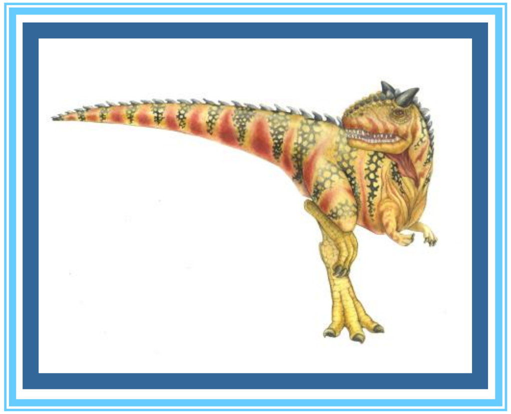
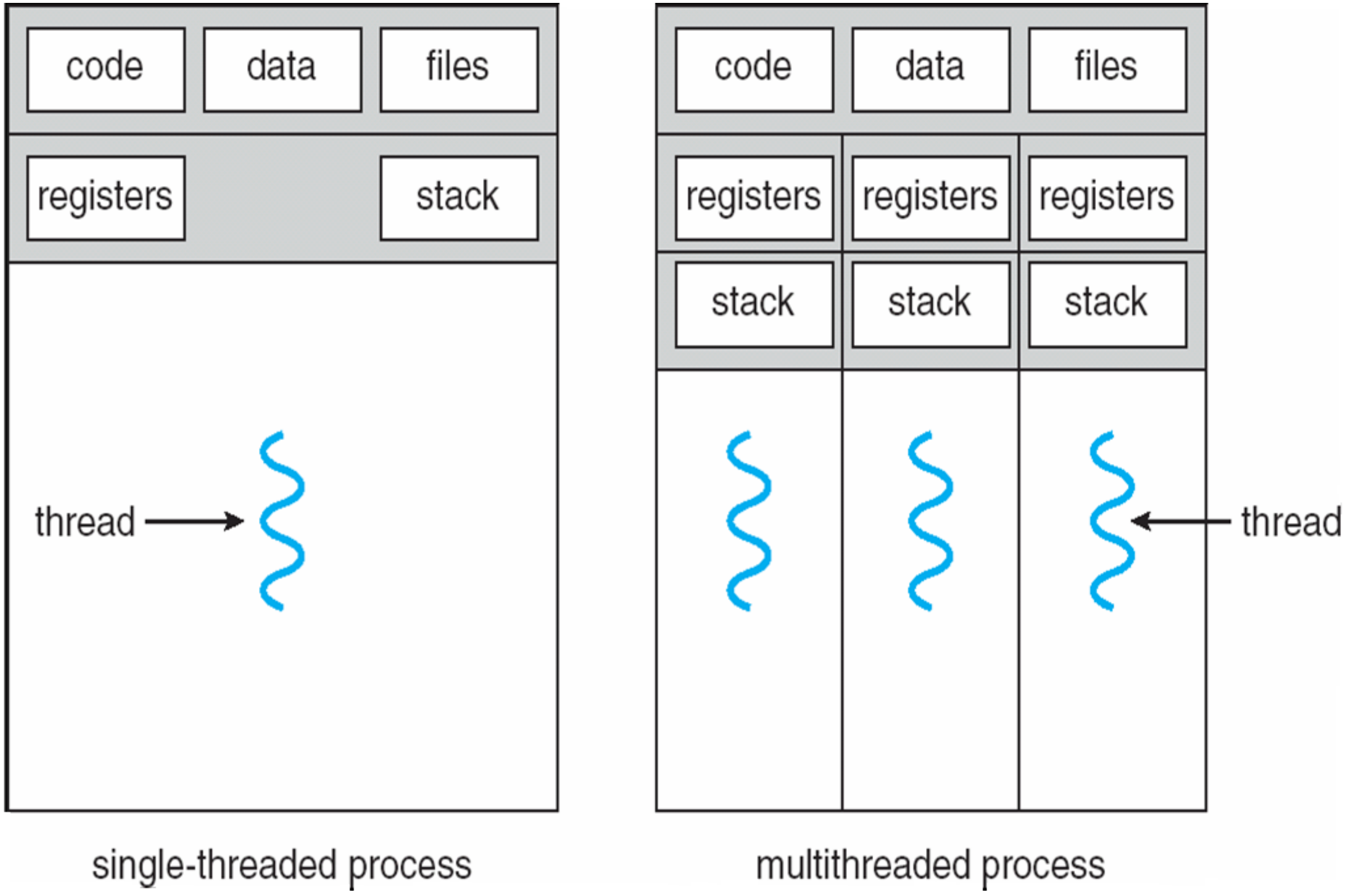
Chapter 4: Threads



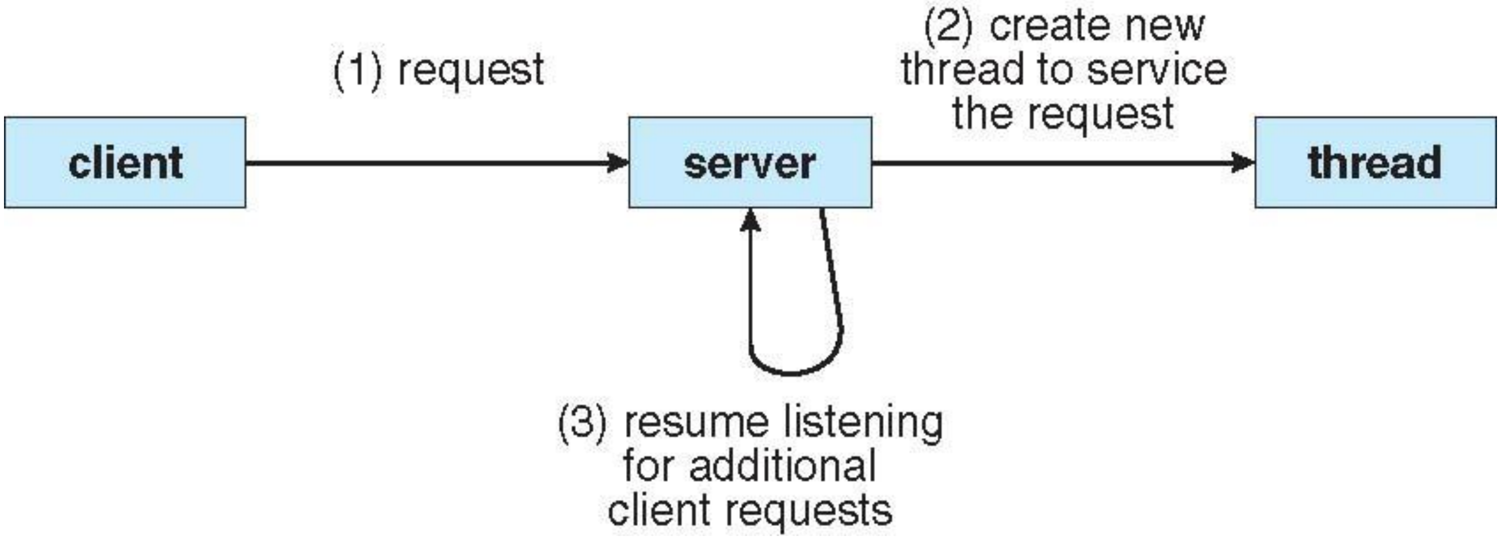
Thread

- Basic unit of CPU utilization
- Thread consist of thread id, program counter, register set, stack
- Thread shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A process can have a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

Single and Multithreaded Processes



Multithreaded Server Architecture


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

Benefits

Responsiveness:

- Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

Resource Sharing:

- threads share the memory and the resources of the process to which they belong by default.

- Processes may only share resources through techniques such as shared memory or message passing. Such techniques must be explicitly arranged by the programmer.

Benefits (Contd…)

Economy

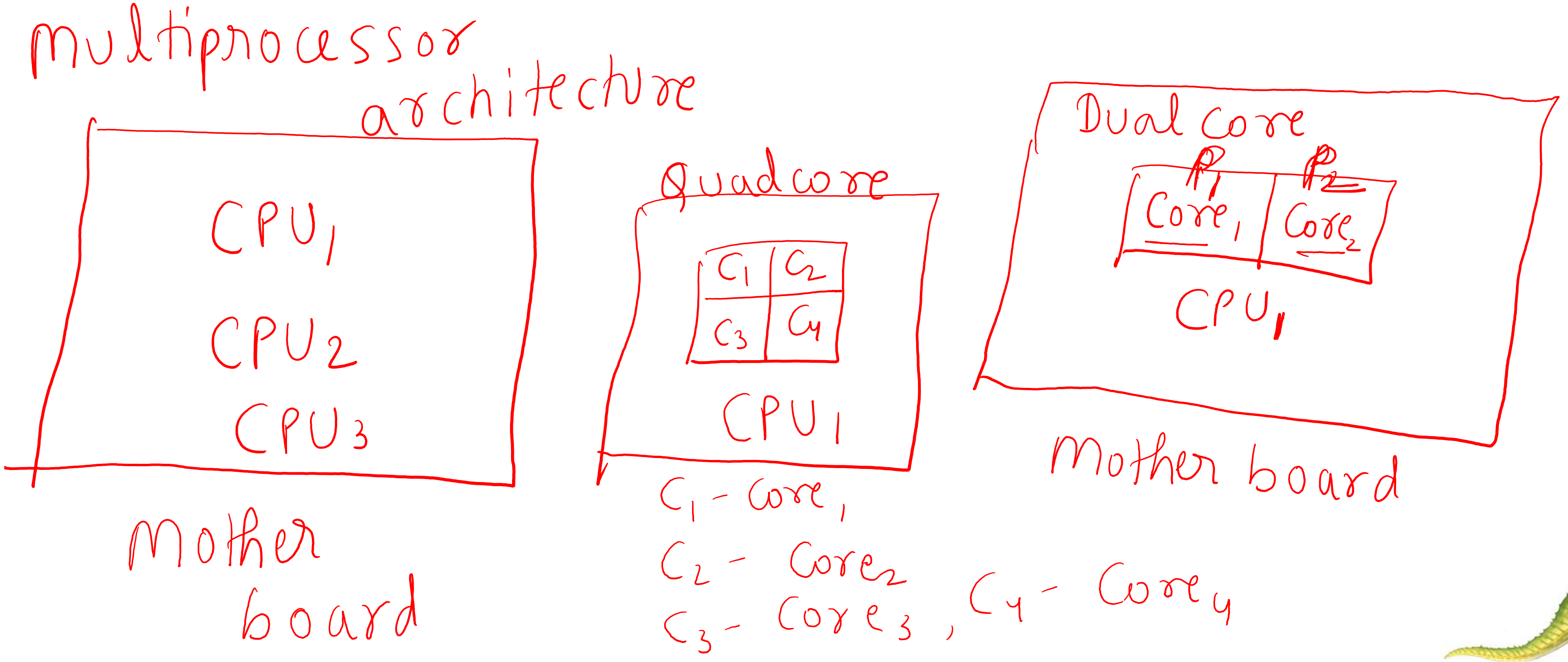
- Because threads share the resources of the process to which they belong (thread is a light-weight process)
- it is more economical to create and context-switch threads.
- Allocating memory and resources for process creation is costly.
- much more time consuming to create and manage processes

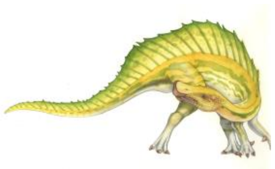
Scalability

- The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. (increases parallelism)

Multicore Programming

- place multiple computing cores on a single chip, where each core appears as a separate processor to the operating system
- Multithreaded programming provides a mechanism for more efficient use of multiple cores and improved concurrency





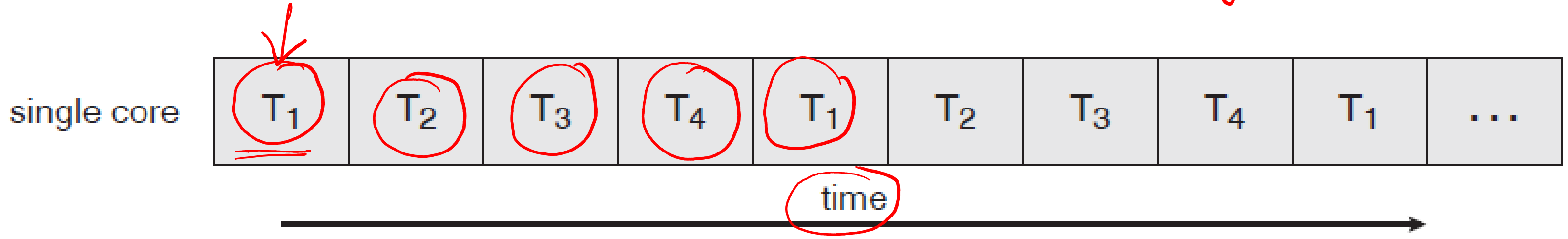
On a system with a single computing core, Single Processor

Concurrency merely means that the execution of the threads will be interleaved over time

as the processing core is capable of executing only one thread at a time.

Single

All 4 threads running



single core

T1	T2	T3	T4	T1	T2	T3	T4	T1	. . .

time

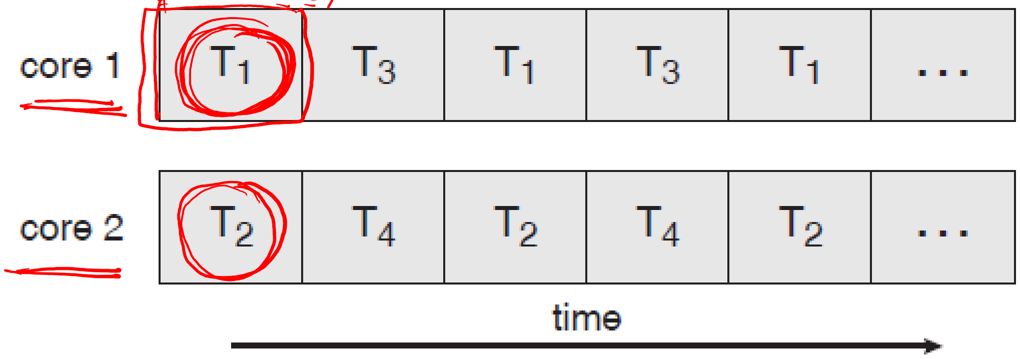
Figure 4.3 Concurrent execution on a single-core system.

On a system with multiple cores, however, concurrency means that the threads can run in parallel, as the system can assign a separate thread to each core

Thread Scheduling

OS designer

0ms 1ms



core 1 T_1 T_3 T_1 T_3 T_1 . . .

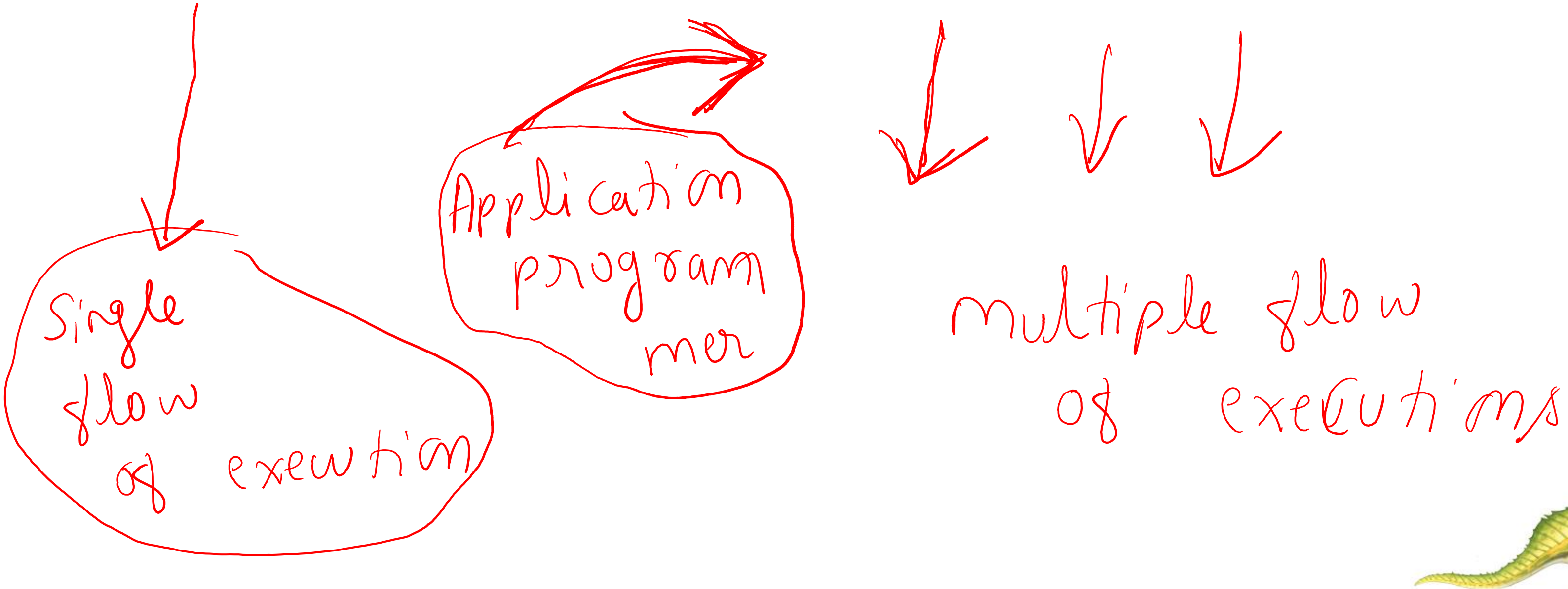
core 2 T_2 T_4 T_2 T_4 T_2 . . .

time

Figure 4.4 Parallel execution on a multicore system.

Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution.

For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded to take advantage of multicore systems.



Multicore Programming

five areas present challenges in programming for multicore systems are:

- Dividing activities: concurrent tasks run in parallel job ⊆ Task
- Balance: equal work of equal value Any ~~one~~ thread must not get overloaded. Equal work for each thread
- Data splitting
- T_1, T_2 Data dependency deposit | withdraw
- Testing and debugging: many different execution paths makes testing and debugging difficult

Easily

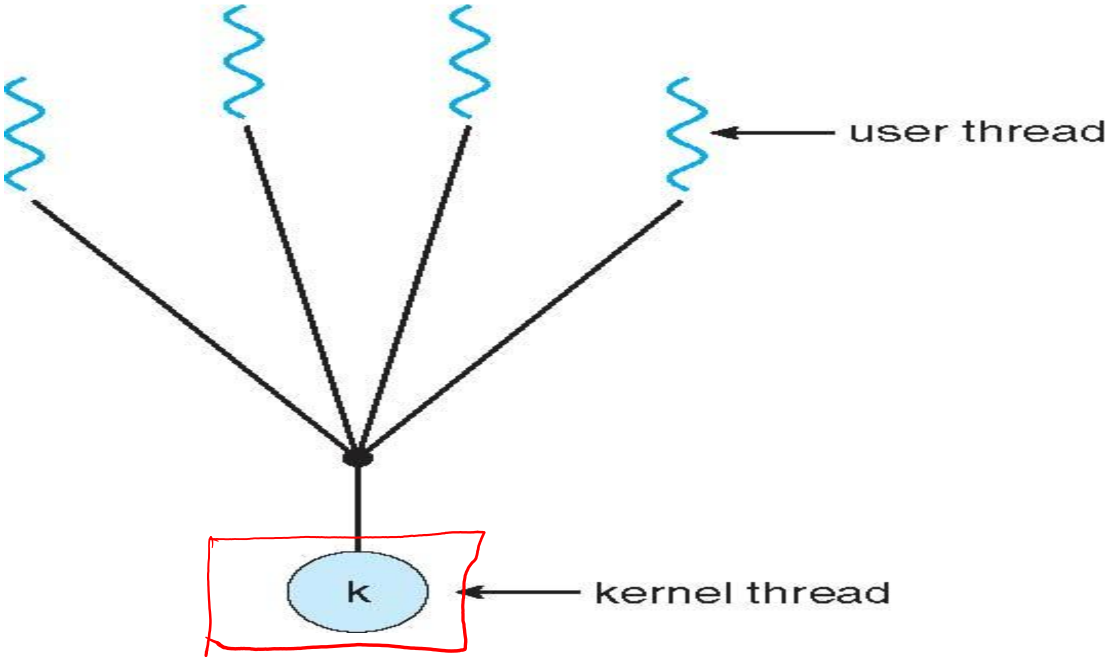
Multithreading Models

- threads may be provided
  - either at the user level, for user threads,
  - or by the kernel, for kernel threads.
- User threads are managed without kernel support, whereas
- kernel threads are supported and managed directly by the operating system.
- Ultimately, a relationship must exist between user threads and kernel threads.
- Three common ways of establishing such a relationship.

Multithreading Models

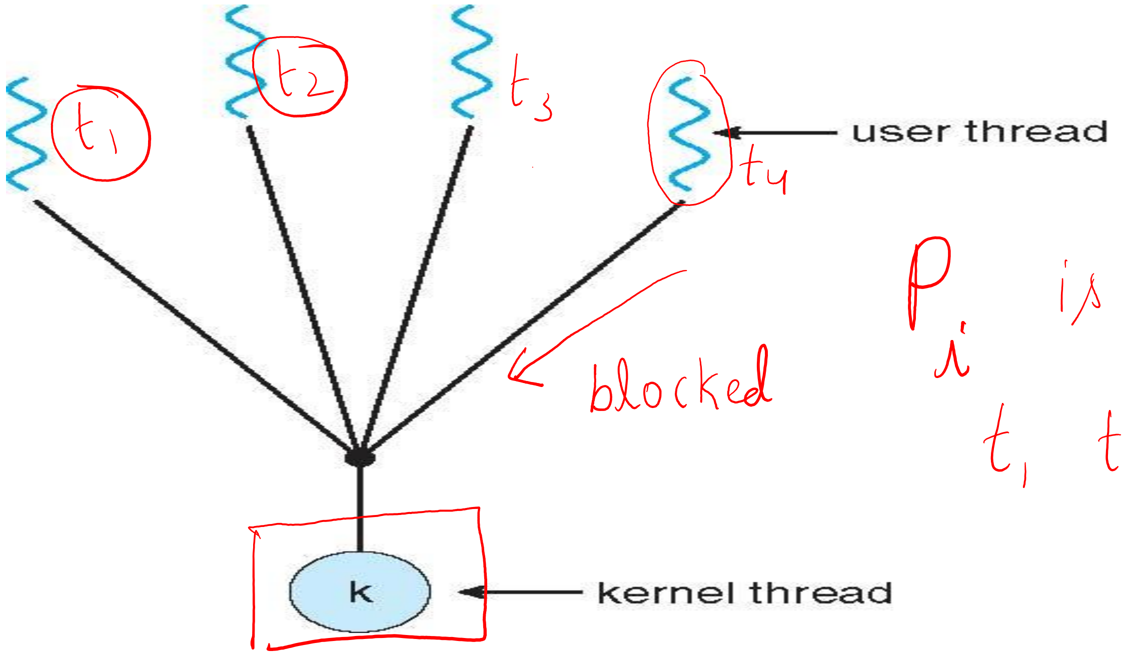
- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One Model



The many-to-one model maps many user-level threads to one kernel thread.

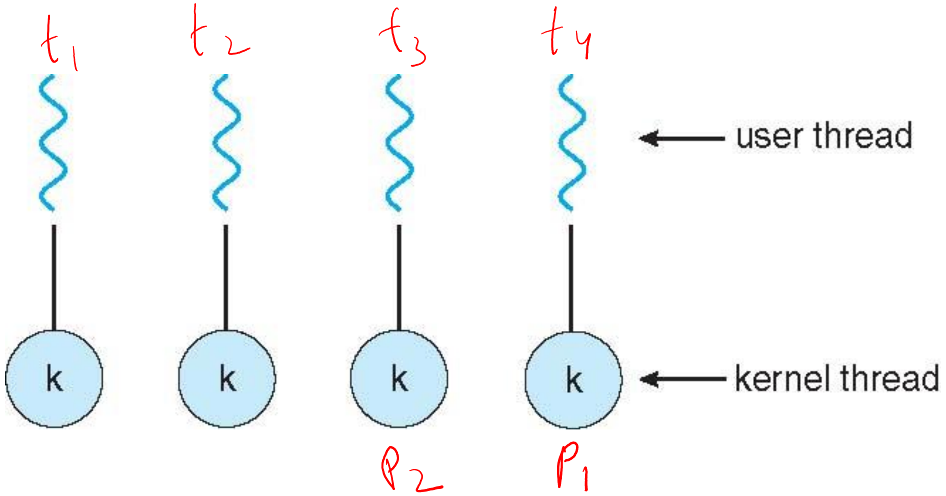
Many-to-One Model



P_i is having t_1 t_2 t_3 t_4

- 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 multiprocessors

One-to-one Model

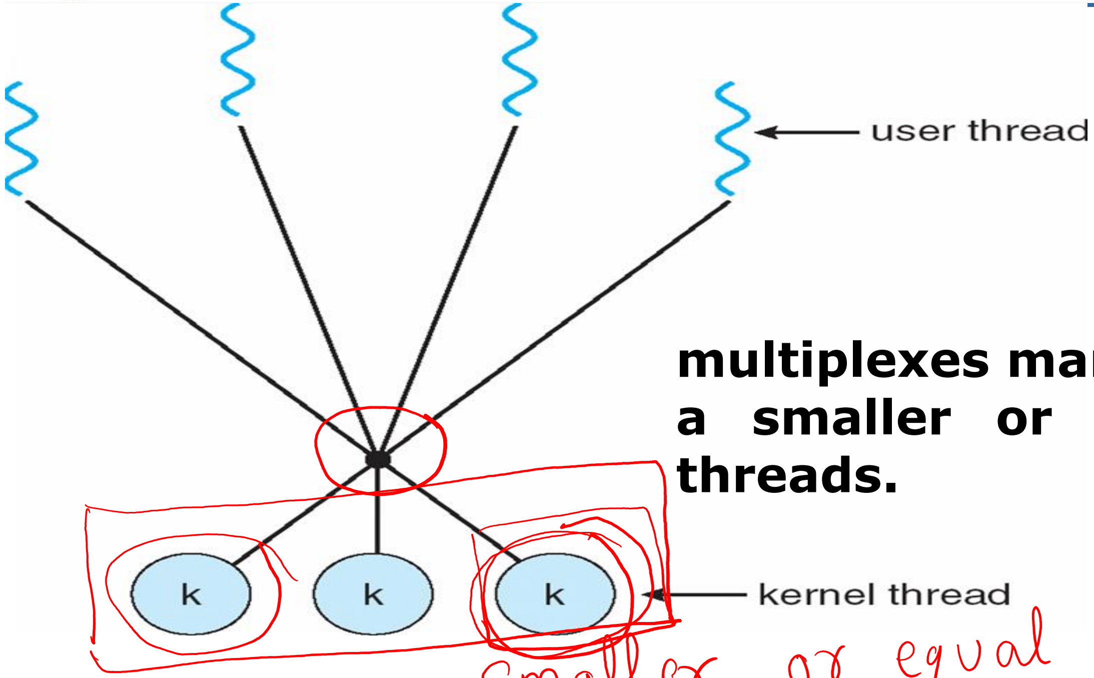


maps each user thread to a kernel thread

- 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.

- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.

Many-to-Many Model



multiplexes many user-level threads to a smaller or equal number of kernel threads.

Smaller or or equal

The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

4.4 Threading Issues

4.4.1 The fork() and exec() System Calls

4.4.2 Cancellation

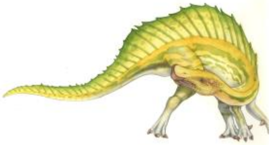
4.4.3 Signal Handling

4.4.4 Thread Pools

4.4.5 Thread-Specific Data

4.4.1 The fork() and exec() System Calls

```
#include <sys/types.h>
main()
{
pid_t pid;
pid = fork(); //Creates a child process

if (pid == 0)
		cout<<“\n I’m the child process”;
else if (pid > 0)
		cout<<“\n I’m the parent process. My child pid is
%d”<<pid;

else
		cout<<“Error in fork”;
}
```



exec system call

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
main(void) {
        pid_t pid = 0;        int   status;        pid = fork();
        if (pid == 0) {                 cout<<"I am the child.\n“;
                                         execl("/bin/ls", "ls", "-l",NULL);
        }
        if (pid > 0)
                cout<<"I am the parent, and the child is %d.\n“<<pid;
            if (pid < 0)           cout<<“In fork():“;
  }
```

4.4.1 The fork() and exec() System Calls

- The semantics of the fork() and exec() system calls change in a multithreaded program.

- Does **fork()** duplicate only the calling thread or all threads?

- Some UNIXes have two versions of fork(one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.)

- **exec()** usually works as normal – replace the running process including all threads

4.4.2 Thread Cancellation

- **Thread Cancellation** is the task of terminating a thread before it has completed.

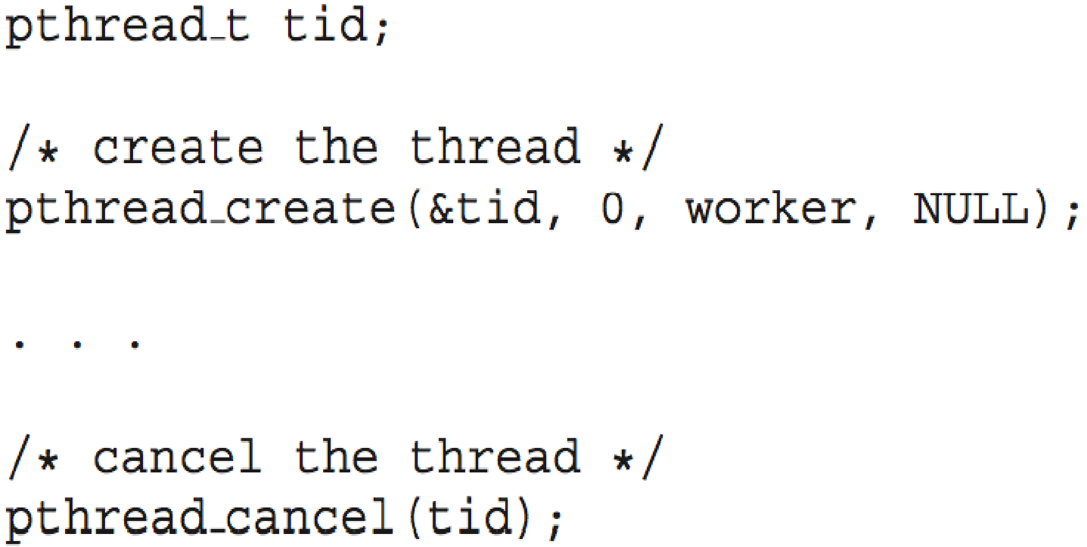
- For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled

- Thread to be canceled (killed) is **target thread**

4.4.2 Thread Cancellation

- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled at **thread cancellation point**. The cancellation point is the time where the thread could be safely be cancelled.

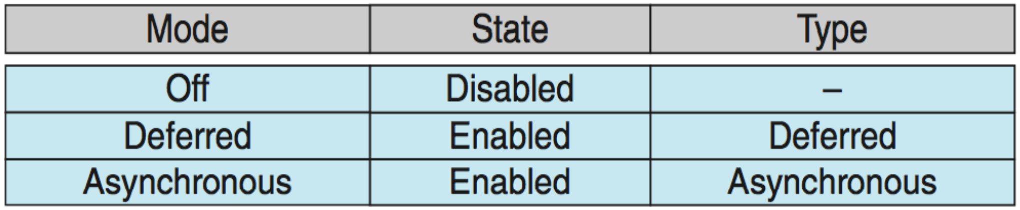
- Pthread code to create and cancel a thread:



```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

4.4.2 Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

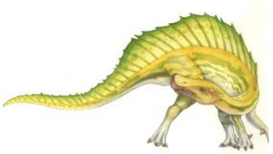


Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - To establish a cancellation point use the system call, **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked(release resources)

4.4.3 Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
    - Signal is generated by particular event
    - Signal is delivered to a process
- Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal.
- When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as <control><C>) and having a timer expire



4.4.3 Signal Handling

Signal is handled by one of two signal handlers:

A default signal handler

user-defined signal handler

Every signal has **default handler** that kernel runs when handling signal **User-defined signal handler** can override default

4.4.3 Signal Handling

- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process.
- However, delivering signals is more complicated in multithreaded programs

Handling signals in multi-threaded programs:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

Eg:Some asynchronous signals—such as a signal that terminates a process (<control><C>, for example)—should be sent to all threads.

4.4.4 Thread Pools

- Unlimited threads could exhaust system resources, such as CPU time or memory.

- One solution to this problem is to use a thread pool.

- The general idea behind a thread pool is to create a number of threads at process startup and place them into a pool, where they sit and wait for work.

- Thread pools offer these benefits:

1. Servicing a request with an existing thread is usually faster than waiting to create a thread.

2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

4.4.5 Thread-Specific Data

- Threads belonging to a process share the data of the process
- In Some situations, each thread might need its own copy of certain data.
- Such thread's own copy of data is referred as **Thread-Local Storage (TLS)**
- For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific data.