A STUDY OF SYNCHRONIZATION MECHANISMS IN UNIX, WINDOWS NT, AND MAC OS

By

RAMASAMY SATISHKUMAR

Bachelor of Engineering

University of Madras

Madras, India

1995

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 1997

OKLAHOMA STATE UNIVERSITY

A STUDY OF SYNCHRONIZATION MECHANISMS IN

UNIX, WINDOWS NT, AND MAC OS

Thesis Approved:

Mansur Samadzadeh

Thesis Advisor

Blayne E. Mayfield

J. Chandler

Wayne B. Powell

Dean of the Graduate College

PREFACE

In the world of operating systems, the wheels of progress turn rather slowly. Operating systems take years to develop. Communication between processes is an important and difficult topic in operating systems. Studies of interactions and communications among processes have resulted in new synchronization primitives. Existing commercial and popular operating systems use different synchronization mechanisms to achieve kernel based synchronization as well as to provide synchronization facilities for applications. Each of these synchronization mechanisms has its advantages and disadvantages. The objective of this thesis work was to conduct a comparative study on how synchronization is achieved in UNIX, Windows NT, and Apple Macintosh operating systems. A detailed study on how synchronization is achieved in these operating systems was carried out. Based on this study the operating systems were compared and the results were tabulated.

The comparative study indicates that among other things UNIX and Windows NT are preemptive multitasking and symmetric multiprocessing operating systems; Apple Macintosh is a cooperative multitasking and master-slave multiprocessing operating system; Thread is the basic unit of scheduling in Windows NT and in recent versions of UNIX such as Solaris 2.5; and in a multiprocessor environment, both UNIX and Windows NT use Spin Locks for achieving synchronization.

ACKNOWLEDGMENTS

I would like to express my appreciation to and thank my graduate advisor Dr. Mansur H. Samadzadeh for his advice, guidance, dedication, encouragement, and instruction throughout my thesis research work. I got inspiration and motivation due to his constant guidance. Without his support and motivation it would not have been possible to complete this work.

I offer my sincere thanks to Drs. B. E. Mayfield and J. P. Chandler for serving on my graduate committee.

Finally, I wish to thank my parents. Without their support and encouragement, it would not have been possible for me to complete my graduate studies.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

An operating system is the set of programs that control a computer. Many books on operating systems describe various operating system concepts [Deitel 92] [Krakowiak 90] [Stallings 95] [Tanenbaum 92] [Tanenbaum and Woodhull 97]. Operating system software includes several levels: kernel-level services, library-level services, and application-level services. Applications are user programs that are linked together with libraries.

The details of what constitutes a process differ from one system to another. At the highest level of abstraction, a process comprises the following [Deitel 92]: a program abstraction that defines the initial code and data, a private address space that is a set of virtual memory addresses that the process can use, and system resources such as semaphores, communication ports, and files, that the operating system allocates to the process as the program executes.

An operating system consists of the following components [Tanenbaum 92]:

i. basic structure;

ii. synchronization and communication mechanisms;

iii. implementation of processes, process management, scheduling, and protection;

iv. memory organization and management, including virtual memory; and

v. input output device management, secondary storage management, and file system management.

A multitasking operating system concurrently executes more than one task or process. A multitasking operating system is basically a logical extension of a multiprogrammed operating system. Multiple jobs are executed by the CPU switching between them, but the switches occur so frequently that the users may interact with each program while it is running. Time slicing, as used in operating systems, is when a process is given a particular time period in which it can utilize the CPU and, when the time period completes, the CPU is allocated to the next process and the current process is sent back to the ready queue.

Synchronization among processes is an important issue in operating systems. One of the primary problems confronting the designers of operating systems is to provide an efficient synchronization mechanism. A considerable amount of research work has been reported in open literature in this area. Many synchronization primitives have been proposed, e.g., Events, Sequences, Queues, and Conditional Critical Regions. Dunstan and Fris did a study on semaphores as implemented in UNIX System V [Dunstan and Fris 95]. Avutu did an extensive study on synchronization mechanisms, and came up with a new synchronization primitive [Avutu 93]. This thesis work comprises of a detailed comparative study on how synchronization is achieved in UNIX, Windows NT, and Apple Macintosh ("Mac") operating systems.

The rest of this thesis report is organized as follows. Chapter II discusses synchronization and different synchronization mechanisms. Chapter III, IV, and V deal

with UNIX, Windows NT, and Apple Macintosh operating systems, respectively. This three chapters introduce the basic architecture of each operating system and then detail how synchronization is achieved in them. Based on the information collected in this study, Chapter VI compares how synchronization is achieved in the three operating systems under consideration. Chapter VII concludes, summarizes, and suggests future work.

CHAPTER II

LITERATURE REVIEW

2.1 Synchronization

Synchronization is the mechanism used to guarantee mutual exclusion among processes when accessing a critical section and to achieve inter-process communication [Wills 96]. A critical section is a sequence of instructions that may be executed by at most one process at a time. Processes involved in synchronization become indirectly aware of each other by waiting on a condition that is set by other processes [Deitel 93].

2.2 Communication Between Processes

Processes communicate with each other using inter-process communication (IPC) mechanisms. Files, pipes, and shared memory are some of the methods used for IPC. Files are the most obvious means of passing information between processes. One process writes to a file and the other reads from that file. Even though files are not interactive, they are often used for IPC.

Another method of connecting the output data stream of one process to the input of another process is known as a pipe. A pipe can be of two types: unidirectional and bi-directional. In unidirectional pipes, the second process cannot talk back to the first process. In bi-directional pipes, actually two unidirectional pipes connect the two processes, so that both of the processes can communicate with each other. Pipes

can hold only a finite amount (10 blocks) of data [Vahalia 96]. Deadlock can occur when using pipes for IPC. For example, while using a bi-directional pipe between two processes, both unidirectional pipes get filled up, then if both processes are blocked writing to their pipes, neither can read any information from their own unidirectional pipe because they haven't finished writing into the other pipe.

The use of shared memory is one of the fastest IPC mechanisms known. Two or more processes share part of the logical memory locations [Wills 96]. Shared memory IPC mechanisms are easy to implement in operating systems with paged memory architecture [Wills 96]. In the case of using shared memory for IPC, the operating system has to keep a link count (similar to the case of using shared files) so that a page can be freed when the link count becomes zero. Implementing shared memory IPC for operating systems without paged architecture is considerably more difficult than for those with paged architecture [Wills 96].

2.3 Synchronization Mechanisms

Synchronization mechanisms can be broadly classified into four basic types based on their level and type of implementation and support: software support, hardware support, operating system support, and language support. In addition, hybrid synchronization solutions exist that combine more than one approach [Wills 96].

2.3.1 Software Support

A correct software based solution for mutual exclusion was first devised by Dekker (as cited in [Dijkstra 68]). He used shared variables to control access to the critical section. Subsequently, other solutions were also proposed. A relatively simpler

solution to the two process mutual exclusion problem was presented by Peterson [Peterson 81]. Dijkstra presented a solution for solving the critical section problem for n processes called the Bakery algorithm [Silberschatz and Galvin 95].

2.3.2 Hardware Support

Hardware based solutions are typically the conceptually simplest solutions. They can be achieved for instance by disabling hardware interrupts at the start of the critical section and enabling them at the end. This will not work in the case of having more than one processor, because even if interrupts are blocked in one processor, all other processors are free to access the critical section, so a different technique has to be followed. For hardware based solution many machines provide special hardware instructions that can be used either to test and modify the contents of a word or to swap the contents of two words atomically (i.e., indivisibly).

The Test-and-Set instruction can be defined as follows [Silberschatz and Galvin 95].

```
Function Test-and-Set (var target: boolean): boolean;

    begin
        Test-and-Set := target;
        target := true;
    end;
```

The Swap instruction swaps the contents of two words atomically and is defined as follows.

```
Procedure Swap (var a, b: boolean);

    var temp: boolean;
    begin
        temp := a;
        a := b;
```

```
        b := temp;
    end;
```

A context switch cannot occur in the middle of the critical section, as these hardware instructions are carried out in an atomic manner, i.e., their execution from beginning to end is indivisible.

2.3.3 Operating System Support

Operating system based solutions can be achieved by adding process-synchronization support to an operating system [Silberschatz and Galvin 95]. The use of semaphores is one example of this type of support. Semaphores can be used to solve most of the synchronization problems. Dijkstra originally defined the semaphore concept [Dijkstra 68]. A semaphore s is a non-negative integer variable that has an implicit queue associated with it. The value of the variable can be handled only by the following two primitive operations.

```
P(s): if s > 0 then s ← s - 1;
      else wait on s;

V(s): s ← s + 1;
```

The mutual exclusion scheme can be coded using a mutual exclusion semaphore called mutex (initialized to 1), as follows.

```
Wait: P(mutex);
        <critical section>
Signal: V(mutex);
```

The Wait or P operation is used by a process wishing to enter a critical section. If the value of the semaphore variable is greater than zero, it is decrement by one and the process is executed. If the value is less than or equal to zero, then the process is added to

the queue associated with the semaphore. The Signal or V operation is used by a process leaving a critical section. It checks the queue to see if there is a process waiting. The processes in the queue are in a passive waiting state. If there is a process, it is activated. If no process is waiting, the semaphore is incremented by one [Silberschatz and Galvin 95]. There are many extensions to the basic definition and implementation of the concept of a semaphore, intended to suit various synchronization requirements, runtime environments, and implementation platforms.

2.3.4 Language Support

Programming language based synchronization can be implemented, for example, by using a construct named monitor or a construct called rendezvous. Implementations of the monitor construct exist in Mesa and JAVA, programming languages from Xerox PARC and Sun Microsystems, respectively. A monitor is characterized by a set of programmer-defined operators. The syntax of a monitor [Silberschatz and Galvin 95] is as follows:

```
type monitor-name = monitor
variable declarations
procedure entry P1 (...);
        begin ... end;
procedure entry P2(...);
        begin ... end;
            .
            .
            .
procedure entry Pn(...);
        begin ... end;
begin
        Initialization Code
end.
```

Detailed information about monitors can be found in the original paper by Hoare [Hoare 74].

Synchronization (or rendezvous) is achieved in the Ada programming language by using the accept statements and entries in a task [Wheeler 96]. A task is a unit of parallelism in Ada. It consists of two parts: task specification and task body. Task specification contains declarations and definitions provided by a task called entry. The task body contains the implementations. The syntax of a task specification is as follows.

```
task [type] <name> is
entry specifications
end;
```

The syntax of a task body is given below.

```
task body <name> is
declarations of local variables
begin
      list of statements
      exceptions
      exception handlers
end;
```

An accept statement is an entry into a task. It is similar to a procedure in conventional languages. There is a one-to-one correspondence between the entry statements in a task specification and the accept statements in a task body. The syntax of the accept statement is given below.

```
accept <entry id> ( <formal parameters> ) do
body of the accept statement
end <entry id>;
```

Once an entry is called, the corresponding accept entry will not be executed until control reaches the accept statement in the task. If the accept statement is reached first, the task is blocked until some other task executes the corresponding entry. When an entry and the

accept connect, it is said that rendezvous occurs. The rendezvous mechanism is more disciplined than a monitor, since the accept statements appear inside a context.

CHAPTER III

UNIX

3.1 Basic Architecture

The UNIX operating system can be divided into two major levels as shown in Figure 1: user level and kernel level. The UNIX kernel can be divided into two major entities: the file subsystem and the process subsystem [Bach 86]. The file subsystem and the process subsystem are shown in the left and right sides of Figure 1 [Bach 86], respectively.

The system call and the library interfaces lie between the user level and the kernel level. The system calls can be further subdivided as those that interact with the file subsystem and those that interact with the process control subsystem. The file subsystem manages files, allocates file space, administers free space, controls access to files, and retrieves data for users. The device drivers block that is shown between the file subsystem and the hardware control are the kernel modules that control the peripheral devices.

The process control subsystem is responsible for process synchronization, inter-process communication, memory management, and process scheduling. Finally, the hardware control, shown above the hardware block in Figure 1, is responsible for handling interrupts and for communicating with the machine.

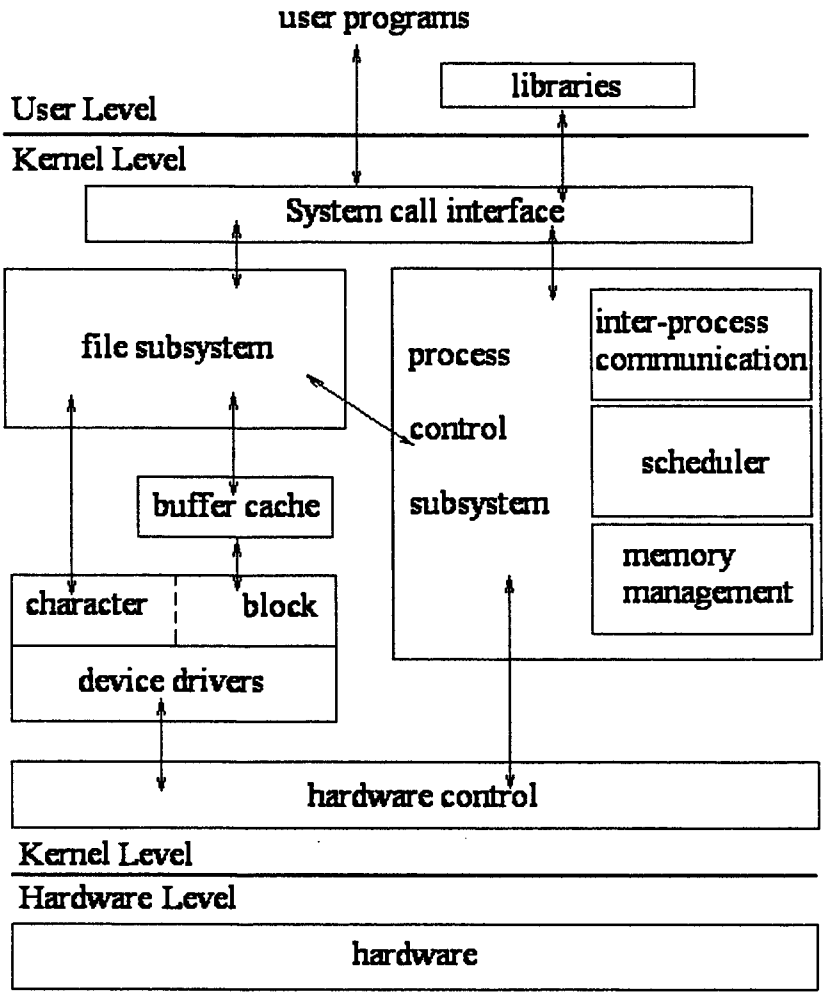

Figure 1. Block Diagram of the UNIX System [Bach 86]

I/O devices and other peripherals may interrupt the operating system while a process is being executed. In such cases the kernel may resume the execution of the interrupted process after servicing the current interrupt. Interrupts are serviced by special functions in the kernel.

3.2 UNIX Processes

A process from the UNIX point of view is an entity that runs a program and provides an execution environment for it [Bach 86]. In other words, it is an instance of a

running program. It comprises of an address space and a control point. Basically, a process is the fundamental scheduling entity, i.e., only one process runs on the CPU at a time. Each process has a definite life time. Most of the UNIX processes are created by a *fork* or *vfork* system call. A process invokes the *exec* system call to run a new program, thus during the life time of a process it may run one or more programs at a time [Bach 86].

UNIX processes have a well-defined hierarchy. Each process has at most one parent process and zero or more child processes. The process hierarchy looks like an inverted tree with the *init* process at the top [Back 86]. The *init* process executes the program located at /etc/init, and it is the first user process that gets created when the system boots. UNIX processes are in well-defined states as shown in Figure 2 [Vahalia 96]. In UNIX , the *fork* system call is used to create a process, until a process is fully created it is in the initial state and then it is moved to the ready-to-run state. The ready-to-run state means that a process is ready to be scheduled by the kernel. When such a process is scheduled, it executes in the kernel mode (kernel running state) still the context switch gets completed. After this, if it was a user mode process, it shifts to the user running state, whereas if it was blocked for a resource while executing a system call, it resumes execution in the kernel mode. Processes switching can occur only in the kernel by explicit calls to the event-wait mechanism [Leffler, et al. 89].

As a result of a system call or an interrupt, a process that is running in the user mode enters the kernel mode, and returns to user mode when the system call/interrupt completes. If a process has to wait for an event or resource that is not available, it calls the *sleep()* function. This will put the process on a queue of sleeping processes (asleep

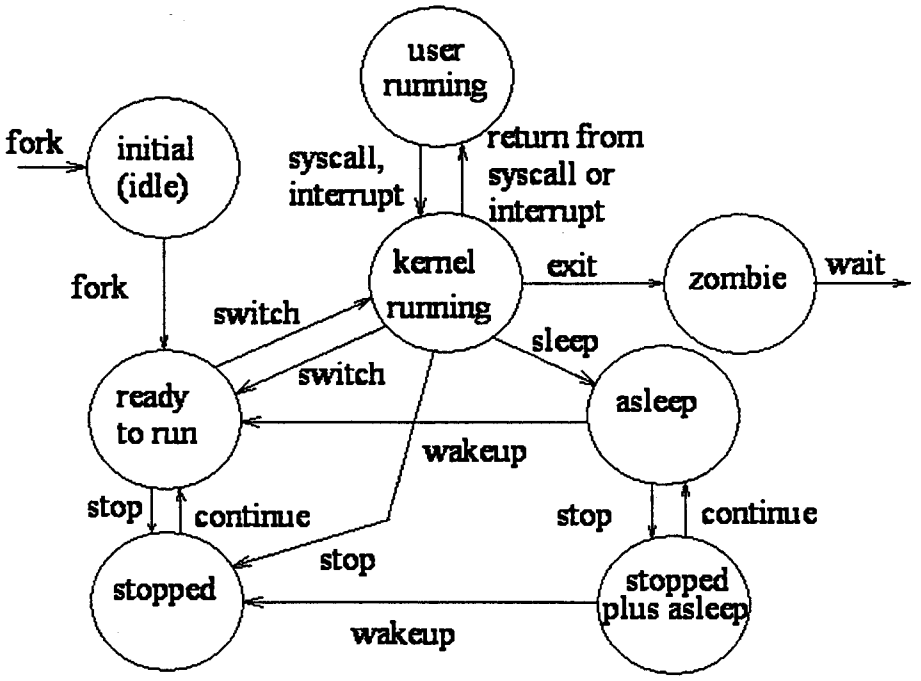

Figure 2. Process States and State Transition [Vahalia 96]

state). When the resource becomes available or the event occurs, the kernel wakes up all of the processes waiting for the same. When a process is stopped or suspended by a stop signal, it moves from the ready-to-run state and asleep state to the stopped state and the stopped-plus-asleep state, respectively. A continue signal helps this process to return to its previous state. A process terminates by calling the *exit* system call. The kernel releases all of the resources of the process, except the exit status and resource usage information, and leaves the process in zombie state. A process remains in this state until its parent generates a wait signal, which destroys the process and returns the exit status to the parent [Vahalia 96].

3.3 Kernel Level Synchronization

UNIX kernel is re-entrant, meaning that several process may be involved in kernel activity concurrently. In fact, processes may even be executing the same routine in parallel. At any instant, several processes may be active but only one is actually running, the others are inactive (blocked, suspended, or asleep). UNIX implements kernel level synchronization to avoid race condition, as all processes share the same copy of the kernel. Race condition is a situation in which several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place. UNIX uses several synchronization techniques. The first and foremost is that initially UNIX kernel was not preemptive [Vahalia 96]. Basically, process synchronization is accomplished by having processes wait for events. We will look at the synchronization issues involving the non-preemptive kernel in this section and then we will investigate additional synchronization techniques that are used in the case of the preemptive kernel.

The situations under which we need synchronization in a non-preemptive UNIX kernel are [Vahalia 96]: when a blocking operation is being carried out, when an interrupt occurs, and for systems with more than one processor. We will next discuss how synchronization is achieved in the case of blocking operations and interrupts. Synchronization issues in the case of multiple processors is discussed in Section 3.4 under the heading of Multiprocessor Synchronization.

3.3.1 Blocking Operations

The blocking operation places a process in the asleep state until the operation on which it blocked completes. Even in the case of a non-preemptive UNIX kernel, most of the objects (data structures and resources) need to be protected across a blocking operation. Let us consider the following example [Vahalia 96]: a process has to read from a file into a disk block buffer in kernel memory. As this is an I/O operation, the process has to wait until it completes. Meanwhile the kernel may schedule another process. Since the disk block buffer is in an inconsistent state, the kernel must ensure that other processes do not access the buffer in any way.

UNIX kernel prevents such accesses by associating a lock with the various objects involved. All processes that need to use an object must check the lock. If it is open, the process can access the object, else the process must go to sleep until the object gets unlocked. Normally, the UNIX kernel associates a wanted flag with an object. This flag is set by a process that needs it while it was locked. When a process is ready to release a lock, it wakes up all the processes that have set the wanted flag for that object. By following this procedure, UNIX allows all other processes to execute safely even when a process has blocked after locking a resource. Figure 3 shows this locking algorithm [Leffler, et al. 89].

3.3.2 Interrupts

In a non-preemptive UNIX kernel, even though a process cannot be preempted by another process, it still can be interrupted by devices. In order to provide proper synchronization, the interrupt handler should not be allowed to access the data that is in

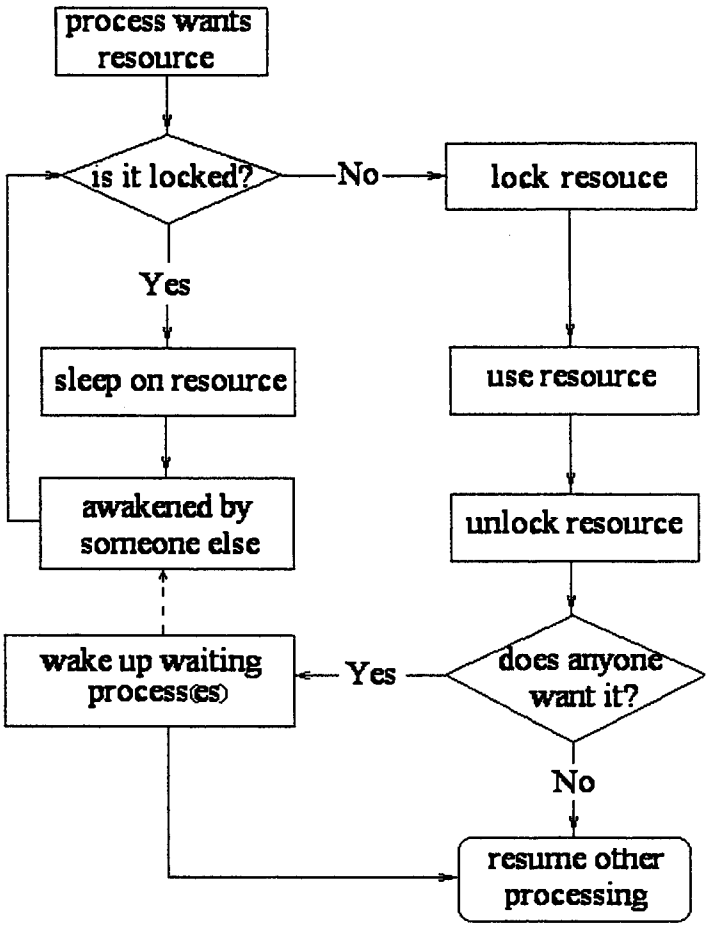

Figure 3. UNIX Locking Algorithm [Leffler, et al. 89]

Table I. Macros used to set the Interrupt Priority Level in SVR4 [Vahalia 96]

Macros	**Purpose**
sp10 of splbase	enable all interrupt
spltimeout	block functions scheduled by timers
splstr	block STREAMS interrupts
spltty	block terminal interrupts
spldisk	block disk interrupts
sp17 or splhi	disable all interrupts
splx	restore *ipl* to previously saved value

an inconsistent state. Synchronization in the case of interrupts is achieved by blocking interrupts while accessing the data that is in an inconsistent state. The UNIX kernel use macros similar to the ones in Table I [Vahalia 96]. These macros help to raise the *ipl* (interrupt priority level) and thereby help to block the interrupts while accessing the critical region. Using these macros, the current *ipl* value is recorded and it is raised to a new value. After raising the *ipl* value, the process enters the critical region. When the process leaves the critical region, the *ipl* value is restored with the recorded value.

3.4 Multiprocessor Synchronization

By increasing the number of processors, the system performance does not increase linearly [Kelley 89]. The need for appropriate synchronization primitives that are needed while accessing the shared data structures, and the extra functionality's like scheduling policies etc., to support multiple processors adds CPU overhead and thus reduces the overall performance gains. The operating system must try to minimize the overhead and allow for optimal CPU utilization. The traditional UNIX kernel needs major modifications to run on multiprocessor systems. One such major area of modification is synchronization.

Synchronization primitives that we discussed earlier under uni-processor environments (see Section 3.3) are inadequate for multiprocessors and must be replaced with more powerful facilities. Synchronization in the case of more then one processor is fundamentally dependent on hardware support provided by the processors. In the subsections below, we will discuss hardware synchronization mechanisms and then

different software synchronization mechanisms for multiprocessor UNIX operating system.

3.4.1 Hardware Support

Let us consider the basic operation of locking a resource, for exclusive use, by setting a locked flag maintained in a shared memory location. This may be achieved by performing the following three operations:

i) read the flag;

ii) if the flag is zero, lock the resource by setting the flag to one; and

iii) return TRUE if the lock was obtained, else return FALSE.

In the case of a multiprocessor systems, two processes on two different processors may simultaneously attempt to carry out this sequence of operations. In order to avoid the race condition (see Section 3.3) that may occur in such situations, the three operations above will have to be performed as one single indivisible operation. Two such indivisible operations that are available on most processors are the atomic Test-and-Set operation and the Conditional-Store instruction. For example, SUN SPARC machines, that run a variant of UNIX, have LDSTUB (LoaD and STore Unsigned Byte) as an atomic Test-and-Set instruction, similarly VAX-11 has BBSSI (Branch on Bit Set and Set Interlock) [Digital 87].

3.4.2 Software Support

Software support for synchronization in an operating system is dependent on the type of multiprocessing technique used in that operating system. There are three types of multiprocessing systems [Vahalia 96]: master-slave, functionally asymmetric and

symmetric. The variants of UNIX such as SVR 4.2 and Solaris 2.5 are symmetric multiprocessing systems [Tanenbaum and Woodhull 97]. In a symmetric multiprocessing system, all CPUs are equal, they share a single copy of the kernel text and data, and compete for system resources such as devices and memory. Each CPU may run the kernel code, and any user process may be scheduled on any processor. It is no longer the case that a thread (see Section 4.3) retains exclusive use of the kernel or block on a resource, as several processors could be executing kernel code at the same time.

We need to protect all kinds of data structures that were not protected in the case of a uni-processor system. The IPC resource table is one such data structure. This structure is not accessed by interrupt handlers and does not support any operations that might block the processes. The kernel manipulates the table without locking it. In the case of multiprocessor environment two threads can access the table simultaneously [Vahalia 96], and hence the kernel must lock the table before using it . The locking primitives and the way interrupts are handled has to be changed so that proper synchronization can be achieved in a multiprocessor environment. In the following subsection let us discuss some of the locking mechanisms.

3.4.2.1 Semaphores

Earlier implementations of UNIX on multiple processors was almost completely dependent on semaphores for synchronization [Kelley 89]. UNIX kernel guarantees that the semaphore operations will be atomic [Lee and Luppi 87], even on a multiprocessor system. Thus, if two threads try to operate on the same semaphore, one operation will complete or block before the other starts.

Semaphores can be used to provide mutual exclusion on a resource. A semaphore, that is initialized to one, can be associated with a shared resource such as a linked list. Each thread does a P operation to lock a resource and a V operation to release it. The first P operation sets the value to zero, causing subsequent P operations to block. When a V is done, the value is incremented and one of the blocked threads is awakened. The code that represents the usage of semaphores in case of controlling the allocation of finite resources [Bach 86], such as message block headers, is as follows.

```
semaphore counter;        // initialization of semaphore followed by
                          // initsem initialization done at the boot
                          // time
initsem (&counter, resourceCount);

P (&counter);             // thread calls P while acquiring an
                          // instance of the resource
V (&counter);             // thread calls V while releasing an
                          // instance of the resource
```

The semaphore is initialized to the number of available instances of the resource under consideration. Threads call P to acquire an instance of the resource, and then call V to release it. Thus, at each point in time, the value of the semaphore indicates the number of pending requests (blocked threads) for that resource.

Semaphores can also be used to cause threads to wait for an event by initializing them to zero. This is shown in the following code.

```
semaphore event;          // initialization
initsem (&event, 0);      // initialized at boot time

                          // code section executed by the thread
                          // that may wait on an event
P (&event);               // blocks if event has not occurred
V (&event);               // called when an event occurs
V (&event);               // each thread call this V upon waking up
```

In this scenario threads doing a P operation will block. When the event occurs, a V operation needs to be done for each blocked thread. This is achieved by calling a single V

operation when the event occurs and having each thread do another V operation upon waking up.

In case of both uni-processor and multiprocessor UNIX systems the use of semaphores has a disadvantage called semaphore convoy. A semaphore convoy is created when there is frequent contention on a semaphore. Figure 4 [Lee and Luppi 87] depicts a semaphore convoy situation. R is a critical region protected by a semaphore, P1 and P2

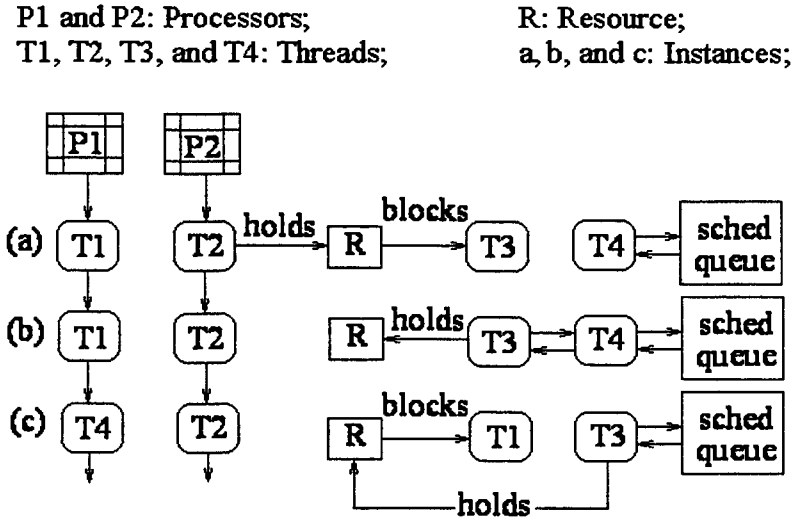

Figure 4. Semaphore Convoy [Vahalia 96]

are two different processors, and T1, T2, T3, and T4 are the threads. At instance 'a', thread T2 holds the semaphore, while T3 is waiting to acquire it. T1 is running on processor P1, T2 is running on processor P2 and T4 is waiting to be scheduled. Now suppose T2 exits the critical region and releases the semaphore. It wakes up T3 and puts it on the scheduler queue. T3 now holds the semaphore, as shown in instance 'b' in Figure 4. Now T1 need to enter the critical region. Since the semaphore is held by T3, T1 will

block, freeing up processor P1. The system schedules thread T4 to run on P1. Hence, at instance 'c' thread T3 holds the semaphore and T1 is blocked on it; neither thread can run until T2 or T4 yields its processor. The problem lies in step 'c'. Although the semaphore has been assigned to T3, T3 is not running and hence it is not in the critical region. As a result, T1 must block on the semaphore even though no thread is in the critical region. The semaphore semantics force allocation in a first-come, first-served order. This forces a number of unnecessary context switches. In more recent variants of UNIX such as Solaris 2.5, semaphore has been replaced by an exclusive lock, or mutex [Vahalia 96].

3.4.2.2 Read-Write Locks

A read-write lock on multiprocessors may permit either a single writer or multiple readers [Vahalia 96]. The basic operations are lockShared(), lockExclusive(), unlockShared(), and unlockExclusive(). In addition, there might be tryLockShared() and tryLockExclusive() (which return FALSE instead of blocking), and also upgrade() and downgrade() (converts a shared lock to an exclusive lock and vice versa). A lockShared() operation must block if there is an exclusive lock present, whereas a lockExclusive() operation must block if there is either an exclusive or a shared lock on the resource. The code that implements a read-write lock is as follows [Vahalia 96].

```
Struct rwlock {
      int nActive;                          // number of active
                                            // readers, or -1 if a
                                            // writer is active

      int nPendingReads;
      int nPendingWrites;
      spinlock_t sl;
      condition canRead;
      condition canWrite;
};

void lockShared (struct rwlock *r)          // this operation blocks
                                            // if there is an
```

```
                                        // exclusive lock
                                        // present
{
      spin_lock (&r->sl);               // acquire the mutex
                                        // spin lock
      r->nPendingReads ++;              // increment the count
                                        // of waiting readers
      if (r->nPendingWrites > 0)        // if there are writers
                                        // waiting then block
          wait (&r->canRead, &r->sl);   // thereby don't starve
                                        // writers
      while (r->nActive < 0)            // someone has an
                                        // exclusive lock
            wait (&r->canRead, &r->sl); // hence block
      r->nActive ++;                    // increment the number
                                        // of readers associated
                                        // with this shared
                                        // lock
      r->nPendingReads --;              // decrement the count
                                        // of waiting readers
      spin_unlock (&r->sl);             // release mutex
                                        // spin lock
}

void unlockShared (struct rwlock *r)
{
      spin_lock (&r->sl);
      r->nActive --;
      if (r->nActive == 0) {            // no other readers has
                                        // the shared lock
            spin_unlock (&r->sl);
            do_signal (&r->canWrite);   // wake up a single
                                        // writer
      } else
            spin_unlock (&r->sl);
}

void lockExclusive (struct rwlock *r)   // this operation blocks
                                        // if there is either an
                                        // exclusive or shared
                                        // lock on the resource
{
      spin_lock (&r->sl);
      r->nPendingWrites ++;             // increment the count
                                        // of waiting writers
      while (r->nActive)                // some one has a shared
                                        // lock associated with
                                        // the resource
            wait (&r->canWrite, &r->sl); // hence block
      r->nPendingWrites --;
      r->nActive = -1;                  // now a writer has an
                                        // exclusive lock
      spin_unlock (&r->sl);
}

void unlockExclusive (struct rwlock *r)
{
      boolean_t wakeReaders;
      spin_lock (&r->sl);
      r->nActive = 0;
      wakeReaders = (r->nPendingReads != 0);
      spin_unlock (&r->sl);
      if (wakeReaders)                  // true if there are
                                        // readers waiting
            do_broadcast (&r->canRead); // wake up all readers
```

```
        else
              do_signal (&r->canWrite);      // wake up a single
                                             // writer
}

void downgrade (struct rwlock *r)            // this operation
                                             // converts a exclusive
                                             // lock into a shared
                                             // lock
{
        boolean_t wakeReaders;
        spin_lock (&r->sl);
        r->nActive = 1;
        wakeReaders = (r->nPendingReads != 0);
                                             // true if there are
                                             // readers waiting for
                                             // the resource
        spin_unlock (&r->sl);
        if (wakeReaders)
              do_broadcast (&r->canRead);    // wake up all readers
}

void upgrade (struct rwlock *r)              // this operation
                                             // converts a shared
                                             // lock to an exclusive
                                             // lock
{
        spin_lock (&r->sl);
        if (r->nActive == 1) {               // no other reader
              r->nActive = -1;
        } else {
              r->nPendingWrites ++;
              r->nActive --;                 // release shared lock
              while (r->nActive)             // some one has a shared
                                             // lock associated with
                                             // the resource
               wait (&r->canWrite, &r->sl); // hence block
              r->nPendingWrites --;
              r->nActive = -1;
        }
        spin_unlcok (&r->sl);
}
```

The UNIX operating system's solution for the Readers/Writers problem is to wake up all the threads waiting for the resource [Vahalia 96]. This is clearly inefficient [Vahalia 96], if a writer acquires the lock next, all readers and other writers will have to go to sleep; if a reader acquires the lock, other writers will have to go to sleep. It is preferable to find a protocol that avoids needless wakeups [Vahalia 96]. If a reader releases a resource, it takes no action if other readers are still active. When the last active reader releases its shared lock, it must wake up a single waiting writer. When a writer releases its lock, it must choose whether to wake up another writer or other readers. If

writers are given preference, the readers could starve under heavy contention [Vahalia 96].

The preferred solution [Vahalia 96] is to wake up all waiting readers when releasing an exclusive lock. If there are no waiting readers, we wake up a single waiting writer. This scheme can lead to writer starvation [Vahalia 96]. If there is a constant stream of readers, they will keep the resource read-locked, and the writer will never acquire the lock. To avoid this situation, a lockShared() request must block if there is any waiting writer, even though the resource is currently only read-locked. Such a solution, under heavy contention, will alternate access between individual writers and batches of readers [Vahalia 96].

The upgrade() function that converts a shared lock to an exclusive lock must be used carefully in order to avoid deadlocks. A deadlock can occur unless the implementation takes care to give preference to upgrade requests over waiting writers [Vahalia 96]. If two threads try to convert a shared lock to an exclusive lock, each would block since the other holds a shared lock. One way to avoid that is for the upgrade() to release the shared lock before blocking, if it cannot get the exclusive lock immediately. This results in additional problems for the user, since another thread could have modified the resource before upgrade() returns. Another solution is for upgrade() to fail and release the shared lock if there is another pending upgrade.

3.4.2.3 Condition Variables

A conditional variable is a complex synchronization mechanism that has a predicate (a logical expression that evaluates to TRUE or FALSE) associated with it

based on some shared data [Vahalia 96]. It basically allows threads to block on it and provides facilities to wakeup one or all blocked threads when the result of the predicate changes. This mechanism is more useful for waiting on events than for resource locking. For example [Vahalia 96], let us assume that one or more server threads are waiting for clients requests. Incoming requests from the clients are to be passed to waiting threads or put on a queue. When a server thread is ready to process the next request, it first checks the queue. If there is a pending message, the thread removes it from the queue and services it. If the queue is empty, the thread blocks until a request arrives. This can be implemented by associating a condition variable with this queue. The shared data is the message queue itself, and the predicate is that the queue be nonempty.

On a multiprocessor, we need to guard against race conditions, such as the lost wakeup problem [Vahalia 96]. Suppose a message arrives after a thread checks the queue but before the thread blocks. The thread will block even though a message is available. We therefore need an atomic operation to test the predicate and block the thread if necessary. Condition variables provide this atomicity by using an additional mutex i.e., a spin lock (see Section 4.6). The mutex protects the shared data and avoids the lost wakeup problem. The server thread acquires the mutex on the message queue and then checks if the queue is empty. If so, it calls the wait() function of the condition with the spin lock held. The wait() function takes the mutex as an argument and atomically blocks the thread and releases the mutex. When the message arrives on the queue and the thread is woken up, the wait() call reacquires the spin lock before returning. The following is a sample implementation of a condition variable [Vahalia 96].

```
struct condition {

      proc *next;              // doubly linked list of blocked
      proc *prev;              // threads

      spinlock_t listlock;     // spin lock protects the list of
                               // threads
};

void wait (condition *c, spinlock_t *s)
{                              // acquire lock on the doubly linked
                               // list of blocked threads
      spin_lock (&c->listLock);
                               // add to the linked list
                               // release the lock on the blocked
                               // threads
      spin_unlock (&c->listLock);
      spin_unlock (s);         // release spin lock on the
                               // predicate before blocking
      swtch ();                // perform context switch
                               // when we return from the swtch(),
                               // the event has occurred
      spin_lock (s);           // acquire the spin on the predicate
      return;
}

void do_signal (condition *c)
                               // wake up one thread waiting on
                               // this condition
{
      spin_lock (&c->listLock);
                               // remove one thread from linked
                               // list, if it is nonempty
      spin_unlock (&c->listLock);
                               // if a thread was removed from the
                               // list, make it runnable
      return;
}

void do_broadcast (condition *c)
                               // wake up all threads waiting on
                               // this condition
{
      spin_lock (&c->listLock);
      while (linked list is nonempty) {
                               // remove a thread from the linked
                               // list and make it runnable.
      }
      spin_unlock (&c->listlock);
}
```

In the above implementation, the predicate itself is not part of the condition variable. It must be tested by the calling routine before calling wait(). The implementation also uses two separate mutexes. One is listLock, which protects the doubly linked list of threads blocked on the condition. The second mutex protects the tested data (predicate) itself. The spin lock mutex is not a part of the condition variable, but is passed as an

argument to the wait() function. The swtch() function and the code to make blocked threads runnable use a third mutex to protect the scheduler queues. We thus have a situation where a thread tries to acquire one spin lock while holding another. This is not disastrous since the restriction on spin locks is only that threads should not be allowed to block while holding one. Deadlocks are avoided by maintaining a strict locking order, the lock on the predicate must be acquired before listLock.

One of the major advantages of a condition variable is that it provides two ways to handle event completion [Vahalia 96]. When an event occurs, there is the option of waking up just one thread with do_signal() or all threads with do_broadcast(). Each may be appropriate in different circumstances. In case of a multithreaded server application, waking one thread is sufficient as each request will be handled by a single thread. However, consider several threads running the same program, thus sharing a single copy of the program text. More than one of these threads may try to access the same nonresident page of the text, resulting in page faults in each of them. The first thread to fault initiates a disk access for that page. The other threads notices that the read has already been issued and blocks waiting for the I/O to complete. When the page is read into memory, it is desirable to call do_broadcast() and wake up all the blocked threads, since at that point they can all access the page without conflict.

3.4.2.4 Sleep Locks

A sleep lock is a nonrecursive mutex lock that permits long-term locking of resources [UNIX 92]. One such example is, the resources that are utilized by a process

can be locked when the process blocks in a blocking operation. It is implemented as a variable of type sleep_t, and provides the following operations:

```
void SLEEP_LOCK (sleep_t *lockp, int pri);          // the processes acquires the
                                                    // lock over the resources by
                                                    // calling this function and
                                                    // this call cannot be
                                                    // interrupted
bool_t SLEEP_LOCK_SIG (sleep_t *lockp, int pri); // same as the above function
                                                    // but can be interrupted
void SLEEP_UNLOCK (sleep_t *lockp);                 // to unlock the resources
```

The pri parameter specifies the scheduling priority to assign to the process after it awakens. If a process blocks on a call to SLEEP_LOCK, it will not be interrupted by a signal. If it blocks on a call to SLEEP_LOCK_SIG, a signal will interrupt the process; the call returns TRUE if the lock is acquired and FALSE if the sleep was interrupted. The lock also provides other operations such as SLEEP_LOCK_AVAIL (checks if the lock is available), SLEEP_LOCKOWNED (checks if the caller owns the lock), and SLEEP_TRYLOCK (returns failure instead of blocking if the lock cannot be acquired).

CHAPTER IV

WINDOWS NT

4.1 Basic Architecture

The structure of Windows NT can be divided into two parts [Custer 93]: the user mode portion of the system and the kernel mode portion of the system. The Windows NT protected subsystems are collectively termed as the user mode portion, and the NT executive is termed as the kernel mode portion as shown in Figure 5.

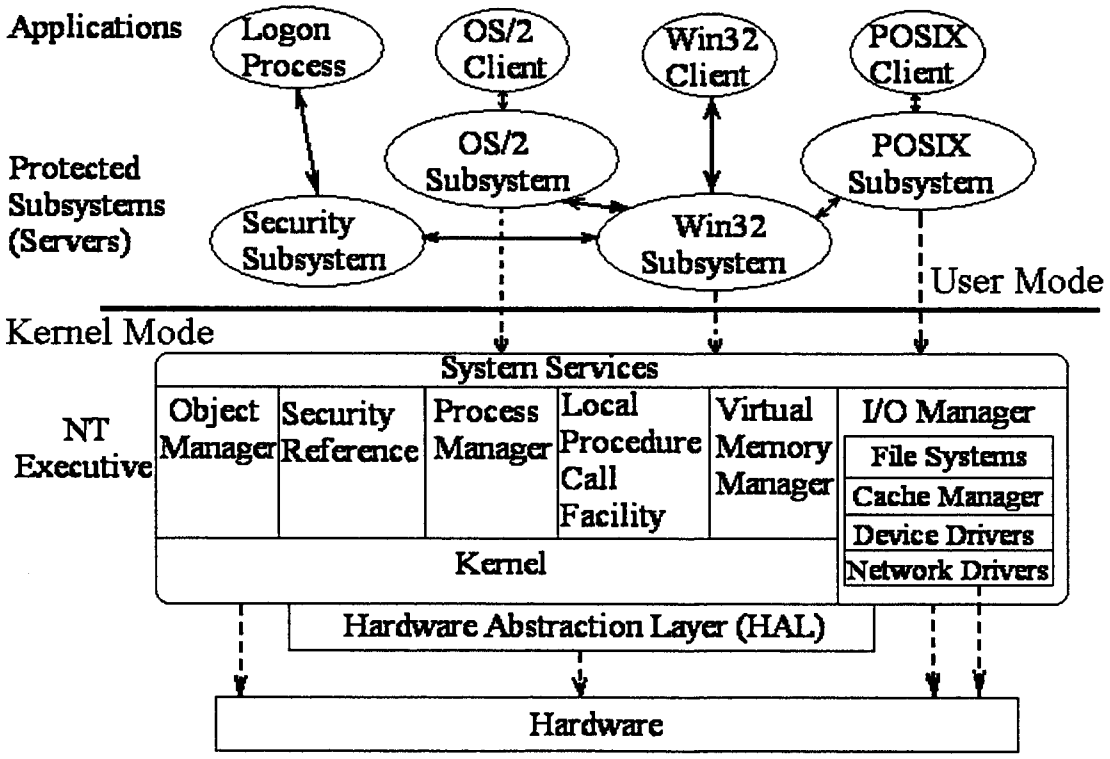

Figure 5. Block Diagram of Windows NT Operating System [Custer 93]

The protected subsystems are also called as the Windows NT servers, as each one of them resides in a separate process whose memory is protected from other processes by the NT executive's virtual memory manager. In Windows NT, the subsystems do not automatically share memory, rather they communicate by passing messages. The NT executive is the operating system engine. The following subsections discuss the protected subsystem and the NT executive.

4.1.1 Protected Subsystem

Windows NT protected subsystems provide API's (e.g., Win32's, OS/2, etc.,) that programs can call [Richter 93a]. When an application calls an API routine, a message is sent to the server that implements the API routine via the NT executive's local procedure call (LPC) facility. LPC is a locally optimized message-passing mechanism in which an application calls an API routine in a DLL (Dynamic Link Library) to which it is linked, and the DLL does the work necessary to send the message to the Windows NT protected subsystem (servers). The server replies by sending a message. The API routine in the DLL receives the message and hands it over to the application. The LPC facility is specific to Windows NT. The protected subsystem can further be divided into the environment subsystems and the integral subsystems, as defined below.

An environment subsystem is a user mode server that provides an API specific to an operating system. When an application calls an API routine, the call is delivered through the LPC facility to an environment subsystem. The chosen environment subsystem executes the API routine and returns the result to the application process by sending another LPC. The Win32 subsystem is the most important environment

subsystem, because it is the one that provides the Microsoft's 32-bit Windows API to the application programs. It also provides the NT's graphical user interface and controls all user input and application output.

Windows NT also provides several other environment subsystems to support each one of the following applications: POSIX, OS/2, 16-bit windows subsystem, and MS-DOS subsystem. All these subsystems still use the Win32 subsystem to receive user input and to display output.

The integral subsystems are the servers that perform major operating system functions. The security subsystem and the components of the networking software are some of the integral subsystems. The security subsystem runs in the user mode and records the security policies in effect on the local computer [Custer 93]. It keeps track of which user accounts have special privileges, it maintains a database of information about user accounts, and it also accepts user logon information and initiates logon authentication.

The NT networking component implements the following two services: Workstation services and the Server services [Custer 93]. Both of these are user mode processes that implements an API to access and manage the LAN Manager network re-director and server, respectively. The re-director is the network component responsible for sending I/O requests across a network when the file or device to be accessed is not local.

4.1.2 NT Executive

The NT executive is the kernel mode portion of Windows NT and, except for the user interface, is a complete operating system unto itself [Richter 93a]. Windows NT kernel is a part of the NT executive. The NT executive consists of a number of components, each of which implements two sets of functions: the system services which the environment subsystems and other executive components can call, and the internal routines which are available only to components within the executive.

NT executive also provides API-like system services, but it does not run continually in a process of its own. Rather, it runs in the context of an existing process by taking over an executing thread when important system events occur. When a thread calls a system service and is trapped by the processor, or when an external device interrupts the processor, the NT kernel gains control of the thread that was running. The kernel calls the appropriate system code to handle the event, executes it, and then returns control to the code that was executing before the interruption.

The Windows NT executive components maintain independence from one another, each creating and manipulating the system data structures it requires. The following are the executive components and their responsibilities. The Object manager creates, manages, and deletes NT executive objects, which are the abstract data types that are used to represent operating system resources. NT executive objects are objects (a single, runtime instance of a statically defined object type) implemented by various components of the NT executive. The security reference monitor enforces security policies on the local computer. The security reference monitor guards operating system resources, and performs run-time object protection and auditing. The process manager

creates and terminates processes and threads. It also suspends and resumes the execution of threads and stores and retrieves information about NT processes and threads. The local procedure call facility passes messages between a client process and a server process on the same computer. It is an optimized version of the remote procedure call i.e., all communications takes place in the local machine. The virtual memory manager implements virtual memory, which is a memory management scheme that provides a large, private address space for each process and protects each processes address space from other processes.

The NT kernel responds to interrupts (asynchronous events, that can occur at any time unrelated to what the processor is executing) and exceptions (a synchronous condition, resulting from the execution of a particular instruction), schedules threads for execution, synchronizes the activities of multiple processors, and supplies a set of objects and interfaces that the rest of the NT executive uses to implement higher-level objects [Custer 93]. The I/O system comprises a group of components responsible for processing input from and delivering output to a variety of devices. The following are the components of the I/O system [Richter 93a]: I/O manager, File systems, Network re-director and Network server, NT executive device drivers, and Cache manager.

The hardware abstraction layer (HAL) places a layer of code between the NT executive and the hardware platform on which Windows NT is running [Custer 93]. The hardware abstraction layer hides hardware-dependent details such as I/O interfaces, interrupt controllers, and multiprocessor communication mechanisms. Rather than access hardware directly, the NT executive components maintain maximum portability by calling the HAL routines when they need platform-dependent information.

Windows NT provides synchronization by means of wait and signal capabilities as part of the executive object architecture. In Windows NT, threads can synchronize by using the synchronization objects. In order to understand the synchronization objects it is necessary to know about the NT's process structure. The fundamental goal of the NT's process manager is to provide a set of native process services that environment subsystems can use to emulate their unique process structures [Custer 93]. This is how NT provides multiple operating system environments that can run in user mode.

4.2 Windows NT Processes and Threads

Windows NT processes have the following characteristics that are different from other operating systems [Custer 93]:

i) NT processes are implemented as objects and are accessed using object services (means for manipulating objects, usually read or change object attributes).

ii) An NT process can have multiple threads executing within its address space.

iii) Both process objects and thread objects have built-in synchronization capabilities (see Section 4.4).

iv) The NT process manager maintains no parent/child or other relationships among the processes it creates.

v) An NT process has to have at least one thread of execution.

NT processes can be in either one of the following two modes: kernel mode or user mode [Richter 93b]. In the kernel mode, processes can execute operating system code or can access operating system memory. The kernel mode processes run in the unrestricted processor mode. The processes that run under restricted processor mode are called user mode processes.

A thread, sometimes called as lightweight process, is the basic unit of scheduling in Windows NT. A thread shares with peer threads its code section, data section, and operating system resources such as open files and signals [Custer 93]. A thread's life cycle start when a program creates a new thread by calling the process manager. The process manager in turn, calls the object manager to create a thread. Similar to NT processes, threads can also be in either one of the following two modes [Richter 93b]: kernel mode or user mode.

A user mode thread gains access to the operating system by calling a system service (services provided by the components of NT executive for the environmental subsystem servers). When the thread calls the service, the processor traps it and switches its execution from user mode to kernel mode. The operating system takes control of the thread, validates the arguments the thread passed to the system service, and then executes the service. The operating system switches the thread back to user mode before returning control to the user's program. By following this procedure, the operating system protects itself and its data from modification by user mode threads. The following section discusses the Windows NT thread states.

4.3 Windows NT Thread States

A thread can be in any of six states at any given time, only one of which makes the thread eligible for execution [Custer 93]. The dispatcher states of a thread are illustrated in Figure 6 [Richter 93b]. Once initialized, the thread progresses through the following states:

i) Ready: When looking for a thread to execute, the dispatcher considers only the pool of threads that are in the ready state. These threads are simply waiting to execute.

ii) Standby: A thread in the standby state has been selected to run next on a particular processor. When the correct conditions exists, the dispatcher performs a context switch to this thread. Only one thread can be in the standby state for each processor in the system.

iii) Running: Once the dispatcher performs a context switch to a thread, the thread enters the running state and executes. The thread's execution continues until either the kernel preempts it to run a higher priority thread, its quantum ends, it terminates, or it voluntarily enters the waiting state.

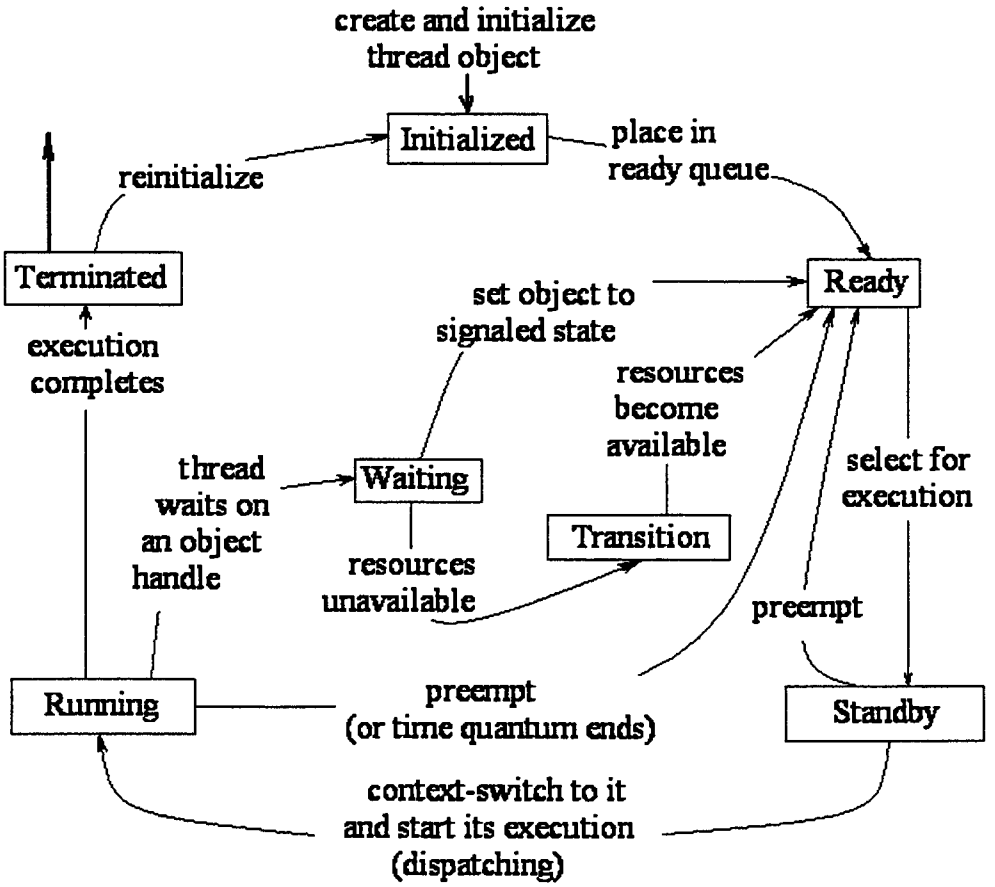

Figure 6. Windows NT Thread States [Richter 93b]

iv) Waiting: A thread can enter the waiting state in several ways: a thread can voluntarily wait on an object to synchronize its execution; the operating system (the I/O system, for example) can wait on the thread's behalf; or an environment subsystem can direct the thread to suspend itself. When the thread's wait ends, the thread moves back to the ready state to be rescheduled.

v) Transition: A thread enters the transition state if it is ready for execution but the resources it needs are not available. For example, the thread's kernel stack might have been paged out of memory. Once its resources are available, the thread enters the ready state.

vi) Terminated: When a thread finishes executing, it enters the terminated state. Once terminated, a thread object might or might not be deleted. If the executive has a pointer to the thread object, it can reinitialize the thread object and use it again.

We will first discuss the objects that provide synchronization for user mode threads, and then the objects that provide synchronization for kernel mode threads in case of having more than one processor.

4.4 User Level Synchronization

The following are the synchronization objects that are used by the user mode threads for synchronization in Windows NT [Custer 93]:

i) Process objects
ii) Thread objects
iii) File objects
iv) Event objects
v) Event pair objects
vi) Semaphore objects
vii) Timer objects
viii) Mutant objects

The first three objects listed serve other purposes in addition to synchronization, but the last five objects are just for synchronization purposes. At any given moment, a synchronization object is in one of two states, either signaled or the non-signaled state.

The signaled state is defined differently for different objects. A thread object is in the non-signaled state during its lifetime, and is set to the signaled state by the NT kernel when the thread terminates. Similarly, the kernel sets a process object to the signaled state when the process last thread terminates. In contrast, the timer object, like a stopwatch, is set to go off at a certain time. When its time comes up, the kernel sets the timer object to the signaled state. The following are the Windows NT objects that does not support synchronization [Custer 93]: section, port, access token, object directory, symbolic-link, profile, and key objects.

To synchronize with an object, a thread calls one of the wait system services supplied by the object manager, passing a handle to the object it wants to synchronize with. The thread can wait on one or several objects and can also specify to the kernel that its wait should be canceled if it is not ended within a certain amount of time. Whenever

Table II. Definitions of Signaled State for Synchronization Objects [Custer 93]

Object Type	Set to Signaled State When	Effect on Waiting Threads
Process	Last thread terminates	All released
Thread	Thread terminates	All released
File	I/O operation completes	All released
Event	Thread sets the event	All released
Event pair	Dedicated client or server thread sets the event	Other dedicated thread released
Semaphore	Semaphore count drops to zero	All released
Timer	Set time arrives or time interval expires	All released
Mutant	Thread releases the mutant	One thread released

the kernel sets an object to the signaled state, it checks to see whether there are any threads waiting on the object. If so, the kernel releases one or more of the threads from their waiting state so that they can continue executing. Table II [Davis 94] shows the effects on the waiting threads when a user mode object is set to the signaled state.

When an object is set to the signaled state, the waiting threads are generally released from their wait states immediately. For example, an event object is used to announce the occurrence of some event. When the event object is set to the signaled state, all threads waiting on the event are released. The exception is any thread that is waiting on more than one object at a time; such a thread might be required to continue waiting until additional objects reach the signaled state. From Table II it is clear that except event pair object and mutant object, all other objects release all the threads while shifting to the signaled state.

Windows NT's executive synchronization semantics are visible to Win32 programmers through the WaitForSingleObject() and WaitForMultipleObjects() API routines [Davis 94], which the Win32 subsystem implements by calling analogous system services supplied by the NT object manager. A thread in a Win32 application can synchronize with a Win32 process, thread, event, semaphore, mutex, or file object.

As an example, let us assume that a user is running a spreadsheet application program under the Windows NT operating system. The application has a main thread that performs ordinary spreadsheet functions and a secondary thread that spools spreadsheet files to the printer. Now suppose the user prints a spreadsheet and, before spooling is completed, enters a command to exit the program. The main thread, which accepts the exit request, doesn't terminate the process immediately. Instead, it calls the

WaitForSingleObject() routine to wait for the spooler thread to finish spooling and terminate. After the spooler thread terminates, the main thread is released from its wait operation and terminates itself, which ends the spreadsheet program and terminates the spreadsheet process.

4.5 Kernel Level Synchronization

Following the foot steps of the recently developed operating systems such as Mach, OS/2 etc., Windows NT also separates the operating system's mechanisms from its policies. The principle of separating policies from mechanisms exists at several levels in Windows NT [Davis 94]. At the highest level, each environment subsystem establishes a layer of operating system policies that differs from that of other subsystems. At the kernel level it avoids policy-making altogether. The kernel performs four main tasks [Custer 93]:

i) Schedules threads for execution.

ii) Performs low-level multiprocessor synchronization.

iii) Transfers control to handler routines when interrupts and exceptions occur.

iv) Implements system recovery procedures after a power failure occurs.

Windows NT is a preemptive multitasking system, thus the operating system does not wait for a thread to voluntarily yield the processor to other threads. Instead, the operating system interrupts a thread after the thread has run for a preset amount of time, called the time quantum, or when a higher priority thread is ready to run.

Windows NT processes are multithreaded. The kernel uses a priority-based scheme to select the order in which threads are executed. The kernel also changes a thread's priority periodically to ensure that all threads will execute. Outside the kernel,

the executive presents threads and other shareable resources as objects. These objects require some policy overhead, such as object handles (an index into the process-specific table that contains pointers to all the objects that the process has opened a handle to) to manipulate them [Custer 93], security checks to protect them, resource quotas to be deducted when they are created, etc. This overhead is eliminated in the kernel, which implements a set of simpler objects, called kernel objects, that help the kernel control central processing and support the creation of executive objects. Kernel objects are a more primitive set of objects implemented by the NT kernel. These objects are not visible to user mode code but are created and used only within the NT executive.

Kernel objects provide fundamental capabilities, such as the ability to alter system scheduling, that can be accomplished only by the kernel. One set of kernel objects, called the dispatcher objects, incorporates synchronization capabilities and alters or affects thread scheduling. The dispatcher objects include kernel thread, kernel mutex, kernel mutant, kernel event, kernel event pair, kernel semaphore, and kernel timer. The Windows NT dispatcher also takes care of context switching, which is the procedure of saving the volatile machine state associated with a running thread, loading another thread's volatile state, and starting the new thread's execution. In the following section we will discuss multiprocessor synchronization and kernel dispatcher objects in detail.

4.6 Multiprocessor Synchronization

Synchronization is a major issue for symmetric multiprocessing operating systems. Analogous to Solaris 2.5 a variant of UNIX, Windows NT is a symmetric multiprocessing operating system. The Windows NT kernel guarantees mutual exclusion

in the case of having multiple processors by utilizing a mechanism called spin lock. The kernel's critical sections are the code segments that modify a global data structure such as the kernel's dispatcher database or its DPC (Deferred Procedure Call) queue.

Before entering the critical section shown in the Figure 7 [Custer 93], the kernel must acquire the spin lock associated with the protected DPC queue. If the spin lock is not free, the kernel keeps trying to acquire the lock until it succeeds. The spin lock is called so because the kernel is held in limbo "spinning" until it gets the lock [Richter 93a].

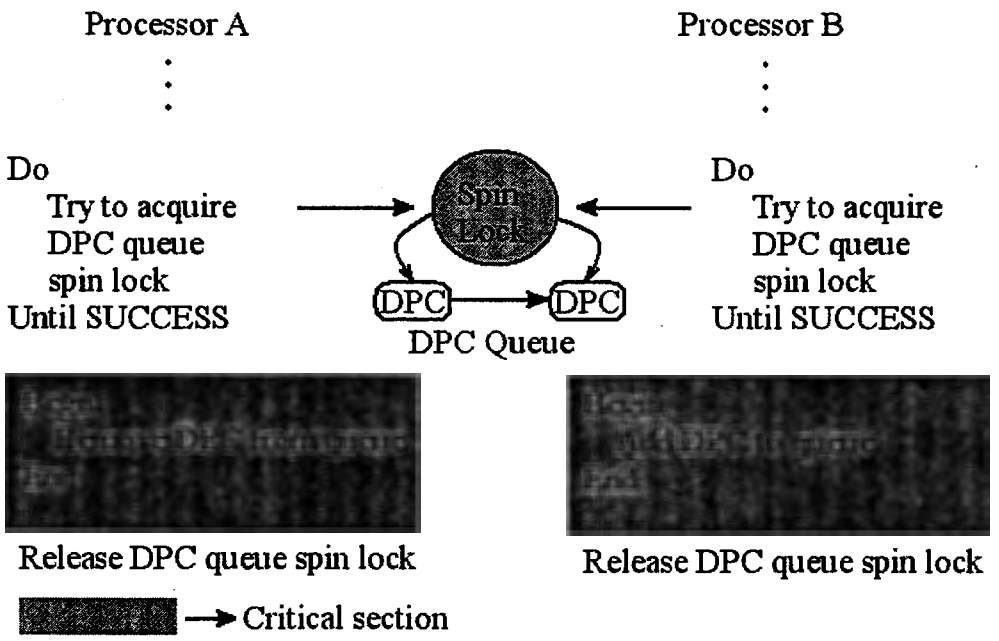

Figure 7. Two Processors Competing for Spin Lock [Custer 93]

Spin locks, like the data structures they protect, reside in global memory. The code to acquire and release a spin lock is written in the host assembly language for speed and to exploit whatever locking mechanism the underlying processor architecture provides.

On many architectures, spin locks are implemented with a hardware-supported test-and-set operation, which tests the value of a lock variable and acquires the lock in one atomic instruction. Testing and acquiring the lock in one instruction prevents a second thread from grabbing the lock between the time the first thread tests the variable and the time it acquires the lock.

When a thread is trying to acquire a spin lock, all other activity ceases on that processor. Therefore, a thread that holds a spin lock is never preempted and is allowed to continue executing so that it will release the lock quickly. The kernel executes minimum number of instructions while it holds a spin lock [Custer 93].

The Windows NT kernel makes spin locks available to other parts of the executive through a set of kernel functions. Device drivers, for example, utilizes spin locks in order to guarantee that the global data structure is accessed by only one part of a device driver at a time.

The executive software outside the kernel also needs to synchronize access to global data structures in a multiprocessor environment. Spin locks only partially fill the executive's needs for synchronization mechanisms. Waiting on a spin lock literally stalls a processor, spin locks can be used only under the following strictly limited circumstances [Custer 93]:

i) The protected resource must be accessed quickly and without complicated interactions with other code.

ii) The critical section code cannot be paged out of memory, cannot make references to pageable data, cannot call external procedures, and cannot generate interrupts or exceptions.

These restrictions cannot be met under all circumstances. The executive needs to perform other types of synchronization in addition to mutual exclusion and it must also provide synchronization mechanisms to the user mode processes.

The Windows NT kernel provides additional synchronization mechanisms to the executive in the form of kernel objects, known collectively as dispatcher objects. A thread can synchronize with a dispatcher object by waiting on the object's handle. Doing so causes the kernel to suspend the thread and change its dispatcher state from running to waiting as shown in Figure 6. The kernel removes the thread from the dispatcher ready queue and no longer considers it for execution. A thread cannot resume its execution until the kernel changes its dispatcher state from waiting to ready. This change occurs when the dispatcher object, whose handle the thread is waiting on, also undergoes a state change, from the non-signaled state to the signaled state. The kernel is responsible for both types of transitions. The kernel dispatcher objects and the system events that induce their state changes are shown in Figure 8 [Custer 93].

Each type of dispatcher object provides a specialized type of synchronization capability. For example, mutex objects provide mutual exclusion, whereas semaphores act as a gate through which a variable number of threads can pass useful information when a number of identical resources are available. Events can be used either to announce that some action has occurred or to implement mutual exclusion. A thread can wait on another thread to terminate, which is useful for synchronizing the activities of two cooperating threads. Together, the kernel dispatcher objects provide synchronization facility for the Windows NT executive.

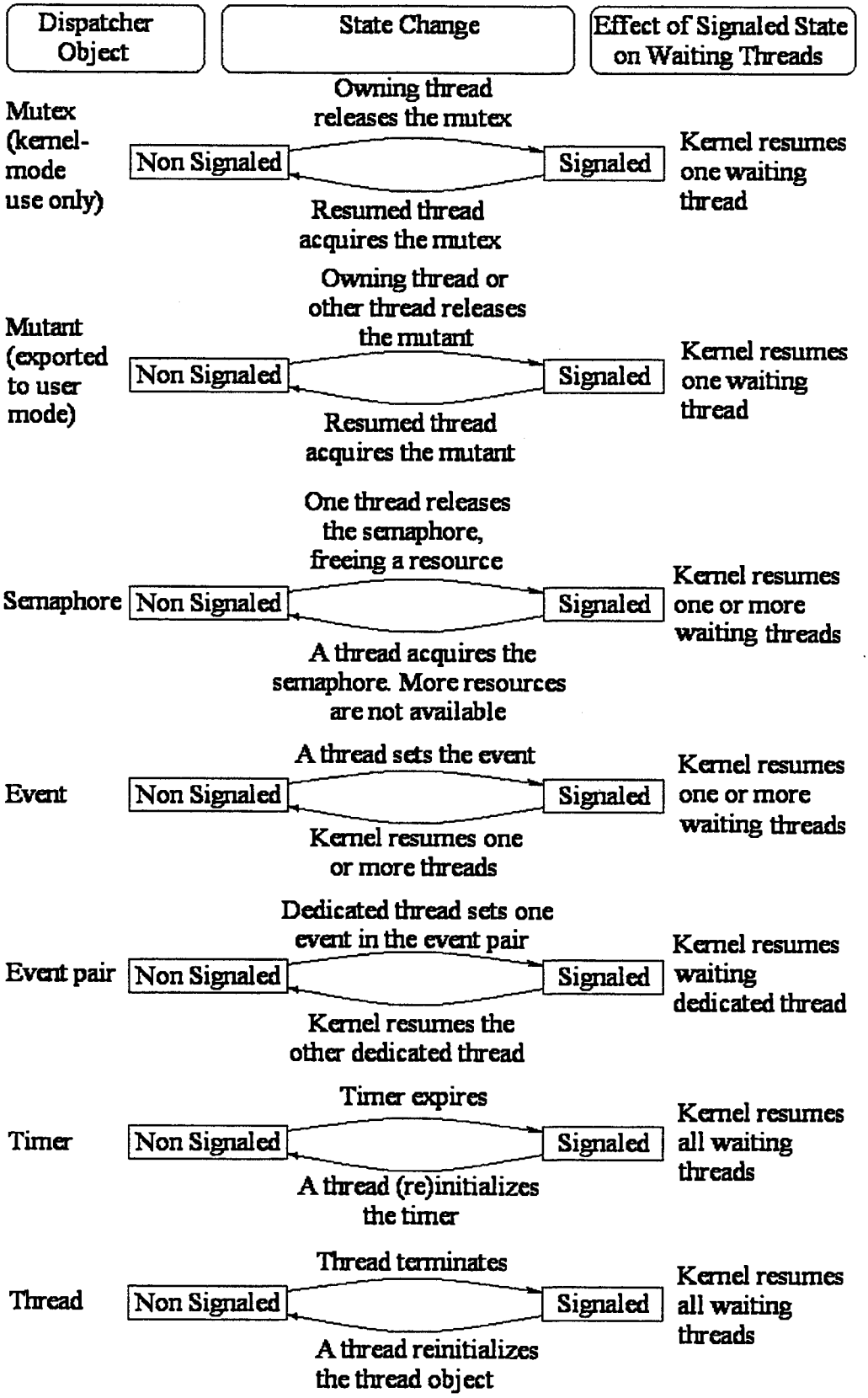

Figure 8. Dispatcher Object-State Changes

The user-visible synchronization objects acquire their synchronization capabilities from kernel dispatcher objects. Each user-visible object that supports synchronization encapsulates at least one kernel dispatcher object. The following example of setting an event illustrates how synchronization interacts with thread dispatching [Custer 93]:

i) A user mode thread waits on an event object's handle.

ii) The kernel changes the thread's scheduling state from ready to waiting and then adds the thread to a list of threads waiting for the event.

iii) Another thread sets the event.

iv) The kernel marches down the list of threads waiting on the event. If a thread's conditions for waiting are satisfied, the kernel changes the thread's state from waiting to ready. If it is a variable priority thread, the kernel might also boost its execution priority.

v) Because a new thread has become ready to execute, the dispatcher reschedules. If it finds a running thread with a lower priority than that of the newly ready thread, it preempts the lower priority thread, issuing a software interrupt to initiate a context switch to the higher priority thread.

vi) If no processor can be preempted, the dispatcher places the ready thread in the dispatcher ready queue to be scheduled later.

CHAPTER V

APPLE MACINTOSH

5.1 Basic Architecture

The Apple Macintosh operating system provides routines that allow a user/programmer to perform basic low-level tasks such as file input and output, memory management, and process and device control [Apple 96g]. The block diagram shown in Figure 9 shows the basic architecture of the Apple Macintosh operating system [Apple 96h].

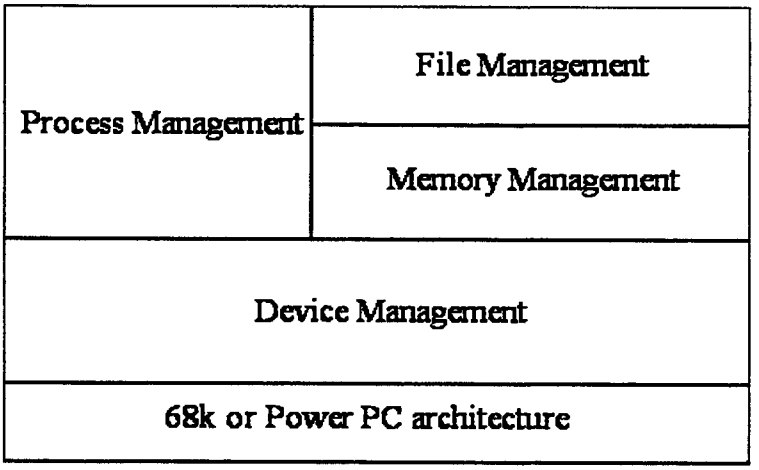

Figure 9. Apple Macintosh Operating System Layers

The 68K and the Power PC are the two different hardware architectures supported by the Apple Macintosh operating system [Apple 96e]. The 68K microprocessors are

manufactured by Motorola. The Power PC microprocessors are manufactured by a collaboration of Apple Computers, Motorola, and IBM. Power PC architecture also supports other major PC operating systems such as Windows NT and Windows 95.

The Device management block shown in Figure 9 constitutes the Device manager, Slot manager, Small Computer System Interface (SCSI) manager, Apple Desktop Bus (ADB) manager, Power manager, and the Serial driver manager [Apple 96a]. The Device manager acts as an interface for all other blocks to interact with the hardware, thus it provides input from and output to the hardware. On system startup the Slot manager examines each slot and initializes any expansion cards it finds. The Slot manager maintains data structures that contain information about each slot and every available system resource, and it provides functions that allow an application developer to get information about expansion cards and their system resources. The Small Computer System Interface (SCSI) manager is a software layer that mediates between device drivers or applications and the SCSI controller hardware in the Apple Macintosh computer.

The Apple Desktop Bus (ADB) manager allows the application developer to get information about and communicate with hardware devices attached to the Apple Desktop Bus. The Apple Desktop Bus is a low speed bus that connects input devices, such as keyboards, mouse devices, and graphic devices, to an Apple Macintosh computer or to other hardware equipment. The Power manager software controls power to the internal devices of portable Apple Macintosh computers. The Serial driver in the Device management block provides low level support for asynchronous, interrupt driven serial data transfers through the modem and printer ports.

The Process management block constitutes the Process manager, Time manager, Vertical retrace manager, Notification manager, Deferred task manager, Segment manager, and Shutdown manager [Apple 96i]. The Process manager handles the launching, scheduling, and termination of applications. It also provides information about open processes (process that are initialized and have not terminated). The Time manager allows a developer to execute a routine periodically or after a specified time delay. The Vertical retrace manager allows a developer to synchronize the execution of an application with the redrawing of the screen. The Notification manager provides notification service. The Notification manager allows applications running in the background to communicate information to the user. The Deferred task manager maintains a deferred task queue of records where each record is a deferred task. A deferred task is an interrupt that will take a long time to process and hence will block all other interrupts of the same or lower priority when it is executed. The Deferred task manager executes interrupts (deferred tasks) whenever there is no other interrupt to process i.e., the interrupt priority level is zero.

The Process Manager loads code segments into memory when an application is launched. The Segment Manager loads code segments whenever any externally referenced routine containing those code segments are called for. Both of these operations occur completely automatically and rely on information stored in the application and in the individual code segments themselves. A segment is locked when it is first read into memory and at any time thereafter when routines in the segment are being executed. This locking prevents the block from being moved during compaction and purging of the applications memory. The Segment manager also has an unload utility to unload the

loaded segments. The Shutdown manager allows a programmer to execute a routine while the computer is shutting down or restarting.

The Memory management block constitutes the Memory manager, Virtual memory manager, and Memory management utility [Apple 96f]. The Memory manager manages the dynamic allocation and release of memory in the application's memory partition. The Virtual memory manager provides virtual memory services, i.e., it provides the ability to have a logical address space that is larger than the total amount of available RAM. The Memory management utility is used to ensure the following [Apple 96f]:

i) the applications call back routines, interrupt tasks, and stand alone codes can access the applications global variables.

ii) the application or driver functions properly in both 24 and 32 bit modes.

iii) the data and instructions in the microprocessor's internal cache remain consistent with data and instructions in the RAM.

The File management block constitutes the File manager, Standard file package manager, Alias manager, and Disk initialization manager [Apple 96b]. The File manager provides access to the file system and allows applications to create, open, read, write, and close files. The Standard file package provides routines that handle the interface between the user and the application when the user saves or opens a document. The Alias manager helps to locate specified files, directories, or volumes. The Disk initialization manager manages the process of initializing disks.

The Apple Macintosh operating system doesn't have a kernel like the ones available in UNIX and Windows NT [Apple 96h]. So there is no such thing as kernel synchronization, but the Apple Macintosh operating system supports multitasking by using the cooperative multitasking mechanism as explained in the next section.

5.2 Cooperative Multitasking

The process manager in the Apple Macintosh operating system resolves the time sharing problem that arises while supporting the multitasking feature [Apple 96i]. This is achieved by cooperative multitasking between applications. In cooperative multitasking systems, each application has to surrender its CPU time to the system at regular intervals, the system then mediates the distribution of processor time to various other applications [May and Whittle 95]. Preemptive multitasking is also made possible in the Apple Macintosh operating system by the Thread manager API [Apple 96j]. Preemptive multitasking is made possible only in the case of the 68k architecture.

Cooperative multitasking doesn't have much in the way of synchronization. It is the programmer's responsibility to make sure that their program is in a suitable state before the program surrenders its CPU time to the system. Basically, the event manager and process manager maintain the cooperative multitasking environment in an Apple Macintosh operating system.

5.3 Processes and Events

In the Apple Macintosh operating system, a process is an open application or, an open desk accessory (small applications that can be opened from the Apple menu in an Apple Macintosh system). The number of processes that can be executed by the operating system is limited only by the available memory [Apple 96i]. The process manager maintains information about each process. It maintains the current state of each process, the address and size of its partition, its type, its creator, a copy of all process specific information such as global system variables, information about its resources, and a

process serial number (similar to process id in UNIX). The process manager assigns a process serial number to identify each process. This number is unique during a single boot of the local machine [Apple 96i]. The process specific information is referred to as the context of each process.

When an application is first launched, it starts executing as a foreground process [Apple 96i]. In the Apple Macintosh operating system, a foreground process has control of the CPU and other system resources, but it can relinquish control of the CPU if there are no events pending for it to process. A process that is open but is not currently a foreground process is said to be a background process.

A context switch can be of two types [Apple 96i]: major switch and minor switch. A major context switch is a complete switch, the application's windows are moved from the back to the front, or vice versa. In a major switch, two applications are involved; the one being switched to the foreground and the one being switched to the background. A minor switch occurs when the process manager gives time to a background process without bringing the background process to the front. The two processes involved in a minor switch can be two background processes or a foreground process and a background process.

Events are usually divided into three categories [Apple 96d]: low-level events, operating-system events, and high-level events. The event manager returns low-level events to applications for occurrences such as the user pressing the mouse button, releasing the mouse button, pressing a key on the keyboard, or inserting a disk [Apple 96d]. The event manager also returns low-level events to the applications if the applications needs to activate a window or update a window. When an application

requests an event and there are no other events to report, the event manager returns a null event.

The event manager returns the operating-system events to the application when the processing status of the application is about to change or has changed. For example [Apple 96d], if a user brings an application to the foreground, the process manager sends an event through the event manager to the application. Some of the work of reactivating the application is done automatically, both by the process manager and by the window manager. The application must take care of any further processing needed as a result of the application being reactivated. The event manager returns high-level events to the application as a result of communication directed to the application from another application or process.

5.4 Thread Manager

In the Apple Macintosh operating system, a thread is defined as a separate process running inside the application space i.e., the memory space occupied by the application [Apple 96j]. This is directly analogous to an application running as a process inside the computer space. It is possible to have several applications sharing memory. With a thread manager, multiple threads can work simultaneously inside an application [Apple 96j]. The thread manager is a simple implementation of concurrent processing within a single application.

5.4.1 Concurrency

Concurrency is a series of processes running simultaneously in a single memory space [Apple 96j]. MultiFinder, introduced in the Apple Macintosh operating system

Version 6.0, brought higher level of concurrency to the Macintosh. Starting from Version 6, the process manager implements cooperative multitasking [Apple 96j]. The Process manager basically depends upon each application to cooperatively surrender time to the system, it then mediates the distribution of processor time to other applications.

At the thread manager level, concurrency means that an application process is divided into simpler sub-processes that run concurrently inside the same application. Each of these sub-processes in the application follows the cooperative multitasking as followed by the application itself. Threads are of two types [Apple 96j]: cooperative and preemptive. The cooperative and preemptive threads exists inside the application.

A multithreaded process is associated with one or more threads. Codes that operate only within an application can use the thread manager. Cooperative threads allow cooperative multitasking. Operationally, cooperative threads yield to other cooperative threads only when the application explicitly makes one of the thread manager yield calls or changes the state of the current cooperative thread. A thread can be in one of the following three states [Apple 96j]: running, ready, or stopped.

Preemptive threads allow true multitasking at the application level. When an application gets control from the process manager, preemptive threads for that application are allowed to run. Preemptive threads differ from cooperative threads because they can interrupt the currently executing thread at any time to resume execution. If the interrupted cooperative thread is in the stopped state when the preemptive thread yields to the system, the next available preemptive thread is scheduled to run [Apple 96j]. Preemptive threads then preempt each other in a round-robin fashion until the interrupted cooperative thread is made ready.

Thread implementation is different for the 68k architecture and the Power Macintosh computers [Apple 96j]. Each thread has an associated data storage that includes the program counter, the registers, and the function stack. The stored stack is not swapped in and out with the main application stack. The stored stack is an independent stack, particular to the thread [Apple 96j]. The thread manager treats the main application as a separate thread.

Although the thread manager preserves each thread's context, threads can work on shared data within the application but outside the threads context. If two or more threads operate on the same data, then the data is at risk. Suppose we have a thread in an application filtering part of an image, at this time a user erases that part of the image. We don't want this to happen. In such cases, the application developer has to provide enough protection using a semaphore or some other locking mechanism. So, it is the developer's responsibility to make sure that his/her data will always be in a consistent state. So all synchronization issues are handled by the developer and not by the operating system. The developer has full control over his/her application as the Apple Macintosh operating system follows the cooperative multitasking mechanism.

In the Apple Macintosh operating system, threads or processes do not have a priority associated with them [Apple 96j]. Basic scheduling unit is the application. Inside the application, the developer can share the processor's time among different threads or yield the processor time to the system voluntarily. The scheduling of the processes and threads inside the application space is done based on the round robin mechanism [Apple 96j]. If an event is waiting, the main application thread receives control. The developer can create threads when needed, or can create a pool of threads and withdraw a thread

from the pool when it is necessary. This mechanism allows the developer to pre-allocate threads at a time when memory is not fragmented.

5.6 Multiprocessing

DayStar Digital developed the Apple Multiprocessing API under a contract signed with Apple Computers, Inc. [Cooksey 96]. Mac OS compatible computers that are compliant with the Apple multiprocessing specifications, have one main processor and one or more attached PowerPC processors. The main processor runs all applications and the Mac OS. The Apple Multiprocessing API provides a set of calls that allow an application to create separate threads of execution called tasks. Tasks are preemptively scheduled on the available processors in the system, even if there is only one.

The command MPProcessors is used to count the number of processors in the system [Cooksey 96]. If there is only one processor, the application may proceed as though the multiprocessing service is not available. However, the developer can still create preemptive tasks in a single processor environment. The count returned by the MPProcessors is usually used as an indication of how many tasks to create. While designing an application for a multiprocessing Mac OS, it is the developer's responsibility to make sure that the application will strive to keep all the processors busy. The simplest way to do this is to create at least as many tasks as there are processors [Cooksey 96]. The application then splits the work to be done into that many pieces and asks each task to work on a piece. An alternative and frequently-adopted technique is to create one less task than there are processors.

Communication among the application and the tasks occurs in two basic ways [Cooksey 96]: shared memory and synchronization methods. Since all memory is shared, anything the application writes into memory is available to the tasks and vice versa. However, before a task tries to access the memory space occupied by the application, the task must synchronize with the application using one of the three methods available in the Multiprocessing API Library [Cooksey 96]: queues, semaphores, and critical regions. Queues are first-in-first-out queues of 96-bit messages, inserting and extracting elements is an atomic operation. Many tasks can try to extract the next message from a given queue but only one will successfully obtain it. Semaphores represent a single 32-bit value that can be atomically incremented up to a predetermined maximum and atomically decremented to a minimum of zero. Critical regions prevent sections of code that they encompass from being executed by more than one task including the application at once.

The PowerPC architecture allows for writes to memory to be deferred [Cooksey 96]. This is a resource management feature that helps the PowerPC achieve its tremendous speed (350 MHz). In order for another processor to see the correct values in memory, certain hardware dependent instructions need to be executed. When a task uses a synchronization method, these instructions are executed, thus ensuring that the processors involved have a consistent view of memory from that point on. It is also important to use synchronization methods so that when one of the communicants is not yet ready to synchronize for some reason, the other one can yield the processor it is on. This makes the processor immediately available to some other task that may be able to make more productive use of it.

Before creating tasks it is usually a good idea to create the means by which to synchronize them with. Queues and semaphores are the two most common methods used. Semaphores are quicker and less memory intensive but do not offer the same degree of flexibility as Queues. Queues and semaphores are usually created in pairs [Cooksey 96]: one by which to signal a request and the other by which to signal results. If a developer creates only one synchronization object and try to use it for both purposes, it will not work. After a request is posted, the application will at some point start waiting for results. If it waits at the same place the request was posted, the request itself may appear to be the result. Since the application clears the request in the mistaken belief that it was a result, no work gets done. So to have the work done successfully, it is important to use two distinct synchronization objects for two-way communication.

CHAPTER VI

COMAPARATIVE EVALUATION

6.1 Comparison

A detailed comparative study on how synchronization is achieved in UNIX, Windows NT, and Apple Macintosh operating system was carried out and the results were tabulated as shown in Table III.

Table III consists of four columns; they are Property, UNIX, Windows NT and Apple Macintosh. For each property in the first column, the remaining three columns contains a Yes, No or NA with a brief explanation. This comparison is carried out based on the information collected in Chapters III, IV and V. Table III is followed by a section on observations concerning this study.

Table III. Comparison of UNIX, Windows NT, and Apple Macintosh Operating Systems Based on Synchronization.

Property	UNIX	WINDOWS NT	APPLE MACINTOSH
Is it a preemptive multitasking operating system?	Yes, recent versions of UNIX such as SVR 4.2, Solaris 2.5 are preemptive multitasking operating systems.	Yes, Windows NT is a Preemptive multitasking operating system.	No, Apple Macintosh is a cooperative multitasking operating system. For details see Section 5.2.
Is thread the basic unit of scheduling?	Yes, in Solaris 2.5 (a variant of UNIX) the basic unit of scheduling is thread.	Yes, in Windows NT the basic unit of scheduling is thread.	No, in Apple Macintosh the basic unit of scheduling is process.
During a blocking operation does the operating system safeguard the data that is in an inconsistent state?	Yes, in case of a blocking operation the UNIX kernel ensures that other processes do not access the data that is in an inconsistent state.	Yes, Windows NT kernel also ensures the same as UNIX.	No, it is the developer's responsibility to lock the data that may be in an inconsistent state.
Does the operating system have functions like sleep() and switch() to block the processes and to initiate a context switch in order to allow another process to execute?	Yes, when the resource is not available, the process blocks itself by calling the sleep() function. After this it calls the switch() function to initiate context switch in order to allow another process to execute.	Yes, Windows NT uses different function calls, but basically does the same operation as UNIX.	Yes, Apple Macintosh also has similar functions which can be called by the current process. The process blocks itself and allows the operating system to schedule the next process in a round robin fashion.

Table III. (Continued) Comparison of UNIX, Windows NT, and Apple Macintosh Operating Systems Based on Synchronization.

Does the operating system raise the interrupt priority level (ipl) in order to block interrupts while accessing the critical section?	Yes, interrupts are blocked while accessing the critical section by raising the *ipl.*	Yes, Windows NT also follows the same procedure as UNIX to block interrupts.	No, the developer has to program the application in such a way that the application by itself will check for events by polling the event manager and yield the CPU back to the operating system so that it can schedule the interrupt routine.
Does the operating system provide kernel synchronization objects?	Yes, Solaris 2.5 (variant of UNIX) uses various kernel synchronization objects such as mutex, locks, and semaphores in order to enhance synchronization while handling interrupts.	Yes, Windows NT also uses other objects to enhance the synchronization in case of interrupts.	NA
Do the processes have a parent/child relationship among them?	Yes, UNIX processes have a parent/child relationship.	No, there is no such relationship among processes.	No, there is no such relationship among processes.
Is it a symmetric multiprocessing operating system?	Yes, UNIX is a symmetric multiprocessing operating system.	Yes, Windows NT is a symmetric multiprocessing operating system.	No, Apple Macintosh follows a Master-Slave relationship among processors to provide the multiprocessing capability.

Table III. (Continued) Comparison of UNIX, Windows NT, and Apple Macintosh Operating Systems Based on Synchronization.

Does the operating system provide a pool of pre-allocated threads in order to reduce the time involved in processing the interrupts?	Yes, Solaris 2.5 (variant of UNIX) maintains a pool of interrupted threads, which are pre-allocated and partially initialized. By default, this pool contains one thread per interrupt level for each CPU, plus a single thread for the clock. These threads use the same synchronization primitives as other threads, and thus can block if they need a resource held by another thread. The kernel blocks interrupts only in a few exceptional situations such as when acquiring the mutex lock that protecting a sleep queue.	No.	No, but Apple Macintosh provides such a pool of pre-allocated threads for entirely different purpose.
Does the operating system use Spin Locks for achieving low level synchronization?	Yes, Spin Locks are used for short term low level synchronization mechanisms.	Yes, Windows NT also provides similar locks.	NA
Does the operating system provide any high level synchronization objects?	Yes, Solaris 2.5 (variant of UNIX) provides semaphores, reader-writer locks, and condition variables as high level synchronization objects.	Yes, Windows NT also provides some, they are process object, thread object, file object, event object, event pair object, semaphore object, timer object, and mutant object.	Yes, Apple Macintosh does provide some, they are semaphores and queues.

Table III. (Continued) Comparison of UNIX, Windows NT, and Apple Macintosh Operating Systems Based on Synchronization.

Does the operating system provide a set of macros to handle interrupts?	Yes, Table I shows the different macros to handle interrupts. These macros help to provide synchronization by raising the *ipl*, and by restoring the *ipl* to its previous value and so on.	Yes, Windows NT also provides similar macros.	NA
Does the operating system utilize the hardware instructions sets in order to provide synchronization?	Yes, different variants of UNIX utilize different hardware instruction sets to provide synchronization, but they all are similar to test-and-set and conditional-store instructions.	Yes, Windows NT also provides synchronization based on hardware instruction sets available on the particular processor on which it runs.	Yes, multiprocessing API developed in collaboration with DayStar Digital uses the hardware instructions sets available in 68k and PowerPC processors in order to provide proper synchronization in case of multiple processors.
Does the operating system provide any provision to remove memory leaks?	Yes, UNIX has some facilities similar to the ones available in Windows NT to remove memory leaks.	Yes, Windows NT provides the termination handlers and exception handlers in order to remove memory leaks caused by abnormal termination of processes while having a memory location locked.	NA

Table III. (Continued) Comparison of UNIX, Windows NT, and Apple Macintosh Operating Systems Based on Synchronization.

Can a thread synchronize with various other objects?	Yes, but UNIX doesn't provide many options as the Windows NT does to provide thread synchronization.	Yes, a thread can synchronize with the executive processes, threads, files, events, event pairs, semaphores, mutant, and timer objects.	NA
Is there any set of objects that a thread cannot synchronize with?	Yes, UNIX has objects that do not support synchronization.	Yes, in Windows NT section, port, access token, object directory, symbolic-link, profile, and key objects do not support synchronization.	NA
Can a thread specify the operating system that its wait should be canceled if it is not ended within a certain amount of time?	Yes, UNIX threads can specify that its wait should be canceled if it is not ended within a certain amount of time.	Yes, in Windows NT a thread can wait on several objects and can also specify that its wait should be canceled if it is not ended within a certain amount of time.	NA

6.2 Observations

The following observations were made in this comparative study: The UNIX operating system is the most reliable of the three operating systems taken into consideration, Windows NT comes next and then the Apple Macintosh operating system. Windows NT is becoming increasingly popular because it helps to build small business machines at a low cost, and it also has an attractive user interface. Windows NT is popular because it is easy to maintain and administer both as a server and as a client operating system. Windows NT and recent versions of UNIX, such as SVR 4.2 and Solaris 2.5, are preemptive multitasking operating systems. The Apple Macintosh operating system is a cooperative multitasking system.

In Solaris 2.5 and Windows NT the basic unit of scheduling is a thread. Some of the better synchronization mechanisms utilized by these three operating systems are hardware dependent. One such hardware dependent synchronization mechanism is the Spin lock. Unix processes have a well defined hierarchy. Each process has at most one parent and zero or more child processes. In Windows NT and Apple Macintosh operating systems there is no parent/child relationship among processes.

UNIX and Windows NT kernels are re-entrant, but the Apple Macintosh operating system is not. Apple does support re-entrant codes to some extent but it is limited to the 68k architecture. UNIX and Windows NT are symmetric multiprocessing operating systems, but Apple Macintosh is not. The Apple Macintosh operating system also supports multiple processors, but it is made possible by using the multiprocessing API developed in collaboration with Daystar International. Thus the operating system by itself

does not support multiprocessing and all the synchronization issues have to be handled by the application developers.

CHAPTER VII

SUMMARY AND FUTURE WORK

7.1 Summary

In Chapter I we discussed operating systems in general, their components, and multitasking operating systems. Various studies carried out in this area (i.e., synchronization) were also mentioned. Chapter II provided information about synchronization, process communication, and different synchronization mechanisms.

The results of the study on synchronization in UNIX, Windows NT, and Apple Macintosh operating systems were listed in Chapters III, IV, and V, respectively. Before discussing synchronization issues, a brief discussion of the internals of the UNIX, Windows NT, and Apple Macintosh operating systems was provided at the beginning of their respective chapters.

Chapter III presented some details about UNIX process states, parent/child relationship among processes, kernel level, and multiprocessor synchronization. Among other synchronization mechanisms and issues, semaphores, semaphore convoy effect, read-write locks, sleep locks, etc., were also discussed. After the discussion about basic architecture of Windows NT, Chapter IV provided some details about the protected subsystem, Windows NT executive, processes, thread states, user-level, and kernel-level synchronization mechanisms, as well as multiprocessor synchronization issues.

We discussed cooperative multitasking, processes and events, thread manager and concurrency, and multiprocessing API in Chapter V. Based only on the synchronization issue, Chapter VI compares these operating systems and tabulates their differences and similarities, followed by the observations of this study.

7.2 Future Work

It is possible to design a new synchronization primitive based on the study conducted in this thesis. It is also possible to extend this study further to explain the reasons for the success or failure of these operating systems based on synchronization. The Apple Macintosh operating system Version 8.0, which is the most recent release, has undergone some major changes. Multitasking capability has improved a lot, but no proper documentation is available on this as of today. This comparative study can be updated by including the latest version of these operating systems.

REFERENCES

[Apple 96a] Apple Technical Library, *Inside Macintosh: Devices*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[Apple 96b] Apple Technical Library, *Inside Macintosh: Files*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[Apple 96c] Apple Technical Library, *Inside Macintosh: Interapplication Communication*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[Apple 96d] Apple Technical Library, *Inside Macintosh: Macintosh Toolbox Essentials*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[Apple 96e] Apple Technical Library, *Inside Macintosh: Mac OS Runtime Architecture*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[Apple 96f] Apple Technical Library, *Inside Macintosh: Memory*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[Apple 96g] Apple Technical Library, *Inside Macintosh: Operating System Utilities*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[Apple 96h] Apple Technical Library, *Inside Macintosh: Overview*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[Apple 96i] Apple Technical Library, *Inside Macintosh: Processes*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[Apple 96j] Apple Technical Library, *Inside Macintosh: System 7.5 Technologies*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[Avutu 93] Raveendra Reddy Avutu, *A General Mutual Exclusion Primitive*, Masters Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, 1993.

[Bach 86] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1986.

[Cooksey 96] Chris Cooksey, "Working with Apple's Multiprocessing API", *Apple Technical Information Library*, Technote 1071, Reading, MA, October 1996.

[Custer 93] Helen Custer, *Inside Windows NT*, Microsoft Press, Redmond, WA, 1993.

[Davis 94] Ralph Davis, *Windows NT Network Programming,* Addison-Wesley Publishing Company, Menlo Park, CA, 1994.

[Deitel 92] H. M. Deitel, *An Introduction to Operating Systems,* Addison-Wesley Publishing Company, Reading, MA, 1992.

[Digital 87] Digital Equipment Corporation, *VAX Architecture Reference Manual*, 1987.

[Dijkstra 68] E. W. Dijkstra, "Co-operating Sequential Processes", In *Programming Languages,* F. Genuys (Ed.); Academic Press, pp. 43-112, Eindhoven, The Netherlands, 1968.

[Dunstan 89] N. Dunstan, "Synchronization Problems and UNIX System V", *ACM Computing Surveys*, Vol. 21, No. 4, pp. 15-19, December 1989.

[Dunstan and Fris 95] Neil Dunstan and Ivan Fris, "Process Scheduling and UNIX Semaphores", *Software: Practice and Experience,* Vol. 25, No. 10, pp. 1141-1153, October 1995.

[Hoare 74] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM,* Vol. 17, No. 10, pp. 549-557, October 1974.

[Kelley 89] M. H. Kelley, "Multiprocessor Aspects of the DG/UX Kernel", *Proceedings of the Winter 1989 USENIX Conference*, pp. 85-99, San Diego, CA, January 1989.

[Krakowiak 90] Sacha Krakowiak, *Principles of Operating Systems,* MIT Press, Cambridge, MA, 1990.

[Lee and Luppi 87] T. P. Lee and M. W. Luppi, "Solving Performance Problems on a Multiprocessor UNIX System", *Proceeding of the Summer 1987 USENIX Conference*, pp. 399-405, Phoenix, AZ, June 1987.

[Leffler, et al. 89] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley, Reading, MA, 1989.

[May and Whittle 95] John C. May and Judith B. Whittle, *Programming Primer for the Macintosh, Volume 1,* AP Professional, Cambridge, MA, 1995.

[McGilton 83] Henry McGilton, *Introducing the UNIX System,* McGraw-Hill, New York, NY, 1983.

[Peterson 81] G. L. Peterson, "Myths About the Mutual Exclusion Problem", *Information Processing Letters,* Vol. 12, No. 3, pp. 115-116, June 1981.

[Richter 93a] Jeff Richter, *Advanced Windows NT,* Microsoft Press, Redmond, WA, 1993.

[Richter 93b] Jeff Richter, "Creating, Managing, and Destroying Processes and Threads Under Windows NT", *Microsoft Systems Journal,* Vol. 8, No. 7, pp. 55-78, July 1993.

[Ritchie 78] D. M. Ritchie, "Synchronization and Scheduling", *The Bell System Technical Journal,* Vol. 57, No. 6, pp. 1935-1937, July 1978.

[Silberschatz and Galvin 95] Avi Silberschatz and Peter Galvin, *Operating System Concepts,* Addison-Wesley Publishing Company, Reading, MA, 1995.

[Stallings 95] William Stallings, *Operating Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Tanenbaum 92] A. S. Tanenbaum, *Modern Operating Systems,* Prentice-Hall Inc., Englewood Cliffs, NJ, 1992.

[Tanenbaum and Woodhull 97] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation,* Prentice-Hall Inc., Englewood Cliffs, NJ, 1997.

[UNIX 92] UNIX System Laboratories, *Device Driver Reference-UNIX SVR4.2,* UNIX Press, Prentice-Hall Inc., Englewood Cliffs, NJ, 1992.

[Vahalia 96] Uresh Vahalia, *Unix Internals: The New Frontiers,* Prentice-Hall Inc., Upper Saddle River, NJ, 1996.

[Wheeler 96] D. A. Wheeler, *Ada 95: The Lovelace Tutorial,* Springer-Verlag Inc., New York, NY, 1996.

[Wills 96] Craig E. Wills, "Process Synchronization and IPC", *ACM Computing Surveys,* Vol. 28, No. 1, pp. 209-211, March 1996.

APPENDICES

APPENDIX A: GLOSSARY

ADB	Apple Desktop Bus.
BSD	Berkeley Software Distribution, a flavor of UNIX.
Critical Section	When a process is accessing shared data, the process must be in its critical section to insure the integrity of the data.
DPC	Deferred Procedure Call.
IPC	Inter-Process Communication.
IPL	Interrupt Priority Level.
LPC	Local Procedure Call.
Mac	Macintosh.
MPProcessors	Command used in an Apple Macintosh multiprocessing operating system to count the number of processors.
Mutual Exclusion	Each process accessing the shared data excludes all other accesses from doing so simultaneously. This is called mutual exclusion.
Object handle	An index into a process-specific table that contains pointers to all the objects that the process has opened a handle to.
P	Proberen, a Dutch word meaning "to test".
SCSI	Small Computer System Interface.
Semaphore	A semaphore is a non-negative integer variable that can be handled only by the P and V operations.
SVR4	System V Release 4.

V	Verhogen, a Dutch word meaning "to increment".
Xerox PARC	Xerox Palo Alto Research Center.

APPENDIX B: TRADEMARK INFORMATION

Macintosh	A registered trademark of Apple Computer, Inc.
UNIX	A registered trademark of AT&T.
Windows NT	A registered trademark of Microsoft Corporation.
Win32	A registered trademark of Microsoft Corporation.

VITA

Ramasamy Satishkumar

Candidate for the Degree of

Master of Science

Thesis: A STUDY OF SYNCHRONIZATION MECHANISMS IN UNIX, WINDOWS NT, AND MAC OS

Major Field: Computer Science

Biographical:

Personal Data: Born in Lakshmipuram, India, July 15, 1974, son of Mr. and Mrs. N. Ramasamy.

Education: Received Bachelor of Engineering in Electronics and Communication Engineering from University of Madras, Madras, India, in July 1995; completed requirements for the Master of Science Degree in Computer Science at the Computer Science Department at Oklahoma State University in December 1997.

Professional Experience: Working as a Database and System Administrator in the Writing Center at Oklahoma State University, from June 1996 to July 1997.