Multithreading Implementations

Article · September 1998

Source: CiteSeer

1 author:

Krishna Kavi
University of North Texas
221 PUBLICATIONS **1,765** CITATIONS

SEE PROFILE

Multithreading Implementations

Krishna Kavi
The University of Texas at Arlington

This research is supported in part by a grant from NSF, MIP 9622593.

Multithreading Implementations

Krishna Kavi
The University of Texas at Arlington

Abstract

A thread is a single sequential flow of control. A multithreaded program allows multiple threads of control within a process. Multithreading was developed because of the unsuitability of the process model as a unit of execution (i.e., a thread of control) for concurrent applications consisting of multiple independent tasks which had to interact with each other. Threads as we know them today were first introduced in the early 1980's in several microkernel based research operating systems. Today almost all modern operating systems offer kernel support for multithreading. In addition, a large number of user level multithreading libraries have also been developed. These user level threads can be used even on systems without kernel thread support

This paper describes some of the more important implementations of multithreading. Both commercial and research multithreading packages are described. The strengths and weaknesses of each implementation are presented. In addition the problems which arise in multiprocessor implementations are also discussed.

I. Introduction

A traditional process is essentially an address space with a single point of control and associated state information and resources. Traditionally a process was considered the atomic unit of control flow in a system and was designed to be an independent, self contained unit of computation. As a result a process was unsuited for applications which consisted of multiple cooperating tasks interacting with each other. The following section describes the weaknesses of the process model which led to the development of threads.

A. Motivation for the development of threads

The concurrent tasks described above did not map well into processes for several reasons. They include,

Process management costs: Processes have associated with them a large amount of state information such as file descriptors, virtual address memory maps, accounting information, etc. In operating systems such as UNIX, process creation involves copying the entire address space of the parent. Even in other operating system environments such as the Win32 subsystem of Windows NT[CUST 93], which does not duplicate address spaces during process creation, there still is a lot of overhead due to the need for initialization of the process. This makes process creation and management very expensive.

Cooperative tasks in a concurrent application, unlike processes did not need to be self contained. In other words these tasks did not need all the state information and resources such as address translation maps, file descriptors, working directory, etc., which are associated with every process. Using a process to implement these tasks this made processes "heavy", imposing unnecessary costs in task management.

Process interaction costs: Cooperating tasks in an application generally do not require address space protection from each other. Cross address space protection prevents processes

including those belonging to the same application from easily sharing memory. While some operating systems do allow processes to share memory, such mechanisms are generally cumbersome and require some form of kernel mediation to set up the shared memory segments and to synchronize access to these segments[VAHA 96]. Synchronization between processes requires system level synchronization primitives. Creating and using these synchronization primitives involves invoking the kernel at each operation. Every kernel invocation involves a system call. System calls are expensive because each argument passed through a system call has to be copied into kernel space and then checked by the kernel to ensure that they are valid before the system call can be executed. Therefore systems calls are expensive operations which tend to degrade performance if used frequently.

Resource costs: Processes use kernel resources which are always limited. The kernel is the core of the operating system. In most operating systems, the kernel is permanently resident in physical memory and is never swapped out to disk. Hence the memory which is available to the kernel itself is always limited by the physical memory available. Every process requires some kernel memory which is used by the kernel for process management. Moreover many data structures which are used by the kernel to control processes such as the process table are fixed length arrays which cannot be dynamically extended. Hence the number of processes that can be active at any given time is limited. This does not allow applications to make use of a large number of processes as threads of control even if performance is not an issue.

Processes have associated with them a large number of resources such as a virtual address space, a root directory, a working directory, a user identifier, accounting information, signal masks etc. Tasks in the same application can share all of these resources and it is unnecessary to have a separate instance of each resource for each task. However using separate processes for each task resulted in unnecessary duplication and waste of resources

B. Threads: The solution

The notion of a thread evolved in response to these problems. The term thread in general refers to a single path of execution or control flow. In a multithreaded process, multiple threads share the same address space. Memory sharing and synchronization is simplified because these threads exist within the same address apace. Much of the state associated with a process can be shared by the threads, making threads lightweight and relatively inexpensive to manage.Each thread also has the same communication facilities as a separate process. Each thread can independently communicate with other processes through normal IPC mechanisms such as semaphores, sockets and pipes. A thread has associated with it only as much state information and resources as is required for it to function as a thread of control. The minimal components associated with a thread are a thread control block and a user level stack. The thread control block stores the context of the thread when it is not executing as well as other thread management information.

Hence threads can be created and managed with very little overhead and threads in the same process can share data without requiring kernel intervention. Threads generally share file descriptors, kernel process data structures, signal masks, virtual address translation maps, the root directory and working directory. Therefore threads make better use of system resources and eliminate unnecessary duplication. Threads provide a very natural model for programming multiple flows of control within a process. Although a similar effect could be obtained with a single threaded process, the resulting control flow would be very complex to develop and maintain. Threads are very useful as a structuring mechanism to cleanly program independent tasks within a single application. Operating system supported threads allow the overlap of computation and I/O on both uniprocessors and multiprocessors producing significant performance benefits.

A threaded program can exhibit improved responsiveness by time slicing of its threads even if the application is mostly compute-bound. In addition, by using priority based scheduling, threaded programs can respond to intermittent events which require real time response. This is especially important in graphical user interfaces where program responsiveness to user actions is important.

A traditional process can only run on a single processor. Threaded programs which use kernel threads can automatically make use of multiple processors if they are available in a transparent manner. Hence the same executable can be used for both uniprocessors and multiprocessors if the same operating system supports both types of machines. For example, a multithreaded program executable written for the Solaris 2.x operating system runs on both uniprocessors and symmetrical multiprocessors running the Solaris 2.x operating system and transparently uses all available processors[EYKH 92, POWE 91, STEI 92].

Over the years several classes of multithreading models have been developed. No single model of multithreading is suitable for all applications. The choice of a particular model directly impacts the performance gains that can be obtained with a multithreaded application. Hence a brief introduction to the various models of multithreading is presented in the following subsection.

The following subsections provide background information on multithreading. Section II examines several thread packages, both commercial and research implementations and discusses their strengths and weaknesses. Finally, section III presents the conclusions of this study.

C. History

Contrary to popular belief, threads are not a new concept. The notion of a thread has been around atleast since 1965 when the Berkeley Timesharing system offered a mechanism similar to threads[comp.os.research FAQ 96]. On this system all resources (including memory) were protected on a user by user basis. Each user had 128K words available where each word consisted of 24 bits. A thread in that system was known as a “process”. Each “process” (i.e. thread) could address 16K words and could map arbitrarily anywhere within the users address space. Moreover any number of threads could be created. Each thread could modify its memory map to point to any region in the user’s address space. Thus by using appropriate mappings, threads could share memory. The threads were also lightweight: their state required only 15 words for storage. These threads were visible to the kernel and could be independently scheduled by the kernel.

Around 1970, a form of multithreading involving multiple stacks within a single process was implemented on Multics for supporting background compilations[comp.os.research FAQ 96]. Threads as they are known today first appeared in the early 1980s in research microkernel based systems such as the V kernel, Chorus, RIG, etc.[comp.os.research FAQ 96]. Commercial versions of multithreaded operating systems were developed atleast as early as 1983. The VAX ELN, a real time operating system from DEC, supported multithreaded processes around 1983[CUST 93].

Today, most modern operating systems support some form of multithreading. In addition, user level thread libraries are available for older systems which do not provide kernel support for multithreading. While thread packages differ widely today in their functionality and interface they share many attributes. The following subsection attempts to classify various thread packages based on various criteria.

D. Classification

Threads can be classified based on various criteria. Some of them are,.

1. Implementation Model

The level of implementation refers to the level of operating system support offered for multithreading. This is the most common classification of threads and thread packages. The level of implementation determines whether such threads are visible to the kernel and whether they can be independently scheduled on a single or multiple processors.

User level threads - These are threads which are created and managed entirely at the user level. They are primarily used as a lightweight program structuring tool. The advantages of such threads are that they are very inexpensive to create and manage. Hence a large number of such threads can be used in an application and very fine grain concurrency can be exploited. Moreover these threads are invisible outside the process that created them and the application has greater control over their scheduling.

However, since the kernel is unaware of the existence of such threads, when one user level thread blocks in the kernel, the entire process blocks since the kernel considers the process as a single threaded entity and cannot schedule user level threads independently. A large number of such thread packages are available. One such thread package is Provenzano's Pthreads package which implements a portable POSIX standard thread interface. Many other thread packages are available because of the relative ease with which such thread packages can be developed and integrated onto existing systems.

Kernel level threads - Kernel threads are essentially lightweight processes which share the same address space if they belong to the same process. Kernel threads generally share most of their state information with other threads in the same process. Hence their creation and management is less expensive then that of a process. Kernel threads in a process can be scheduled independently. Hence a blocking system call by one thread does not block all the other threads in the process. Moreover kernel threads, unlike user level threads can be scheduled independently on multiple processors when available.

However, most of the operations on kernel threads(creation, synchronization, stopping, etc.,) require kernel intervention. Hence kernel thread management is quite expensive compared to user level threads. Moreover, each kernel thread requires some kernel resources. This limits the number of threads that can be created in a system. The relatively high cost of kernel threads allows only coarse grain concurrency to be exploited with such threads. Several commercial operating systems offer kernel level threads including Windows NT, Digital UNIX 3.x and Linux 2.x.

Hybrid thread model - Thread packages in this category multiplex one or more user level threads on one or more kernel level threads in a process. Each process may have multiple kernel threads which may be scheduled independently by the kernel. A user level threads library multiplexes one or more user level threads on top of each kernel level thread. This model combines the advantages of both user level threads and kernel level threads.

In hybrid systems scheduling occurs at two distinct levels. At the user level, a user level threads library schedules the user threads while the kernel schedules the kernel level threads and neither scheduler is aware of the scheduling decisions of the other. Hence scheduling conflicts can arise which can degrade performance. A modification of the hybrid

thread model, known as scheduler activations has been proposed by Anderson et. al. [FEEL 93]. However no commercial operating system currently offers this facility[FEEL 93].

2. Scheduling model

Threads can also be classified based on the scheduling policy. While not commonly used to classify thread packages, this classification is useful when considering the performance aspects of multithreading

Nonpreemptive scheduling - In nonpreemptive scheduling a thread runs until it either blocks for a resource or voluntarily gives up the processor. Such threads are also called coroutines. This type of scheduling is not practical at the kernel level as an ill behaved application could refuse to yield the processor to other applications. However this model can be implemented at the user level as threads in an application are generally cooperative. This is a very simple scheduling model with excellent performance because there is very little scheduling overhead.

Another advantage of this model is that it reduces the dependence on locks in a uniprocessor as a thread knows when it is giving up control of the processor. The reduced use of locks in turn results in reduced overhead. The disadvantage of this approach is that it is not possible to implement preemptive priorities for threads. Hence real time applications and GUIs cannot use this model. Moreover since timeslicing is not available, improved program responsiveness through timeslicing also cannot be implemented. An example of this approach can be seen in the Windows 3.x operating system (strictly speaking a GUI). All Windows 3.x applications in a system run as non preemptive threads within a single process[CUST 93].

Preemptive priority non time sliced scheduling - In this approach the highest priority runnable thread runs until it either voluntarily yields the processor, or blocks for a resource or is preempted by a higher priority thread. The priorities of the threads are generally fixed. This approach is followed by a number of user level thread packages such as the Solaris 2.x user level thread library[STEI 92]. This approach is well suited for user level thread packages. Real time threads and GUI threads which wait for user input can use high priorities to ensure a prompt response. However this policy is not appropriate for kernel level threads as this policy is unfair at best and may even cause starvation of low priority threads in the system.

Preemptive time sliced (Round robin) scheduling - Here threads are allotted a time slice and a thread runs until it either blocks for a resource, yields the processor or uses up its time slice. All threads have equal priority. Threads may be chosen for execution either on a FIFO or LIFO basis. However using a LIFO strategy may lead to starvation. This is not very useful for user level thread packages as it can result in unnecessary context switching of threads and the absence of priorities means that GUI and real time applications cannot use such threads. Since no priorities are implemented, this policy cannot be used for kernel level threads either. In practice this scheduling mechanism by itself is not commonly used for general purpose operating systems, but certain operating systems use them with other scheduling policies. For instance, Digital UNIX has a scheduling class called SCHED_RR which has similar functionality and is one of three scheduling classes available on the system.

Preemptive priority time sliced scheduling - This scheduling algorithm is commonly used for kernel level threads. It is fair and prevents low priority processes from starvation. Each process is associated with a priority and there is generally a separate run queue associated with each priority. The priority of a process is increased or decreased depending on

how much CPU time it has already obtained. Threads with the same priority are preemptively time sliced as above. Higher priority threads can preempt lower priority threads and can run beyond their time slice if no thread of equal or higher priority is runnable. While this could also be used for user level threads, the scheduling algorithm is complex because priorities are constantly recomputed. It is generally not advisable to use a complex scheduler for user level threads because this can add unnecessary overhead to context switches and overwhelm any benefit gained from using user level threads. Since user level threads belong to the same process, fairness is not generally an issue and hence the overhead imposed by this algorithm is not justified for user level thread libraries. This scheduling mechanism is most commonly used for kernel level threads. Windows NT, Digital UNIX and Solaris use such scheduling algorithms for scheduling kernel threads.

3. Programming interface

The programming interface is the interface provided to the developer of multithreaded applications. Thread interfaces can be broadly categorized into two classes, those similar to the POSIX standard interface and those closely related to the Win32 thread interface. The former are known as POSIX style threads and includes most thread packages offered by UNIX variants. The latter are known as Microsoft style threads and include Win32 threads and OS/2 threads. Some of these thread interfaces are described below.

POSIX - POSIX standard 1003.1c was established in 1995. It specifies a portable thread programming interface without specifying the underlying implementation. Hence there are user level thread packages (Provenzano's Pthreads[1]), kernel level threads (LinuxThreads) as well as hybrid thread packages (Solaris threads) which provide this thread interface.

UNIX International threads - This interface which is also known as the Solaris thread interface and is offered on Solaris 2.x from Sun Microsystems and UNIX Ware 2 from SCO. This interface closely resembles the POSIX interface.

DCE threads - This interface actually corresponds to Draft 4 of the POSIX standard and is quite similar to the POSIX standard. Several operating systems such as Digital UNIX 3.2x, HP/UX from HP and AIX from IBM provide this interface. Moreover kits which implement this interface on top of native Win32 threads on Windows NT are available[comp.programming.threads FAQ 96].

Win32 threads - This interface is available on Windows NT and Windows 95 and is quite different from the POSIX interface. Currently threads on the Windows platform are kernel level threads.

OS/2 threads - These threads were initially developed by Microsoft, but have since been reimplemented by IBM. This interface resembles Win32 threads to some extent.

Java threads - Java is a programming language which has integrated support for multithreading. The interface provided by Java is quite different from the POSIX interface and the Win32 interface.

[1] Provenzano's Pthreads is still undergoing development and does not fully comply with the POSIX Pthreads standard yet.

4. Multiplexing model

Threads can be classified based on their multiplexing model which describes all threads based on an abstract two level model. In other words all thread packages are assumed to consist of two levels of thread-like entities and the models differ in the manner in which the upper level threads are multiplexed on the lower level threads. The upper level threads and lower level threads correspond respectively to the user level threads and the kernel level threads of the implementation model.

Many to one model: In this model many upper level threads or user threads are multiplexed on one lower level thread. All systems which have traditional single threaded processes and user level thread libraries correspond to this model. A commercial example is the SunOS 4.x operating system which has user level threads known as lightweight processes (LWP) multiplexed on top of a traditional single threaded process. These LWPs should not be confused with Solaris 2.x LWPs which are kernel level threads. Other examples are threads in the Java language[BERG 95], various portable thread packages such as Provenzano's Pthread library.

One to One model: This corresponds to a user level thread bound permanently to a kernel level thread. This is essentially a kernel level thread and in all implementations of kernel level threads there is generally no visible distinction between the two levels. Currently many commercial examples of this model exist, such as Win32 threads (Windows NT/95), Digital UNIX 3.x kernel threads, etc.

Many to many model: This is the most flexible model of multithreading. It allows for multiple lower level (kernel level) threads within each process on which multiple user level threads can be multiplexed. The hybrid threads implementation model can be considered to be a variant of the many to many model in that the hybrid model allows both the one to one model and the many to many model to coexist in the same process. The Solaris 2.x thread model can therefore be considered as a variation of the many to many model[STEI 92].

II. Implementations

This section describes several common implementations. The thread packages were chosen based on how widely they are used. Hence most of the emphasis is on commercial implementations. However research implementations such as Mach and its C-threads package, scheduler activations and Opal threads are also discussed.

A. Solaris threads

The Solaris thread implementation belongs to the hybrid category, i.e. it offers both user level threads and kernel level threads. The Solaris thread model can also be considered a variation of the many to many model. The Solaris thread model offers three types of thread objects, kernel threads, lightweight processes (LWPs) and user level threads.

Kernel threads - This is the most fundamental thread entity in Solaris. Each kernel thread can be scheduled independently by the kernel. A kernel thread need not have a corresponding process. In fact, kernel threads are used in Solaris to perform many kernel level functions, and the kernel itself consists of a set of kernel threads[VAHA 96]. Kernel threads which perform internal kernel functions do not have a corresponding process and share the same address space which makes context switching between such kernel threads inexpensive.

A kernel thread essentially consists of a thread data structure and a stack. The thread data structure contains[VAHA 96],

1. Contents of the kernel threads register context (when the thread is blocked).
2. Priority and scheduling information.
3. Pointers for putting the thread in scheduler queues and queues for resources.
4. Pointer to the thread's stack.
5. Pointers to the associated LWP if the kernel thread has an associated LWP else this contains NULL.
6. Pointers to the queue of all threads in the system. Also pointer to the queue of all threads in the associated process if the kernel thread has a process associated with it.
7. Information about the associated LWP if there is an associated LWP.

Bare kernel threads are not exported outside the kernel. The user interface for using kernel threads are the LWPs. In other words, all LWPs are built on top of kernel threads. Confusingly, the LWPs of Solaris terminology corresponds to the kernel thread of the implementation model described in a previous section.

Lightweight processes (LWPs) - These are actually kernel level thread wrappers which are exported by the kernel. Hence, LWPs can be scheduled independently by the kernel and can transparently make use of multiple processors when available. The thread model in Solaris is a variant of the many to many model. Hence every LWP has a user level thread associated with it either temporarily or permanently bound to it. LWPs by themselves cannot run user level code. They have to be associated with a user level thread. LWPs are invisible to processes other than the one which contains it.

Traditionally a UNIX single threaded process has two important kernel level per process data structures, the *proc* structure and user area (also known as *u* area)[VAHA 96]. In fact the process table in the UNIX kernel is actually an array of proc structures with one *proc* structure per process in the system. The *proc* structure generally contains process information which is required even when the process is not running. The *proc* structure is never paged out to disk. The *u* area on the other hand contains information which is required only when the process is running. The *u* area can be paged out to disk.

These traditional UNIX per-process data structures do not map well to multithreaded processes where kernel data structures can be per process or per LWP. In Solaris[EYKH 92, POWE 91], all process specific and LWP independent information is grouped into the *proc* data structure. Thus the Solaris proc structure corresponds to the traditional *proc* structure and the LWP independent part of the traditional *u* area. As before the *proc* data structure cannot be paged out.

The per LWP information is stored in a new per LWP data structure called the *lwp* structure[POWE 91]. The *lwp* structure can be paged out. The *lwp* structure contains,[VAHA 96]

1. The user level register context of the user level thread associated with the LWP if the LWP is not running.
2. Signal handling information.
3. System call arguments, results and error code
4. Accounting information and profiling data
5. Virtual time alarms
6. User time usage and CPU usage
7. Pointer to the kernel thread associated with the LWP

8. Pointer to the associated *proc* structure.

User level threads - These entities actually run the user level code. In order for a user level thread to run it must have an associated LWP. The binding may be temporary or permanent. They are managed by a user level thread library which multiplexes them on top of LWPs. When a multithreaded process begins, the thread library creates a pool of LWPs. The size of the pool is a default which depends on the number of user level threads and processors.

However there is a library call, *thr_setconcurrency(int n)*, which allows an application to set the number of LWPs. The thread library does not automatically create the number of LWPs requested by the application. It only creates LWPs if there are user threads which can run on them. Thus as long as the number of user threads is less than or equal to n, the number of LWPs will be equal to the number of user level threads. However, when the number of user level threads exceeds n, the number of LWPs stays fixed at n. The thread library automatically destroys LWPs if they are idle for 5 minutes. Hence if the number of user level threads falls below n, the number of LWPs will also fall until the number of LWPs equals the number of user level threads.

Each user level thread also has state which must be kept separate from LWP state as threads and LWPs have separate identities. Each thread has a user level stack and thread data structure associated with it.

The thread data structure contains[VAHA 96],

1. Thread's ID
2. The saved register context of the user level thread (when the thread is not running)
3. The thread signal mask
4. Priority and scheduling information
5. Thread specific private data - This is used to support reentrant versions of C libraries.

Signal management - All LWPs in a process share a common set of signal handlers. Each LWP may have its own signal mask so that it can block unwanted signals. An LWP may also specify its own alternate stack for handling signals. Traps which are synchronous signals produced by the action of an LWP are delivered to the LWP that produced them. Interrupt signals (which are asynchronous) can be delivered to any LWP that has not masked the signal. Moreover since LWPs are invisible outside the process, an LWP in one process cannot send a signal to a specific LWP in another process.

Fork semantics - One of the issues with multithreaded processes relates to the behavior of the fork system which creates a child process. There are two possible behaviors for the fork system call. Either the entire process with all its threads may be duplicated or else only the process and the calling thread may be duplicated. In Solaris, the first behavior is exhibited by the fork() system call while the second behavior is exhibited by the fork1() system call. The fork1() system call is useful if the child process is not meant to be a duplicate of the parent process and an exec type system call follows the fork system call.

Multiprocessing

Solaris provides a mutual exclusion mechanism known as adaptive lock which is useful in multiprocessing. An adaptive lock is based on the premise that if a thread holding a lock is active it is likely to release the lock soon. On the other hand if the thread is blocked it is not likely that the lock will be released quickly. Hence in adaptive mutex, if a thread attempts to lock a mutex held by an active thread, the unsuccessful thread spins on the lock. If on the other hand, the thread holding the adaptive lock is blocked, then the thread trying to lock the mutex also blocks. On uniprocessors the unsuccessful thread must always block because if it spins it will impede the execution of the thread holding the lock.

B. Windows NT threads

Windows NT has a process and thread model quite different from Solaris and indeed all UNIX variants. The Windows NT architecture was partly inspired by the Mach operating system. It uses a client server architecture in that operating system services are structured as a number of user mode servers. These services can then be obtained by clients (either applications or other operating system components or servers) by passing messages to the server. Message passing between servers is handled by the microkernel a small minimal privileged mode component of the operating system. The microkernel also implements some very fundamental operating system services such as process creation, management and scheduling, virtual memory and message passing.. The advantage of this architecture is that the kernel offers services which any operating system is minimally required to provide. Any additional services can be provided by optional user mode servers. This makes the operating system efficient on any hardware. Only those services which are required need be added to the operating system.

It is frequently assumed that the most common environment provided by Windows NT, i.e., the Win32 subsystem is the only environment supported by Windows NT. However this not true. The Windows NT operating system is capable of providing environments ranging from DOS and Windows 3.x to POSIX and OS/2 in addition to the Win32 environment. In fact, Windows NT can provide support for any environment likely to come up in the near future. This is because all these environments can be implemented as user level servers (also know as protected subsystems). All these subsystems use the facilities provided by the kernel mode component of the operating system which is called the NT executive. The NT executive provides services like,

1. Virtual memory management
2. Resource management
3. I/O and file system management
4. IPC

Although NT is not an object oriented operating system, it treats system resources including native processes and threads as objects to provide a uniform object based approach to resource management. Here the term native is used to distinguish the process objects created by the NT executive from the process abstraction provided by the NT user level environment subsystems. The user level server subsystems (POSIX, OS/2, Win32, etc.,) use the native process objects to emulate processes belonging to those environments.

Processes in NT - Native NT processes differ from processes in UNIX like operating systems in three major ways,

1. Processes and threads are treated as objects
2. There is no parent child relationship between processes.
3. Processes and threads have integrated synchronization capabilities.

Since Windows NT treats system resources as objects it has an object manager which is responsible for creating all objects. All operations involving objects must involve the object manager at some point or the other to allow centralized control of system resources. Although all objects are created by the object manager, their creation is generally requested by some component (which in this paper will be called object creator for simplicity although strictly it is not correct) of the executive. In the case of processes this component is the process manager. Every object has two parts, the object header and the object body. The object header contains attributes which every object has, such as an object name, a security descriptor (which describes which objects may access it and the types of access allowed), accounting information, a list of processes which have handles (i.e. references) to that object, etc. This information is used and controlled by the object manager. The object creator controls and uses the object body. The object body constitutes the variable component of the object. The creator supplies the services (or methods) associated with the object body.

Some of the object body attributes associated with the process object are[CUST 93],

1. Process ID
2. Base priority - This is the reference priority about which a thread's priority can vary in a limited range. This enables a priority variation among the threads in the process. At the same time having a different base priority between processes enables a form of priority to be established between processes as well.
3. Default processor affinity - The set of processors on the system on which the process' threads can run.
4. Quota limits - Memory (paged and non paged), paging space (or swap space) and processor time.
5. Execution time - Total execution time of all threads of process.
6. Miscellaneous accounting information
7. Exception/debugging ports - IPC channels on which messages about exceptions due to one of the process' threads are sent. For instance, a separate debugger process can monitor the exception port of a client process it is debugging and know when a breakpoint is reached.
8. Exit status

The process manager provides various services to create, manage and destroy processes as well as obtain process information. As stated earlier, Windows NT allows various environments to be emulated. This makes native NT process creation and management difficult because of the need to support different notions of process creation and management. For instance, the Windows Win32 environment does not have a parent child relationship between processes, while the POSIX and OS/2 subsystems maintain such relationships. To overcome these problems the Windows NT process manager does not regulate how processes and their relationships should be structured and allows maximum flexibility. For instance it allows the process requesting creation of another process to optionally specify itself or a third process as parent. This is used by the POSIX subsystem to emulate fork semantics. In Win32 and OS/2 on the other hand , a new process does not inherit a parent's address space. Rather they specify an executable image to be loaded into the new process's address space. Hence in NT, the creating process can also request that the new process inherit the parent's address space.

Threads in NT

Another difference between Windows NT and some operating systems with user or kernel level threads is that, in Windows NT a process is essentially a static object. In other

words, a process is by default not associated with a thread of execution. A thread has to be separately created before a process can execute.

An object of type thread has like all objects,. an object header and body. The thread object body attributes and services are defined by the process manager. Some of the thread attributes are[CUST 93],

1. Client ID - A unique Id associated with each thread in the system. This is used when a thread invokes a server in the system.
2. Thread context - Contains the thread execution context (when the thread is not running)
3. Dynamic priority - A thread's actual execution priority at any given point of time.
4. Base priority - The minimum value which the threads dynamic priority can take.
5. Thread processor affinity - A nonproper subset of the corresponding process' processor affinity. Thus a thread can run on some or all of a process's processors depending on the thread's processor affinity.
6. Thread execution time
7. Termination port - The IPC channel on which a message is sent if the thread terminates. This is used by the environment subsystems to know about the termination of a thread in a process.
8. Thread exit status.

A thread's base priority can take on any one value in a range between two less than and two more than the base priority of a process. The thread's dynamic priority on the other hand can vary from the threads base priority to the maximum priority possible, which is 15 for variable priority threads.

Synchronization in Windows NT - In NT since most system resources are objects, synchronization is integrated into those objects which are capable of synchronization. Thus, unlike Solaris and many UNIX variants, two threads which need to synchronize need not use a separate synchronization object. However, these features are available only with native NT threads and a subsystem (like the POSIX subsystem) may choose to hide these capabilities. Objects which have synchronization facilities are called synchronization objects in NT. These include,

1. Process objects
2. Thread objects
3. File objects
4. Event objects
5. Event pair objects
6. Semaphore objects
7. Timer objects
8. Mutant objects

Objects four through eight have synchronization as their only function. Process, threads and files obviously have other functions as well. A synchronization object in NT can be in one of two states, signaled or nonsignaled. What characterizes a signaled state depends on the type of object. In the case of a process, it goes into the signaled state when the last thread in the process terminates. A thread goes into the signaled state when it terminates. A timer goes into signaled state when it "expires", i.e. when the time it is set to expire arrives.

A thread synchronizes with another object by calling a wait routine and specifying the object on which it wants to wait. Solaris and POSIX compatible thread packages supply the option of waking up either one (for example the cond_signal() call) or all threads (the cond_broadcast()) waiting on a synchronization object. In NT on the other hand, a signaled state may or may not have broadcast semantics depending on the type of object. For instance

when a thread terminates all waiting threads are woken up. On the other hand when a mutant object (a NT native object used among other things to implement mutexes in the Win32 subsystem) is set to the signaled state (which occurs when it is released by its current owner), only one of the waiting threads is woken up.

Thread scheduling - Windows NT has two types of native objects, executive objects and kernel objects. Executive objects are those which are created by an executive component and can be exported to the user level. User level code can access these objects by calling native NT services. Kernel objects on the other hand are created by the kernel and are restricted to kernel mode use i.e. they can only be used by the executive. Thread scheduling is the function of the kernel. The kernel does not manipulate the thread objects which are exported to user level code. Rather it schedules the kernel thread object. The kernel thread object is a kernel object on top of which the user level exportable thread object is built. The latter is known as the executive thread object. Similar terminology applies to process objects. Since the kernel is a low level entity, it avoids high level constructs such as handles and uses simple pointers instead. The kernel process object has a pointer to a linked list of all kernel thread objects associated with the process.

A kernel thread object can be one of six states,
1. Ready - The thread is waiting for a processor to run.
2. Standby - Thread which has been selected to run next on a specified processor.
3. Running - A thread running on a processor.
4. Waiting - A thread waiting on some resource or just suspended.
5. Transition - A thread which is ready for execution but is waiting for some resource necessary for execution before it enters the ready state.
6. Terminated. - A terminated thread has finished its execution but has not yet been deleted. Such a thread can be reinitialized and restarted.

As stated earlier, every process has a base priority. A thread obtains a base priority when it is created which it can modify about a small range before it starts executing for the first time. This forms the base priority of the thread. The threads actual or dynamic priority can vary above (but not below) the base priority upto a maximum of 15. Variable native priorities in NT vary from 0 to 15. The Windows NT executive has 32 priority levels (these are executive priorities which may or may not be exported by subsystems to the user). These priority levels are divided into two scheduling classes, variable priority and real time classes. The real-time class has priorities from 16 to 31, while priorities 1 through 15 are used by variable priority threads. Priority 0 is reserved for use by the system.

The kernel scheduler is a preemptive priority timesliced scheduler. The scheduler assigns each thread a time quantum. Variable priority threads have their priorities adjusted depending on the amount of CPU usage of the threads. One scheduling policy is to give a high priority to interactive threads and low priority to compute bound threads and intermediate priority to I/O bound threads. The scheduler allots processors to threads starting from the highest priority level and then works its way downwards. Within each level, threads are scheduled in round robin fashion. A thread which is runnable and has an appropriate priority may still be passed over for a lower priority thread if none of the processors available belongs to the thread's processor affinity. A thread which serves its full quantum is placed at the back of the queue for the appropriate priority level. However a thread which has been preempted before the end of its quantum is placed at the front of its priority level's queue. The kernel in Windows NT is interruptible but is nonpreemptible. Thus threads running kernel code cannot be preempted nor is the kernel ever paged out and is permanently resident in memory.

Native NT threads and Multiprocessors - Windows NT is a symmetric multiprocessing operating system. Hence operating system code can run on more than one processor which

concurrently access global operating system data structures in shared memory. To prevent the integrity of these shared data structures from being violated, the kernel provides synchronization mechanisms which are used by the kernel itself as well as the executive.

The kernel code includes critical sections such as those which update global kernel data structures, which must be executed at any given by only one processor. On a uniprocessor this can be achieved by disabling interrupts as this prevents both preemption and interrupt handlers from running. However this does not work on multiprocessors because disabling interrupts on another processor does not prevent code currently running on the other processor from accessing global data structures. Instead, the kernel uses spin locks which are generally implemented using hardware test and set instructions.. Kernel code must acquire a spin lock before entering critical sections. The spin locks provided by the kernel are not preemptible and the processor keeps spinning on the lock until the lock is acquired. The kernel also exports these spinlocks to the executive through kernel functions.

However spin locks are expensive as they waste CPU cycles and threads spinning on locks cannot be preempted. Moreover there are additional restrictions on the use of these spin locks[CUST 93]. Hence the kernel thread provides other kernel objects to the executive for synchronization. These objects which are called dispatcher objects are essentially the kernel objects on top of which the executive synchronization objects mentioned previously are built. As stated previously, these executive synchronization objects are native NT objects which can be exported by the executive to user mode code. However there is one dispatcher object which is not encapsulated by an executive object. This is the mutex dispatcher object which can be used in kernel mode only.

This section described only the native Windows NT multithreading capabilities. However this interface is seen only by Windows NT system programmers and is comparable to the kernel threads (not LWPs) of Solaris. Most application developers use the Win32 subsystem for application development and hence are only exposed to the Win32 thread interface. The following section describes the Win32 multithreading interface which built using the Windows NT native thread services.

C. Win32 threads

The Win32 subsystem is the most common subsystem used on Windows NT based systems. In addition it is the native environment in Windows 95. This section describes the Win32 multithreading interface.

In the Win32 environment, a process consists of a virtual address space and associated state information and resources. The threads are kernel threads and correspond to the one to one multiplexing model with no user visible distinction between the user level context and kernel level context of the thread. Hence, the thread consists of a thread data structure known as a thread environment block, thread's register context, a user level stack and a kernel level stack. The threads in a system are preemptively scheduled on a system wide basis.

Processes and threads are created by the CreateProcess() and CreateThread() calls respectively. Each process in a system has a priority class associated with it. There are four priority classes, real-time, high, normal and idle. The idle priority class is for processes which run only when system is idle. Each thread in a process has a priority level within the priority class of its process. Every priority class has five priority levels associated with it - lowest l(-2), below normal (-1), normal (0), above normal (1) and highest (2). The default process priority class is normal and the default thread priority level is 0 (normal). Parent processes

can request a specific priority class for the process's while using the CreateProcess() call. The priority of a process can subsequently be examined (GetPriorityClass()) and changed (SetPriorityClass()). Similarly the priority of a thread can be determined and changed by using GetThreadPriority() and SetThreadPriority() respectively. As in native NT threads, the priority level of a thread decays with CPU usage and increases (this is known as a dynamic priority boost in NT) if it has been blocked.

Clearly the priority classes and levels are different numerically from those used by native NT threads. However these do map directly into native NT primitives and priorities. For instance, the priority class corresponds to the base priority of a native NT process. The priority level of a thread corresponds to the base priority of a native NT thread. As with native NT threads there is the concept of dynamic priority which is a thread's actual priority at any given time. The dynamic priority of a thread can never fall below the thread's priority level (which is the thread's base priority.).

Unlike the Pthreads interface, a thread can suspend other threads by calling SuspendThread() and resume them by calling ResumeThread(). In addition threads can be created in the suspended state. Threads can suspend themselves for a certain time period by calling Sleep. The Win32 API provides several synchronization calls. These include wait calls such as WaitForSingleObject() and WaitForMultipleObjects(). The objects on which a thread can wait include mutex objects (not to be confused with NT native mutex objects), event objects, process objects, thread objects, semphore objects, file objects, etc. Synchronization follows native NT semantics with synchronization objects being in either signaled or nonsignaled state. One or all waiting objects are woken up (depending on synchronization object) when the synchronization object goes into the signaled state.

The Win32 API specifies only three objects whose sole function is synchronization. These are the mutex (built using the Windows NT mutant native object), semaphore and event objects. The event object is quite different from synchronization primitives found in other thread implementations. An event object is used to signal the occurrence of an event. There are two types of event objects, manual-reset and auto-reset. A manual-reset event has to be explicitly set to the nonsignaled state after it has been signaled, whereas an auto-reset object automatically returns to the nonsignaled state once one waiting thread has been woken up. The event object by itself has no specific event associated with it. It is merely used as a means to signal the occurrence of an event. Thus, it has to be explicitly set into the signaled state by the thread which wants to specify the occurrence of the event. There are two calls for this, SetEvent() and PulseEvent(). SetEvent() sets the event object to the signaled state and the event object goes into the nonsignaled state only after either 1 thread has woken up (auto-reset) or the object has been explicitly reset by using ResetEvent() i.e. by manual-reset. A PulseEvent() causes an object to go into the signaled state, release all (manual reset) or one (auto-reset) waiting threads and then resets the object into the nonsignaled state. In other words the PulseEvent() attempts to free only threads which are currently waiting.

Threads in a process can also have thread local storage(TLS) i.e. data which is private to a thread. This is useful if a function called by more than one thread needs to return data to the calling thread. Using a global or static variable is not safe in a multithreaded environment. TLS can be used to ensure that each thread gets its own version of the data. TLS is handled through the use of a TLS index. A TLS index can be thought of as representing a data variable with many versions each version specific to one of the threads using the TLS index. A process is guaranteed to have available to it a minimum of 64 TLS indices. A TLS index is created by one thread. Subsequently, each thread allocates dynamic memory for its TLS and then calls TlsSetValue() with the index (say TlsIndexVar) and pointer to the dynamic storage (say TlsDynDataVarPtr) as arguments as follows,

TlsSetValue(TlsIndexVar, TlsDynDataVarPtr);

Subsequently any function can retrieve the data from the TLS by using TlsGetValue() as follows,

TlsGetValue(TlsIndexVar);

The value retrieved by the function will be the value corresponding to the TLS of the thread in whose context the function is executing.

Process and thread termination have semantics somewhat similar to those in Solaris threads. A process exits if any thread calls ExitProcess() or if the primary (i.e. main thread) returns. As in Solaris, the primary or main thread can still exit without causing termination of the process. It can do so by explicitly calling ExitThread(). A process also terminates if the last thread in a process terminates or if some other process calls TerminateProcess() and has the authority to do so. Threads can terminate by explicitly calling ExitThread() or in the case of non-main threads by simply returning. A thread can also be terminated by another thread calling TerminateThread() on it provided it is authorized to do so. As in the case of native Windows NT native threads and processes, Win32 thread objects and process objects are set to the signaled state on termination.

D. Java threads

Java is a new language developed primarily for cross-platform and World Wide Web based applications(applets). Hence the language is designed to be portable and architecture neutral. It compiles to bytecodes which can run on any platform equipped with a Java Virtual Machine. One feature of Java which makes it different from other languages like C and C++ is that it has integrated multithreading. However, the internal characteristics and implementation of multithreading (such as whether the threads are kernel level or user level entities) can vary from platform to platform depending on the multithreading support available. Hence this section describes the constructs provided by the Java language rather than the internals of any particular implementation. However some information on the Solaris implementation is provided.

Java is a pure object oriented language and hence all data and procedures are members of some class. Unlike C++, there are no global functions or data. Naturally, threads are also objects and there is a Thread class in Java. One way in which threads can be created in Java is by subclassing(extending in Java terminology) the Thread class and instantiating objects of the subclass[BERG 95]. The Thread class has to be subclassed because the Thread class does not have much behavior associated with it. The developer has to add behavior by overriding methods of the Thread class. Instantiating a Thread object only creates the object and a thread starts executing only after the start() method of the object is called. The start method in turn calls the run() method (which for the Thread class is a null function). The run method corresponds to the start_routine passed to Pthread_create in thread interfaces like POSIX Pthreads. Thus when the Thread class is extended, the run() method can be overridden appropriately.

One of the major applications of the Java language is in developing applets. To facilitate this Java has an Applet class. Applets are created by extending the applet class. But this creates a problem in using threads because Java does not support multiple inheritance.. Hence a subclass of Applet cannot also extend the Thread class. In Java, multiple inheritance

is replaced by interfaces. An interface in Java is essentially a declaration of constants and method interface declarations and is of the form..

```
public interface WillImplement {
        static final float PI = 3.141;   //Constant declaration
        public abstract void WillDefine();  //Method declaration
}
```

An interface is identical to a class in almost all respects except that it does not provide definitions of methods. A class can then declare that it implements an interface if it provides the method definitions (i.e. body) for all the methods in the interface. Unlike subclassing, many interfaces can be implemented by a class. Threads can thus be used by Applets by implementing the Runnable interface. The Runnable interface has only one method and is declared as,

```
public interface Runnable       {

        public abstract void run();
}
```

Thus if an Applet subclass wants to use threads, it creates thread objects from the Thread class without subclassing it, but passes a reference to an object of a class which implements the Runnable interface. Then the thread object's start method is called as usual. Thus essentially a thread object is given the run() method of some other object whose class implements the Runnable interface. As an example, assume class MeImplementRunnable implements the Runnable interface

```
class MeImplementRunnable implements Runnable     {

        public void run()       {
                //Put all code required to be run by thread here
        }
}
```

Then assume that class WWWApplet needs a thread to do the work in MeImplementRunnable's run method.

```
class WWWApplet extends Applet implements Runnable      {
        public void appletCode(){


                                MeImplementsRunnable threadsWorkObject = new
MeImplementsRunnable();
                Thread thread1 = new Thread(threadsWorkObject);

                thread1.start();
        }
}
```

One of the disadvantages of using the runnable interface is that none of the methods of the Thread class can be used by the thread which uses the interface's run method.

However, this is the only technique available to classes like the subclass of Applet which need to use threads.

As in other implementations, the Java also provides the ability to suspend, resume and kill threads. In Java, these are implemented as methods of the thread class. The suspend() and resume() methods suspend and resume threads respectively, while the stop() method effectively kills the thread. Java also provides the ability for one thread to join with another thread through an overloaded join method in the Thread class. The version invoked depends on the number of arguments passed to the join method. If invoked with no arguments, the join method does not time-out, whereas when a single argument is passed to the join method, that value is used as a time-out value. Java also seems to have borrowed the notion of daemon threads from native Solaris threads. Daemon threads are essentially threads which terminate automatically when all the threads in the process terminate. They are created by calling a setDaemon(true) method on a thread which is then transformed into a daemon thread.

Java also introduces a new concept called thread groups. A thread group is essentially a group of threads on which methods can be called collectively so that thread management is simplified. A thread can be made to belong to a particular group only when it is first created. Thereafter, its thread group cannot be changed. There is a ThreadGroup class with many methods available to manipulate thread groups. However this is not a commonly used feature of Java threads[BERG 95].

Scheduling in Java

Threads in Java are preemptive priority scheduled. There are 10 priority levels, with higher numbers having higher priorities. A thread with a higher priority preempts a thread with a lower priority. Within each level, threads are scheduled on a round robin basis. There may or may not be time slicing of threads depending on the virtual machine implementation and underlying system. Programs can use the yield method of the Thread class to ensure that each thread gets a fair share of CPU time on systems where the threads are not time sliced.

Synchronization in Java

Java does not provide all the features available in thread libraries such as those provided by Solaris or POSIX implementations. However, Java does simplify some tasks such as synchronization. For instance in Java, methods and data to which access needs to be synchronized can be declared as such and the Java runtime ensures transparent use of appropriate synchronization mechanisms.

One such synchronization mechanism is a monitor, which is based on the Xerox Mesa version (The other version of monitors which is the original version is slightly different and is called the Hoare type after its creator T. Hoare). Every class and object can potentially have a monitor associated with it. This monitor is allocated when an object has one or more of its methods declared as synchronized. Similarly, a class's monitor is allocated if a static method is declared synchronized. There is only one object monitor per object and only one class monitor per class. An object monitor ensures that, only one thread is executing in any of the object's synchronized methods. Similar arguments apply to the class monitor. One of the advantages of the language based approach is that monitors are acquired and released transparently and correctly by the runtime system without producing deadlocks. With thread libraries, this is a major source of programming errors. In addition, the monitors can be called recursively without causing deadlock and hence synchronized methods can be also be called recursively.

The synchronized method and monitors discussed above only allow mutual exclusion. Java also has facilities to enable threads to wait for some event. This is provided by the wait(), notify() and notifyall() methods. These methods are associated with every object and can only be called from within synchronized methods i.e., only when a thread is in possession of a monitor. The wait() method causes a thread to relinquish control of the monitor and wait until someone calls the notify() or notifyall() method. When a thread calls the notify() method one of the threads waiting gets control of the monitor after the notifying thread finishes execution of the synchronized method (and implicitly releases the monitor). The notifyall() is the broadcast version of notify which wakes all waiting threads. The wait() call takes an optional time-out parameter also. The monitor together with the wait() and notify() methods plays the same role as a mutex and a condition variable in other implementations like Solaris native threads and POSIX Pthreads implementations.

In addition to synchronizing access to methods, Java allows designating variables as volatile. For such variables, the Java virtual machine always reads from main memory and always writes to main memory. This is essentially used to ensure that all reads and writes to such variables are from a single coherent memory location. This prevents threads from writing such variables to caches or CPU registers without committing the changes to main memory atomically. Reads are thus guaranteed to obtain a coherent committed version of such variables.

Problems with multithreading in Java

One of the problems associated with the Java approach is that it is designed to be portable to many platforms all of which may not have operating system support for multithreading. In addition, Java VM(virtual machine) implementations may or may not use the underlying support for threads. Thus whether or not all the threads map to a single kernel thread or each Java thread maps to a kernel thread is operating system and implementation dependent. For instance, the Java Development Kit 1.02 version on Solaris has a single threaded VM implementation which means that all Java threads are multiplexed on a single native Solaris kernel thread. A similar problem arises with respect to time slicing. Java threads may be time sliced on some machines and not time sliced on others. If a consistent time sliced behavior is desired on all platforms, explicit calls to the yield() method have to be used to yield the CPU.

An application developed using Java threads may show different performance depending on the platform on which it is implemented. While this is probably irrelevant at this time because of the relatively slow execution of Java code compared to compiled C/C++ native code, it may be an issue in the future if Java's speed improves and it is used to develop conventional applications in the same way that C and C++ are used. At that point it may become difficult for developers to determine how best threads should be used (in large numbers as lightweight structuring primitives or in minimal numbers for overlapping I/O and computation), as threads may map differently on different systems.

E. Mach

Mach is a microkernel based operating systems and is based on some of the earliest microkernel and multithreaded operating systems such as RIG and Accent. Mach not only supports kernel threads but also provides a very flexible user level threads library known as C-threads. Mach, one of the earliest multithreaded operating systems, replaced the traditional UNIX process with two entities, a static entity known as a task which consisted of an address

space and associated resources and threads which were the dynamic executing entities. Mach uses a client server architecture, consisting of a microkernel and user level servers which can implement services and environments needed to run programs. Windows NT's client server architecture is conceptually based on Mach's architecture and is partly based on Mach's architecture. Like Windows NT and Solaris, not all kernel supported threads in Mach have a user level context. As in Windows NT, the Mach kernel does not maintain a parent child relationship between process although user level server environments (such as a BSD UNIX server) may choose to maintain them. While in Windows NT, a handle is the minimal requirement to operate on a process, in Mach it is the task identifier which is used to operate on a task.

Every task has associated with it a task structure, which contains[VAHA 96],

1. Pointer to the virtual memory address translation maps.
2. A pointer to the list of threads in the task.
3. Information related to ports - Ports are Mach's abstractions for interprocess communication channels.

In addition to the task data structure of which there is only one per task, there are per thread data structures called thread structures. A thread structure contains[VAHA 96],

1. Pointer to the task structure of the task to which it belongs.
2. Pointer to the list of threads in the task.
3. Pointer to the thread's kernel stack.
4. State of the thread and priority information.
5. Pointer for putting thread on various queues (run queue, etc.).
6. Per thread port related information.

In addition to kernel supported threads, Mach provides a very flexible user level threads library known as C-threads library. In terms of the interface provided, C-threads are very similar to the POSIX Pthreads interface but for a few features of Pthreads which are not available in C-threads. These include, the ability to set various attributes of threads such as stack sizes, thread cancellation, timed waits, signal handling, etc.[BOYK 93].

However, the one feature of the C-threads library which makes it particularly flexible is that there are different implementations of the C-threads interface, each of which maps a C-thread differently to the underlying Mach task and thread constructs. These implementations are [VAHA 96],

1. Coroutine - In this implementation, each C-thread maps to a user level thread. All threads in an application are multiplexed on a single Mach kernel thread in a single task. Thus in this implementation a C-thread behaves like a user level thread implementation.
2. Kernel thread implementation - Here each C-thread maps to a single Mach kernel level thread. Hence C-threads behave like kernel threads in this implementation.
3. Task implementation - Here each C-thread maps to a Mach task. This implementation is not commonly used.[VAHA 96].

Thus an application has the flexibility of choosing the implementation suited for its goals while maintaining the same thread interface.

F. Digital UNIX Threads

Digital UNIX developed by DEC, was previously known as OSF/1 as it was based on work done by the OSF. There are two versions of Digital UNIX available(versions 3.2x and

4.0) which differ widely in their implementation of threads. This paper talks about Digital UNIX version 3.2x. Digital UNIX is based on Mach 2.5. Internally it has a Mach core on which a BSD UNIX style interface was built. In Mach 2.5, the kernel itself contains BSD code. Later versions of Mach (version 3.0 onwards), implement the BSD environment as a user level server process. Hence most of the discussion above on Mach kernel threads applies to Digital UNIX as well. As stated above, Mach is organized in terms of tasks and threads. However, the BSD interface code was ported from actual BSD 4.3 source code[VAHA 96]. Hence to make porting easier, it was necessary to maintain the traditional UNIX proc and u area data structures, even if in a modified form. The u area was replaced by two data structures, a utask structure which was for the task as a whole and a uthread structure for each thread.

The utask area contains task specific information which includes[VAHA 96],

1. Pointers to vnodes (virtual inodes which allow an operating system to support multiple different filesystem types on a single computer system)
2. Pointer to the proc structure
3. Signal handlers.
4. Open file descriptor table.
5. Resource usage information.

The uthread data structure on the other hand contains thread specific information such as[VAHA 96],

1. Pointer to user level register context(when thread is not running)
2. List of current and pending signals
3. Signal handlers specific to the thread.

The proc structure was not changed much in Digital UNIX, but is also not used much because the task and thread data structures now perform most of its functions. The proc structure consists of,

1. Pointers to put the proc structure (and hence the process) on the free or zombie list.
2. Signal masks.
3. Process Id, parent's PID, links to PIDs of related processes.
4. Group information.
5. Resource usage and exit status information
6. Pointers to task, utask data structures and to thread data structure of first thread.

Thus although the task and thread data structures form the core, the proc and the u area (in modified form) are maintained for compatibility with the UNIX code layered on top of Mach.

As in Solaris, threads may also be created by the kernel task for kernel level work. Such threads have no user context and neither does the kernel task. However, unlike Solaris, Digital UNIX provides only one fork system call which has the same semantics as Solaris' fork system call i.e., only the calling thread and the process are duplicated. As in Solaris, synchronous signals (SIGSEGV - segmentation fault, etc.) are sent to the thread which caused them. However, asynchronous signals are sent to one of the threads in the process and all threads in a process share the same signal mask.

Digital UNIX provides the draft 4 version of the POSIX Pthread multithreaded programming interface standard. However, the standard does not specify the level of implementation. Digital UNIX implements Pthreads as kernel threads with each Pthread being built on top of a single Mach thread. As stated earlier, the disadvantage of using kernel

threads is that the kernel has to be invoked for every thread operation. In the case of Digital UNIX (and Solaris), any synchronization which does not require a thread to block in the kernel (such as when a lock is acquired without contention) can be done without kernel involvement. Digital UNIX also uses threads to implement asynchronous I/O. Whenever an asynchronous function is called, the thread library creates a thread to service the call and the calling thread is free to continue executing. When the call completes the calling thread is notified by a signal.

G. Scheduler Activations

One of the problems with hybrid thread packages is disconnected two level scheduling. This problem arises because the user level thread scheduler has no control or knowledge of processor allocation which is handled by the kernel. The user level thread scheduler can only assume that each of its kernel threads is running on its own processor. Whenever this assumption is violated, two level scheduling problems can result. The key to solving this problem is to ensure that the number of runnable kernel threads (kernel threads not blocked in the kernel) is the minimum of the number of runnable threads and the number of processors available to the process. Thus the number of runnable kernel threads can never exceed the number of processors available to the process. Moreover the user level thread scheduler should know how many processors it has available to it and hence the maximum number of concurrently running kernel threads. Thus the user level thread scheduler's assumption that all runnable kernel threads have their own processors always holds true.

There are solutions to this problem such as scheduler activations[VAHA 96]. In this scheme, the kernel informs the processor of any changes in the assignment of processors to the process. A kernel thread in this scheme is known as a scheduler activation and a process has as many scheduler activations as processors currently assigned to it. Whenever a user level thread blocks in the kernel, the kernel assigns the process a new scheduler activation on the processor relinquished by the old activation when it blocked and informs the thread library about the change. The thread library saves state from the old activation and returns it to the kernel. The thread library can then schedule another thread on the new activation. Similarly when the kernel preempts a processor currently assigned to the process, it informs the thread library so that it can make an appropriate decision. Thus the kernel controls processor allocation and informs the user library of all allocation decisions affecting it. The thread library is responsible for scheduling user level threads on available scheduler activations and it is helped in its task by the information provided by the kernel. Although scheduler activations solves the two level scheduling problem, its implementation requires changes to the kernel. Currently no commercial operating system implements this technique.

H. Opal threads

One of the disadvantages of scheduler activations is that its implementation requires kernel modifications. However solutions involving better communication between applications and thread library have been proposed[FEEL 93] which can solve some but not all problems associated with two level scheduling. This solution is used in the Opal single address space operating system which is a research implementation. The solution avoids the two level scheduler problem for blocking calls which can be anticipated by the application. However for changes that cannot be anticipated by the application such as preemption of processors from the process, this technique will not work. In the solution proposed by Feeley et. al[FEEL 93], a separate component called the activation pool is added to the thread library. This component is in charge of creating and destroying kernel threads. The application informs the activation pool when it makes a blocking system call. The activation pool in turn informs the scheduler about the blocking of the kernel thread. The thread library

can then ask the activation pool to create a new kernel thread to replace the one blocked in the kernel if it has runnable user level threads and the local invariant is not exceeded. When the blocking system call completes, the activation pool informs the thread scheduler that the kernel thread has unblocked. The scheduler can then either take back the kernel thread and its associated user level thread and let them run (if the local invariant is not exceeded) or the scheduler can put the user level thread returning from the system call on the runnable queue and ask the activation pool to destroy the associated kernel thread (if the local invariant is exceeded).

Even one level schedulers are not entirely problem free. The following example illustrates the priority inversion problem commonly found in schedulers. Consider a low priority thread A running on a uniprocessor which holds a resource R. If a high priority thread C attempts to obtain this resource it will block. Now if a medium priority thread B preempts the low priority thread A and takes over the processor, thread A cannot run. Meanwhile the high priority thread C cannot run because the resource which it requires is being held by thread A which cannot run and finish using the resource. Thus a higher priority thread C is prevented from running by a lower priority thread B which is able to use the processor. This is known as priority inversion. A solution to this problem is to have the low priority thread A raise its priority to a high enough level so it can run. This is inconvenient for the thread itself to do since it has to know about what other threads are involved in the priority inversion and what their priorities are. A better job could be done by the scheduler. It should raise the priority of the thread holding the resource at least as high as the highest priority thread blocked on the resource as long as the thread holds the resource. This technique is called priority inheritance and is used in Solaris 2.x.

III. Summary

Threads are lightweight process-like executing entities which share memory within a process. Threads are better suited than processes for many concurrent applications. They are useful as mechanisms for structuring programs. In addition they provide better performance on uniprocessors by allowing computation and I/O to be overlapped. On multiprocessors, kernel supported threads can transparently be scheduled on multiple processors. Threads and multithreading are supported on most modern operating systems. However implementations and functionality vary widely. Threads can be classified based on various criteria including level of operating system support, scheduling model, programming interface and the multiplexing model. The level of implementation is commonly used to classify implementations based on their internal structure. User level threads are not visible to the kernel and are useful primarily for structuring programs. They are lightweight and can be used in large numbers with very little overhead. Kernel threads can provide improved performance with certain applications and on multiprocessors. They are also relatively expensive and must be used judiciously. Hybrid threads provide both user and kernel threads and offer the advantages of both models.

Several thread packages are available which vary in their functionality and interface. The Solaris operating system offers a very flexible hybrid type implementation with both the POSIX standard Pthreads interface and the UI (UNIX International) programming interface. Windows NT offers a kernel threads implementation which is very flexible because of its ability to emulate various environments. The most commonly used environment on Windows NT is the Win32 subsystem which is widely available. The Win32 interface offers

kernel threads and a programming interface which is quite different from the POSIX interface. Yet another interface is provided by Java, which is a language with integrated multithreading support. Java borrows some concepts like daemon threads from Solaris, but its interface is quite different from Solaris and is more limited in terms of the number of features offered. However since multithreading is integrated with the language, many routine and error-prone synchronization tasks are automated. Java is a cross platform language and relies on operating system facilities to provide services like multithreading. Hence benefits from Java may vary from platform to platform.

Mach was one of the earliest research operating systems to both support multithreading and gain widespread usage. The Mach kernel supports threads and in addition has a user level thread library known as C-threads. C-threads has multiple implementations each of which maps C-threads differently to the underlying Mach tasks and threads. Digital UNIX is an operating system which is based on the Mach kernel. It offers a Pthreads like interface which is based on an early draft of the Pthreads standard. Pthreads on Digital UNIX are kernel threads and are built on top of the underlying Mach threads.

It appears likely that the POSIX interface will predominate on UNIX based systems while Win32 will dominate the Windows desktop systems. The MacOS is currently undergoing a major change and is likely to be based on a kernel derived from Mach. Java is likely to dominate as a crossplatform language and as a language for WWW based applications (applets).

Since the hybrid implementation model offers a very flexible, it is likely to predominate as an implementation model in the future. However this model suffers from disconnected two level scheduling. Solutions such as scheduler activations which involve better coordination between the user level library and the kernel have been developed to overcome these problems.

References

1. Berg D. J., *Java Threads*, White paper, Sun Microsystems, 1995.
2. Boykin, J. et. al., *Programming under Mach*, Addison-Wesley, Reading, Mass, 1993.
3. comp.os.research FAQ 1996.
4. comp.programming.threads FAQ, 1996.
5. Custer H., *Inside Windows NT*, Microsoft Press, Redmond, Washington, 1993.
6. Eykholt J. R., S.R. Kleiman, Barton S., Faulkner R., Shivalingaiah A., Smith M., Stein D., Voll, J., Weeks M., Williams D., - SunSoft Inc., *Beyond Multiprocessing ..., Multithreading the SUNOS Kernel*, Summer '92 USENIX Technical Conference, San Antonio, TX, June 1992.
7. Feeley M. J., Chase J. S. and Lazowska E. D., *User-level Threads and Interprocess Communication*, Technical report 93-02-03, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.
8. Powell M. L., Kleiman S. R., Barton S., Shah D., Weeks M., Sun Microsystems Inc., *SunOS Multi-thread Architecture*, Winter '91, USENIX Technical Conference, Dallas, TX, Jan. 1991.
9. Stein D., Shah D., - SunSoft Inc., *Implementing Lightweight Threads*, Summer '92 USENIX Technical Conference, San Antonio, TX, June 1992.
10. Vahalia U., *UNIX Internals The New Frontiers*, Prentice -Hall Inc., Upper Saddle River, New Jersey, 1996.