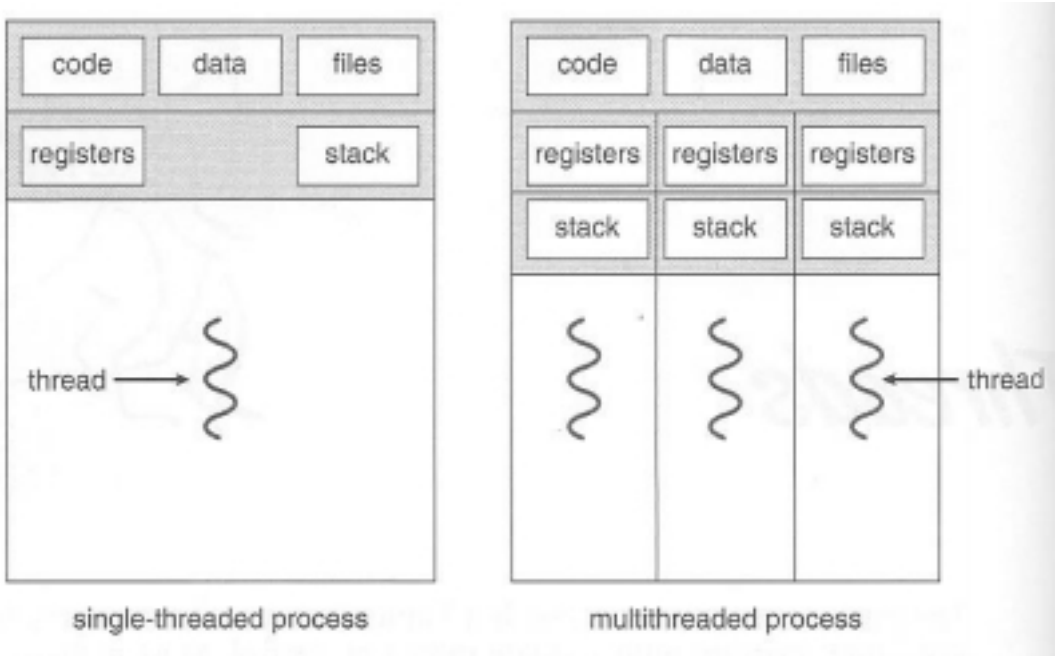
Multi threaded programming

· A ***thread*** is a basic unit of CPU utilization. It consists of a thread ID, program counter, a stack, and a set of registers.

· Traditional processes have a single thread of control. It is also called as **heavyweight process**. There is one program counter, and one sequence of instructions that can be carried out at any given time.

· A multi-threaded application 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. Such process are called as **lightweight 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.

· In a web server - Multiple threads allow for multiple requests to be served simultaneously. A thread is created to service each request; meanwhile another thread listens for more client request.

· In a web browser – one thread is used to display the images and another thread is used to retrieve data from the network.

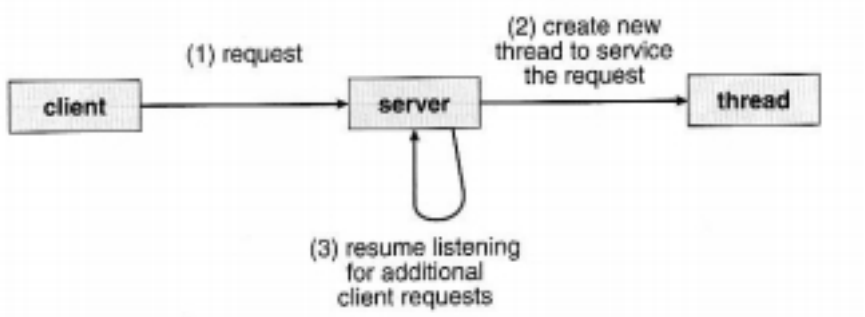


Figure 4.2 Multithreaded server architecture.

Benefits

The four major benefits of multi-threading are:

1. **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
   Multi threading allows a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
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 is much faster than performing the same tasks for processes. Context switching between threads takes less time.

4. **Scalability**, i.e. **Utilization of multiprocessor architectures** – Multithreading can be greatly utilized in a multiprocessor architecture. A single threaded process can make use of only one CPU, whereas the execution of a multi threaded application may be split among the available processors. Multithreading on a multi-CPU machine increases concurrency. In a single processor architecture, the CPU generally moves between each thread so quickly as to create an illusion of parallelism, but in reality only one thread is running at a time.

Multicore Programming

· A recent trend in computer architecture is to produce chips with multiple ***cores***, or CPUs on a single chip.
· A multi-threaded application running on a traditional single-core chip, would have to execute the threads one after another. On a multi-core chip, the threads could be spread across the available cores, allowing true parallel processing.

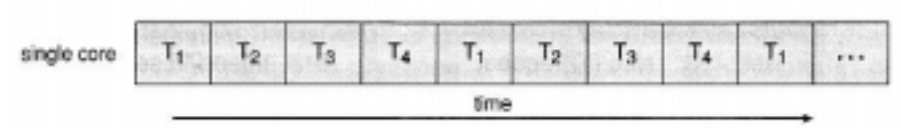


Figure 4.3 Concurrent execution on a single-core system.

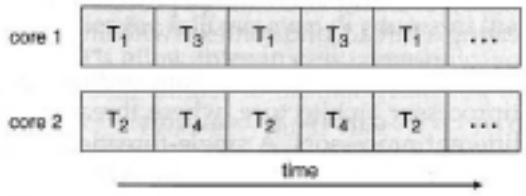


Figure 4.4 Parallel execution on a multicore system.

· For operating systems, multi-core chips require new scheduling algorithms to make better use of the multiple cores available.
· For application programmers, there are five areas where multi-core chips present new challenges:

1. Dividing activities - Examining applications to find activities that can be performed concurrently.
2. Balance - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.
3. Data splitting - To prevent the threads from interfering with one another.
4. Data dependency - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
5. Testing and debugging - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

Multithreading Models

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

Many-To-One Model

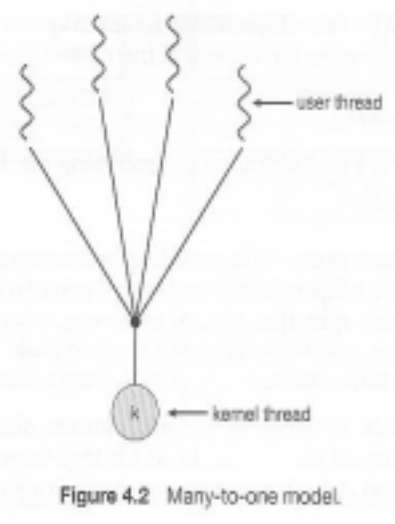


Figure 4.2 Many-to-one model.

In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.

· Thread management is handled by the thread library in user space, which is very efficient.
· If a blocking system call is made by one of the threads, then the entire process
blocks. Thus blocking the other user threads from continuing the execution.
· Only one user thread can access the kernel at a time, as there is only one kernel thread. Thus the threads are unable to run in parallel on multiprocessors.
· Green threads of Solaris and GNU Portable Threads implement the many-to
one model.

b) 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.
· This model places a limit on the number of threads created.
· Linux and Windows from 95 to XP implement the one-to-one model for threads.

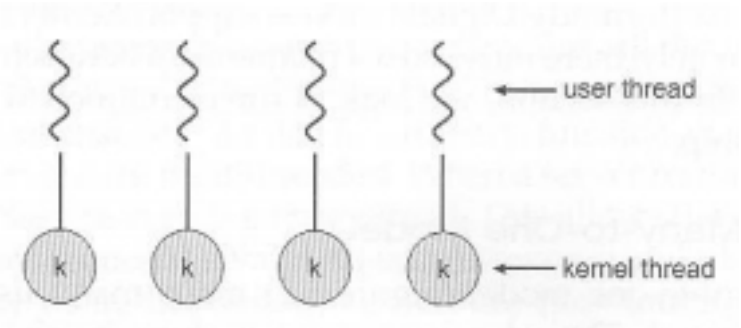


Figure 4.3 One-to-one model.

c) 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.

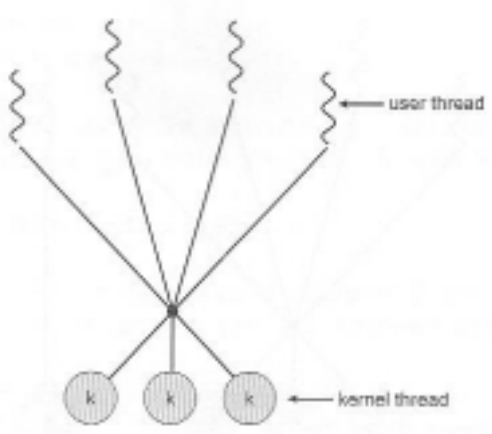


Figure 4.4 Many-to-many model.

· 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.
· This model is also called as two-tier model.
· It is supported by operating system such as IRIX, HP-UX, and Tru64 UNIX.

Threading Issues

a) The fork( ) and exec( ) System Calls

The fork( ) system call is used to create a separate, duplicate process. When a thread program calls fork( ),

· The new process can be a copy of the parent, with all the threads
· The new process is a copy of the single thread only (that invoked the process)

If the thread invokes the exec( ) system call, the program specified in the parameter to exec( ) will be executed by the thread created.

b) Cancellation

Terminating the thread before it has completed its task is called thread cancellation. The thread to be cancelled is called **target thread**.

Example : Multiple threads required in loading a webpage is suddenly cancelled, if the browser window is closed.

Threads that are no longer needed may be cancelled in one of two ways:

1. **Asynchronous Cancellation** - cancels the thread immediately.

2. **Deferred Cancellation** – the target thread periodically check whether it has to terminate, thus gives an opportunity to the thread, to terminate itself in an orderly fashion. In this method, the operating system will reclaim all the resources before cancellation.

c) Signal Handling

A signal is used to notify a process that a particular event has occurred.

All signals follow same path

1) A signal is generated by the occurrence of a particular event.
2) A generated signal is delivered to a process.
3) Once delivered, the signal must be handled.

A signal can be invoked in 2 ways : synchronous or asynchronous.

Synchronous signal – signal delivered to the same program. Eg – illegal memory access, divide by zero error.

Asynchronous signal – signal is sent to another program. Eg – Ctrl C

In a single-threaded program, the signal is sent to the same thread. But, in multi threaded environment, the signal is delivered in variety of ways, depending on the type of signal –

· Deliver the signal to the thread, to which the signal applies.
· Deliver the signal to every threads in the process.
· Deliver the signal to certain threads in the process.
· Deliver the signal to specific thread, which receive all the signals.

A signal can be handled by one of the two ways –

Default signal handler - signal is handled by OS.

User-defined signal handler - User overwrites the OS handler.

d) Thread Pools

In multithreading process, thread is created for every service. Eg – In web server, thread is created to service every client request.

Creating new threads every time, when thread is needed and then deleting it when it is done can be inefficient, as –

Time is consumed in creation of the thread.

A limit has to be placed on the number of active threads in the system. Unlimited thread creation may exhaust system resources.

An alternative solution is to create a number of threads when the process first starts, and put those threads into a ***thread pool***.

· Threads are allocated from the pool when a request comes, and returned to the pool when no longer needed(after the completion of request).
· When no threads are available in the pool, the process may have to wait until one becomes available.

Benefits of Thread pool –

· Thread creation time is not taken. The service is done by the thread existing in the pool. Servicing a request with an existing thread is faster than waiting to create a thread.
· The thread pool limits the number of threads in the system. This is important on systems that cannot support a large number of concurrent threads.

The (maximum) number of threads available in a thread pool may be determined by parameters like the number of CPUs in the system, the amount of memory and the expected number of client request.

e) Thread-Specific Data

· Data of a thread, which is not shared with other threads is called thread specific data.
· Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data.

Example – if threads are used for transactions and each transaction has an ID. This unique ID is a specific data of the thread.

f) Scheduler Activations

Scheduler Activation is the technique **used** for communication between the user thread library and the kernel.
It works as follows:

- the kernel must inform an application about certain events. This procedure is known as an **upcall.**
- Upcalls are handled by the thread library with an **upcall handler,** and upcall handlers must run on a virtual processor.

Example - The kernel triggers an upcall occurs when an application thread is about to block. The kernel makes an upcall to the thread library informing that a thread is about to block and also informs the specific ID of the thread.

The upcall handler handles this thread, by saving the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.

The upcall handler then schedules another thread that is eligible to run on the virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. Thus assigns the thread to the available virtual processor.

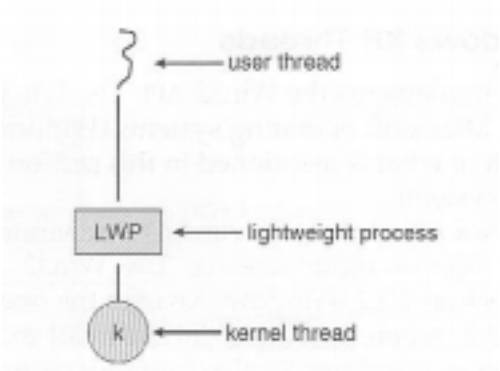


Figure 4.9 Lightweight process (LWP)

Thread Libraries

· Thread libraries provide an API for creating and managing threads. · Thread libraries may be implemented either in user space or in kernel space. · There are three main thread libraries in use –

1. POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
2. Win32 threads - provided as a kernel-level library on Windows systems.
3. Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

· The following sections will demonstrate the use of threads in all three systems for calculating the sum of integers from 0 to N in a separate thread, and storing the result in a variable "sum".

4.3.1 Pthreads

· The POSIX standard (IEEE 1003.1c) defines the *specification* for pThreads, not the *implementation*.
· pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
· Global variables are shared amongst all threads.
· One thread can wait for the others to rejoin before continuing. · pThreads begin execution in a specified function, in this example the runner( ) function.
· Pthread_create() function is used to create a thread.

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   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\n",sum);
}

/* The thread will begin control 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);
}
```

Figure 4.6 Multithreaded C program using the Pthreads API.

Refer example programs discussed inn the class using Pthread