Thread:

A thread a single sequential flow of control. In a high level language you normally program a thread using procedures, where the procedure calls follow the traditional stack discipline.

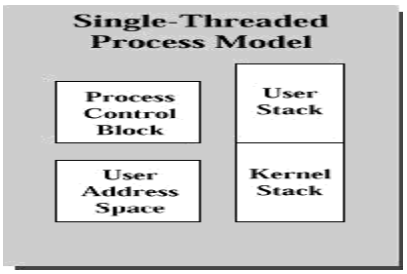
Within a single thread, there is at any instant a single point of execution.

Having "multiple threads" in a program means that at any instant the program has multiple points of execution, one in each of its threads. The programmer can mostly view the threads as executing simultaneously, as if the computer were endowed with as many processors as there are threads. The programmer is required to decide when and where to create multiple threads, or to accept such decisions that are made by implementers of existing library packages or runtime systems. Additionally, the system might not execute all the threads simultaneously. Having the threads execute within a "single address space" means that the computer's addressing hardware is configured so as to permit the threads to read and write the same memory locations. In a high-level language, this usually corresponds to the fact that the off-stack (global) variables are shared among all the threads of the program. Each thread executes on a separate call stack with its own separate local variables. The programmer is responsible for using the synchronization mechanisms of the thread facility to ensure that the shared memory is accessed in a manner that will give the correct answer.

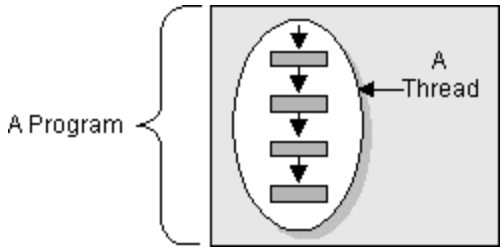
Thread facilities are always "lightweight". This means that thread creation, existence, destruction and synchronization primitives are cheap enough that the programmer will use them for all his concurrency needs. A thread is also called *lightweight process* instead of thread. A thread is similar to a real process in that a thread and a running program are both a single sequential flow of control. However, a thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment.

As a sequential flow of control, a thread must carve out some of its own resources within a running program. (It must have its own execution stack and program counter for example.) The code running within the thread works only within that context. Thus, some other texts use *execution context* as a synonym for thread.

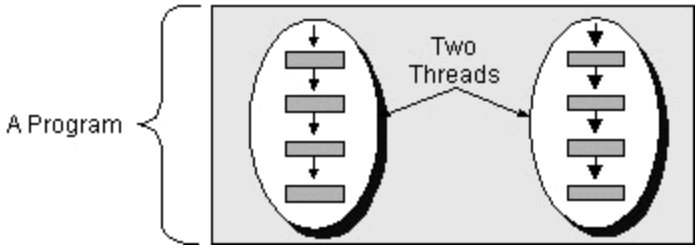
A thread is similar to the sequential programs .A single thread also has a beginning, a sequence, and an end and at any given time during the runtime of the thread, there is a single point of execution.



However, a thread itself is not a program; it cannot run on its own. Rather, it runs within a program.



Multiple threads can be used in a single program, run at the same time and perform different tasks.



Web browser is an example of a multithreaded application. Within the HotJava browser scrolling a page while it's downloading an applet or image, play animation and sound concurrently, print a page in the background while downloading a new page can be done simultaneously.

Reason for using threads:

With multiprocessor machines, there are multiple simultaneous points of execution, and threads are an attractive tool to take advantage of the available hardware. The alternative, with most conventional operating systems, is to configure as multiple separate processes, running in separate address spaces. This is expensive to set up, and the costs of communicating between address spaces is high, even in the presence of shared segments. By using a lightweight multi-threading facility, the processors can be economically used.

A second area where threads are useful is in driving slow devices such as disks, networks, terminals and printers. In these cases an efficient program should be doing some other useful work while waiting for the device to produce its next event (such as the completion of a disk transfer or the receipt of a packet from the network) by adopting device requests that are all sequential (i.e. they suspend execution of the invoking thread until the request completes), and meanwhile other work is done in other threads.

Third, threads are a convenient way of doing two to three things at a time. The typical arrangement of a modern window system is that each time the user invokes an action (by clicking a button with the mouse, for example), a separate thread is used to implement the action. If the user invokes multiple actions, multiple threads will perform them in parallel. (The implementation of the window system also uses a thread to watch the mouse actions themselves, since the mouse is an example of a slow device.)

A final source of concurrency appears when building a distributed system.
Frequently encounter with shared network servers (such as a file server or a spooling print server), where the server is willing to service requests from multiple clients. Use of multiple threads allows the server to handle clients' requests in parallel, instead of

artificially serializing them (or creating one server process per client, at great expense).
Concurrency can be deliberately added to in order to reduce the latency of operations (the elapsed time between calling a procedure and the procedure returning). Often, some of the work incurred by a procedure can be deferred, since it does not affect the result of the procedure. For example, when you add or remove something in a balanced tree it could be returned to the caller before re-balancing the tree. With threads this can be achieved easily: do the re-balancing in a separate thread. If the separate thread is scheduled at a lower priority, then the work can be done at a time when its less busy (for example, when waiting for user input). Adding threads to defer work is a powerful technique, even on a uni-processor. Even if the same total work is done, reducing latency can improve the responsiveness.

Design:

The various systems that support threads offer primitives that are provided by a multi-threading facility. In general, there are four major mechanisms: thread creation, mutual exclusion, waiting for events, and an arrangement for getting a thread out of an unwanted long-term wait.

Benefits:

Threads can be used to improve program's perceived performance. Threads can simplify a program's code or architecture.

Situations where threads are used:

- To move a time-consuming initialization task out of the main thread, so that the GUI comes up faster. Examples of time-consuming tasks include making extensive calculations and blocking for network or disk I/O (loading images, for example).
- To move a time-consuming task out of the event-dispatching thread, so that the GUI remains responsive.
- To perform an operation repeatedly, usually with some predetermined period of time between operations.
- To wait for messages from other programs.

Benefits of multithreading:

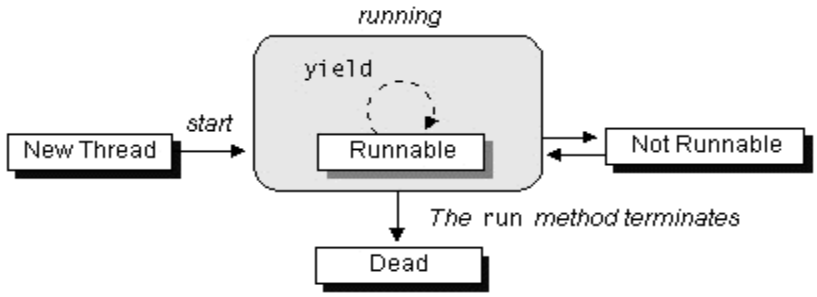
- Responsiveness
- Resource sharing
- Economy
- Utilization of multiprocessor architecture

Multithreaded programs introduces challenges

- Semantics of fork and exec system call
- Thread cancellation
- Signal handling
- Thread specific data

The Life Cycle of a Thread:

.



E.g.: States that a Java thread can be in during its life

Creating a Thread/Thread creation

Calling "Fork", giving it a procedure and an argument record, creates a thread.
The effect of "Fork" is to create a new thread, and start that thread executing asynchronously at an invocation of the given procedure with the given arguments. When the procedure returns, the thread dies.
Usually, "Fork" returns to its caller a handle on the newly created thread. Most forked threads are permanent dæmon threads, or have no results, or communicate their results by some synchronization arrangement
When a thread is a New Thread, it is merely an empty Thread object; no system resources are allocated for it yet. When a thread is in this state, the thread can only be started.

Starting a Thread

This creates the system resources necessary to run the thread, schedules the thread to run, and calls the required thread method. A thread that has been started is actually in the Runnable state.
Many computers have a single processor, thus making it impossible to run all "running" threads at the same time.
A scheduling scheme must be implemented that shares the processor between all "running" threads. So at any given time, a "running" thread actually may be waiting for its turn in the CPU.

Thread Priority

Most computer configurations have a single CPU, so threads actually 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 an order, is called *scheduling*. The Java runtime supports scheduling algorithm known as *fixed priority scheduling*. This algorithm schedules threads based on their *priority* relative to other runnable threads.

When a thread is created, it inherits its priority from the thread that created it. thread's priority can be modified at any time after its creation using a created method. Thread priorities are integers ranging between a minimum and maximum. The higher the integer, the higher the priority. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. Only when that thread stops, yields, or becomes not runnable for some reason will a lower priority thread start executing. If two threads of the same priority are waiting

for the CPU, the scheduler could choose one of them to run in a round-robin fashion. The chosen thread will run 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.

Thread scheduling algorithm is *preemptive* that is 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 higher priority thread is said to *preempt* the other threads. At any given time, the highest priority thread is running. However, this is not guaranteed. The thread scheduler may choose to run a lower priority thread to avoid starvation. For this reason, priority is used only to affect scheduling policy for efficiency purposes and is not reliable for algorithm correctness.

Once the scheduler chooses a thread with this thread body for execution, the thread never voluntarily relinquishes control of the CPU--the thread continues to run until the while loop terminates naturally or until the thread is preempted by a higher priority thread. This thread is called a *selfish thread*.

In some situations, having selfish threads does not cause any problems because a higher priority thread preempts the selfish one. However, in some other situations, threads with CPU-greedy run methods, can take over the CPU and cause other threads to wait for a long time before getting a chance to run. Some systems, such as Windows 95/NT, fight selfish thread behavior with a strategy known as *time-slicing*.

Time-slicing comes into play when there are multiple "Runnable" threads of equal priority and those threads are the highest priority threads competing for the CPU. A time-sliced system divides the CPU into time slots and iteratively gives each of the equal-and-highest priority threads a time slot in which to run. The time-sliced system iterates through the equal-and-highest priority threads, allowing each one a bit of time to run, until one or more of them finishes or until a higher priority thread preempts them. Time-slicing makes no guarantees as to how often or in what order threads are scheduled to run. When running this program on a non-time-sliced system, messages from one thread finish printing is seen before the other thread ever gets a chance to print one message. This is because a non-time-sliced system chooses one of the equal-and-highest priority threads to run and allows that thread to run until it relinquishes the CPU (by sleeping, yielding, finishing its job) or until a higher priority preempts it. Writing CPU-intensive code can have negative repercussions on other threads running in the same process. In general, "well-behaved" threads must be written so that they voluntarily relinquish the CPU periodically and give other threads an opportunity to run.

Making a Thread Not Runnable

A thread becomes Not Runnable when one of these events occurs:

- Its sleep method is invoked.
- The thread calls the wait method to wait for a specific condition to be satisifed.
- The thread is blocking on I/O.

For each entrance into the Not Runnable state, there is a specific and distinct escape route that returns the thread to the Runnable state. An escape route works only for its corresponding entrance. For example, if a thread has been put to sleep, then the specified number of milliseconds must elapse before the thread becomes Runnable again. The escape route for every entrance into the Not Runnable state:

- If a thread has been put to sleep, then the specified number of milliseconds must elapse.
- If a thread is waiting for a condition, then another object must notify the waiting thread of a change in condition
- If a thread is blocked on I/O, then the I/O must complete.

with independent, asynchronous threads. That is, each thread contained all of the data and methods required for its execution and didn't require any outside resources or methods. The threads run at their own pace without concern over the state or activities of any other concurrently running threads.

Some concurrently running threads do share data and must consider the state and activities of other threads. One such set of programming situations are known as producer/consumer scenarios where the producer generates a stream of data, which then is consumed by a consumer.

For example, an application where one thread (the producer) writes data to a file while a second thread (the consumer) reads data from the same file. Or, as you type characters on the keyboard, the producer thread places key events in an event queue and the consumer thread reads the events from the same queue. Both of these examples use concurrent threads that share a common resource: the first shares a file, the second shares an event queue. Because the threads share a common resource, they must be synchronized in some way.

Locking an Object

The code segments within a program that access the same object from separate, concurrent threads are called *critical sections*. A critical section can be a block or a method and are identified with a keyword. Usage: Semaphores, Mutex

A lock is associated with every object that has synchronized code. The system associates a unique lock with every instance. Whenever control enters a synchronized method, the thread that called the method locks the object whose method has been called. Other threads cannot call a synchronized method on the same object until the object is unlocked. The acquisition and release of a lock is done automatically and atomically by the Java runtime system. This ensures that race conditions cannot occur in the underlying implementation of the threads, thus ensuring data integrity. The two threads must also be able to notify one another when they've done their job.

These timed wait methods to synchronize threads can be used in place of sleep. Both wait and sleep delay for the requested amount of time, but you can easily wake up wait with a notify but a sleeping thread cannot be awakened prematurely. This does not matter too much for threads that don't sleep for long, but it could be important for threads that sleep for minutes at a time.

Poor performance through lock conflicts

In general, to get good performance you must arrange that lockconflicts are rare events. The best way to reduce lock conflicts is to lock at a finer granularity; but this introduces complexity. It is atrade-off inherent in concurrent computation.

The most typical example where locking granularity is important is in a module that manages a set of objects, for example a set of open buffered files.

The simplest strategy is to use a single mutex for all the operations: open, close, read, write, and so forth. But prevents multiple writes on separate files proceeding in parallel. So a better strategy is to use one lock for operations on the global list of open files, and one lock per open file for operations affecting only that file. This can be achieved by associating a mutex with the record representing each open file.

There is an interaction between mutexes and the thread scheduler that can produce particularly insidious performance problems. The scheduler is the part of the thread implementation (often part of the operating system) that decides which of the non-blocked threads should actually be given a processor to run on. Generally the scheduler makes its decision based on a *priority* associated with each thread. (priority might be fixed or dynamic, programmer assigned or computed by the scheduler.)

Lock conflicts can lead to a situation where some high priority thread never makes progress at all, despite the fact that its high priority indicates that it is more urgent than the threads actually running.

Starvation and Deadlock

If several concurrent threads are competing for resources, precautions must be taken to ensure fairness. A system is fair when each thread gets enough access to limited resource to make reasonable progress. A fair system prevents *starvation* and *deadlock*.

Starvation occurs when one or more threads in your program is blocked from gaining access to a resource and thus cannot make progress. When you are using a mutex, fairness is achieved by a first-in-first-out rule for each priority level.
This is also true for condition variables.

Deadlock is the ultimate form of starvation; it occurs when two or more threads are waiting on a condition that cannot be satisfied. Deadlock most often occurs when two (or more) threads are each waiting for the other(s) to do something. Deadlocks can also be introduced by using condition variables

Condition variable: scheduling shared resources

A condition variable is used when the programmer wants to schedule the way in which multiple threads access some shared resource, and the simple one-at-a-time mutual exclusion(eg provided by mutexes) is not sufficient.
For example, when one or more producer threads are passing data to one or more consumers.
The data is transferred through an unbounded buffer formed by a linked list whose head is the global variable "head". If the linked list is empty, the consumer blocks on the condition variable "nonEmpty" until the producer generates some more data. The list and the condition variable are protected by the mutex "m".

Spurious wake-ups
If condition variables are kept very simple, it might introduce the possibility of awakening threads that cannot make useful progress.

Spurious lock conflicts
The straightforward use of condition variables can lead to excessive scheduling overhead.

Complexity
Spurious wake-ups, lock conflicts and starvation makes the program more complicated. The first solution of the reader/writer problem that for each case, it has to be considered whether the potential cost of ignoring the problem is enough to merit writing a more complex implementation. This decision depends on the performance characteristics of threads implementation, on whether a multi-processor is being used, and depends on the expected load on the resource. In particular, if resource is mostly *not* in use then the performance effects will not be a problem

Stopping a Thread

A program doesn't stop a thread. A thread arranges for its own death by having a method that terminates naturally

The isActive Method

The API for the Thread class includes a method called isActive. The isActive method returns true if the thread has been started and not stopped. If the isActive method returns false, you know that the thread either is a New Thread or is Dead. If the isActive method returns true,then the thread is either Runnable or Not Runnable.Ther is no differentiation between a New Thread or a Dead thread nor between a Runnable thread and a Not Runnable thread.

Mutual exclusion

The simplest way that threads interact is through access to shared memory. In a high-level language, this is usually expressed as access to global variables. Since threads are running in parallel, it must be explicitly arrange to avoid errors arising when more than one thread is accessing the shared variables. The simplest tool for doing this is a primitive that offers mutual exclusion (sometimes called critical sections), specifying for a particular region of code that only one thread can execute there at any time.

Using fork: working in parallel

Situations where thread is forked:

- to utilize a multi-processor;
- to do useful work while waiting for a slow device;
- to satisfy human users by working on several actions at once;
- to provide network service
- to multiple clients simultaneously;
- and to defer work until a less busy time.

There are application programs that use several threads.

For example, one thread could do main computation, a second thread writing some output to a file, a third thread waiting for (or responding to) interactive user input, and a fourth thread running in background to clean up data structures (for example, re-balancing a tree).

When programming with threads, slow devices are driven through synchronous library calls that suspend the calling thread until the device action completes, but allow other threads in the address space to continue. To avoid waiting for the result of a device interaction,it could be invoked it in a separate thread. To have multiple device requests outstanding simultaneously, they can be invoked in multiple threads.

If program is interacting with a human user, it must be required to make it responsive even while it is working on some request. E.g. window oriented interfaces. Responsiveness can be achieved by using extra threads. The window system could call the program in a single thread synchronously with the user input event. Whether the requested action is short enough to do it synchronously, or whether it should be forked needs to be decided.

The problem with complexity introduced by using forked threads is in accessing data from the interactive interface (for example, the value of the current selection, or the contents of editable text areas) since these values might change once execution is started asynchronously.

Network servers are usually required to service multiple clients concurrently. If the network communication is based on RPC, since the server side of the RPC system will invoke each concurrent incoming call in a separate thread, by forking a suitable number of threads internally to its implementation.

For example, in a traditional connection-oriented protocol (such as file transfer layered on top of TCP), fork is done- one thread for each incoming connection. For client program and there is no need to wait for the reply from a network server, invoke the server from a separate thread.

The technique of adding threads in order to defer work is quite valuable. There are several variants of the scheme. The simplest is that as soon as the procedure has done enough work to compute its result, a thread can be forked to do the remainder of the work, and then return to caller in the original thread.

This reduces the latency of the procedure(the elapsed time from being called to returning),and the deferred work can be done more economically later (for example, because a processor goes idle).

The disadvantage of this simplest approach is that it might create large numbers of threads, and it incurs thecost of calling "Fork" each time.

Often, it is preferable to keep a single housekeeping thread and feed requests to it. For example, when the housekeeper is responsible for maintaining a data structure in an optimal form, although the main threads could get the correct answer without this optimization. The housekeeper could be programmed either to merge similar requests into a single action, or to restrict itself to run not more often than a chosen periodic interval.

On a multi-processor use of "Fork" is to utilize as many of processors as possible.

Pipelining

On a multi-processor, a chain of producer-consumer relationships can be built known as a *pipeline*.

For example, when thread A initiates an action, all it does is enqueue a request in a buffer.
Thread B takes the action from the buffer, performs part of the work, then enqueues it in a second buffer. Thread C takes it from there and does the rest of the work. This forms a three-stage pipeline. The three threads operate in parallel except when they synchronize to access the buffers, so this pipeline is capable of utilizing up to three processors. At its best, pipelining can achieve almost linear speed-up and can fully utilize a multi-processor.

A pipeline can also be useful on a uni-processor if each thread will encounter some realtime delays (such as page faults, device handling or network communication).

There are two problems with pipelining:
First, how much of the work gets done in each stage. The ideal is that the stages are equal: this will provide maximum throughput, by utilizing all processors fully. Achieving this ideal requires hand tuning, and re-tuning as the program changes.
Second, the number of stages in the pipeline determines statically the amount of concurrency. If the number of processors is known, and exactly where the real-time delays occur, this will be fine. For more flexible or portable environments it can be a problem.
Despite these problems, pipelining is a powerful technique that has wide applicability.

The impact of environment
The design of the operating system and runtime libraries will affect the extent to which it is desirable or useful to fork threads. The operating system should not suspend the entire address space just because one thread is blocked for an i/o request (or for a page fault). The operating system and the libraries must permit calls from multiple threads in parallel. Generally, in a well-designed environment for supporting multi-threaded programs, the facilities of the operating system and libraries are available as synchronous calls that block only the calling thread .Some of the performance parameters of the threads implementation needs to be known- cost of creating a thread, cost of keeping a blocked thread in existence, cost of a context switch to decide to what extent it is feasible or useful to add extra threads.

Potential problems with adding threads
If there are more threads ready to run than there are processors, then performance degrades. This is partly because most thread schedulers are quite slow at making general re-scheduling decisions. If there is a processor idle waiting for thread, the scheduler can probably get it there quite quickly. But if thread has to be put on a queue, and later swapped into a processor in place of some other thread, it will be more expensive. A second effect is that if there are lots of threads running they are more likely to conflict over the resources managed .

Mostly, when threads are added for performance purposes (such as performing multiple actions in parallel, or deferring work, or utilizing multiprocessors), its possible to overload the system. This applies only to the threads that are ready to run.
The expense of having threads blocked on condition variables is usually less significant,being just the memory used for scheduler data structures and the thread stack.
In most systems the thread creation and termination facilities are not cheap. The threads implementation takes care to cache a few terminated thread carcasses, to reduce cost of stack creation on each fork, but a call of "Fork" incurs a total cost of about two or three re-scheduling decisions. So too small a computation should not be forked into a separate thread.
One useful measure of a threads implementation on a multi-processor is the smallest computation for which it is profitable to fork a thread.

Using alert: diverting the flow of control
The purpose of alerts is to cause termination of a long running computation or a longterm wait.
The problem with alerts or any other form of asynchronous interrupt mechanism, is that they are, by their very nature, intrusive. A straightforward-looking flow of control in one thread can suddenly be diverted because of an action initiated by another thread. Advantage of using alerts is that they are a single unified scheme for provoking thread termination.
The alternatives to using alerts is by setting a boolean flag and signalling the condition variable.

Additional techniques

Up-calls
In this methodology, a pool of threads is maintained that is willing to receive incoming data link layer (e.g. ethernet) packets. The receiving thread dispatches on ethernet protocol type and calls *up* to the network layer (e.g. DECnet or IP), where it dispatches again and calls up to the transport layer (e.g. TCP), where there is a final dispatch to the appropriate connection. In some systems, this up-call paradigm extends into the application. The attraction here is high performance: there are no unnecessary context switches. All the top-performing network implementations are structured this way.
Downside is performance ,also synchronization problem

Version stamps
Sometimes concurrency can make it more difficult to use cached information. This can happen when a separate thread executing at a lower level in system invalidates some information known to a thread currently executing at a higher level. In the low level abstraction a counter is maintained that is associated with the true data. Whenever the data changes, the counteris incremented. Whenever a copy of some of the data is issued to a higher level, it is accompanied by the current value of the counter. If higher-level code is caching the data, it caches the associated counter value. Whenever a call back down to the lower level is made, and the call or its parameters depend on previously obtained data, the associated value of the counter is included. When the low level receives such a call, it compares the incoming value of the counter with the current truth-value. If they are different it returns an exception to the higher level, which then knows to re-consider its call. This technique is also useful when maintaining cached data across a distributed system.

Work crews
When thereis more concurrency than can be efficiently accommodated on a machine, an abstraction can be used to control the forking The basic idea is to enqueue requests for asynchronous activity and have a fixed pool of threads that perform the requests. The complexity comes in managing the requests, synchronizing between them, and coordinating the results. Another method is to implement "Fork" in such a way that it defers actually creating the new thread until there is a processor available to run it-- "lazy forking".

Reasons for using thread management:

- Scalability: More requests are serviced with no change in resources.
- Safety: Application is protected from reaching the maximum allocated memory
- Effiency: Make good use of the available computer resources, and therefore produces its answer quickly.
- Thread management helps to have controlled and known variants, resolves issues of the unknown: the number of threads running in the system, memory required for thread stack is known.
- In comparison to creating a process it takes less time to create, less time to switch, to terminate and more intuitive to implementation of concurrency.

Advantages:

- Only a specific number of threads are allowed to run in the system.
- Memory for total thread stack size is known and constant.
- Fixed and known number of threads contend for execution time per CPU
- Thread objects need not be constructed during runtime.

Disadvantages:

- Additional object synchronization is required.
- Working thread may block on a specific task forever, this might reduce the effective working threads by a significant number, one/pool size. A maintenance thread is required to check for blocked threads and assumptions must be made on expected execution timing.

Implementation:

As regarding our project, we developed a thread management class and in order to meet the requirement , the class was developed in Unix environment using c++.

Operations of thread management:

int CreateRootThread (string);
//Creates parent thread

int CreateThread(int,string);
//Creates child thread

int ThreadTerminate();
//Terminates the thread and cleans up memory

int GetThreadID(string)
//Returns thread id of the given named thread

string GetThreadName(int threadId)
//Returns the name of a thread

int GetThreadID ();
//Returns the thread ID

int GetNoOfThread ();
//Returns the no of threads

int SendMessage (int , int, string);
//Sends message from one thread to another

int StartThread(int)
//Starts a thread's work

int PauseThread(int)
//Pauses a thread that is working

void ResumeThread(int)
//Resumes a thread's work that was paused

int StopThread(int)
//Stops thread's work

int ReturnNoOfThreads()
//Returns the current no of threads

void PrintAllThreads(ostream&)
//Prints thread information for all threads

int PrintThreadInfo(ostream&, int)
//Prints information about a thread

void PrintThreadHeader(ostream&)
//Prints a header for printing thread info

void PrintOneThread(ostream&,string, int,int,string)
//Prints the information for one thread

bool IdInRange(int)
//Determines if the thread is in the valid range

bool IsActive(int)
//Determines if thread is active or not

Application :

The code was written to provide a framework for an efficient implementation of the thread management class. Currently, we have laid the groundwork for such an implementation. The user based on his/her needs could further enhance this.

Project:

Problems:

The original project was implemented inside Microsoft windows As we chose to do it in C++ we had to work on the code from scratch.We have tried to incorporate the functionalities of the thread and the thread manager class but due to time constraint we were only able to design a decent framework which is easy to understand and easy to enhance.We would have like to implement priority other that FIFO which would have led to further complications for which again time constraint was a big issue.

Learning experience:

Creation and managing of threads , priority scheduling and messaging between threads.
Threads are required to run in a given memory allocation and CPU time .The CPU time is determined by various methods e.g. threads could be prioritized based on priority levels ,First come First serve. Our choice was first come first serve . This choice was based on couple of factors-one that we were time restrained and secondly in terms of our requirements it suited our purpose to effectively show how multiple threads can be effectively run. thereby giving a degree of control as needed.
Threads might share data and this could be a problem, as threads usually are not concerned with each other. Data shared between many threads is called Critical section data. . To avoid conflict one hread will not be able to access that data while another thread is using it e.g. semaphore, mutex

References:

ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-035.pdf

http://www.cs.wustl.edu/~schmidt/UCLA.html

Attachments:

Source code

Output