THREADS

We are happy with each *process* that has *an address space* and *a single thread of control* – and the concept of processes (an instant of a running program) has made high performance OS design possible.
However, in many situations, it is desirable to have

- multiple threads of control in the same address space
- running in quasi-parallel, as though they were (almost) separate processes (except for the shared address space).

The main reason to go for the threads is that in certain cases **multiple activities are going on simultaneously and some of them block from time to time. By splitting the activities into mutiple threads running in parallel sharing the same address space makes the programming model easier and offers performance gain and flexibilities.** This cannot be achieved by having separate processes as they do not share the same address space and the data elements.

THREADS–Example

Other benefits of threads are

- Threads (technically light weight processes) are faster to create and kill (10 to 100 times) than that of the processes
- Performance benefits – if the threads can overlap for a good mix of CPU and I/O bound activities by the process that may be true in many cases.
- More performance benefits for multi-core CPU where the threads can achieve real parallelism.

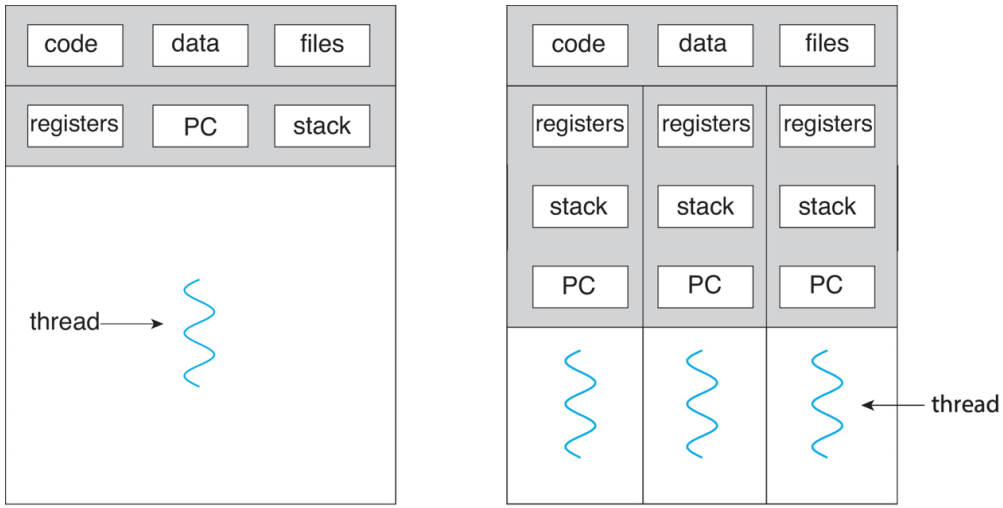
THREADS–Example

A practical example of the use of thread – in all modern word processor spelling mistakes are detected and flagged as you type the word. Here, possibly three threads are working under a single process; one for getting the input from the keyboard; the other is showing you the document on the screen (managing the format etc.) and the third is checking what is being typed and flagging spelling errors (as well as offering possible correct options) with the help of a built-in dictionary.
If the process is single threaded then surely there would be delays in getting the next caharacter from the keyboard – perhaps to manage spell checking and displaying the formatted contents. It cannot be managed with three separate processes as well as they do not share the data elements (residing in different address spaces). Now with three threads and sharing the common data elements (the document) in memory we can offer a very elegant, fast, and low cost solution.

THREAD: Intra process concurrency/parallelism

For more powerful hardwares (particularly with multiple cores) a process with several threads yields a lot of benefits.

- A thread is a basic unit of CPU utilisation with its PC, registers, stack etc,
- A thread shares the code, data section along with open files and signals with the peer thread
- Each thread belongs to exactly one process and no thread can exist outside a process.
- Each thread represents a separate flow of control within a process



single-threaded process

multithreaded process

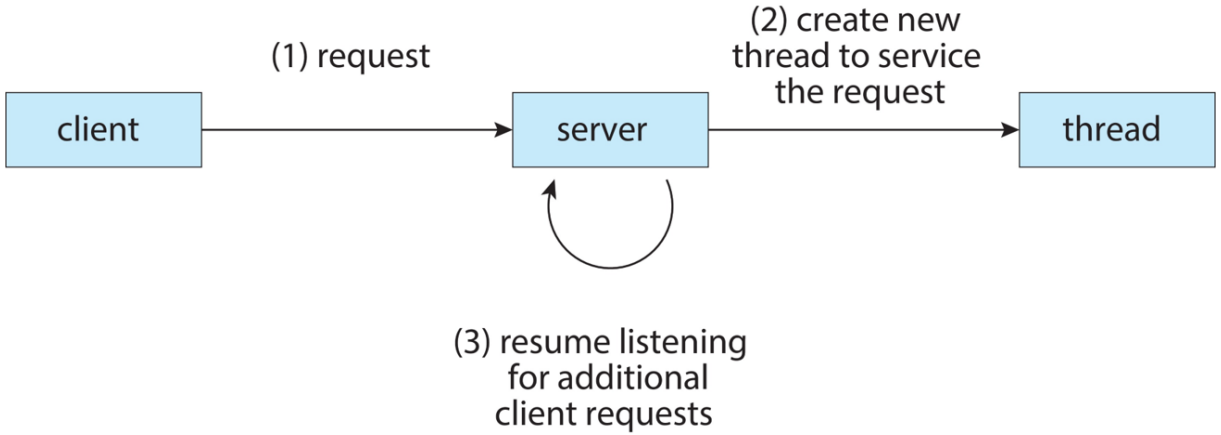
THREADS – Why?

In certain cases, a single application may required to perform several similar tasks. For example a web-server accepts hundreds of client requests for pages etc. The server may run as a single process and can create several child processes to comply the client requets. However;

- Process creation takes time and it is resource intensive
- Moreover, these chlid processes would basically do the same thing (say, return the pages requested by the clients)

A much better proposition is to go for a multi threaded process

- So, the server creates threads (to serve the client request)
- And the server continues listening to the client rquests



THREADS – complications

Thread has benefits but introduces complications in the programming model that requires careful considerations.

- A parent has a number of threads and now it has created a child through *fork()* system call – will the child gets all the threads of the parent? If yes
  - What happens if the thread in parent blocks due to a read() (from keyboard) then corresponding child thread blocks also (i.e., two threads waiting)
  - A new input line is typed at the keyboard – does it mean both the threads would receive the same ?
- Threads share many data structures.
  - If one thread closes a file while another is trying to read from the same file.
  - One thread notices memory shortage and allocates more – another thread (running after thread switch) may repeat the same – essentially allocation extra (double) memory space

To solve these problems can be solved extra effort and care are to be taken.

THREAD: Benefits

- **Responsiveness:** May allow a program to continue running even if part of it is blocked or is performing a lengthy operation. This quality is especially useful in designing responsive user interfaces.
- **Economy:** Thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
- **Resource sharing:** Threads share the memory and the resources of the process to which they belong by default.
- **Scalability:** Threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

Process and Thread : A comparison

PROCESS

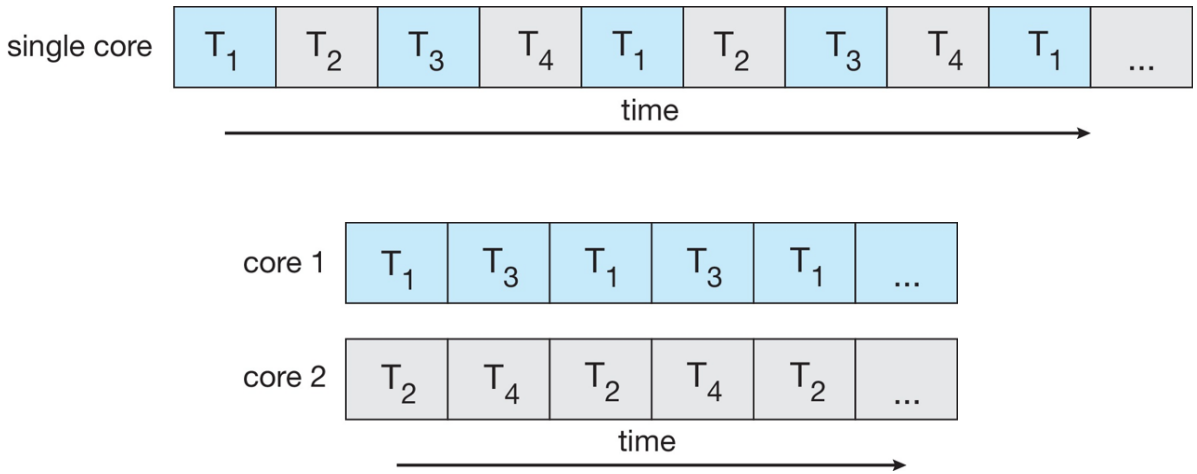
- Heavy weight or resource intensive
- Process switching needs interaction with operating system.
- Each process executes the same code but has its own memory and file resources.
- If one process is blocked, then no other process can execute until the first process is unblocked.
- Multiple processes without using threads use more resources.
- In multiple processes each process operates independently of the others.

THREAD

- Light weight, taking lesser resources
- Thread switching does not need to interact with operating system
- All threads can share same set of open files, child processes.
- While one thread is blocked and waiting, a second thread in the same task can run.
- Multiple threaded processes use fewer resources.
- One thread can read, write or change another thread's data

THREAD: multicore CPUs

Multiple cores in single CPU is logically equivalent to multiple CPUs to the OS and we may get more concurrency and real parallelism by having threads running on differt cores.
On a system with single core mutiple threads can only interleaved in time if they are blocking. However, in multicore system we can have more concurrency and parallelism by allocting threads to different cores albeit accepting increasing programming complexity.



THREAD: multicore CPUs contd.

- **Testing and debugging:** When a program is running in parallel on multiple cores, many different execution paths are possible. Testing (and debugging) such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

With more cores the speedup approximately follows Amdahl Law

$$speedup(s\_up) \leq (1/(S + (1 - S)/N)$$

where S is the portion of the code that must be executed serially and N is the number of cores. For a job where $S = 25\%$ the speed-up will be 1.6 times for 2 cores and 2.28 for 4 cores.

Seriality	Core	s_up	Seriality	Core	s_up	Seriality	Core	s_up
0.05	2	2	0.05.	4	4	0.05	16	16
0.10	2	1.9	0.10	4	3.8	0.10	16	8.6
0.50	2	1.7	0.50	4	1.8	0.50	16	1.96

For increasing N the speed-up converges to 1/S. So, it is dictated by S that corroborates the result.

THREADS : Challenges in multicore system

- **Identifying tasks:** This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.
- **Balance:** While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. Using acore for not so important task does not fit the bill.
- **Data splitting:** Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
- **Data dependency:** The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, synchronisation is needed

THREAD: Data or Task parallelism

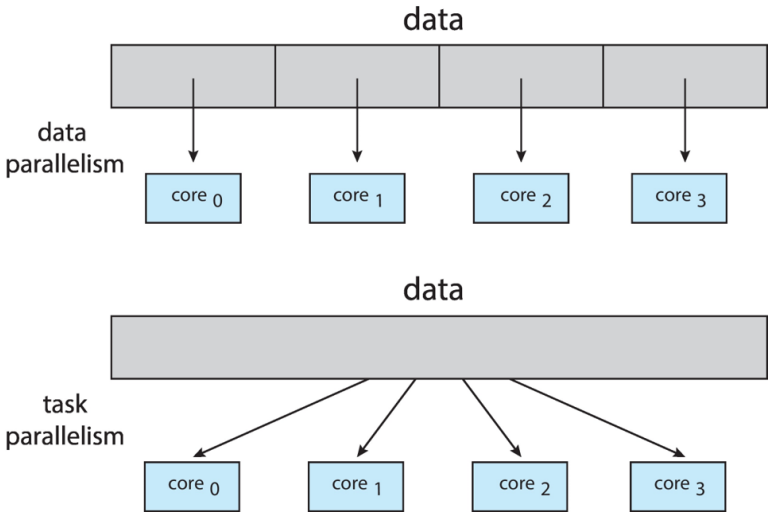
In general we have two type of parallelism; i) *data* and ii) *task*.

i) Data Parallelism

- Focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core

II) Task Parallelism

- involves distributing tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.



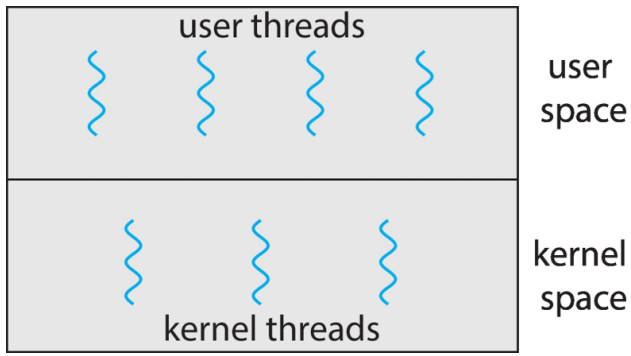
USER/KERNEL THREADS

Support for thread may be offered in user level or kernel level

- User level threads act on above the kernel threads and
- needs no kernel support

Kernel threads

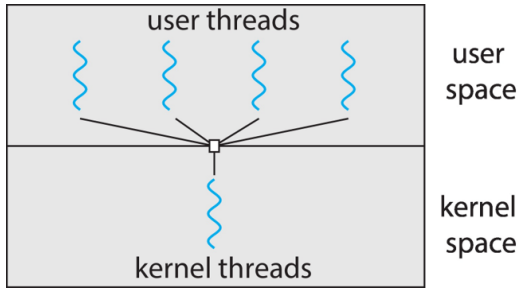
- Are managed by kernel
- All modern OS supports kernel thread



This two-tier model has many variations establishing different relationships between the kernel and user level threads.

THREAD MODELS: Many to One

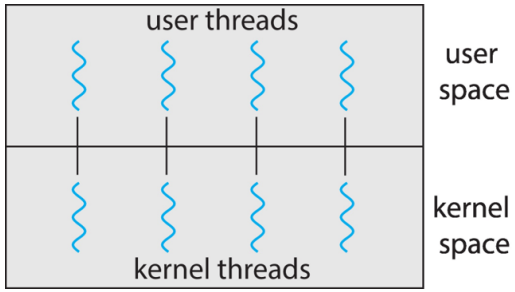
Many to one model



- Many user level thtread is managed by a single kernel thread
- Mangement is done by the thread library in users space– efficient; however
- entire process blocks for a blocking system call
- Only one thread has access to kernel at a time; cannot exploit multicore system– hence obsolete

THREAD MODELS: One to One

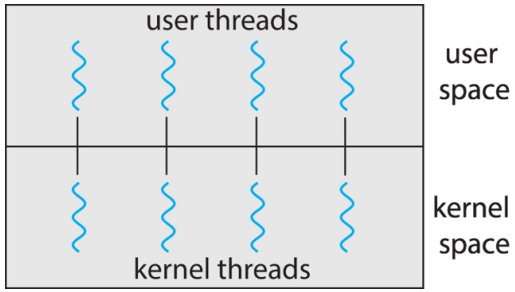
One to one model; used in Linux and windows



- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread – so there may be hundreds of kernel threads leads to performance degradation
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

THREAD MODELS: Many to Many

One to one model; used in Linux and windows



- Allows many user level threads to be mapped to equal or less no of kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the ThreadFiber package
- Otherwise not very common

THREAD : POSIX

In order to support portable threaded program IEEE (standard no. 1003.1c) has defined a thread package (**pthreads** .Most UNIX systems support it. Some calls (there around 60 in total) are listed below.

Thread call [1]	Action taken
pthread_create	Create a new thread
pthread_exit	Terminate the callling thread
pthread_join	Wait for a specific thread to exit
pthread_yield	Releases the CPU to let another thread run
pthread_attr_int	Create and initilise the attribute structure of a thread
pthread_attr_destroy	Remove the attribute structure of a thread
pthread_detach	To detach a thread
pthread_once	To initialize the state associated with a thread routine

[1] See *man pages for full description*

THREAD: Example

The following program creates a peer thtread from the main thread and prints two different strings (one in main and the other in the peer thread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void *thread1(void *vargp);
int main()
{
    pthread_t tid1;
    pthread_create(&tid1, NULL, thread1, NULL);
    pthread_join(tid1, NULL);
    printf("Hello \n");
    return 0;
    }
void *thread1(void *vargp)
{
    printf("hello world1\n");
    return NULL;
}
```

THREAD: Example ... contd.

Some explanation in order.

- The main program (main thread, here) creates a peer thread using the pthread_create call and the single local variable tid1 (the thread1 id) that will be used to identify the peer thread
- When the call returns from creating the thred we have two concurrent threads running; one is the main and the other is the peer thread (thread1).
- The main thread waits for the peer thread to terminate with the call to pthread_join
- Finally the main thread call exit() that terminates all the threads (in this case only the main thread) concurrently running in the process.

THREAD: Example ... contd.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */
int main(int argc, char *argv[])
{
pthread_t tid; /* the thread identifier */
pthread_attr_t attr; /* set of thread attributes */
/* set the default attributes of the thread */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid, &attr, runner, argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);
printf("sum = %d", sum);
}
```

THREAD: Example

```
/* The thread will execute in this function */
void *runner(void *param)
{
int i, upper = atoi(param);
sum = 0;
for (i = 1; i <= upper; i++)
sum += i;
pthread_exit(0);
}
```