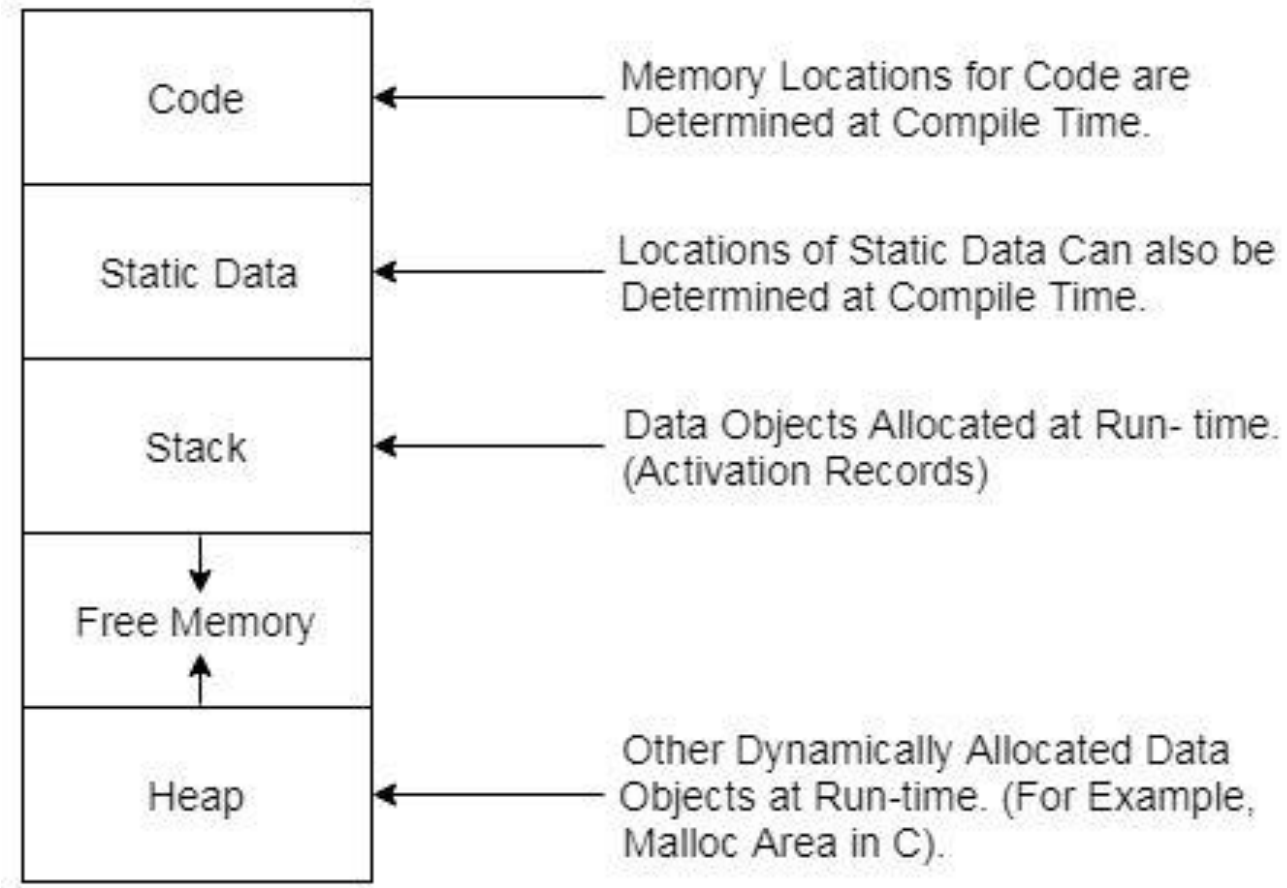
# Fork

- Requirement of Fork command
  - In number of applications specially in those where work is of repetitive nature, like web server i.e. with every client we have to run similar type of code. Have to create a separate process every time for serving a new request.
  - So it must be a better solution that instead to creating a new process every time from scratch we must have a short command using which we can do this logic.
- Idea of fork command
  - Here fork command is a system command using which the entire image of the process can be copied and we create a new process, this idea help us to complete the creation of the new process with speed.
  - After creating a process, we must have a mechanism to identify weather in newly created process which one is child and which is parent.
- Implementation of fork command
  - In general, if fork return 0 then it is child and if fork return 1 then it is parent, and then using a programmer level code we can change the code of child process to behave as new process
- Advantages of using fork commands
  - Now it is relatively easy to create and manage similar types of process of repetitive nature with the help of fork command.
- Disadvantage
  - To create a new process by fork command we have to do system call as, fork is system function
    - Which is slow and time taking
    - Increase the burden over Operating System
  - Different image of the similar type of task have same code part which manes we have the multiple copy of the same data waiting the main memory.




So, in general $2^n$ times a statement will get printed, where n is the number of times fork() system call was executed. $2^n$-1 Child processes and 1 parent process.

The number of child processes created will be $2^n$-1.

**Q** Fork is **(ISRO 2008)**

a) the creation of a new job

**b)** the dispatching of a task

**c)** increasing the priority of a task

**d**) the creation of a new process

**Ans. D**

**Q** The following C program is executed on a Unix/Linux system:

```
#include <unistd.h>
int main ()
{
    int i ;
    for (i=0; i<10; i++)
        if (i%2 == 0) fork ( ) ;
    return 0 ;
}
```

The total number of child processes created is _____(**GATE-2019) (1 Marks)**

**Answer: 31**

**Q** A process executes the code

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

the total number of child processes created is **(GATE - 2012) (1 Marks)**

**a)** 3 **b)** 4 **c)** 7 **d)** 8

**Answer: (C)**

**Q** A process executes the following code **(GATE - 2008) (2 Marks)**

```
for (i=0; i<n; i++)
fork();
```

**a)** n **b)** $(2^n) - 1$ **c)** $2^n$ **d)** $(2^{n+1}) - 1$

**ANSWER B**

**Q** if (fork() == 0)
{

a = a + 5;

printf ("%d, %d /n", a, &a);

}
else

{

a = a –5;

printf("%d, %d /n", a, &a);

}

Let u, v be the values printed by the parent process, and x, y be the values printed by the child process. Which one of the following is TRUE? **(GATE-2005) (2 Marks)**

**a)** u = x + 10 and v = y **b)** u = x + 10 and v != y
**c)** u + 10 = x and v = y **d)** u + 10 = x and v != y

**Q** a process ecexutes the following segment of code:
fork(i=1;i<=n;i++)
fork();
the number of new processes created is **(GATE-2004) (1 Marks)**

**a)** n **b)** (n(n+1))/2 **c)** $2^n$-1 **d)** $3^n$-1
**Answer: (C)**

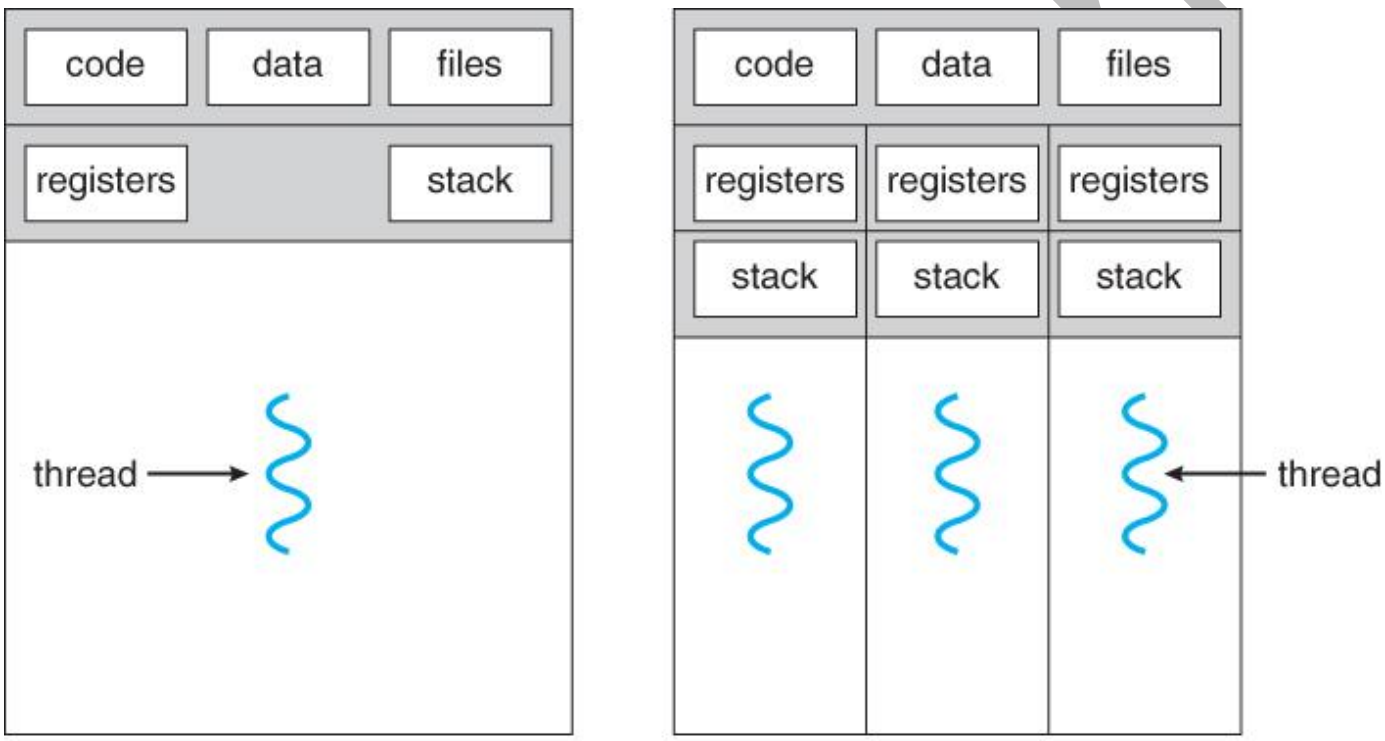
**Q** What is the output of the following program?

```
main( )
{
    int a = 10;
    if ((fork ( ) == 0))
        a++;
    printf (“%dn”, a );
}
```

**a)** 10 and 11 **b)** 10 **c)** 11 **d)** 11 and 11
**Ans. A**

# Threads

- A ***thread*** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, ( and a thread ID. )
- Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.




single-threaded process

multithreaded process

# Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input ( keystrokes ), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request. ( The latter is how this sort of thing was done before the concept of threads was developed. A daemon would listen at a port, fork off a child for every incoming request to be processed, and then go back to listening to the port. )

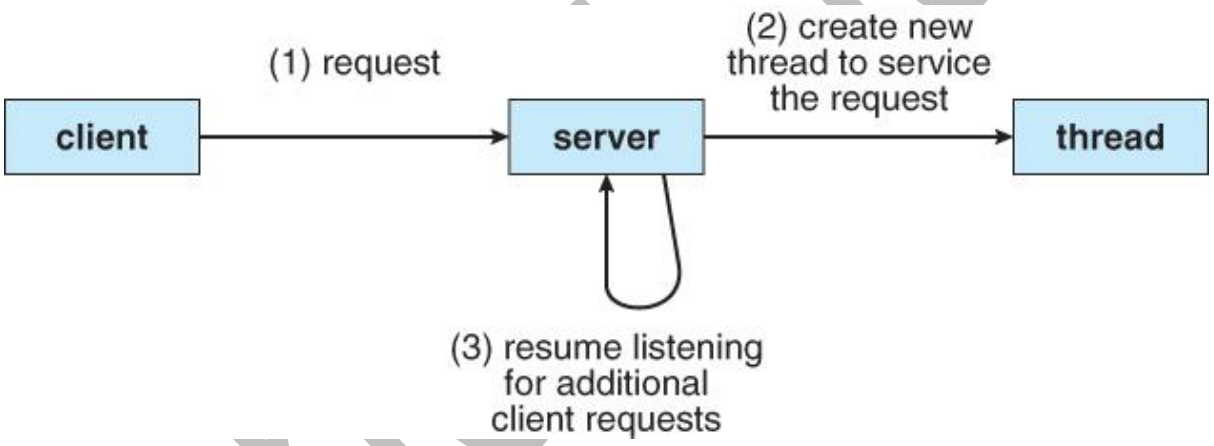



**Figure 4.2 - Multithreaded server architecture**

There are four major categories of benefits to multi-threading:

1. Responsiveness - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
2. Resource sharing - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
3. Economy - Creating and managing threads ( and context switches between them ) is much faster than performing the same tasks for processes.
4. Scalability, i.e. Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. ( Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold. )

**Q** Which of the following is/are shared by all the threads in a process? **(GATE - 2017) (1 Marks)**

**I.** Program Counter **II.** Stack **III.** Address space **IV.** Registers

**(A)** I and II only **(B)** III only **(C)** IV only **(D)** III and IV only

**Answer: (B)**

**Q** Threads of a process share **(GATE - 2017) (1 Marks)**

**(A)** global variables but not heap **(B)** heap but not global variables

**(C)** neither global variables nor heap **(D)** both heap and global variables

**Answer: (D)**

**Q** A thread is a light weight process. In the above statement, weight refers to **(NET-DEC-2012)**

**(A)** time **(B)** number of resources

**(C)** speed **(D)** All the above

**Answer: (B)**

**Q** A thread is usually defined as a “light weight process” because an operating system (OS) maintains smaller data structures for a thread than for a process. In relation to this, which of the following is TRUE? **(GATE - 2011) (1 Marks)**

**(A)** On per-thread basis, the OS maintains only CPU register state

**(B)** The OS does not maintain a separate stack for each thread

**(C)** On per-thread basis, the OS does not maintain virtual memory state

**(D)** On per-thread basis, the OS maintains only scheduling and accounting information

**Answer: (C)**

# Multithreading Models

- There are two types of threads to be managed in a modern system: User threads and kernel threads.
- User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.
- Kernel threads are supported within the kernel of the OS itself. All modern OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

# Many-To-One Model

- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.
- Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs.
- Green threads for Solaris implement the many-to-one model in the past, but few systems continue to do so today.

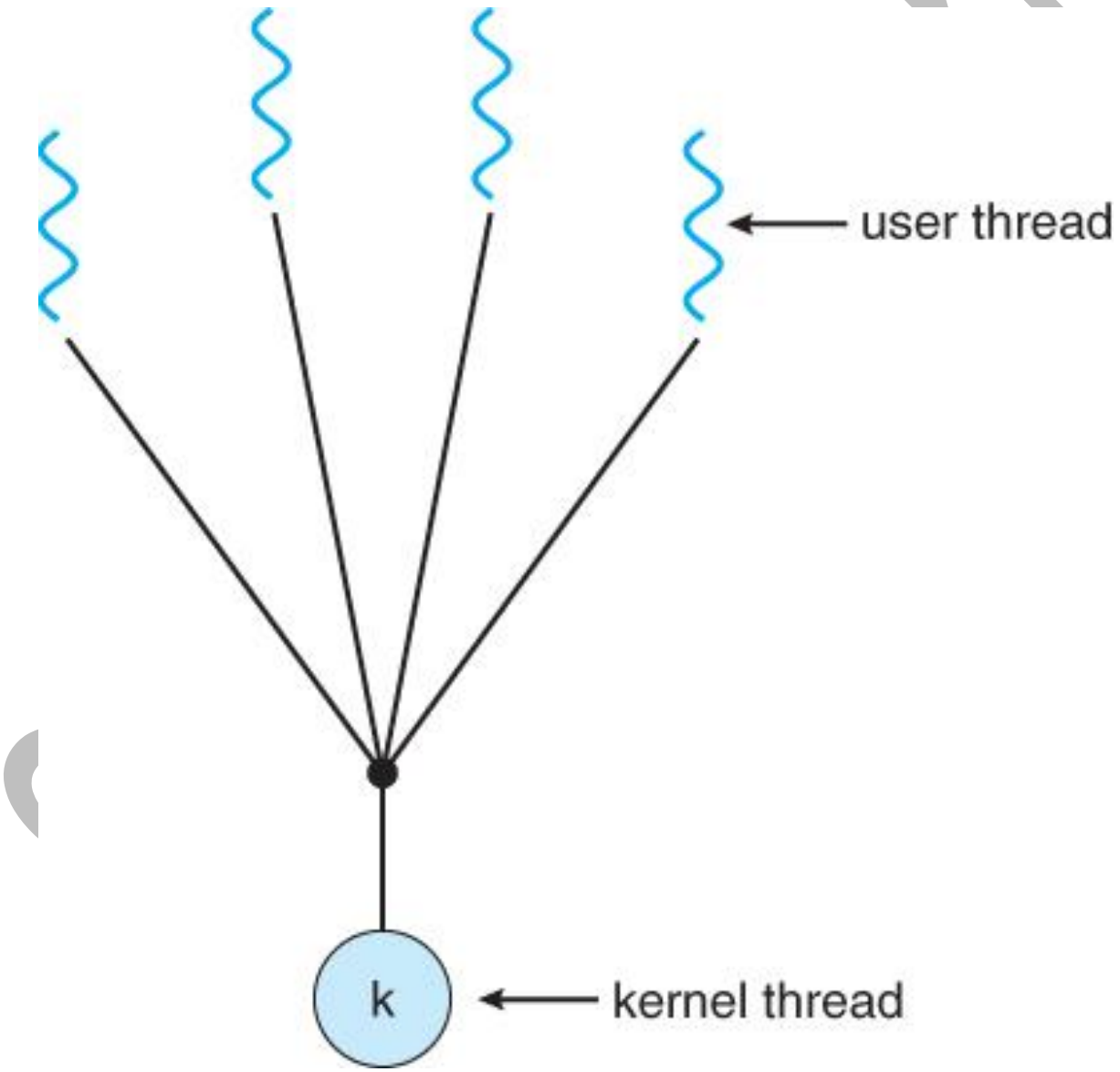



**Figure 4.5 - Many-to-one model**

# One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However, the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.

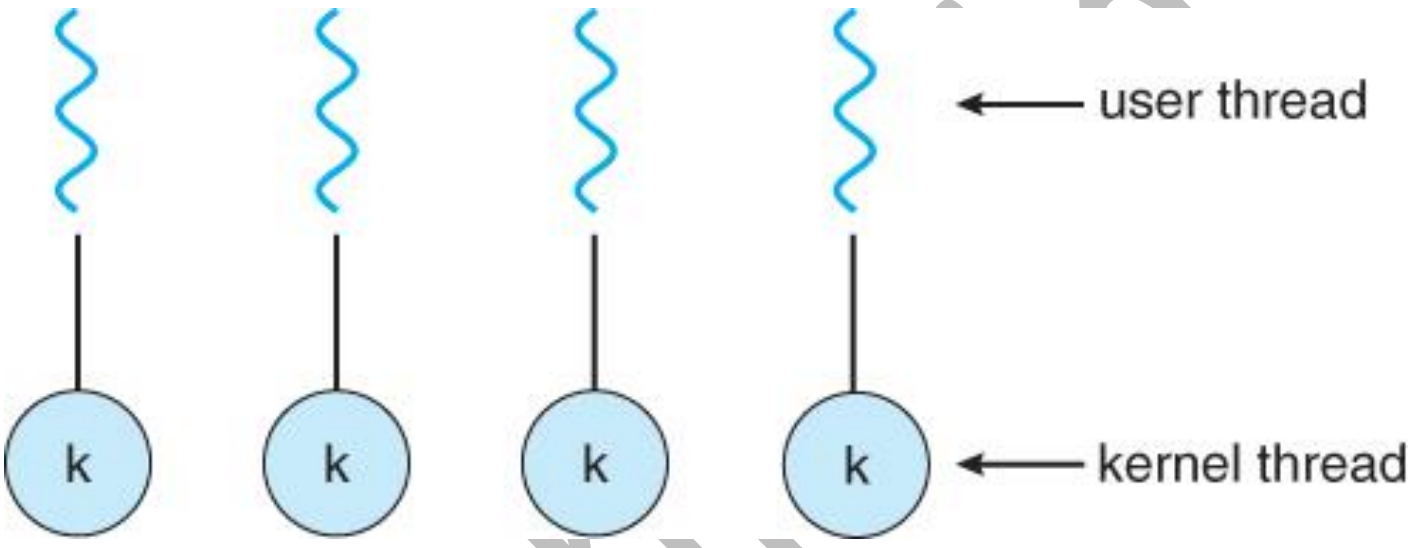



**Figure 4.6 - One-to-one model**

# Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.

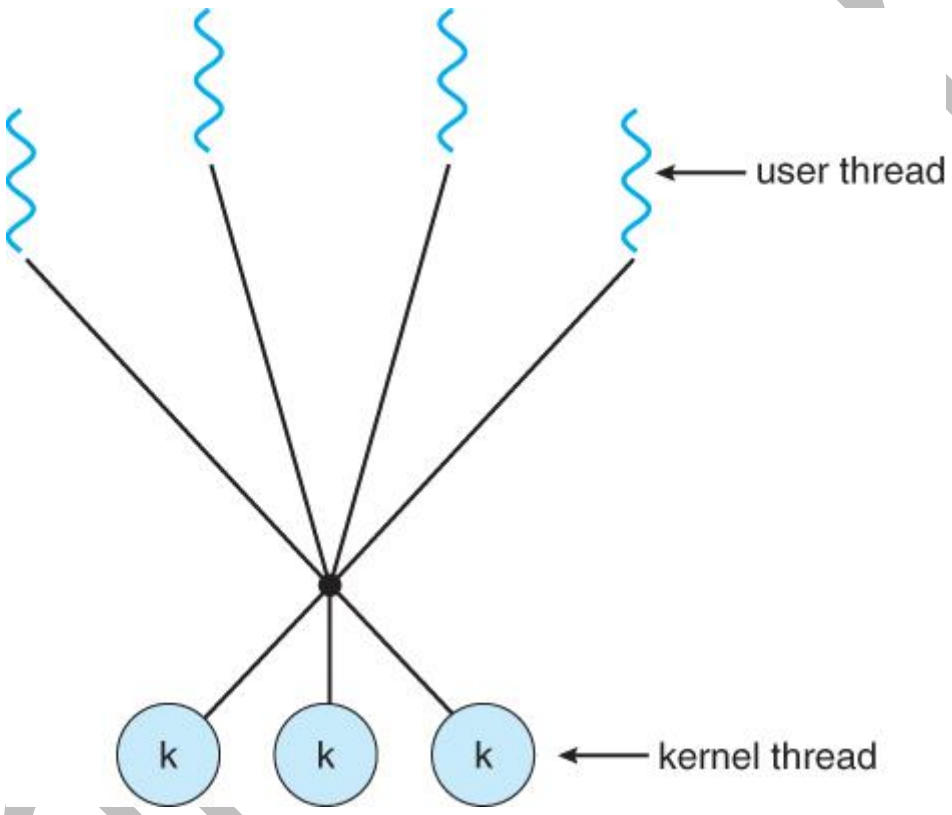



**Figure 4.7 - Many-to-many model**

**Q** User level threads are threads that are visible to the programmer and are unknown to the kernel. The operating system kernel supports and manages kernel level threads. Three different types of models relate user and kernel level threads. Which of the following statements is/are true? **(NET-NOV-2017)**

**(a)** (i) The Many - to - one model maps many user threads to one kernel thread
(ii) The one - to - one model maps one user thread to one kernel thread
(iii) The many - to - many model maps many user threads to smaller or equal kernel threads

**(b)** (i) Many - to - one model maps many kernel threads to one user thread
(ii) One - to - one model maps one kernel thread to one user thread
(iii) Many - to - many model maps many kernel threads to smaller or equal user threads

**A)** (a) is true; (b) is false
**B)** (a) is false; (b) is true
**C)** Both (a) and (b) are true
**D)** Both (a) and (b) are false
**Answer: A**

**Q** One of the disadvantages of user level threads compared to Kernel level threads is **(NET-JAN-2017)**
**a)** If a user level thread of a process executes a system call, all threads in that process are blocked.
**b)** Scheduling is application dependent.
**c)** Thread switching doesn't require kernel mode privileges.
**d)** The library procedures invoked for thread management in user level threads are local procedures.
**Answer: (A)**

**Q** Which one of the following is FALSE? **(GATE - 2014) (1 Marks)**
**(A)** User level threads are not scheduled by the kernel.
**(B)** When a user level thread is blocked, all other threads of its process are blocked.
**(C)** Context switching between user level threads is faster than context switching between kernel level threads.
**(D)** Kernel level threads cannot share the code segment
**Answer: (D)**

**Q** Let the time taken to switch between user mode and kernel mode of execution be T1 while time taken to switch between two user processes be T2. Which of the following is correct? **(NET-JUNE-2013)**
**a)** T1 < T2
**b)** T1 > T2
**c)** T1 = T2
**d)** Nothing can be said about the relation between T1 and T2.

Ans: a

**Q** Let the time taken to switch between user and kernel modes of execution be $t_1$ while the time taken to switch between two processes be $t_2$. Which of the following is TRUE? (**GATE-2011) (1 Marks)**

**(A)** t1 > t2 **(B)** t1 = t2

**(C)** t1 < t2 **(D)** nothing can be said about the relation between t1 and t2

**Answer: (C)**

**Q** Consider the following statements about user level threads and kernel level threads. Which one of the following statements is FALSE? **(GATE - 2007) (1 Marks)**

**A)** Context switch time is longer for kernel level threads than for user level threads.

**B)** User level threads do not need any hardware support.

**C)** Related kernel level threads can be scheduled on different processors in a multi-processor system

**D)** Blocking one kernel level thread blocks all related threads

**ANSWER D**

**Q** Consider the following statements with respect to user-level threads and kernel supported threads

i. context switch is faster with kernel-supported threads

ii. for user-level threads, a system call can block the entire process

iii. Kernel supported threads can be scheduled independently

iv. User level threads are transparent to the kernel

Which of the above statements are true? **(GATE-2004) (1 Marks)**

**(A)** (ii), (iii) and (iv) only **(B)** (ii) and (iii) only

**(C)** (i) and (iii) only **(D)** (i) and (ii) only

**Answer: (A)**