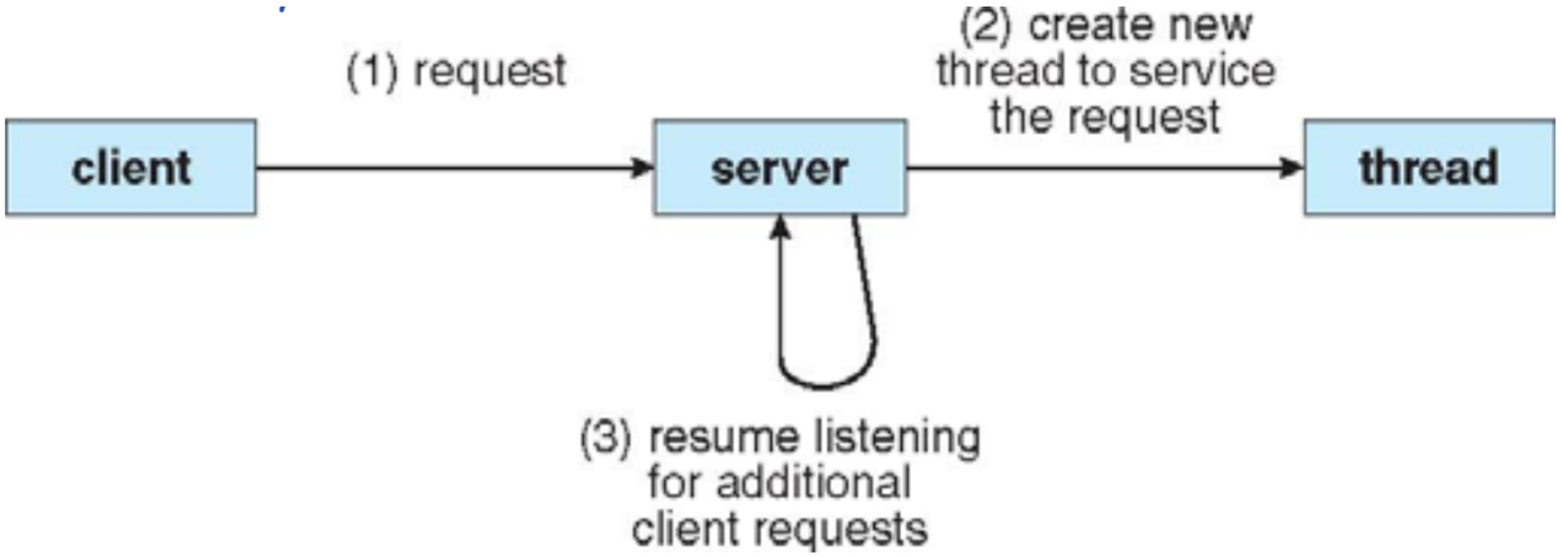
❑ pthread_detach example

```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!
```

Multithreads Application Usage

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



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.