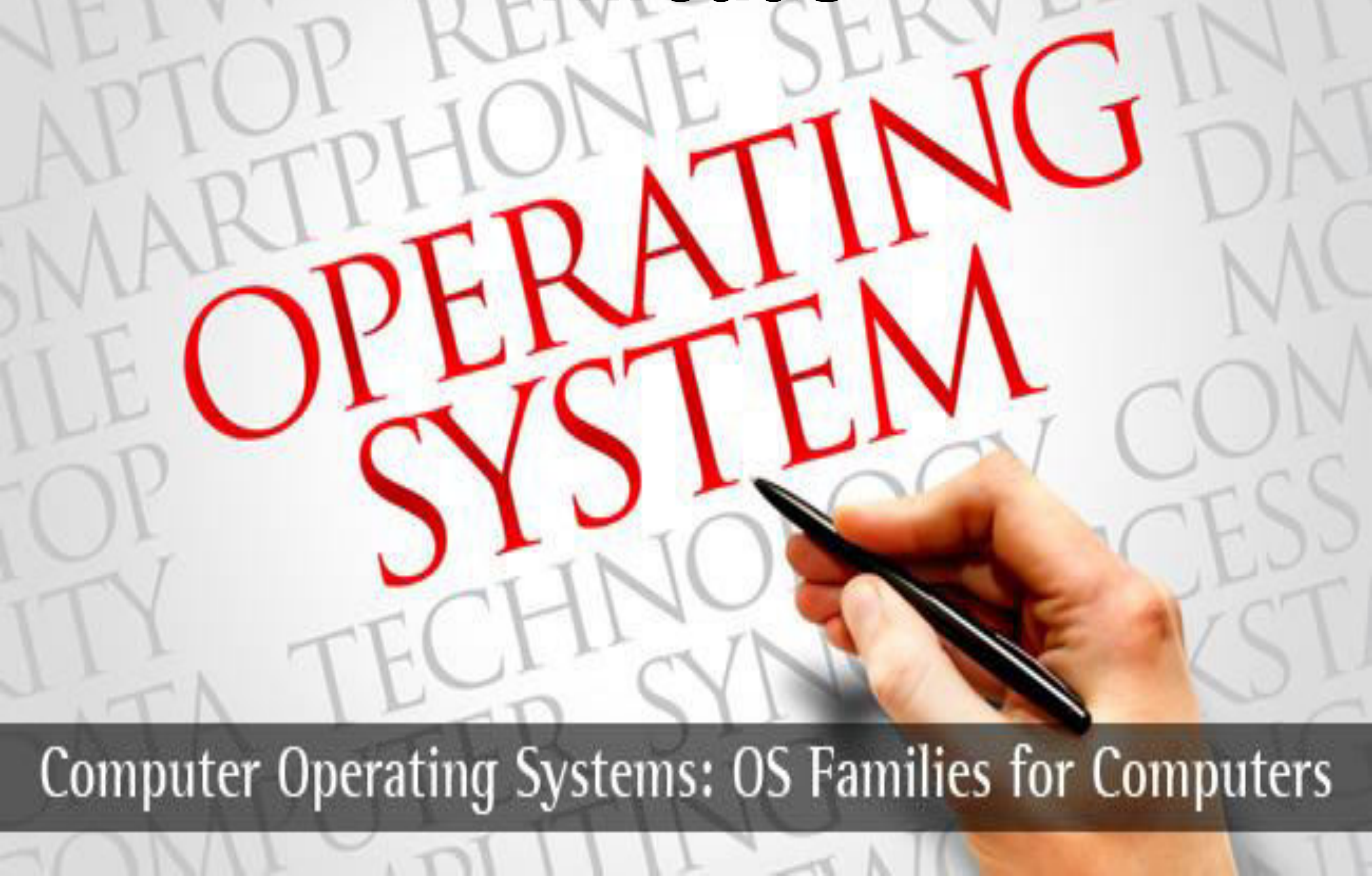
CS-3103 : Operating Systems : Sec-A (NB) : Threads



Interprocess Communications

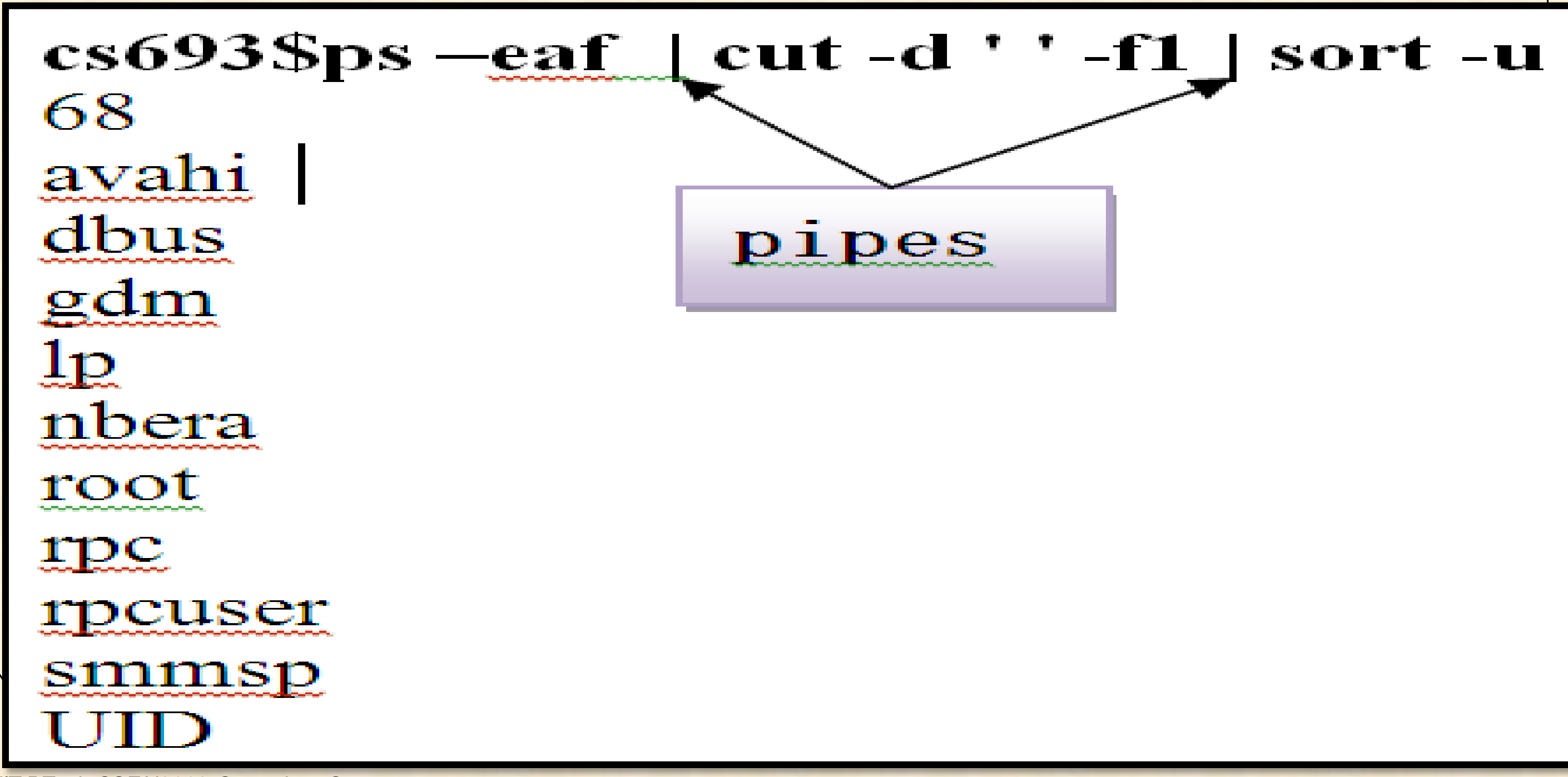
- Communication is a means of sharing information between at least two agent.
- A communication consists of an individual sending some information and another receiving that information.
- Five elements are involved in a communication: 1) ***the situation (when, where and what circumstance) that triggers the communication***, 2) ***the information to be communicated***, 3) ***the sender of the information***, 4) ***the medium of transmitting the information***, and 5) ***the receiver of the information***.

Interprocess Communications…

- The agents involved in communications in operating systems are ***processes***, ***threads***, ***kernel*** and ***processors***.
- In systems involving multiple processes, concurrent processes may interact with one another to 1) ***exchange information to achieve some common goal***, and 2) ***coordination (i.e., synchronization) of activities of the participating processes***.
- Each interaction involves exchanging a finite amount of information (i.e., data transfer) between two or more processes.

IPC – PIPE …

- In UNIX systems, pipes are normally used as unidirectional byte streams which connect the standard output from one process to the standard input of another process.

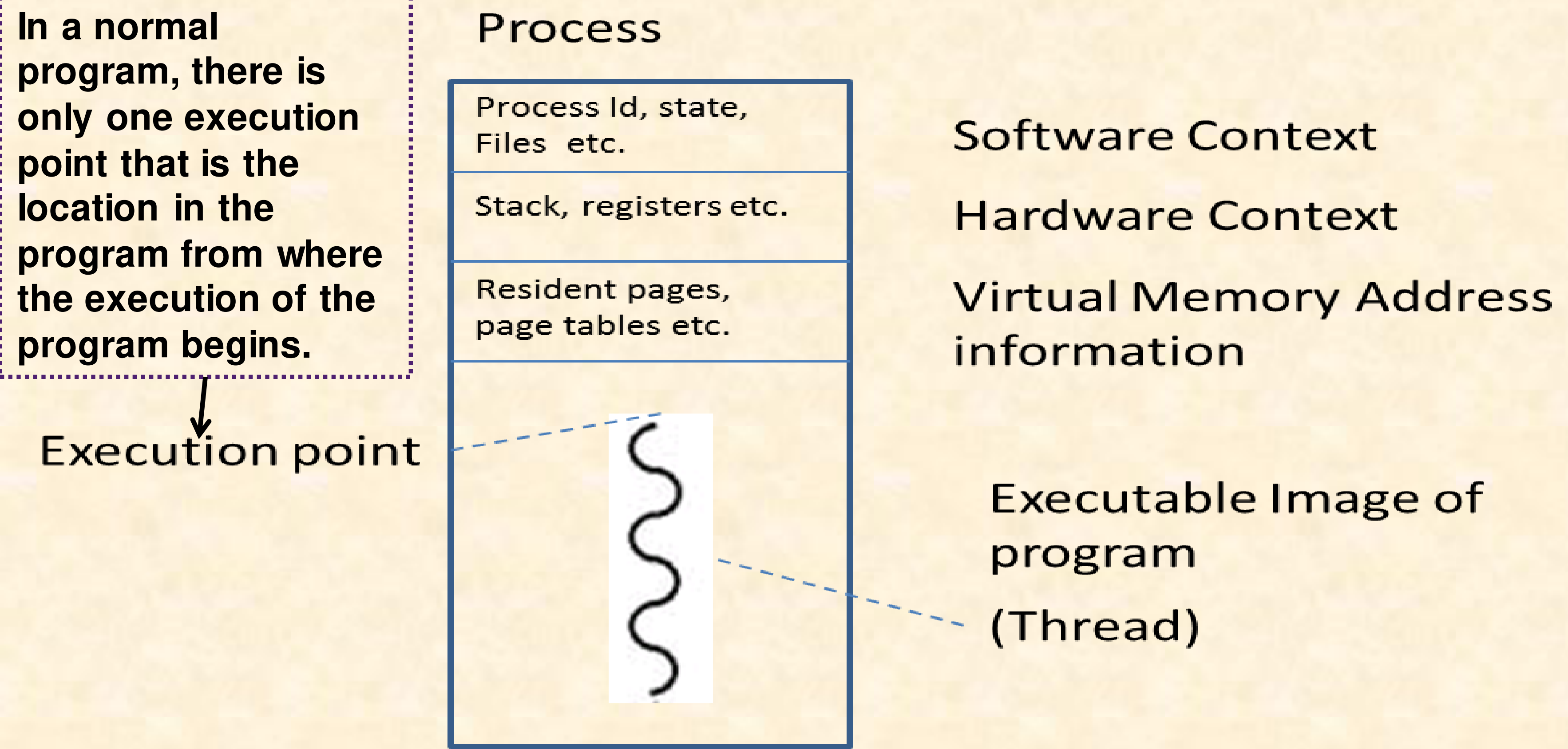


```
cs693$ps —eaf | cut -d ' ' -f1 | sort -u
68
avahi |
dbus
gdm
lp
nbera
root
rpc
rpcuser
smmsp
UID
```

Let's learn Threads

- A process comprises four components:
  - Executable image of the program
  - Software context
  - Hardware context
  - Virtual memory address information

In a normal program, there is only one execution point that is the location in the program from where the execution of the program begins.



Execution point
Process
Process Id, state, Files etc.
Stack, registers etc.
Resident pages, page tables etc.
Software Context
Hardware Context
Virtual Memory Address information
Executable Image of program
(Thread)

Let’s learn Threads …..

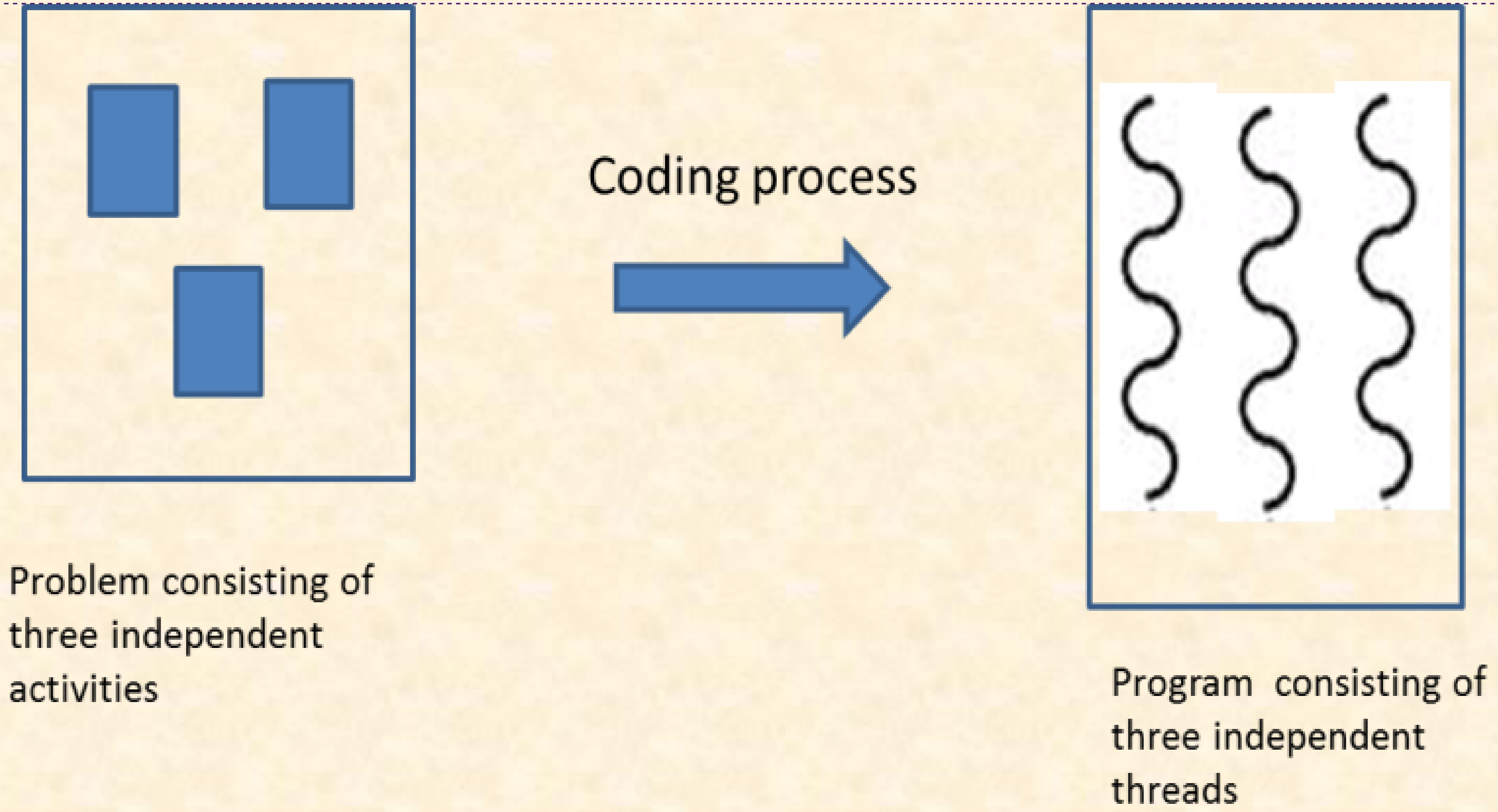
- The execution point is the address of the first executable statement in a program;
- Thus, this address is initially stored in the hardware context.
- When the process is dispatched for execution, this address gets automatically loaded into the program counter (PC) with the context of the process.
- As execution progresses, the contents of the program counter are incremented or changed.
- **Note**: *There is only one sequence of instruction in a normal process*.
- ***This sequence of instruction is also called a thread***.

Let’s learn Threads

- ***A thread is a sequence of instructions within a process.***
- ***It is an executable entity that runs in the address space of its container process, and uses the resources of the process.***
- We have seen that a process switch involves an overhead in the form of saving and loading outgoing and incoming processes respectively.
- This overhead becomes more significant when the process is I/O bound because such a process frequently occupies and leaves the CPU.
- In order to reduce this overhead, modern OS allow the inclusion of multiple execution streams in a single process. This means if a programmer can identify different activities in the program that can run independently of each other, then he/she may include ***multiple threads*** in the program (figure in the next slide)…..

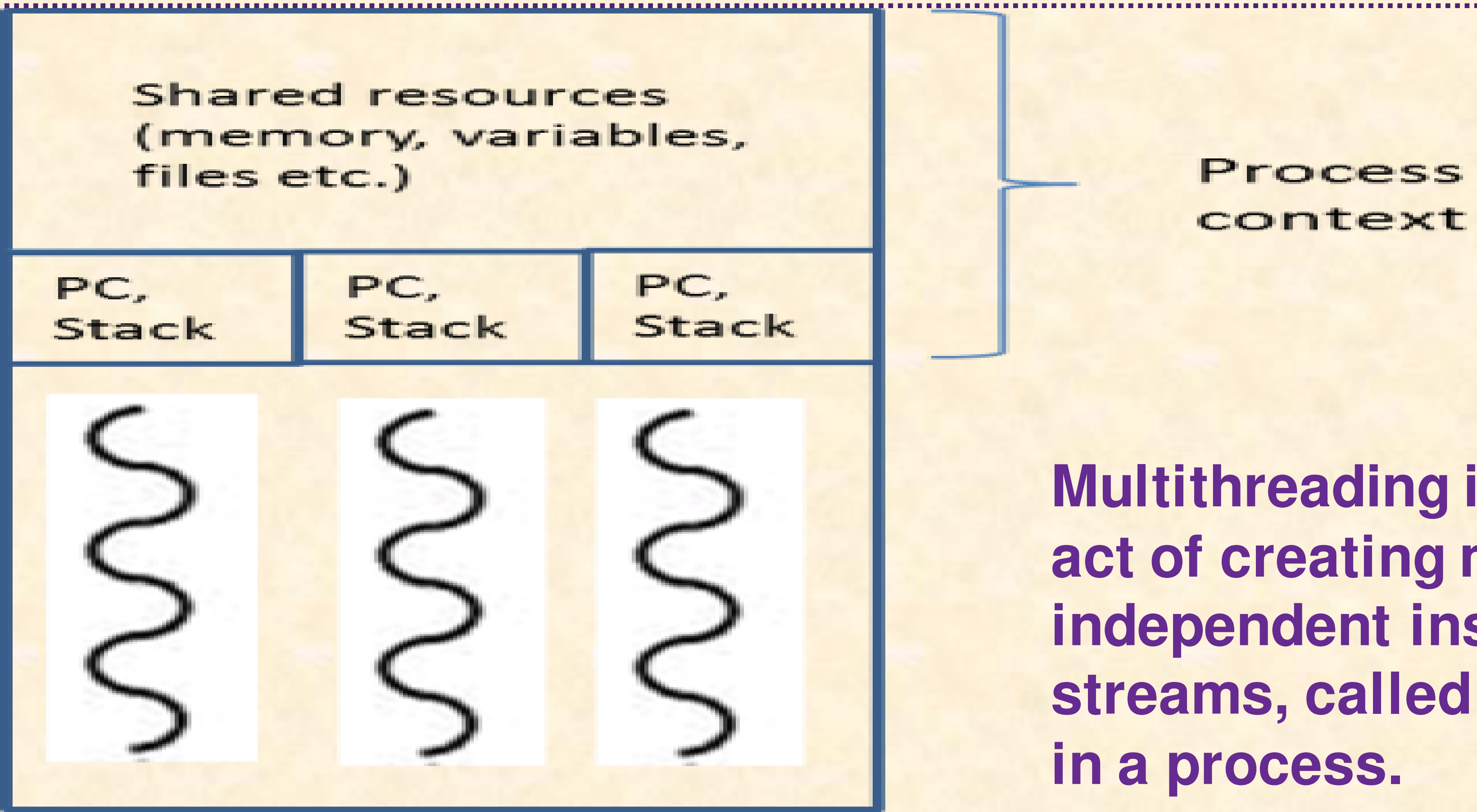
Let’s learn Threads

❑ **Note**: The program represented in the following figure has three concurrent threads; hence three execution points. The term “***concurrent threads***” means that the threads have an overlap of CPU execution time.


Coding process
Problem consisting of three independent activities
Program consisting of three independent threads

Let’s learn Threads

When the OS loads this ***multithreaded program*** into main memory, it opens a process for this program. Various threads of the process (in this case 4, one for the main program where the entire code of execution is written plus three threads created through pthread_create(a,b,c,d) system call) will share most of the resources such as heaps, file, tables, devices etc. However, each thread will have its own thread context consisting of stack and program counter, state, and registers. Therefore, a separate area is allocated for a thread in the hardware context of process control block, for storing thread context. Shown in the picture ➔



Multithreading is the act of creating multiple independent instruction streams, called threads, in a process.

Pthreads

- Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.
- pthread_t tid → declares the identifier for the thread we will create.
- Each thread has a set of attributes including stack size and scheduling information.
- Pthread_attr_t attr → represents attributes for the thread.
- Set the attributes → pthread_attr_init(&attr)
- A separate thread is created with pthread_create function call with 4 arguments.

Pthreads (continued)

- pthread_create(&tid, &attr, function, parameters for func)
- &tid → pass thread identifier
- &attr → pass attributes for the thread
- Function → where the new thread will begin execution
- Parameters (if any) required for the Function execution

Program to create a thread with Pthreads library… Run this program and check the attributes for thread creation, pthread_create(a,b,c,d) and pthread_join(a,b)….

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

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
  sleep(1);
  printf("Printing  GeeksQuiz  from  Thread  \n");
  return  NULL;
}

int main()
{
  pthread_t  thread_id;
  printf("Before  Thread\n");
  pthread_create(&thread_id,  NULL,  myThreadFun,  NULL);
  pthread_join(thread_id,  NULL);
  printf("After  Thread\n");
  exit(0);
}
```

Result of running a Thread program…

In main() we declare a variable called thread_id, which is of type pthread_t, which is an integer used to identify the thread in the system. After declaring thread_id, we call pthread_create() function to create a thread.

pthread_create() takes 4 arguments.

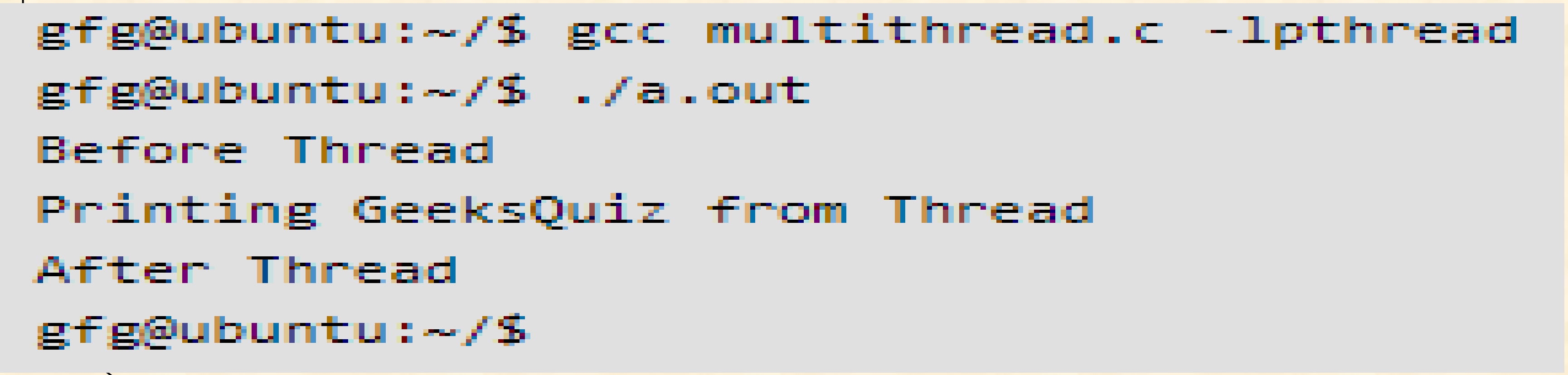
The first argument is a pointer to thread_id which is set by this function.

The second argument specifies attributes. If the value is NULL, then default attributes shall be used.

The third argument is name of function to be executed for the thread to be created.

The fourth argument is used to pass arguments to thread.

The pthread_join() function for threads is the equivalent of wait() for processes. A call to pthread_join blocks the calling thread until the thread with identifier equal to the first argument terminates.

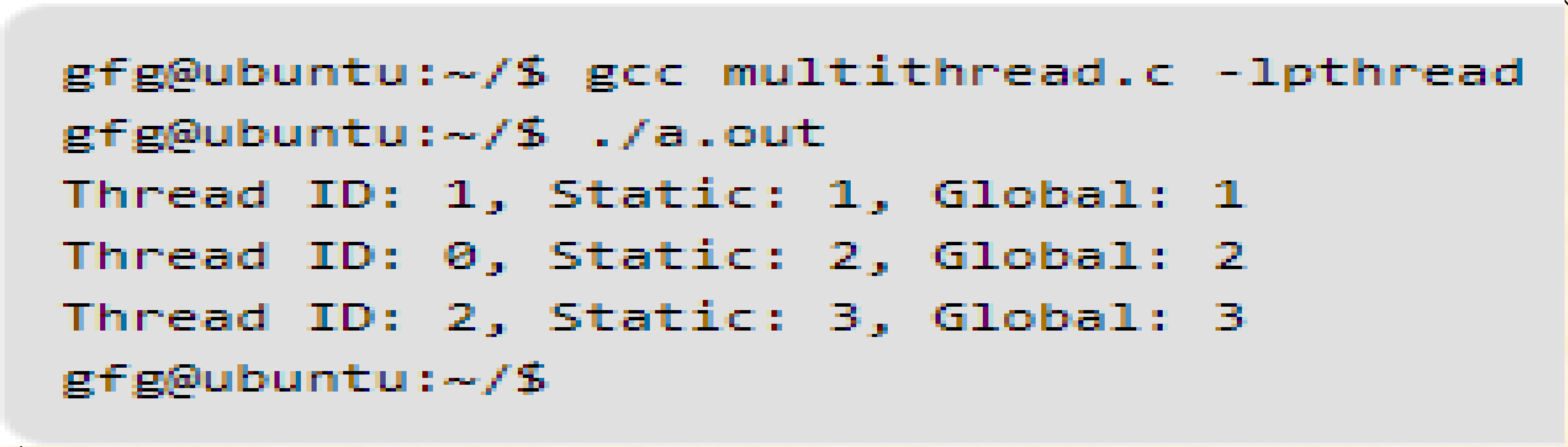


```
gfg@ubuntu:~/$ gcc multithread.c -lpthread
gfg@ubuntu:~/$ ./a.out
Before Thread
Printing GeeksQuiz from Thread
After Thread
gfg@ubuntu:~/$
```

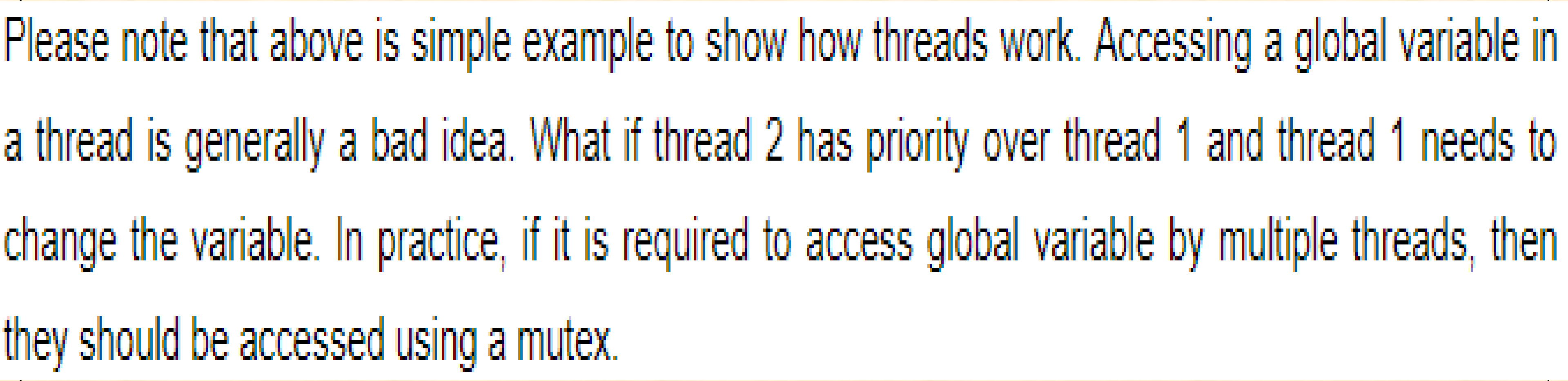
Program to show Multithreading with Pthreads library…

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

// Let us create a global variable to change it in threads
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void  *vargp)
{
    // Store the value argument passed to this thread
    int *myid = (int *)vargp;  // variable-global-pointer

    // Let us 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);
}

int main()
{
    int i;
    pthread_t tid;

    // Let us create three threads…………………………………….. Check the output
    for (i = 0; i < 3; i++)
        pthread_create(&tid, NULL, myThreadFun,  (void *)&i);

    pthread_exit(NULL);
    return 0;
```

Result of running Multiple Threads…



```
gfg@ubuntu:~/$ gcc multithread.c -lpthread
gfg@ubuntu:~/$ ./a.out
Thread ID: 1, Static: 1, Global: 1
Thread ID: 0, Static: 2, Global: 2
Thread ID: 2, Static: 3, Global: 3
gfg@ubuntu:~/$
```



Please note that above is simple example to show how threads work. Accessing a global variable in a thread is generally a bad idea. What if thread 2 has priority over thread 1 and thread 1 needs to change the variable. In practice, if it is required to access global variable by multiple threads, then they should be accessed using a mutex.

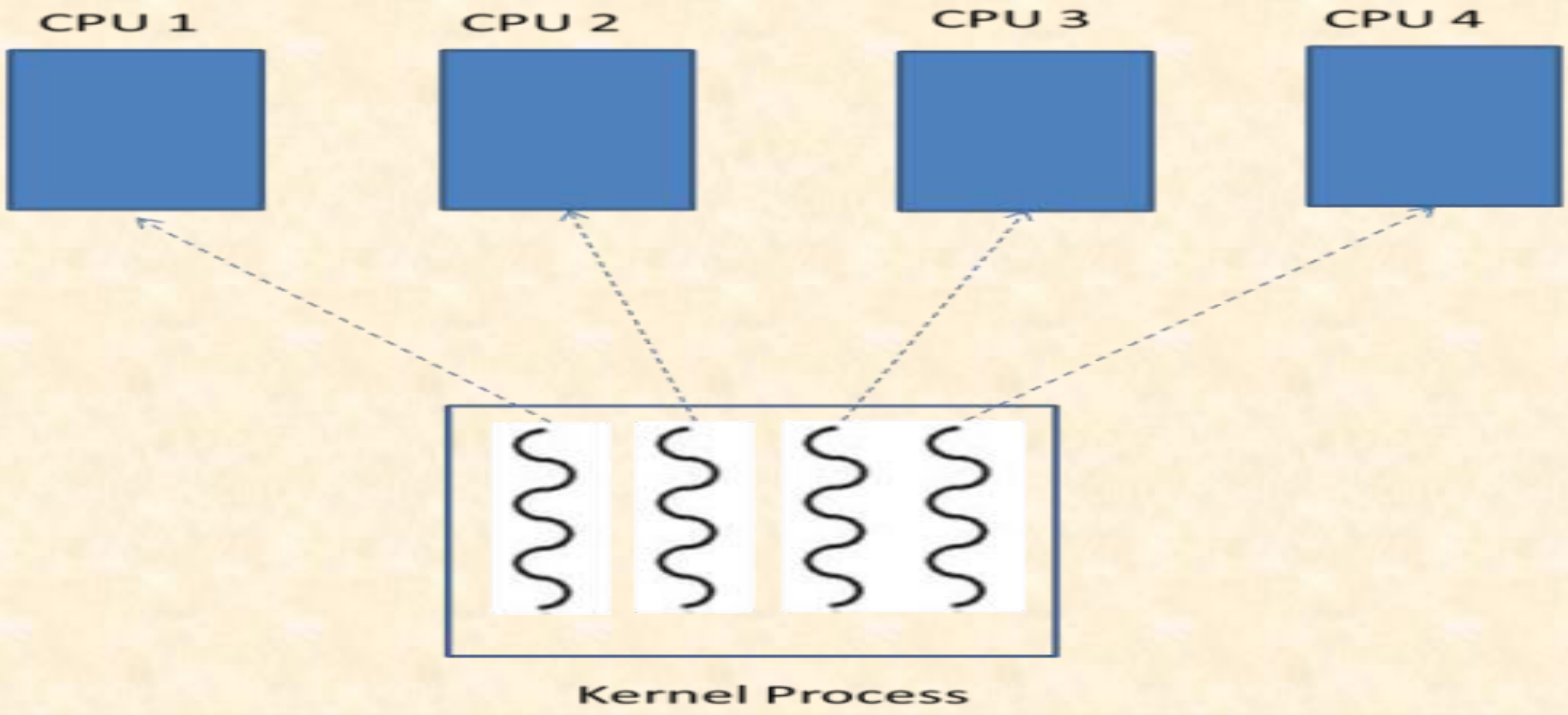
Types of Threads

- There are two types of threads:
  - User level threads
  - Kernel level threads
- There is a major difference in terms of visibility of threads.
  - ***The kernel level threads are visible to the OS*** whereas the ***user level threads are not visible to the OS. The user level threads are visible only to the user. Thus, threads are visible only to their owners.***

Scheduling of Threads

❑ Since the OS is the owner of kernel level threads, it can schedule these threads. In a multiprocessor environment, the OS can map its threads on different processors. *In the following figure, the OS has scheduled 4 threads on 4 different CPUs ➔ utilizing the power of parallel hardware available to the system.*

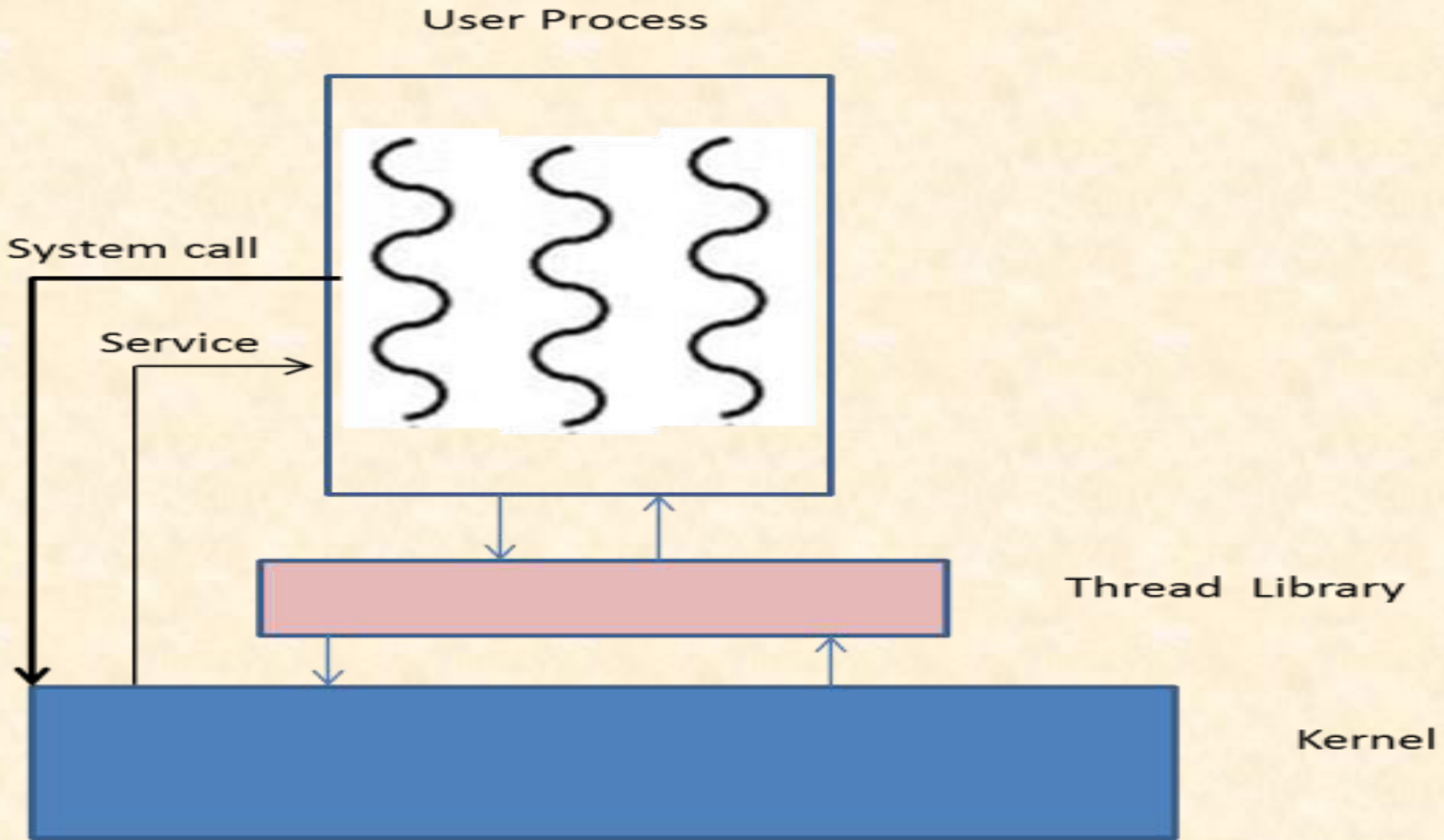
❑ *Example: the file server, exception handler, memory manager, and other processes of the kernel and coded as multithreaded processes.*



Lets Get Started

Scheduling of Threads

- User level threads are not visible to the OS.
- The kernel cannot directly schedule these threads.
- The user can manage his threads with the help of a thread library.



Scheduling of Threads (*We are talking about java threads*)

- ❑ Considering a single CPU design, threads run one at a time in such a way as to provide an illusion of concurrency.
- ❑ Execution of multiple threads on a single CPU in some order is called ***scheduling***.
- ❑ The Java runtime environment supports a very simple, deterministic scheduling algorithm called ***fixed-priority scheduling***.
- ❑ This algorithm schedules threads on the basis of their priority relative to other ***Runnable*** threads
- ❑ When a thread is created, it inherits its priority from the thread that created it. (example: main thread of execution of the running program, i.e., a process)
- ❑ We also can modify a thread's priority at any time after its creation by using the ***setPriority*** method.

Scheduling of Threads

❑Thread priorities are integers ranging between MIN_PRIORITY and MAX_PRIORITY (constants defined in the Thread class).

❑The higher the integer, the higher the priority.

❑At any given time, when multiple threads are ready to be executed, the runtime system chooses for execution the Runnable thread that has the highest priority.

❑Only when that thread stops, yields, or becomes Not Runnable will a lower-priority thread start executing.

❑If two threads of the same priority are waiting for the CPU, the scheduler arbitrarily chooses one of them to run. The chosen thread runs until one of the following conditions is true:

✓A higher priority thread becomes runnable.

✓It yields, or its run method exits.

✓On systems that support time-slicing, its time allotment has expired.

❑Then the second thread is given a chance to run, and so on, until the interpreter exits.

❑The Java runtime system's thread scheduling algorithm is also preemptive.

❑If at any time a thread with a higher priority than all other Runnable threads becomes Runnable, the runtime system chooses the new higher-priority thread for execution.

❑The new thread is said to *preempt* the other threads.

❑ http://www.iitk.ac.in/esc101/05Aug/tutorial/essential/threads/priority.html

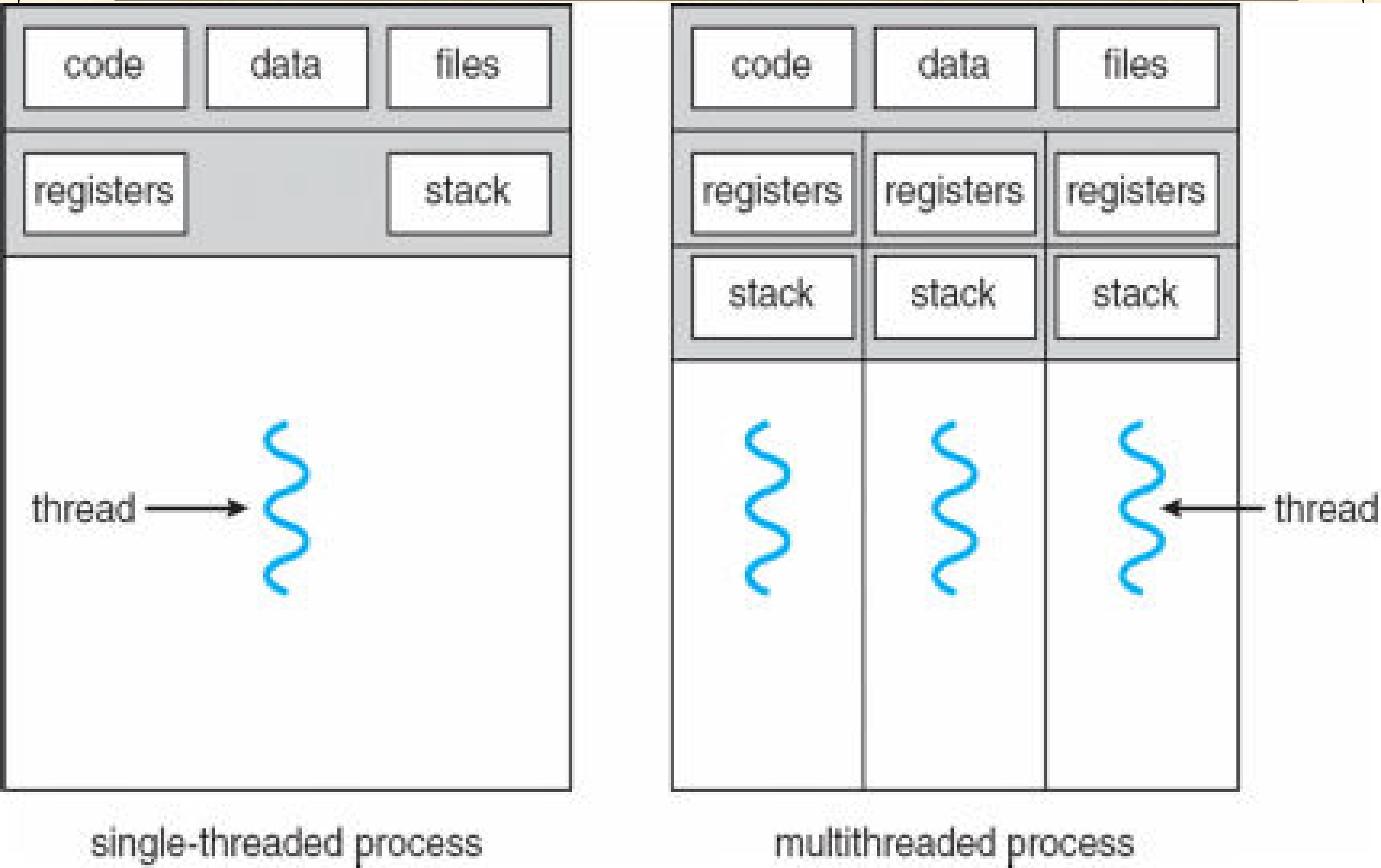
Threads (more slides follows… same idea is explained in a different way…from dhamdhere… leisure time reading….)

- A ***thread*** (or ***lightweight process*** <u>***but not a process by itself***</u>) is a basic unit of CPU utilization; it consists of: a ***thread ID***, **program counter, register set, stack space**
- A thread shares with its peer threads its: **code section**, **data section**, **operating-system resources** collectively know as a *task*.
- A traditional or ***heavyweight*** **process** is equal to a task with a <u>***single thread of control***</u>.
- If the process has ***multiple threads of control***, it can do <u>***more than one task at a time***</u>.

Threads

- ***Thread cannot run on its own***;
- ***It always runs within a process.***
- Thus, a multithreaded process may have multiple execution flows, different ones belonging to different threads.
- These threads share the same private address space of the process, and they share all the resources acquired by the process.
- They run in the same process execution context, and therefore, one thread may influence other threads in the process.

Single and Multithreaded Processes



Let’s look at thread creation….

- pthreads - POSIX threads (http://man7.org/linux/man-pages/man7/pthreads.7.html)
- **Pthreads function return values** Most pthreads functions return 0 on success, and an error number on failure.
- **Thread IDs** Each of the threads in a process has a unique thread identifier (stored in the type *pthread_t*). This identifier is returned to the caller of pthread_create(), and a thread can obtain its own thread identifier using pthread_self(void) [http://man7.org/linux/man-pages/man3/pthread_self.3.html]
- pthread_create - create a new thread [http://man7.org/linux/man-pages/man3/pthread_create.3.html]

#include <pthread.h>

int pthread_create(pthread_t ***thread*****, const pthread_attr_t** ***attr*****, void *(****start_routine*) **(void *), void** **arg***);**

Compile and link with *-lpthread*.

Let’s look at void * pointer needed for thread creation….

- A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be type-casted to any type.

```
#include<stdio.h>

int main() {

   int a = 10;

   void *ptr = &a;

   printf("%d", *(int *)ptr);

   return 0;

}
```

- Output : 10

Let's look at thread creation….

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

void* say_hello(void* data)
{
    char *str;
    str = (char*)data;
    while(1)
    {
        printf("%s\n",str);
        sleep(1);
    }
}

void main()
{
    pthread_t t1,t2;

    pthread_create(&t1,NULL,say_hello,"hello from 1");
    pthread_create(&t2,NULL,say_hello,"hello from 2");
    pthread_join(t1,NULL);
}
```

Explanation of previous program….

Each thread executes a function. In this case the function prototype is predefined. The function needs to have a void* pointer as argument and must return a void* pointer (void* can be interpreted as a pointer to anything).

The function converts the **data** pointer to a string pointer **str** and runs a while loop to print it,the **sleep** function pauses the thread for 1 second as specified. Needless to say that the function never returns.

```
pthread_t t1,t2;
```

pthread_t is a data type that holds information about threads. Each thread requires one **pthread_t** variable

```
pthread_create(&t1,NULL,say_hello,"hello from 1");
```

thread is a **pthread_t** variable in our case **t1 and t2**.

attr

is a variable of type **pthread_attr_t**, if specified it holds details about the thread, like scheduling policy, stack size etc. Since we have specified **NULL** the thread runs with default parameters.

start_routine

is the function the thread executes. In our case it is the ever so friendly **say_hello** function.

arg

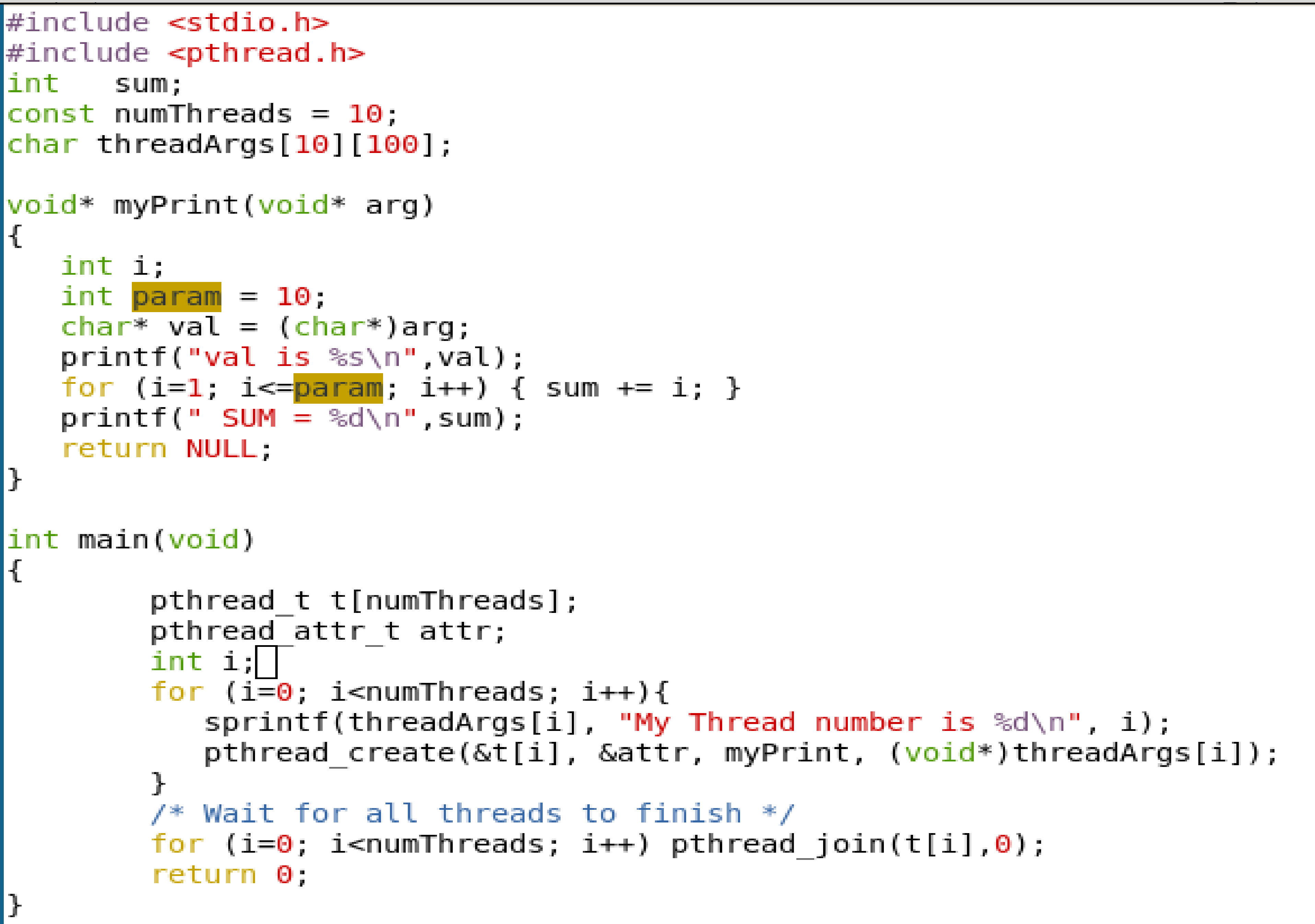
is a void pointer which points to any data type. This same pointer is passes to the **start_routine** function when it is called. In our case the strings **hello from 1/hello from 2** are passed to the **say_hello** function in the **data** variable.

```
pthread_join(t1,NULL);
```

pthread_joinwaits for the thread specifies in the first argument to exit, which in our case doesn't happen, so the program runs forever, unless you interrupt it with Ctrl-C etc.

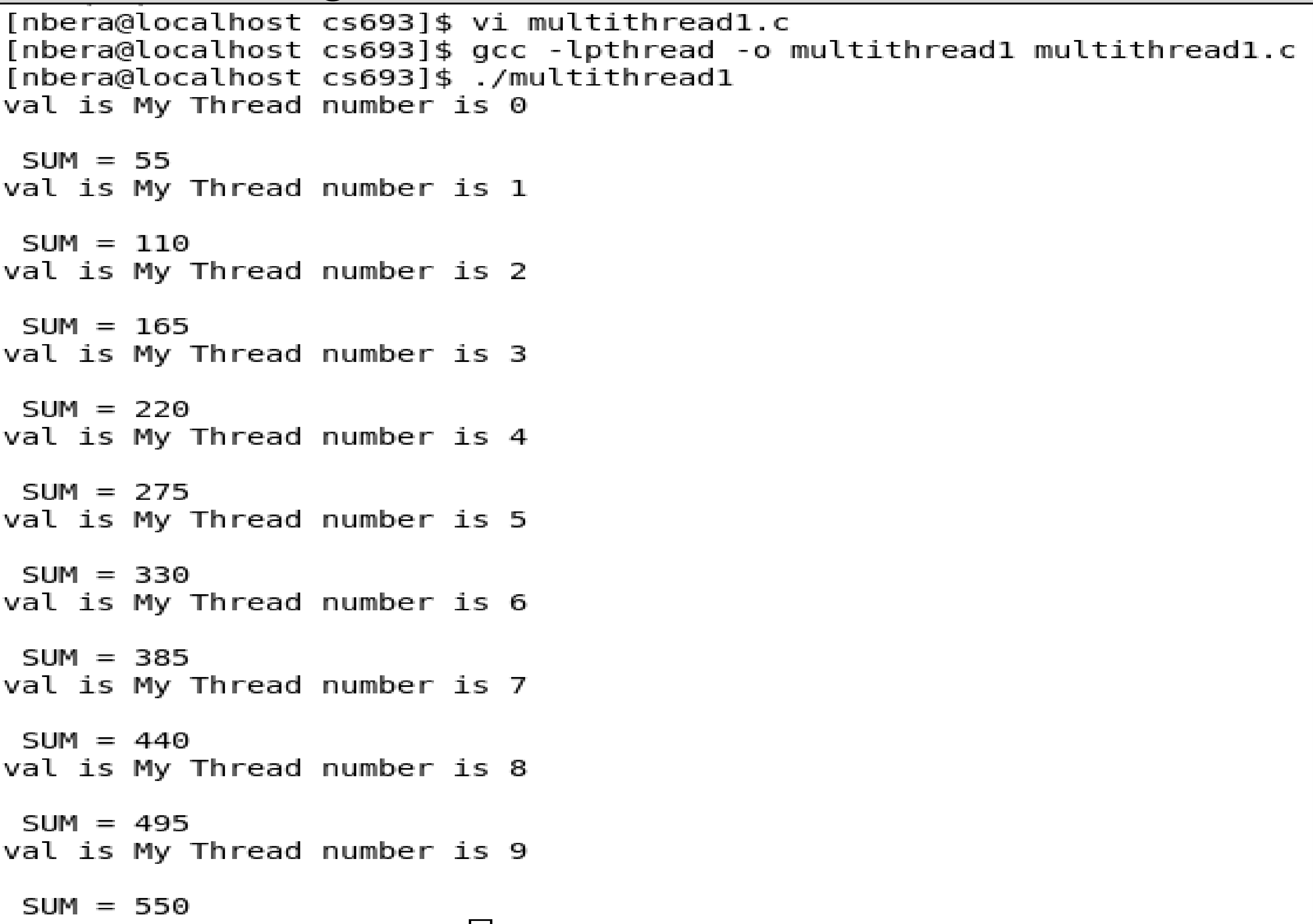
The second argument is a pointer to the variable specified by **pthread_exit** in case the thread calls the function to exit. Since it is NULL, it is ignored.

Multithreaded Processes



```c
#include <stdio.h>
#include <pthread.h>
int   sum;
const numThreads = 10;
char threadArgs[10][100];

void* myPrint(void* arg)
{
   int i;
   int param = 10;
   char* val = (char*)arg;
   printf("val is %s\n",val);
   for (i=1; i<=param; i++) { sum += i; }
   printf(" SUM = %d\n",sum);
   return NULL;
}

int main(void)
{
        pthread_t t[numThreads];
        pthread_attr_t attr;
        int i;
        for (i=0; i<numThreads; i++){
           sprintf(threadArgs[i], "My Thread number is %d\n", i);
           pthread_create(&t[i], &attr, myPrint, (void*)threadArgs[i]);
        }
        /* Wait for all threads to finish */
        for (i=0; i<numThreads; i++) pthread_join(t[i],0);
        return 0;
}
```

Running Multithreaded Processes - Result



```
[nbera@localhost cs693]$ vi multithread1.c
[nbera@localhost cs693]$ gcc -lpthread -o multithread1 multithread1.c
[nbera@localhost cs693]$ ./multithread1
val is My Thread number is 0

 SUM = 55
val is My Thread number is 1

 SUM = 110
val is My Thread number is 2

 SUM = 165
val is My Thread number is 3

 SUM = 220
val is My Thread number is 4

 SUM = 275
val is My Thread number is 5

 SUM = 330
val is My Thread number is 6

 SUM = 385
val is My Thread number is 7

 SUM = 440
val is My Thread number is 8

 SUM = 495
val is My Thread number is 9

 SUM = 550
```

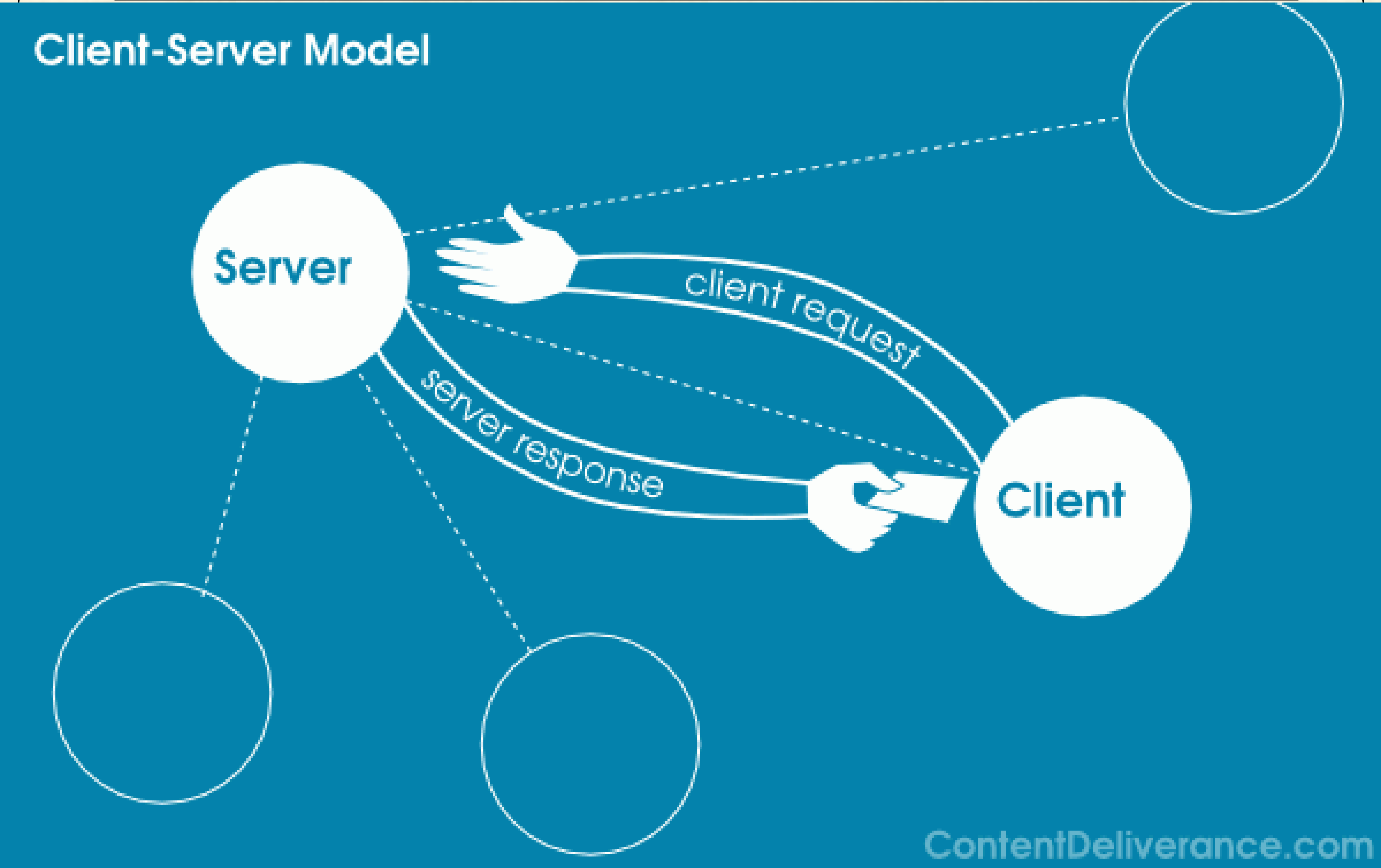
Motivation of thread creation

- Assertion: Many software packages that run on modern desktop PCs are ***multithreaded***.
- An application is implemented as a **separate process** with ***several threads of control***.
- A word processor may have 1) **a thread** for displaying graphics; 2) **another thread** for reading keystrokes from the user; 3) **third one** for performing spelling and grammar checking in the background.
- In certain situation a single application may be required to perform several similar tasks.

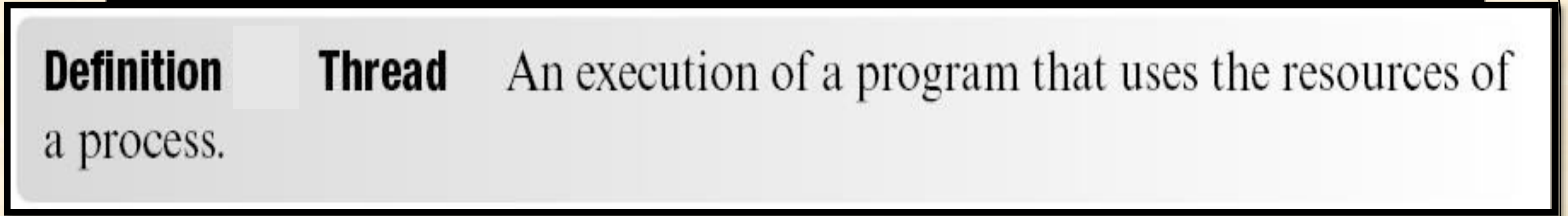
Motivation of thread creation

- Example: A web server accepts client requests for web pages, sound, and so forth.
- A busy web server may have several (e.g., hundreds) clients concurrently accessing it.
- If the word server ran as a traditional ***single-treaded*** process, only one client request may be served at a time. The amount of time that a client might have to wait for its request to be serviced could be enormous.
- Solution: Server creating multiple threads.

Application of Threads



Threads…


Definition **Thread** An execution of a program that uses the resources of a process.

- A thread is an alternative model of program execution
- A process creates a thread through a system call
- Thread operates within process context
- Use of threads effectively splits the process state into two parts
  - Resource state remains with process
  - CPU state is associated with thread
- ***Switching between threads incurs less overhead than switching between processes***

Threads…

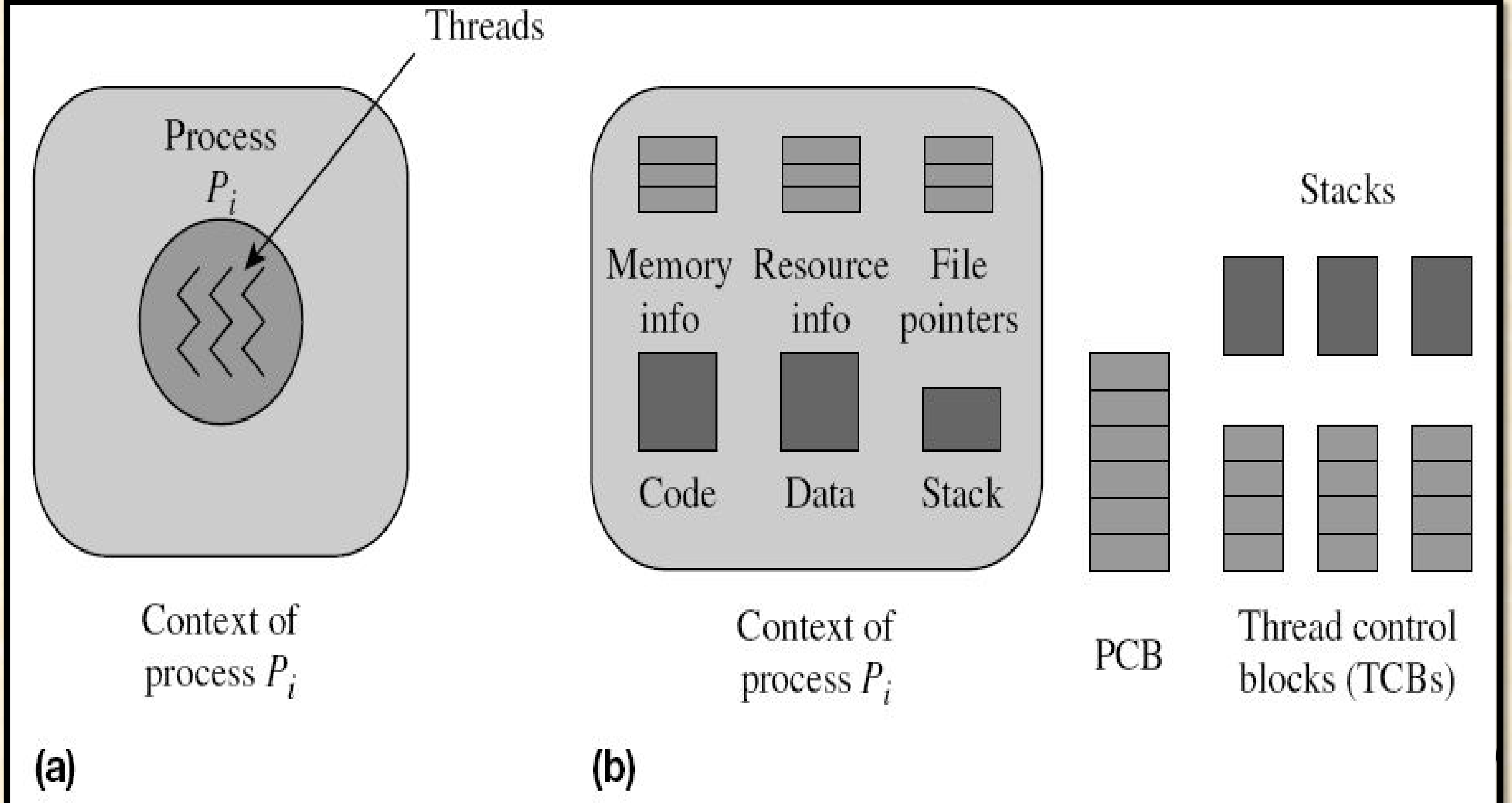


Figure Threads in process P_i: (a) concept; (b) implementation.

Table Advantages of Threads over Processes

Advantage	Explanation
Lower overhead of creation and switching	Thread state consists only of the state of a computation. Resource allocation state and communication state are not a part of the thread state, so creation of threads and switching between them incurs a lower overhead.
More efficient communication	Threads of a process can communicate with one another through shared data, thus avoiding the overhead of system calls for communication.
Simplification of design	Use of threads can simplify design and coding of applications that service requests concurrently.

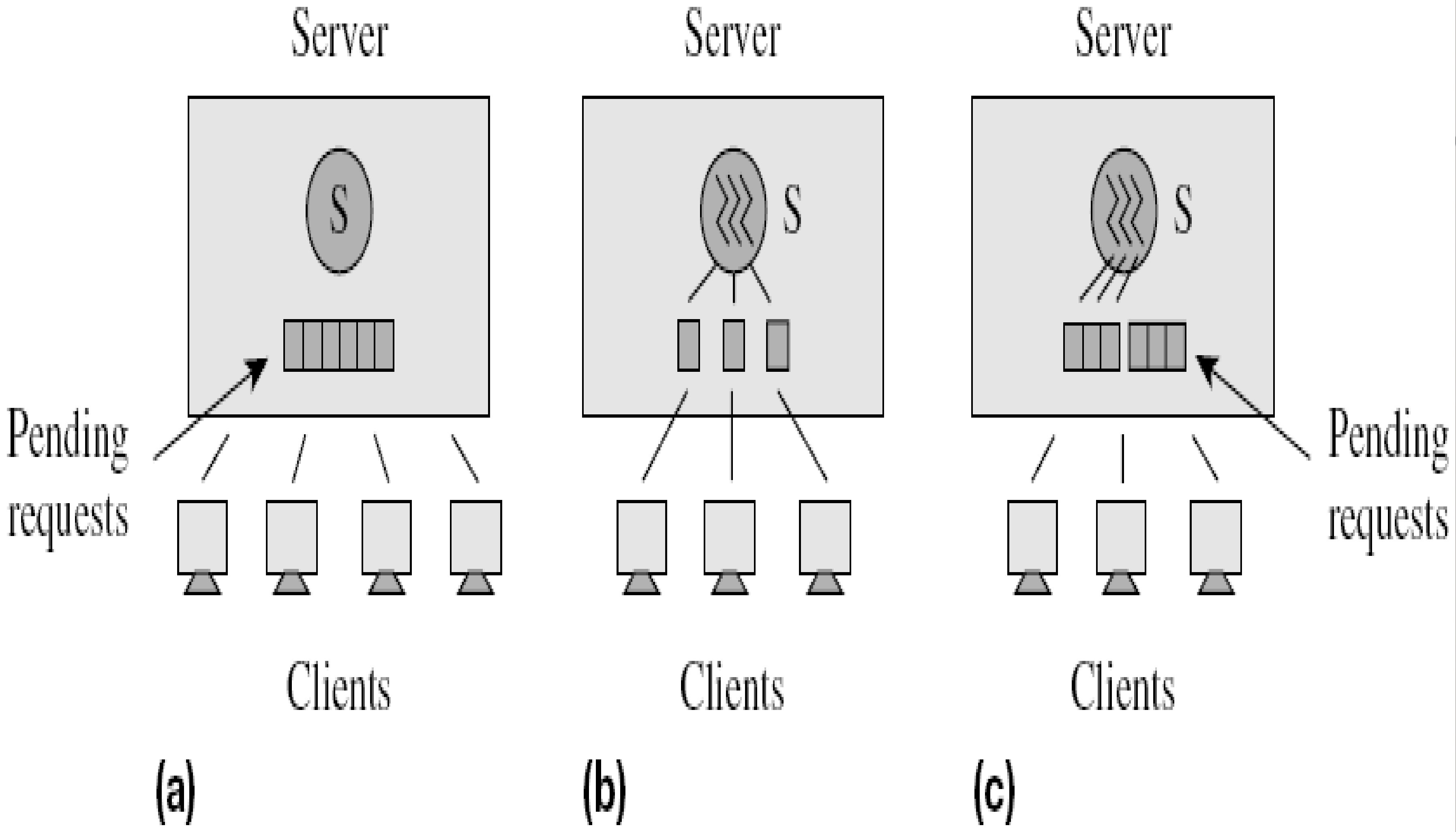


Figure Use of threads in structuring a server: (a) server using sequential code; (b) multithreaded server; (c) server using a thread pool.

Assignments on Process and Scheduling

A process executes the code:

```
fork();
fork();
fork();
```

The total number of child process created is: a) 3 b) 4 c) 7 d) 8

Answer: (C)

Exp:- If fork is called n times, there will be total 2^n running processes including the parent process. So, there will be 2^n-1 child processes.

Answer in detail….

```
#include <stdio.h>
#include <sys/types.h>
#define N 3
int main()
{
  int i;
  printf("Hey! I am the PARENT %d\n", getpid());
  for (i = 1; i <= N; i++) fork();
  printf("Hey! from %d\n", getpid());
  return 0;
}
```

```
nilina@nilina-HP-Pro-3330-MT:~/Desktop/cs693$ gcc -o process4
process4.c
nilina@nilina-HP-Pro-3330-MT:~/Desktop/cs693$ ./process4
Hey! I am the PARENT 2344
Hey! from 2344
Hey! from 2346
Hey! from 2345
Hey! from 2350
Hey! from 2347
Hey! from 2349
Hey! from 2348
Hey! from 2351
```

Process Scheduling….

Consider three processes, P1, P2 and P3 shown in the table.

Process	Arrival Time	Time units required
P1	0	5
P2	1	7
P3	3	4

The completion order of the 3 processes under the policies FCFS and RR2 (round robin scheduling) with CPU quantum of 2 time units) are:

A) FCFS: P1, P2, P3; RR2: P1, P2, P3

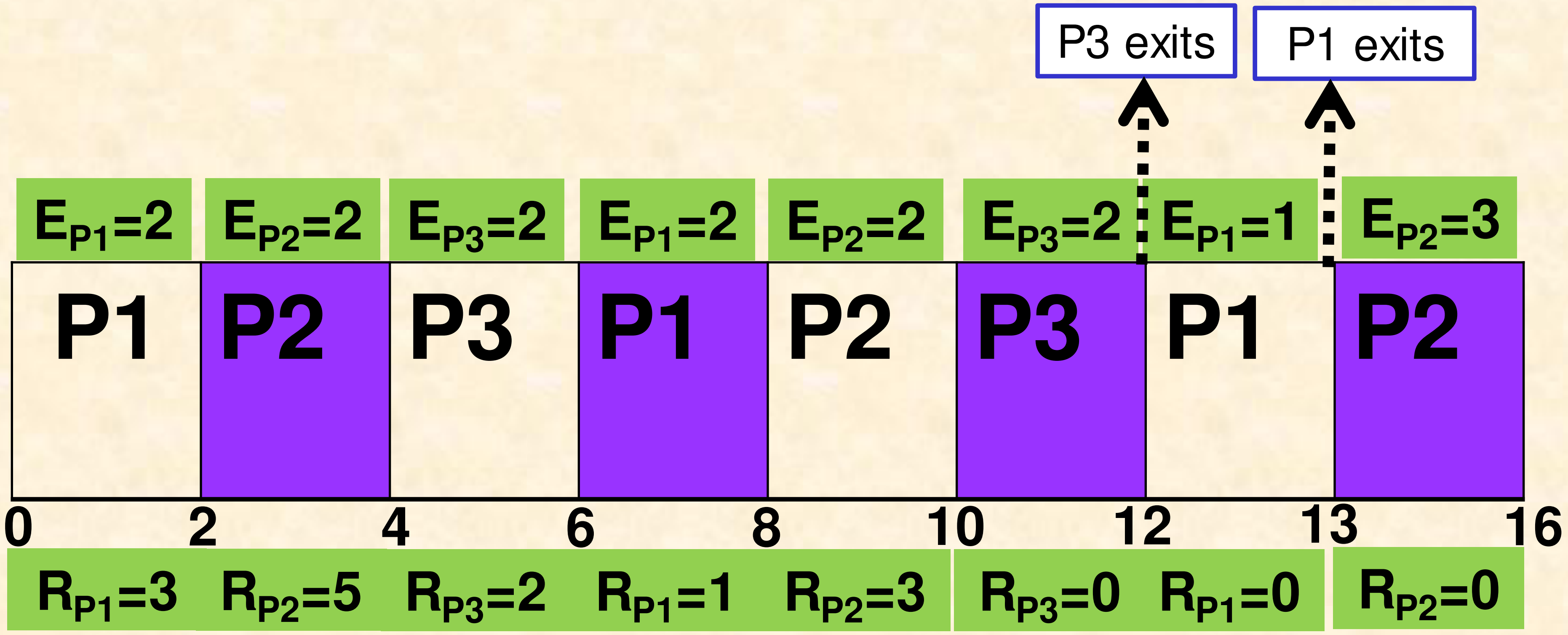
B) FCFS: P1, P3, P2; RR2: P1, P3, P2

C) FCFS: P1, P2, P3; RR2: P3, P1, P2

D) FCFS: P1, P3, P2; RR2: P1, P2, P3

Ans: C

Answer in detail….



UNIX PROCESS MANAGEMENT

- Unix assigns a process number to every process present in the system.
- At the time of booting, a special process called ***process 0*** is created. It is called the ***swapper*** process.
- Thereafter, every process in Unix is created by the '***fork***' system call.
- Process 0 becomes a parent process and forks a child process called ***process 1*** that is also named ***'init'*** process which, in turn, spawns its child processes.
- The child processes spawn their own child processes, and so on. Therefore, the '***init***' process becomes an ancestor of all the processes present in the system.

UNIX PROCESS MANAGEMENT …..

- ❑ An executable file in Unix has many components, including the program text and data.
- ❑ The executable file is loaded into the system by the '***exec***' system call.
- ❑ The operating System opens a process for the loaded file and assigns a stack to it.
- ❑ Thus, the process space consists of three regions → ***text***, ***data*** and ***stack***.
- ❑ The process gets an entry into the process table and the kernel reserves a user area (also called '***u area***') for the process.

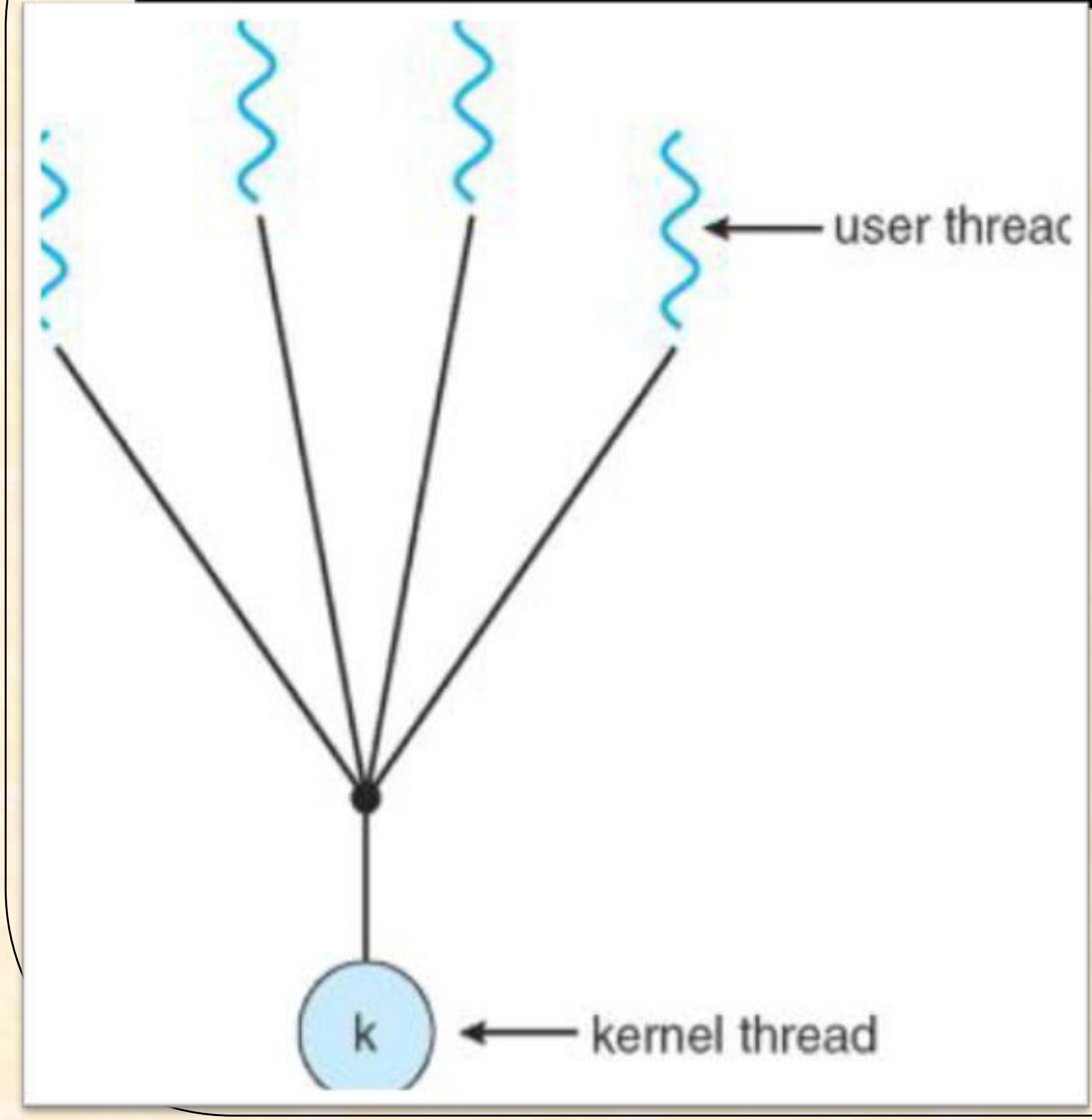
Kernel-Level, User-Level, and Hybrid Threads

- User-Level Threads
  – Threads are managed by thread library.
  – This library provides support for thread creation, scheduling, and management with no support from kernel.
  – Because the kernel is unaware of user-level threads, all thread creation and scheduling are done in user space without the need for kernel intervention.
  – Therefore, user-level threads are generally fast to create and manage.

Kernel-Level, User-Level, and Hybrid Threads

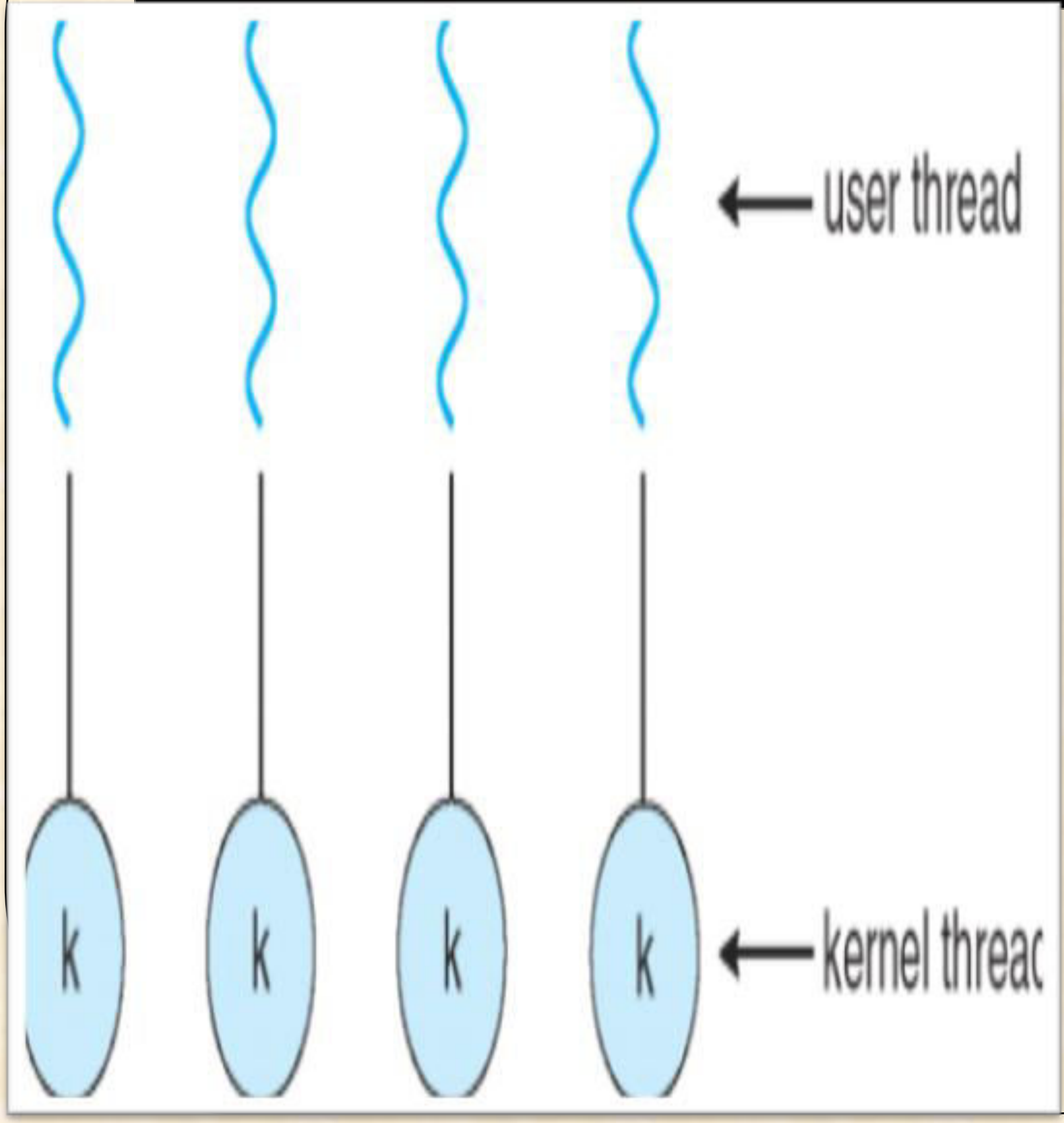
- Kernel-Level Threads
  - Thread-management is done by the operating system
  - The kernel performs thread creation, scheduling and management in kernel space.
  - Kernel threads are generally slower to create and manage than are user threads.
  - Since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execute.
- Hybrid Threads
  - Combination of kernel-level and user-level threads

Many-to-One Model



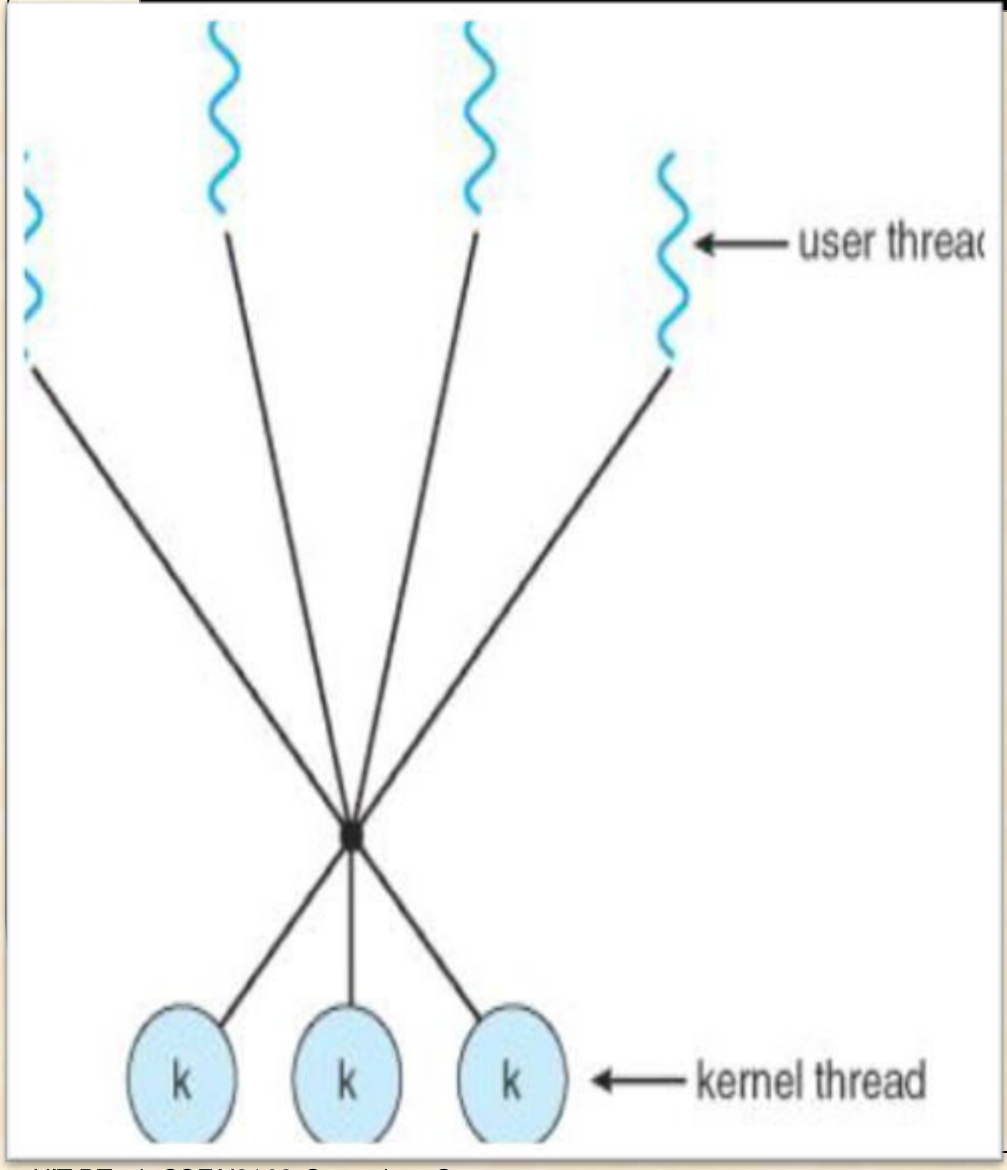
- This model maps many user-level threads to one kernel thread. Thread management is done at user space, so it is efficient, but the entire process will block if a thread makes a blocking system call.
- Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessor.

One-to-one Model



- This model maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.

Many-to-Many Model



- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor).

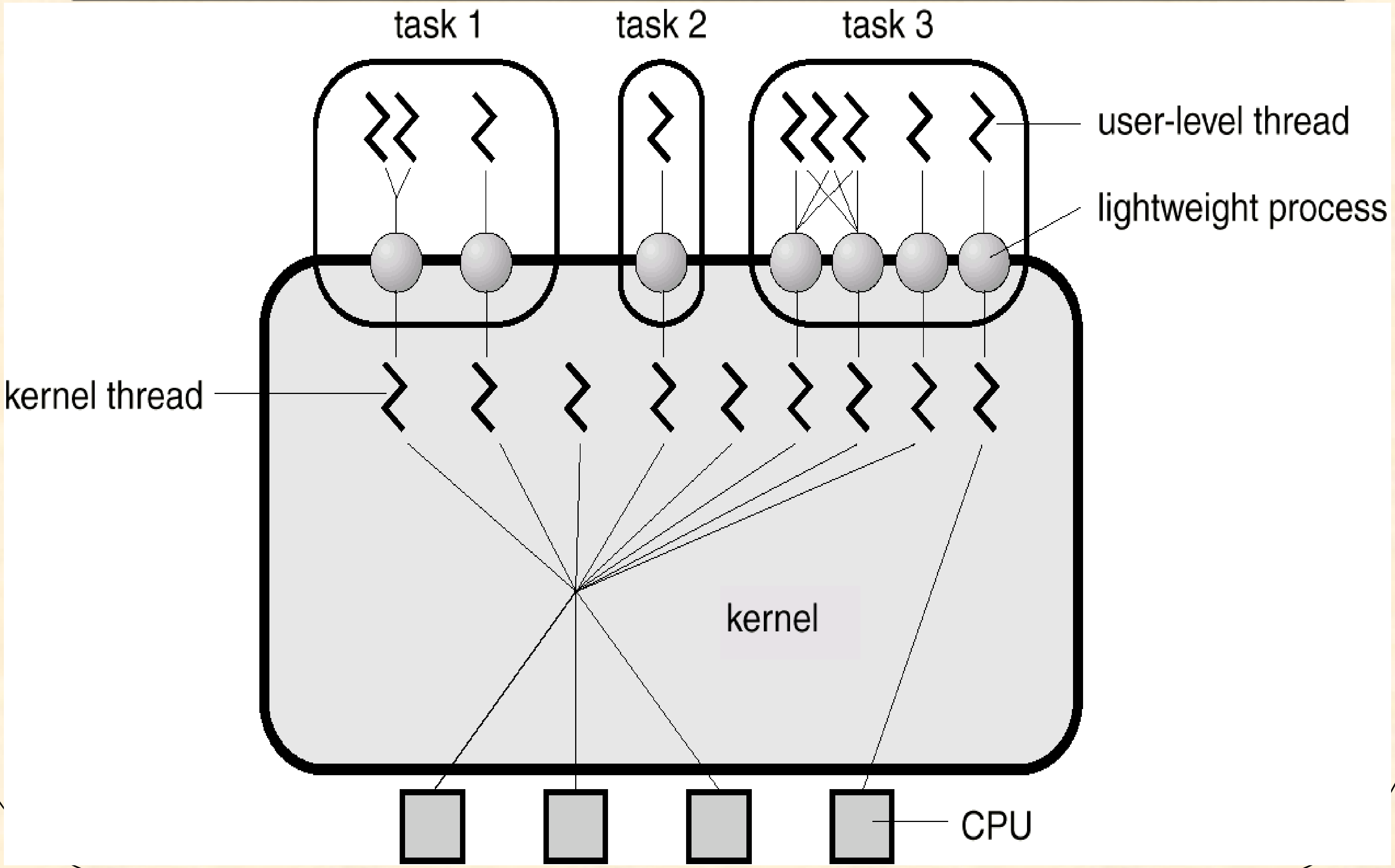
Threads (Cont.)

- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
  - Cooperation of multiple threads in same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.
- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.
- Kernel-supported threads (Mach and OS/2).
- User-level threads; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU).
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

Threads Support in Solaris 2

- Solaris 2 is a version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.
- LWP – intermediate level between user-level threads and kernel-level threads.
- Resource needs of thread types:
  - Kernel thread: small data structure and a stack; thread switching does not require changing memory access information – relatively fast.
  - LWP: PCB with register data, accounting and memory information,; switching between LWPs is relatively slow.
  - User-level thread: only ned stack and program counter; no kernel involvement means fast switching. Kernel only sees the LWPs that support user-level threads.

Solaris 2 Threads



Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically.
  - A link is associated with exactly one pair of communicating processes.
  - Between each pair there exists exactly one link.
  - The link may be unidirectional, but is usually bi-directional.

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports).
  - Each mailbox has a unique id.
  - Processes can communicate only if they share a mailbox.
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link may be unidirectional or bi-directional.
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

Indirect Communication (Continued)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A.
  - P_1, sends; P_2 and P_3 receive.
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Buffering

- Queue of messages attached to the link; implemented in one of three ways.
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous).
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full.
  3. Unbounded capacity – infinite length
     Sender never waits.

Exception Conditions – Error Recovery

- Process terminates
- Lost messages
- Scrambled Messages