

---

# Operating Systems

## CS2006

### **Chapter: 4**

### **Threads**

# Background



# Basic concepts

---

- **Multiprogramming:** Many processes one CPU. Interleaving between multiple processes. Process relinquish CPU itself.
- **Multiprocessing:** refers to more than one CPU. Systems can be both multiprogramming and multiprocessing at the same time.
- **Multitasking:** refers to modern operating systems doing preemption for switching processes/threads. OS gives a time quantum to each process/thread.
- **Time sharing:** When CPU time is being shared between processes, it is called time sharing system. Multiprogramming & multitasking systems are time sharing systems.
- **Multithreading:** is an execution model that allows a single process to have multiple code segments (threads) that run concurrently within the context of that process.

# Motivation

---

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
  - MS Word
    - One thread for spell checking
    - One thread for output of keystrokes
- Process creation is heavy-weight while thread creation is light-weight
- Increases efficiency
- Kernels are generally multithreaded

# Introduction

---

- Each process has
  1. Own Address Space
  2. Single thread of control
- A process model has two concepts:
  1. Resource grouping
  2. Execution
- Sometimes it is useful to separate them

# Unit of Resource Ownership

---

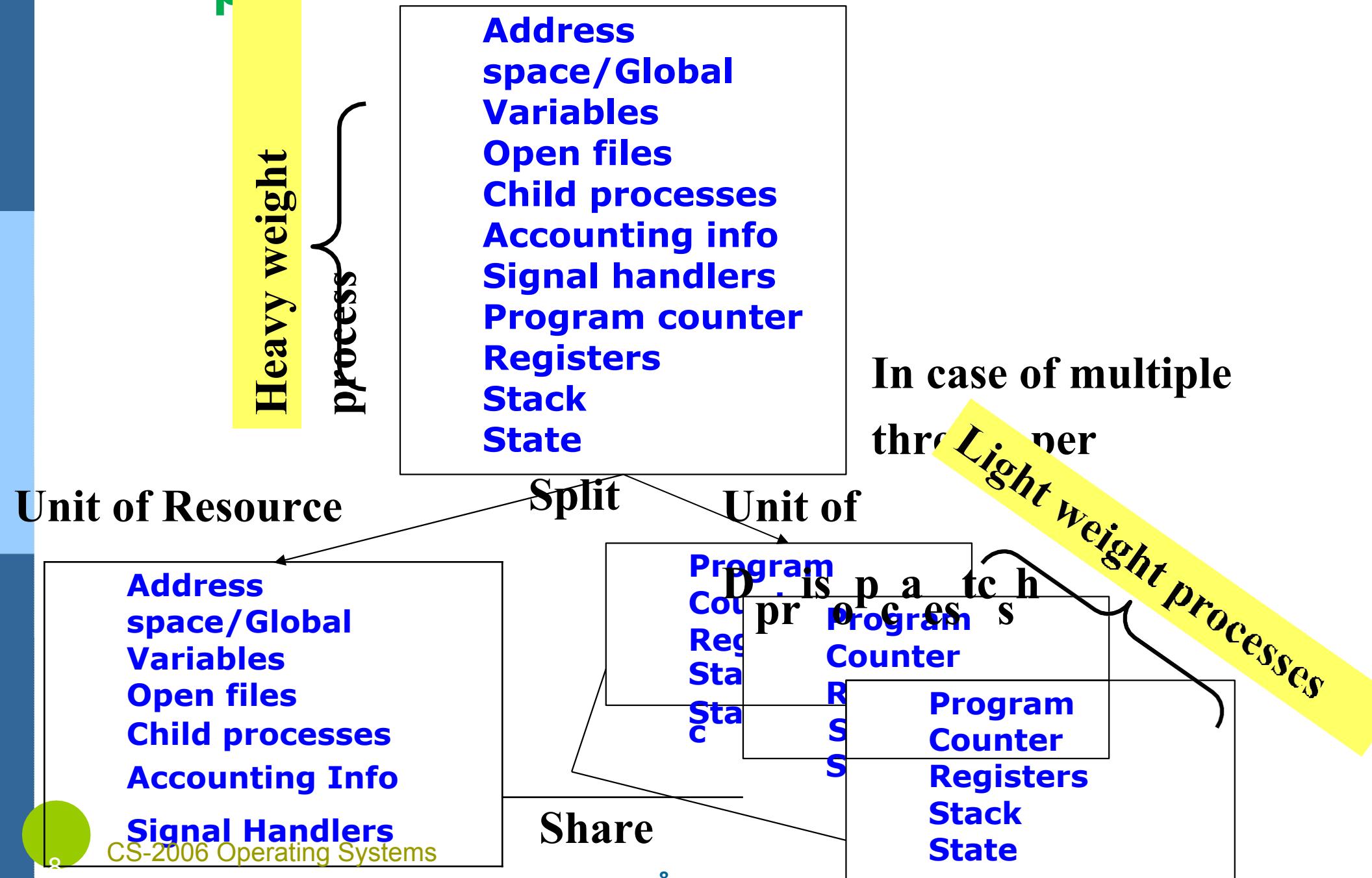
- A process has an
  - Address space
  - Open files
  - Child processes
  - Accounting information
  - Signal handlers
- If these are put together in a form of a process, can be managed more easily

# Unit of Dispatching

---

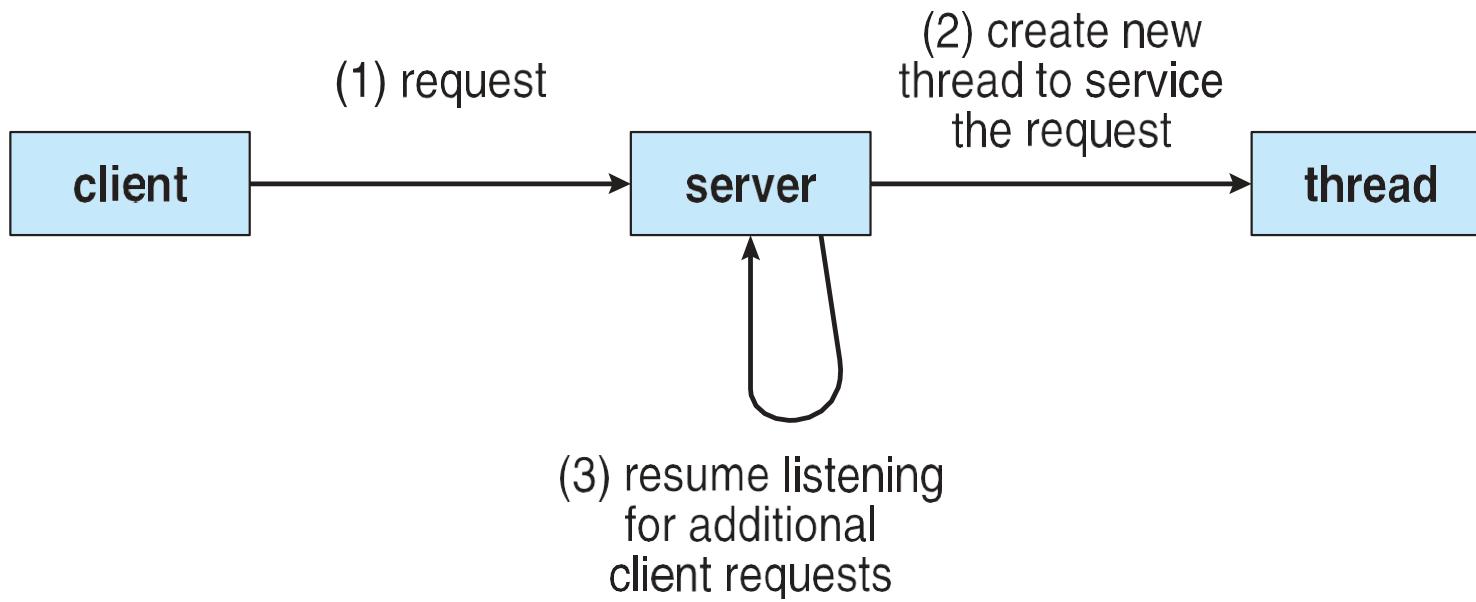
- Path of execution
  - Program counter: which instruction is running
  - Registers:
    - holds current working variables
  - Stack:
    - Contains the execution history, with one entry for each procedure called but not yet returned
  - State
- Processes are used to group resources together
- Threads are the entities scheduled for execution on the CPU
- Threads are also called *lightweight* process (LWP)

# Its better to distinguish between the two concepts



# Multithreaded Server Architecture

---



# What are threads?

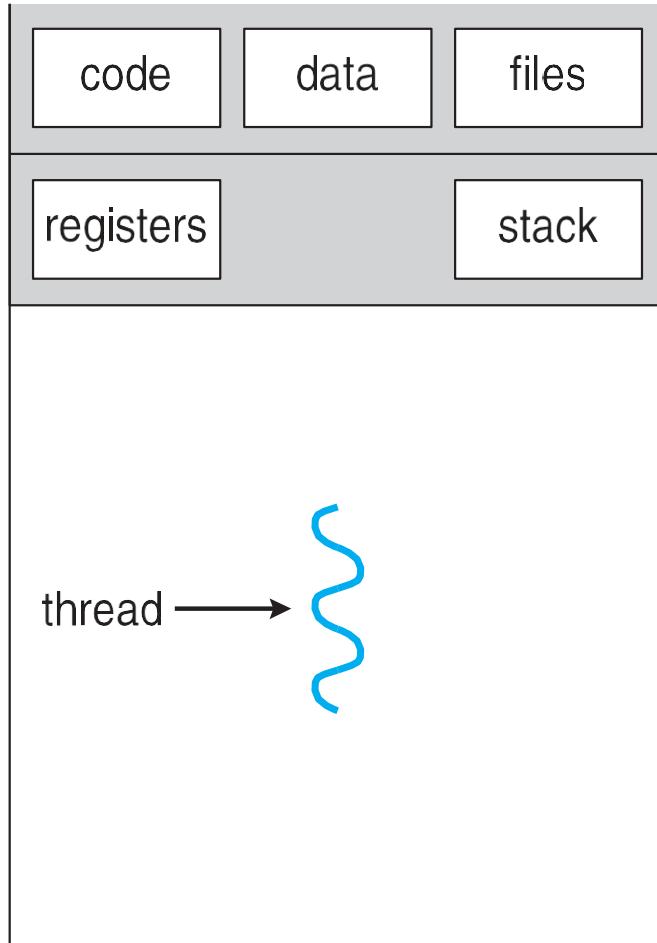


# Thread

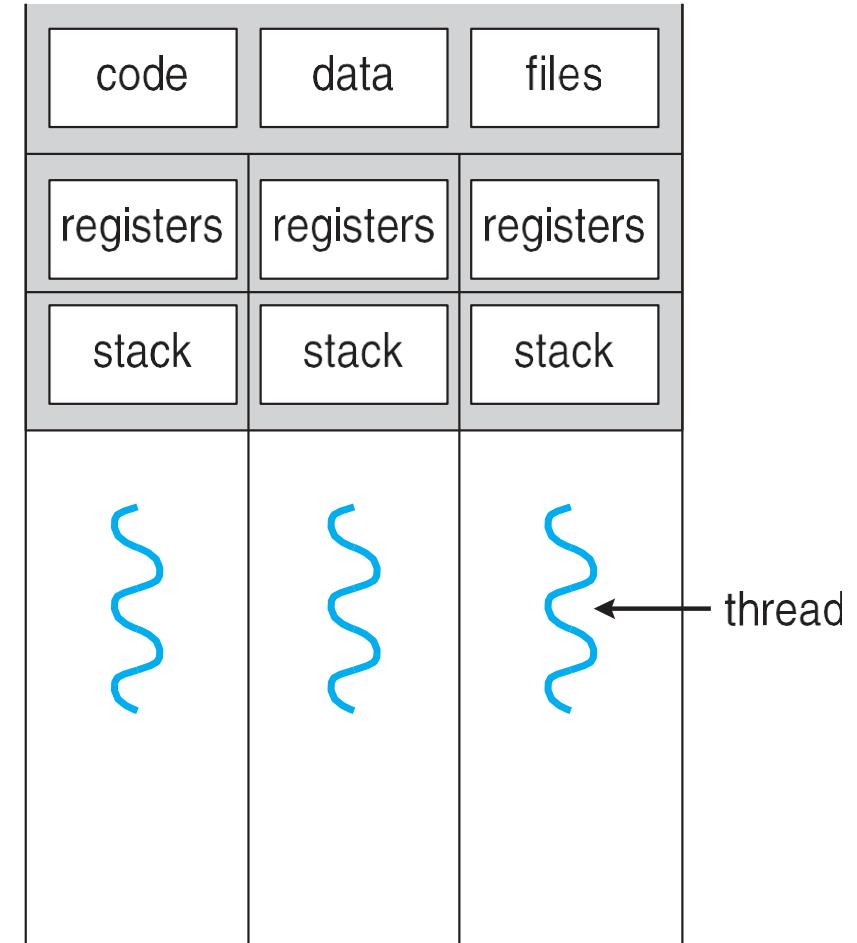
---

- A thread is a flow of execution through the process code.
- A thread will always belong to some process.
- A process can have multiple threads but starts with one only
- Threads share:
  - Same data as that of its process
  - Same code as that of its process
  - Same files as that of its process
- Each thread has its own
  - Register.
  - Stack.
  - Program Counter.

# Single and Multithreaded Processes



single-threaded process



multithreaded process

# Benefits

---

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures

# Process vs. Thread

---

- Process is heavy weight or resource intensive.
- Process switching needs interaction with operating system.
- In multiple processes each process operates independently of the others.
- Thread is light weight, taking lesser resources than a process.
- Thread switching does not need to interact with operating system.
- One thread can read, write or change another thread's data.

# Thread Properties



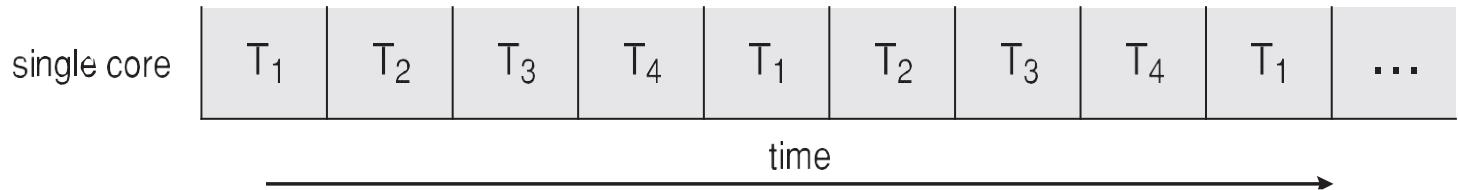
# Multicore Programming

---

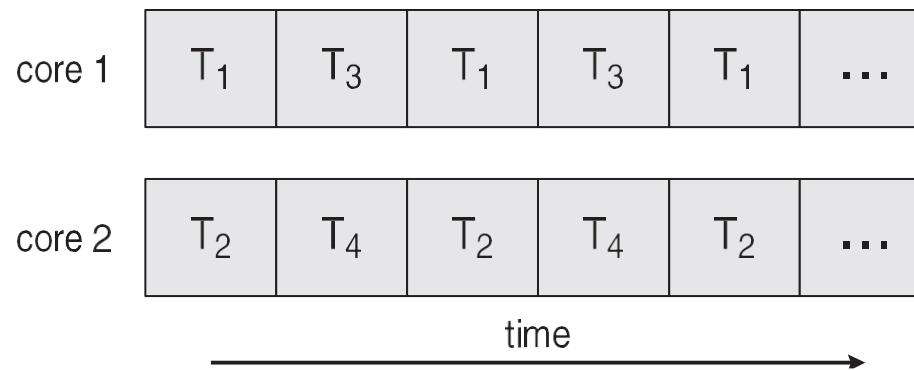
- **Multicore or multiprocessor** systems offer real speed up if multiple threads are used
- Programmers must accurately use threads
  - **Divide** independent problems and assign to threads
  - Keep a **balance** in dividing activities, do not overburden one or more threads
  - **Split Data** too, so threads can work in parallel
  - **Data** should be split properly, so that **dependency** among threads remains minimum
  - **Testing and debugging** of the multi threaded applications is difficult
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** enables more than one task making progress

# Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:



# Multicore Programming (Cont.)

---

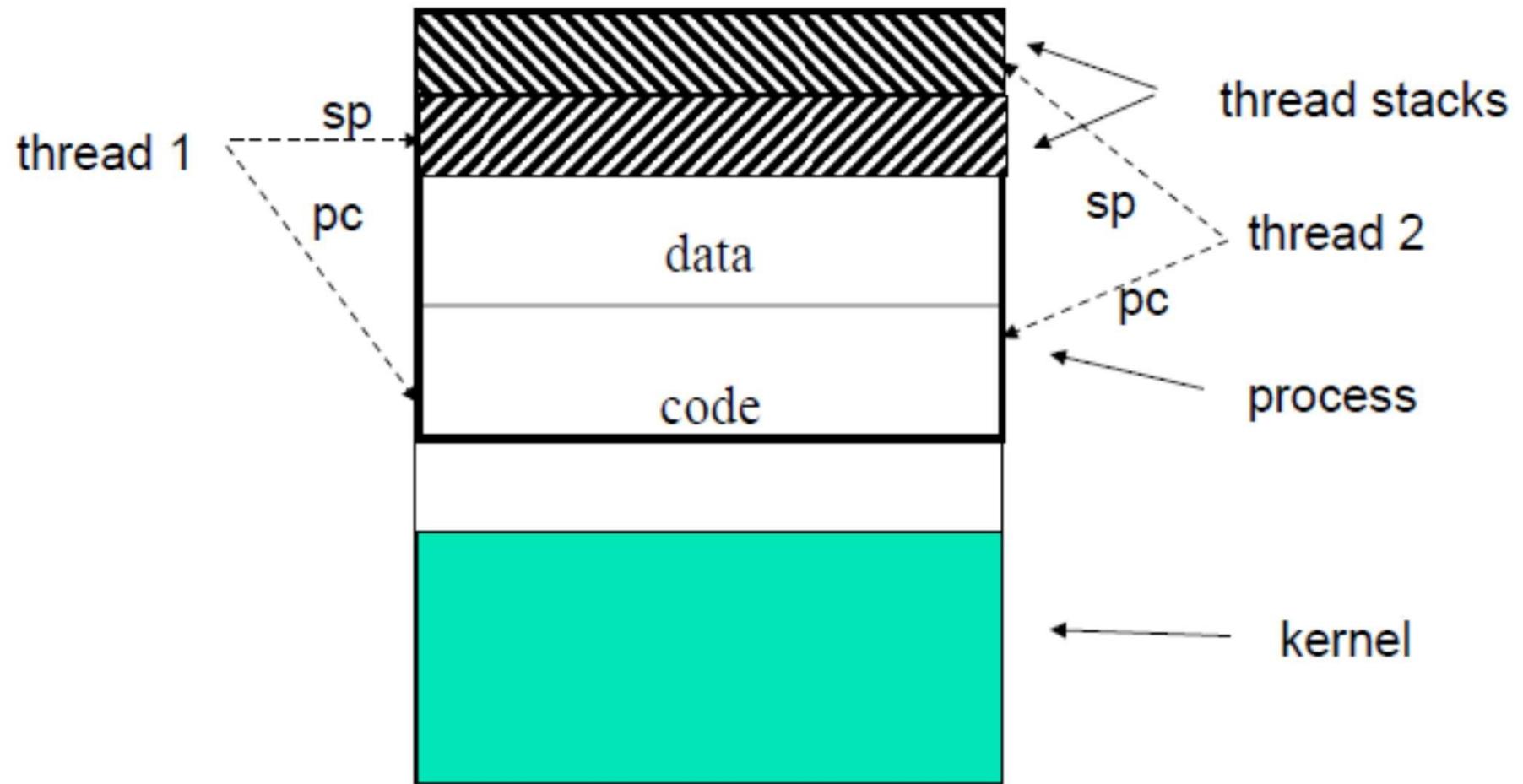
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as *hardware threads*
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

# User Threads and Kernel Threads

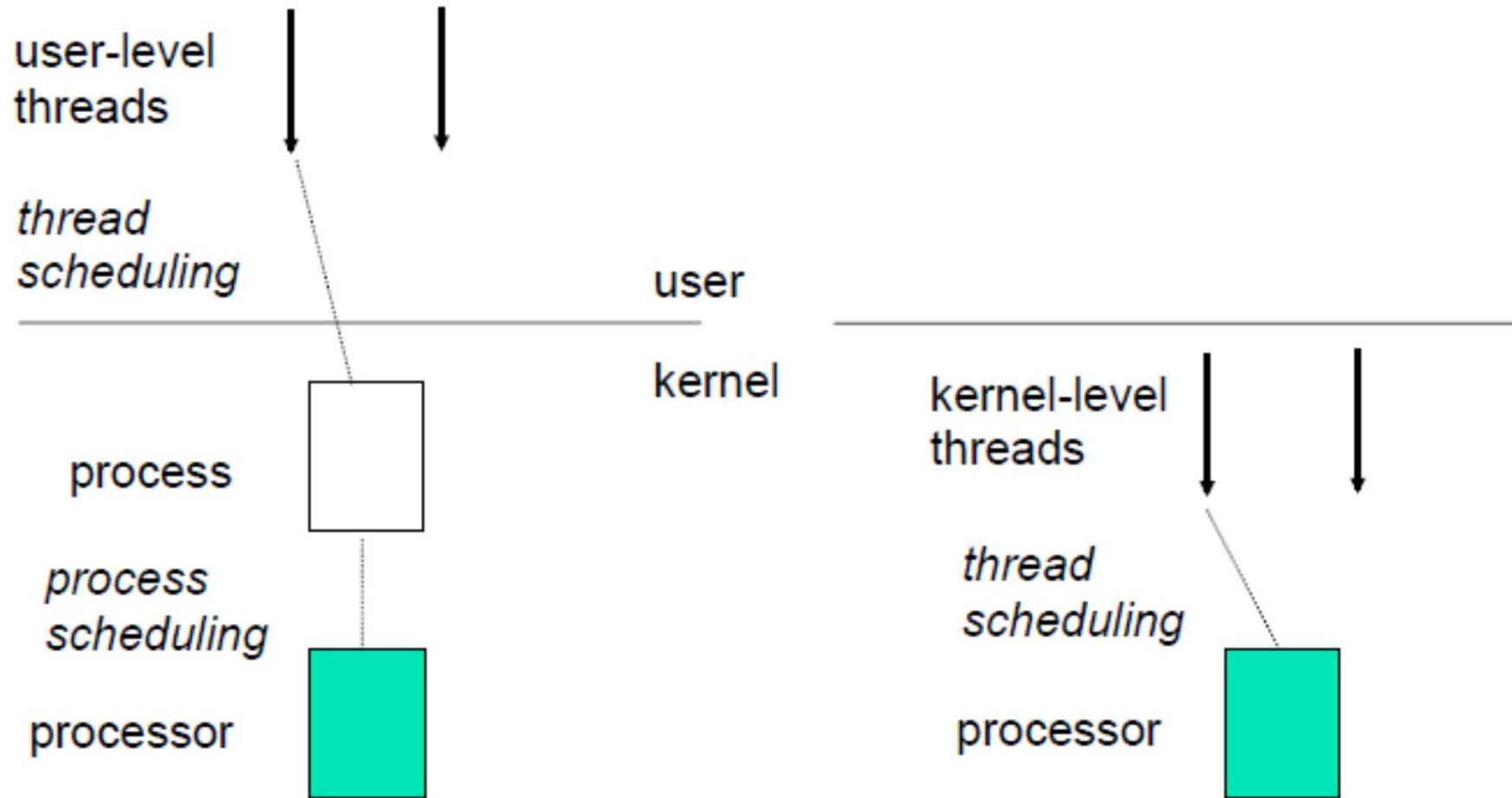
---

- **User threads** - management done by user-level threads library
- The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Operating System managed threads acting on kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

# User-Level Thread Implementation



# User-Level vs. Kernel-Level Threads



# User-level threads vs. kernel-level threads

---

- User-level threads are faster to create and manage.
- Implementation is by a thread library at the user level.
- OS doesn't recognize user level threads.
- Multi-threaded applications cannot take advantage of multiprocessing
- If one user level thread perform blocking operation then entire process will be blocked.
- Kernel-level threads are slower to create and manage.
- Operating system supports creation of Kernel threads.
- OS recognizes kernel-level threads
- Kernel routines themselves can be multithreaded.
- If one kernel thread perform blocking operation then another thread can continue execution.

# Multithreading Models

---

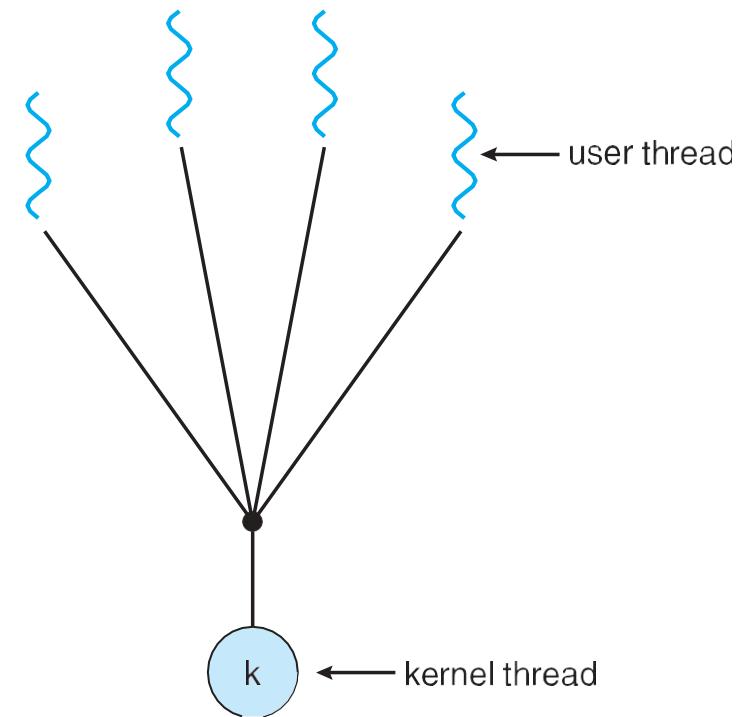
User-level threads can be implemented using any of the three models:

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

# Many-to-One

---

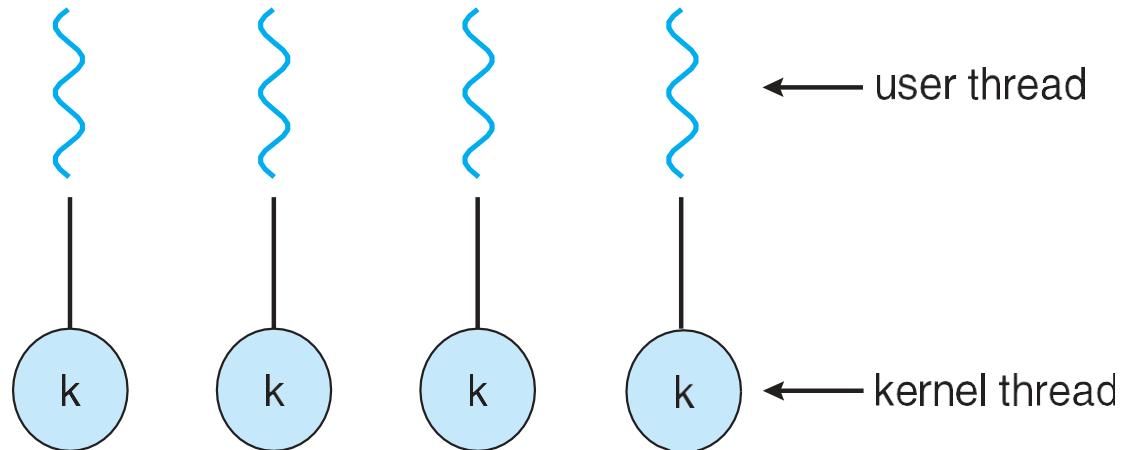
- Synonym of user-level threads
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



# One-to-One

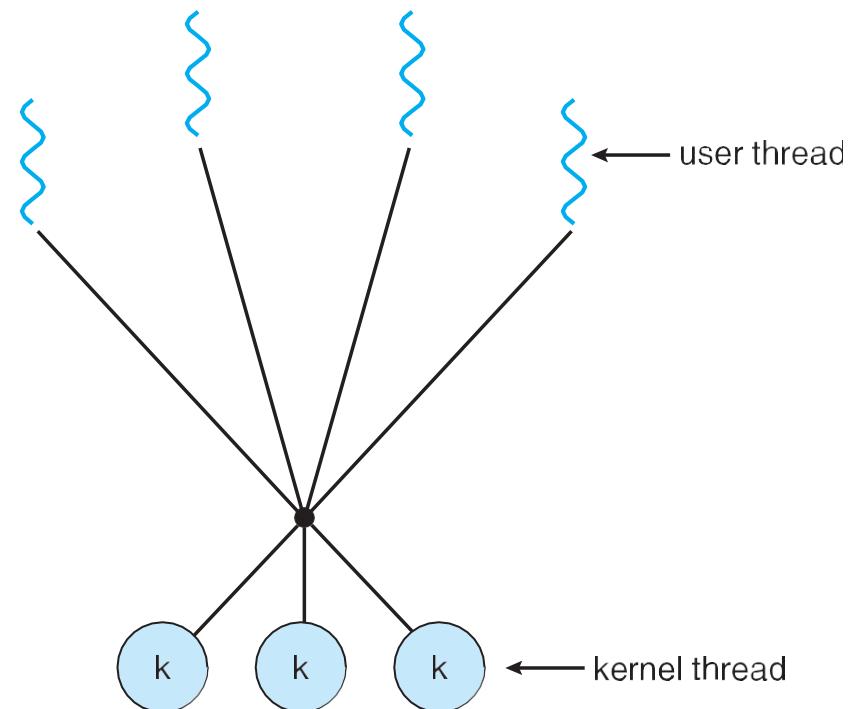
---

- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- One user-level thread is mapped onto one kernel level thread.
  - Even if one thread is blocked, others will still run
- Allows multiple threads to run on multiprocessor system
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



# Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- The number of kernel level threads maybe equal to or less than user level threads
- Allows the operating system to create a sufficient number of kernel threads
- Many-to-one: suffered from lack of concurrency
- One-to-one: suffered from overhead of too many kernel level threads
- Many-to-many suffers from none

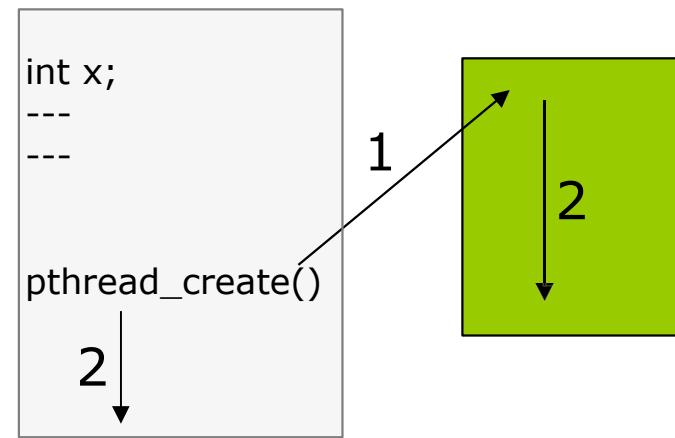
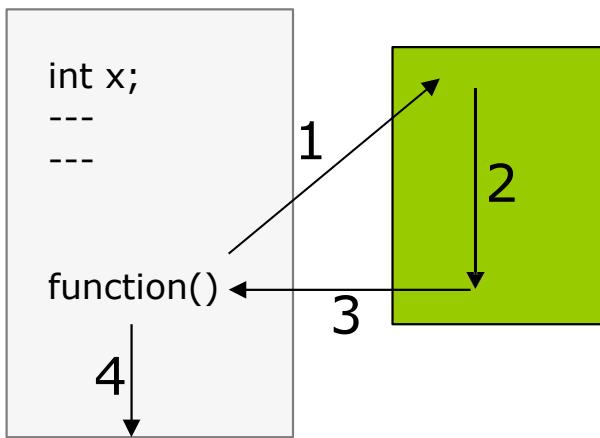


# Threads In Action



# C function call vs. thread creation

---



# Pthread Operations

---

POSIX function	description
<code>pthread_create</code>	create a thread
<code>pthread_detach</code>	set thread to release resources
<code>pthread_equal</code>	test two thread IDs for equality
<code>pthread_exit</code>	exit a thread without exiting process
<code>pthread_kill</code>	send a signal to a thread
<code>pthread_join</code>	wait for a thread
<code>pthread_self</code>	find out own thread ID

# Practical work (explanation)

---

`pthread_create(pthread_t* , NULL, void*, void*)`

- First Parameter is pointer of thread ID it should be different for all threads.
- The second argument points to a `pthread_attr_t` structure whose contents are used at thread creation time to determine attributes for the new thread. Null means use default attr.
- Third perimeter is address of function which we are going to use as thread.
- Forth parameter is argument to function. Only one is allowed.

# Pthread\_create Returning value

---

Returned value:

If **successful**, `pthread_create()` returns 0.

If **unsuccessful**, `pthread_create()` returns -1 and sets `errno` to one of the following values:

# Practical work (explanation)

---

`pthread_join(i pthread_t, void**)`

Pthread join is used in main program to wait for the end of a particular thread.

- First parameter is Thread ID of particular thread
- Second Parameter is used to catch return value from thread.

# pthread\_join - wait for thread termination

---

## **RETURN VALUE**

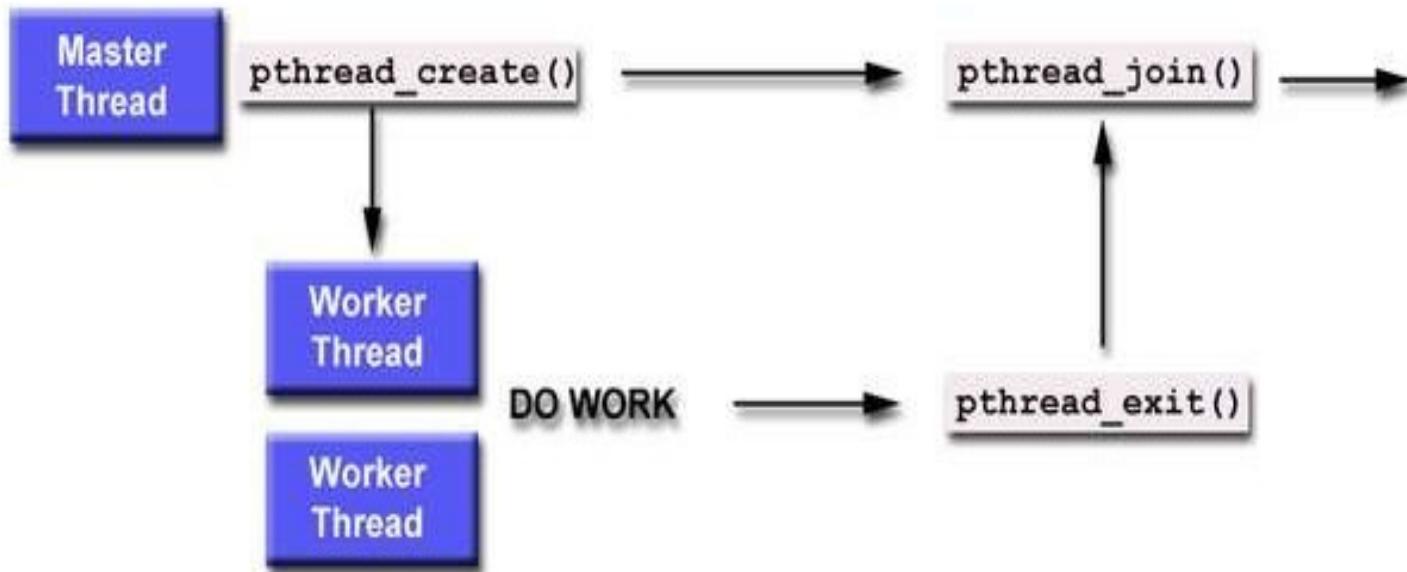
If successful, the `pthread_join()` function returns zero. Otherwise, an error number is returned to indicate the error.

The `pthread_join()` function suspends execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated. On return from a successful `pthread_join()` call with a non-NULL *value\_ptr* argument, the value passed to `pthread_exit()` by the terminating thread is made available in the location referenced by *value\_ptr*. When a `pthread_join()` returns successfully, the target thread has been terminated.

## Exit() function

---

The *pthread\_exit()* function terminates the calling thread and makes the value *value\_ptr* available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and not yet popped are popped in the reverse order that they were pushed and then executed.



```
1 // Importing the Pthread Library
2 #include <pthread.h>
3 // Importing the library for printf
4 #include <stdio.h>
5
6 // This function takes in a pointer
7 void * ThreadFunction(void *arguments)
8 {
9     printf("Thread Running\n");
10    return NULL;
11 }
12
13 int main() {
14
15     // Creating the Location where the ID of the newly created
16     // thread will be stored
17     pthread_t thread1;
18
19     printf("Calling Thread \n");
20
21     // Creating the thread
22     pthread_create(&thread1, NULL, ThreadFunction, NULL);
23     return 0;
24 }
```

# Mutithreading

---

- Threads in Arrays
- Threads in Arrays with command line arguments

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

#define N 5

void *worker_thread(void *arg)
{
    printf("This is worker_thread # %ld\n", (long) arg);
    pthread_exit(NULL);
}

int main()
{
    pthread_t my_thread[N];

    long id;
    for(id = 1; id <= N; id++) {
        int ret = pthread_create(&my_thread[id], NULL,
&worker_thread, (void*)id);
        if(ret != 0) {
            printf("Error: pthread_create() failed\n");
            exit(EXIT_FAILURE);
        }
    }

    pthread_exit(NULL);
}
```

# Output

---

```
This is worker_thread #5  
This is worker_thread #4  
This is worker_thread #3  
This is worker_thread #2  
This is worker_thread #1
```

```
void * f1(void * arg);
int main(int argc, char* argv[]){
    char* msg[] = {"Thread1", "Thread2", "Thread3", "Thread4", "Thread5"};
    pthread_t tids[5];
    for(int i=0; i<5; i++)
        pthread_create(&tids[i], NULL, f1, (void*)msg[i]);
    for(int i=0; i<5; i++)
        pthread_join(tids[i], NULL);
    printf("main(): Reporting that all child threads have terminated\n");
    exit(0);
}
```

```
void * f1(void * arg){
    printf("I am child %s\n", (char*)arg);
    pthread_exit(NULL);
}
```

---

**with command line arguments**

```
int thread_no = 0;

void * f1(void * arg);
int main(int argc, char* argv[]){
    if(argc != 2){
        printf("Invalid arguments, must pass one integer value...\n");
        exit(1);
    }
    int ctr = atoi(argv[1]);
    pthread_t* tid = (pthread_t*)malloc(sizeof(pthread_t)*ctr);
    for(int i=0;i<ctr;i++)
        pthread_create(&tid[i], NULL, f1, NULL);
    for(int j=0;j<ctr;j++)
        pthread_join(tid[j], NULL);
    printf("main(): Reporting that all child threads have terminated\n");
    exit(0);
}

void * f1(void * arg){
    int thread_id = thread_no; thread_no++;
    fprintf(stderr, "I am child thread number %d \n", thread_id);
    pthread_exit(NULL);
}
```

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

void * PrintHello(void * data)
{
    int my_data = (int) data;

    printf("\n Hello from new thread - got %d !\n", my_data);
    pthread_exit(NULL);
}

int main()
{
    int rc;
    pthread_t thread_id;

    int t = 11;

    rc = pthread_create(&thread_id, NULL, PrintHello, (void*)t);
    if(rc)
    {
        printf("\n ERROR: return code from pthread_create is %d \n", rc);
        exit(1);
    }
    printf("\n Created new thread (%u) ... \n", thread_id);

    pthread_exit(NULL);
```

# Output

---

Created new thread (4) ...

Hello from new thread - got 11

# Questions

---

- What happens when a thread calls an `exit()` function?
- Can a function that is a part of the program be called from a thread?
- What happens when you call a `fork()` in a thread?
- What happens when you call `pthread_exit()` in the main thread?

```
void *functionC(void *);

int main ()
{
    int rc;
    pthread_t th;

    if(rc = pthread_create(&th, NULL, &functionC, NULL))
    {
        printf("Thread creation failed, return code %d, errno %d",
rc,
               errno);
    }

    printf("Main thread %lu: Sleeping for 20 seconds\n", pthread_self());
    fflush(stdout);
    sleep(20);
    pthread_exit(NULL);
    printf("Main thread %lu: This will not be printed as we already
called
               pthread_exit\n", pthread_self());
    exit(0);
}

void *functionC(void *)
{
    printf("Thread %lu: Sleeping for 20 second\n", pthread_self());
    sleep(20);
    printf("Thread %lu: Came out of first and sleeping again\n",
pthread_self());
    sleep(20);
    printf("CThread %lu: Came out of second sleep\n", pthread_self());
}
```

# Output

---

```
Main thread 140166909204288: Sleeping for 20 seconds
Thread 140166900684544: Sleeping for 20 second
Thread 140166900684544: Came out of first and sleeping again
CThread 140166900684544: Came out of second sleep
```

```
void *thread_fnc(void * arg);

char thread_msg[] = "Hello Thread!"; // global

int main()
{
    int ret;
    pthread_t my_thread;
    void *ret_join;

    ret = pthread_create(&my_thread, NULL, thread_fnc, (void*) thread_msg);
    if(ret != 0) {
        perror("pthread_create failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    ret = pthread_join(my_thread, &ret_join);
    if(ret != 0) {
        perror("pthread_join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined, it returned %s\n", (char *) ret_join);
    printf("New thread message: %s\n", thread_msg);
    exit(EXIT_SUCCESS);
}

void *thread_fnc(void *arg)
{
    printf("This is thread_fnc(), arg is %s\n", (char*) arg);
    strcpy(thread_msg, "Bye!");
    pthread_exit("Exit from thread");  
48
}
```

# Output

---

```
Waiting for thread to finish...
This is thread_fnc(), arg is Hello Thread!
Thread joined, it returned 'Exit from thread'
New thread message: Bye!
```

# Practice Problem

---

Write a program in which you are going to create a thread. The main thread will pass an integer value N to the newly spawned thread. This thread will sum the numbers starting from 1 till the number N. The main thread will display the updated sum. Remember that the main thread will have to wait for the spawned thread to complete its work.

**SKELETON:**

```
int sum;
void *runner(void
*param); main()
{
    pthread_create(tid, attr, runner,
    int_arg);    Pthread_join(tid);
}
void * runner(void *param)
{
    sum...
    pthread_exit(sum);
}
```

# Practice

---

- Write a program that gets the following problem solved in two threads.
  - $(3 * 10) + (5 * 4)$
  - First half via thread 1 and second half via thread 2
- Write the program in which you will search for the two halves of an int array for a value, each half using one thread.

# Non-deliverable assignment

---

- Write a program in which you create a thread that calculates the factorial of a number n that is input by a user. The thread attributes must be changed are:
  - Scope
  - Scheduling priority
- Write a program that prints the first 15 numbers(starting from 1-15) each after a delay of one second using threads
- Why do we fork when we can thread?

---

# Example Solution

# Solution Practice example\_2 1/5

---

```
#include <iostream>
#include<pthread.h>
#include<stdio.h>
using namespace std;
#include <string.h>

//arguments that we pass in the function
// we create struct if we have to pass multiple arguments

struct arg_struct {
    int array[10]={1,2,3,4,5,6,7,8,9,10};
    int size=10;
    int search=11;
};
```

# Solution Practice example\_2 2/5

---

```
//function that we use in threads
// to search in 1st half
void* myFunc1(void* arguments)
{
    struct arg_struct *args = (struct arg_struct *)arguments;
    int *val_p=args->array;

    int *val=new int;

    int i = 0;
    for( i = 0; i < args->size/2; i++){
        if(val_p[i]==args->search){
            *val=val_p[i];
        }
    }
    pthread_exit( (void*) val);
}
```

# Solution Practice example\_2 3/5

---

```
// to search in 2nd half  
void* myFunc2(void* arguments)  
{  
    struct arg_struct *args = (struct arg_struct  
    *)arguments; int *val_p=args->array;  
    int *val=new int;  
  
    int i = 0;  
    for( i = args->size/2; i < args->size;  
        i++){if(val_p[i]==args->search){  
  
        *val=val_p[i];  
    }  
}  
pthread_exit( (void*) val); }
```

# Solution Practice example\_2 4/5

---

```
int main()
{
    // creating id for thread
    pthread_t thread1_id,thread2_id;
    struct arg_struct thread_var;

    // creating threads
    if (pthread_create(&thread1_id, NULL, &myFunc1,(void *) &thread_var)==-1)
    {
        cout<<"Thread Creation Failed!"<<endl;
        return 1;
    }
    if (pthread_create(&thread2_id, NULL, &myFunc2,(void *) &thread_var)==-1)
    {
        cout<<"Thread Creation Failed!"<<endl;
        return 1;
    }
```

# Solution Practice example\_2 5/5

---

```
// variables to store the return value  
int *ptr_t1_ret,*ptr_t2_ret;  
  
// wait for the end of a particular thread  
pthread_join(thread1_id, (void**) &ptr_t1_ret);  
pthread_join(thread2_id, (void**) &ptr_t2_ret);  
  
// display result  
if(*ptr_t1_ret>0){  
    cout<<"found in first half of array";  }  
else if(*ptr_t2_ret>0){  
    cout<<"found in 2nd half of array";  }  
else  
{cout<<"not found";}  
}
```

---

**Thanks**