REAL-TIME OPERATING SYSTEMS

1. **INTRODUCTION** **2**

1.1. WHAT IS REAL-TIME COMPUTING? 2
1.2. WHAT EXACTLY IS MEANT BY REAL-TIME? 2
1.3. COMMON MISCONCEPTIONS 3
1.4. HARD REAL-TIME SYSTEMS 4
1.5. GENERAL REQUIREMENTS 5
1.6. DRIVING FORCES 7

2. **WHY MULTICOMPUTERS?** **7**

3. **REAL-TIME OPERATING SYSTEMS** **8**

4. **KERNEL** **9**

4.1. WHAT IS A KERNEL? 9
4.2. WHAT SERVICES DOES A KERNEL PROVIDE? 10
4.3. BASIC ABSTRACTIONS 10
4.4. SINGLE PROCESSOR KERNEL 12
4.5. MULTIPROCESSOR KERNEL 13
4.6. DISTRIBUTED KERNEL 14

5. **KERNEL STRUCTURE** **15**

5.1. MONOLITHIC 15
5.2. LAYERED ARCHITECTURE 16
5.3. MICROKERNEL-BASED 16
5.4. MICROKERNELS-MACROKERNELS-AND-THE-IN-BETWEENS 18
5.5. ADAPTABLE KERNEL 18
5.6. IMPLEMENTATION OF THE KERNEL 18

6. **PROCESS MANAGEMENT** **19**

6.1. GOALS 20
6.2. ADDRESS SPACES 21
6.2.1. *Libraries:* *21*
6.2.2. *Kernel:* *21*
6.2.3. *Data sharing and communication:* *21*
6.3. CREATION OF A NEW PROCESS 21
6.3.1. *Choice of a target host* *22*
6.3.2. *Creation of a new execution environment* *22*

7. **THREAD MANAGEMENT** **23**

7.1. THREADS VERSUS MULTIPLE PROCESSES 23
7.2. MULTI-THREADED PROCESSES 25
7.3. SOFTWARE MODELS FOR MULTITHREADED PROGRAMMING: 28
7.4. THREAD SCHEDULING 28
7.5. THREADS VERSUS MULTIPLE PROCESSES 29

8. **SCHEDULING** **31**

8.1. THE SCHEDULING PROBLEM 32

8.2. TYPES OF SCEDULING 32
8.3. SCEDULING ALGORITHMS 33
8.4. MULTIPROCESSOR SCHEDULING 34
8.5. SCHEDULING FOR SHARED-MEMORY MULTIPROCESSORS 36
8.6. REAL-TIME SCHEDULING 38

9. **APPLICATION INTERFACE 41**

9.1. STANDARD INTERFACES 41
9.2. WHAT IS POSIX? 45
9.3. WHAT IS THE DIFFERENCE BETWEEN POSIX CONFORMANCE AND POSIX COMPLIANCE? 45
9.4. WHAT IS POSIX.1 (SPEC. 1003.1)? 45
9.5. WHAT IS POSIX.4 (SPEC. 1003.4)? 46
9.6. WHAT IS POSIX.4A (SPEC. 1003.4A)? 47
9.7. WHAT IS POSIX.1B (SPEC. 1003.1B)? 47

1. INTRODUCTION

1.1. What is real-time computing?

In real-time computing the correctness of a system depends not only on the logical result of the computation but also on the time at which the results are produced. Real-time computer systems play a vital role in our society and they cover a spectrum from the very simple to the very complex. Examples of current real-time computing systems include the control of laboratory experiements, the control of automobile engines, command-and-control systems, nuclear power plants, process control plants, flight control systems, aircraft avionics, and robotics. The more complicated real-time systems are expensive to build and their timing constraints are clarified with ad hoc techniques, or with costly and extensive simulations. Minor changes in the system result in another round of extensive testing. Different system components are extremely difficult to integrate and consequently add to overall system cost.

1.2. What exactly is meant by real-time?

There are several definitions of real-time, most of them contradictory. Unfortunately the topic is controversial, and there doesn't seem to be 100% agreement over the terminology.

1. The canonical definition of a real-time system (from Donald Gillies), is the following:

"A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."

Others have added:

"Hence, it is essential that the timing constraints of the system are guaranteed to be met. Guaranteeing timing behavior requires that the system be predictable. It is also desirable that the system attain a high degree of utilization while satisfying the timing constraints of the system."

A good example is a robot that has to pick up something from a conveyor belt. The piece is moving, and the robot has a small window to pick up the object. If the robot is late, the piece won't be there anymore, and thus the job will have been done incorrectly, even though the robot went to the right place. If the robot is early, the piece won't be there yet, and the robot may block it.

Another example is the servo loops in an airplane when on auto-pilot. The sensors of the plane must continuously supply the control computer with proper measurements. If a measurement is missed, the performance of the airplane can degrade, sometimes to unacceptable levels.

2. One will occasionally see references to "real-time" systems when what is is meant is "on-line", or "an interactive system with better response time than we used to have". Often, this is just marketing hype. For instance, although some have queried whether running "rn" is real-time, it is not, as it is interacting with a human who can tolerate hundreds of milliseconds of delays without a problem. Similarly, on-line stock quotation systems interact with humans.

3. One will also see references to "real-time" systems when what is meant is just "fast". It might be worth pointing out that "real-time" is not necessarily synonymous with "fast"; that is, it is not the latency of the response per se that is at issue (it could be of the order of seconds), but the fact that a bounded latency sufficient to solve the problem at hand is guaranteed by the system. In particular, frequently, algorithms that guarantee bounded latency responses are less efficient overall than algorithms that don't.

4. One will also occasionally see discussions of "soft" vs. "hard" real-time systems. In many of these discussions, "hard" real-time means the type of real-time system discussed above, and "soft" real-time means systems which have reduced constraints on "lateness" but still must operate very quickly and repeatably. However, the definition is controversial, as some mean by "hard" and "soft" the degree of time constraints. For instance, a real-time process attempting to recognize images may have only a few hundred microseconds in which to resolve each image, but a process that attempts to position a servo-motor may have tens of milli- seconds in which to process its data.

1.3. Common misconceptions

Real-time system design has not attracted the attention from academic computer scientists and basic research funding agencies that it deserves. This lack of adequate attention is due, at least in part, to some common misconceptions about real-time systems.

There is no science in real-time system design.

It is certainly true that real-time system design is mostly ad hoc. This does not mean, however, that a scientific approach is not possible. Most good science grow out of attempts to solve practical problems, and there is plenty of evidence that engieers of real-time systems need help. For example, the first flight of the space shuttle was delayed, at considerable cost, because of a subtle timing bug that arose from a transient CPU overload during system initialisation. Can we then develop a scientific basis for varifying that a design is free of such subtle timing bugs?

Investigations are beginning into the subtleties of including a time metric in system specification methods and semantic theories for real-time languages.

Advances in supercomputer hardware will take care of real-time requirements

Advances in supercomputer design will likely exploit parallel processors to improve system throughput, but this does not mean that timing constraints will be met automatically. Unless the architecture of the computing system is carefully tailored to match that of the application, the processors and their communication subsystems may not be able to handle all of the process load and time-critical traffic. In fact, real-time process-and-communication scheduling problems will likely get worse as more hardware is used.

Realistically, the history of computing shows that the demand for more computing power has always outstripped the supply. If the past is any guide to the future, the availability of more computing power will only open up real-time applications requiring greater functionality, thus exacerbating the timing process.

Real-time computing is equivalent to fast computing

The objective of fast computing is to minimize the average response time of a given set of processes. However, the objective of real-time computing is to meet the individual timing requirement of each process. Rather than being fast (which is a relative term anyway) the most important property of a real-time system should be predictability; that is its functional and timing behaviour should be as deterministic as necessary to satisfy system specifications. Fast computing is helpful in meeting stringent timing specifications, but fast computing alone does not guarantee predictability.

Real-time programming is assembly coding, priority interrupt programming and device driver writing

To meet tight timing constraints current practice in real-time programming relies heavily on machine-level optimization techniques. Reliance on clever hand-coding and difficult-to-trace timing assumptions is a major source of bugs in real-time programming, especially in modifying large real-time programs. A primary objective in real-time systems research is in fact to automate, by expoliting optimizing transforms and scheduling theory, the synthesis of highly efficient code and customized resource schedulers from timing-constraint specifications.

1.4. Hard real-time systems

Computer systems are real-time to the extent that they have computation completion time constraints which must be satisfied with predictability that is acceptable under the current execution environment circumstances. A special case is hard real-time systems, in which all time constraints are hard deadlines and the only acceptable predictability is that all deadlines must always be satisfied.

1.5. General requirements

Most real-time applications, from controlling a microwave oven to running a race car engine, require multiple operations to be controlled simultaneously. Today's microprocessors have the performance and capability needed for this if you can program them to do it. This is not easy since any normal processor can only execute one instruction at a time. However a multiprocessing operating system can make the job very straightforward

In The Days Before Multiprocessing

A typical real-time application, such as a microwave oven, needs to respond to user requests from a keypad, measure some variable, such as temperature, control the magnetron for power level and time, while updating a display in real time. Often an engineer will start developing a program like this by by putting everything in a simple loop repeating at regular intervals. For example:

```
do forever{

        check keypad;

        measure temperature;

        control oven power;



        decrement timer;

        update display;

        wait for clock tick;

}
```

This can work for simple cases, but what happens if one of the operations takes a long time? Or you can't run the loop fast enough to give a good response to the keypad? You may be able to improve things a bit by repeating some operations more than once in the loop. eg.

:

- check keypad
- measure temperature
- check keypad
- control oven
- check keypad

- etc...

You can keep on doing this type of fine tuning as demands get more complex but at a great cost in debugging time.These and other similar solutions, such as using many interrupt service routines are useful for small applications but quickly become unmanageable and unreliable as the system becomes more complex. Adding a simple feature can easily break an existing application.

The modern way of simplifying the development and improving maintainability of such programs is to incorporate a Multiprocessing Operating System into the design. You can then divide your problem into a number of separate processes and let the operating system take responsibility for managing the CPU so that each function thinks it owns the entire system. Multiprocessing operating systems are available for most processors from 8031 and 68HC11 8 bit processors through 16 bit processors such as the 68HC16 to large mainframes.

Non Preemptive Scheduling

There are two main classes of multiprocessing, non-preemptive or cooperative multiprocessing and preemptive multiprocessing. Non-preemptive multiprocessing relies upon each part of program voluntarily giving up the CPU to another process every so often. This has the advantage that the application program remains in control, but an unsociable process can hog the entire CPU if it does not let the operating system take control every so often. Windows 3.x uses this form of multiprocessing. You may have noticed that when the mouse button is pressed all other activity such as the clock, stops. This type of multiprocessing is not very common for real-time real time systems, although it has the advantage of simplicity.

Preemptive Operating Systems

In a preemptive multiprocessing system, control can be taken away from a process forcibly by the operating system, and then resumed later, without the process even noticing that it was paused for a while. Real-time operating work this way and are often called Real Time Operating Systems (RTOS).

Processes

You will probably see many definitions of process, and you may see the words process or thread used as well, but they mean pretty much the same thing. A process is usually a fragment of a program that performs one function and may execute in parallel with other such processes. An individual program may contain a few or many processes. At any moment a process can be in one of a number of states. It can be actually running, it could be suspended but ready to run when the CPU is available, it could be blocked waiting for an event or it may not be ready to run because it has not yet been initialized.

Who Gets to Go First?

In real-time applications each process is usually given a priority and the operating system will always attempt to run the highest priority process. Even the highest priority process will have to wait for something occasionally, such as when it is waiting for data from a keypad. When this

happens the CPU cycles are not wasted, the next highest priority process is restarted, if that has to wait for something the next highest process is allowed to execute and so on.

Let Me Sleep Until there is Something to Do.

One of the most common events used to control a program is the passage of time. The simplest operation is for a process to pause for a defined number of ticks from the real time clock. Rather than doing this by a delay loop which will be inaccurate if any interrupts are received, the operating system provide functions that pause the process while allowing other processes to run. When the required number of clock ticks have happened the process is reawakened. While this is happening other processes can still run.

No Process is an Island

Usually you do not just want to have multiple separate programs running on a processor, normally you need to have the processes communicate with one another. For example the keypad process in the example will need to communicate with the process that controls the oven temperature and the display process. You may also want processes to wait until another process has performed an operation or processed some data. Most real time operating systems provide various system calls to synchronize processes. Sophisticated RTOS' support semaphores, message passing and queues.

Avoiding Fights Over Resources

Whenever you have multiple processes running, you have to be careful about two processes accessing a common item at the same time. For example if they both attempted to send data out of the serial port at the same time the data received at the other end would probably be garbled.

An important operating system function called a semaphore can be used to synchronize two processes. It does this by performing a test & set of a flag in one atomic uninterruptible operation. The test function will block a process if the flag is already set and wait until the flag is cleared by a signal operation. Whichever process arrives first at the semaphore can guarantee to be allowed to carry on, the second process will be blocked until the first clears the semaphore.

1.6. Driving forces

Two major forces are pushing real-time systems into the next generation: their need for artificial intelligence capabilities and the rapid advance in hardware. These forces are exacerbating the difficult scientific and engineering problems faced in real-time systems. They add complex entities that must be integrated into current and future applications.

2. WHY MULTICOMPUTERS?

As more and more capabilities are added to avionics and other real-time or embedded systems, it becomes necessary to increase the processing power of the underlying executors. At any point, technology imposes limits on the available performance of individual processors. Increasing the computing power beyond those limits requires the use of multiple processors.

Multiprocessor architectures offer integrators of realtime systems numerous advantages not found in single processor systems. Multiprocessor systems are inherently configurable, allowing both functional specification at processor level and the ability to add computing capacity as required. Multiprocessor systems generally use proven conservative hardware technologies that can provide reliability and resiliency to outages. Multiprocessor systems, however, present a special set of problems to designers of realtime operating systems. **The model of processor connectivity** and the **demands of realtime processing** force treatment of issues not addressed in other systems.

Sophisticated software tools makes the development process extremely tedious and error prone. In addition, many architectural difficulties must be overcome. Some real-time systems must be implemented on top of existing machines that do not lend themselves well to multitasking systems that depend on shared-memory. Other real-time systems are built from components that present particular architectural problems. Data caches, in particular, introduce the cache consistency problem. Consistency protocols designed to keep the caches consistent are not always usable, and they often introduce substantial performance penalties. Without consistency protocols, because of false sharing, shared variables may become inconsistent even if mutual exclusion mechanisms are used.

The most recent and advanced species is the message-passing cuncurrent computer: the multicomputer. Multicomputers embody the latest manifestation of computer systems engineering techniques and very large scale integrated (VLSI) circuit manufacturing process into a scalable edifice, and this feature affords a substantional advantage in cost and performance. Multicomputers can be used to design and build ever more complicated and powerful systems, and discern solutions to problems and phenomena that their predecessors could not. This existence substantiates the revolution and rejuvenates the evolutionary techniological process. Multicomputers are vehicles that possess an eath-shaking potential for discovery.

3. REAL-TIME OPERATING SYSTEMS

An operating system is interposed between applications and the physical hardware. Therefore, its structure has a dramatic impact on the performance and the scope of applications that can be built on it. Since its inception, the field of operating systems has been attempting to identify an appropriate structure: previous attempts include the familiar monolithic and micro-kernel operating systems as well as more exotic language-based and virtual machine operating systems

One major focal point for developing next generation real-time systems is the operating system. The operating system must provide basic support for guaranteeing real-time constraints, supporting fault tolerance and distribution, and integrating time-constrained resource allocations and scheduling across a spectrum of resource types. Given that the system is distributed, we face a complicated end-to-end timing analysis program.

All but the most primitive operating systems protect the operating system from being damaged by applications, by placing the operating system functions in a separate address space (or several separate address spaces) from the applications and by letting the applications run in an unprivileged mode of the processor.

The part of the operating system that has access to the memory management hardware and switches between the applications' protection domains is referred to as the kernel. The kernel's code runs in a special, privileged mode called supervisor mode or kernel mode.

Until quite recently, the operating system was the kernel, because all of the operating system ran in supervisor mode. This is now changing because of two reasons:

- Operating systems are becoming very large which makes them increasingly hard to maintain as a single monolithic entity running in supervisor mode where bugs can create so much damage.

- Distributed systems place different functions on different machines, so not all operating system functions are needed in every copy of the operating system.

The result has been a shift towards microkernels, operating systems in which only functions that need access to privileged instructions, the MMU, or peripherals are placed in the kernel. All other operating system functions are carried out by processes running in user mode.

A microkernel usually contains the memory manager, process scheduler and device drivers. User space operating services usually encompass file server, window system, and terminal handling. Services for interprocess communication are sometimes placed in the kernel and sometimes in user space.

Note that the difference between structuring an operating system as a microkernel plus user-space services and structuring it as a monlithic kernel is purely one of modularization. Microkernels with user-space services are, in general, easier to test, debug and maintain than monolithic kernels, but monolithic kernels have potentially better performance because there are fewer context switches.

A ready explanation for the popularity of microkernel-based systems in the operating systems research community is that they are easy to modify and to experiment with. Monolithic versions of UNIX, for instance, are so large today, that it takes students a long time before they even find the place where modifications for experimentation with research prototypes can be made.

An important concern in the design of microkernel-based operating systems is the realization of a very efficient context-switching mechanism. This is because the number of context switches is larger in such systems. Achieving this is made more difficult by the fact that faster (RISC) processors invest heavily in a context by collecting state in their cache; on a contexct switch much of the cache contents becomes obsolete.

4. KERNEL

4.1. What is a Kernel?

Most programmers tend to think of a real-time operating system, RTOS, instead of a real-time multiprocessing kernel. RTOS's typically are large, complex, slow, and expensive -- not the right thing for small to medium size real-time systems. What, then is the difference between an RTOS and a kernel? An RTOS has a kernel inside of it. In addition, it usually has a "shell" (operator

interface), file I/O services, application load and management services, etc. It usually runs itself and applications from RAM.

A kernel, on the other hand, does not assume the availability of a keyboard, monitor, or disk drive. Instead, it is targeted at black boxes which do not have these ammenities and it is assumed to run from ROM. Also, a kernel is normally linked with the application code to form a single, executable program. Hence, kernels are exactly right for small to medium size real-time systems.

4.2. What Services Does a Kernel Provide?

If you are going to use a kernel, it must provide the services you need. Typically, real-time, multiprocessing kernels provide the following services:

- process management
- interprocess communication
- memory management
- message management
- timing
- i/o management
- error management

Of course, you can write your own code to perform these functions. But why should you? Why not build on other people's work? Also you might wonder "Why do I need processes?"

4.3. Basic abstractions

What are Processes, Semaphores, Etc.?

People new to multiprocessing kernels usually have difficulty grasping what the kernel constructs actually are. In this era of object-oriented programming, it is appropriate to invoke the object paradigm: Processes, semaphores, etc. are objects. Each has information and code associated with it. Usually, the information is stored in a control block. For example, a semaphore has the following control block:

```
struct SCB   {   // SEMAPHORE CONTROL BLOCK

CB_PTR  fl      // forward link

CB_PTR  bl      // backward link

byte    cbtype   // control block type
```

```
byte    ctr      // signal counter

word    thres:8  // signal threshold

word    tplim:7  // process priority limit

word    tq:1     // process queue present

};
```

There is a control block, like this, for every semaphore defined by the application.The code associated with an object is the set of services provided by the kernel for that object. For example: signal(sem) signals a semaphore and test(sem) tests a semaphore. The principle of information hiding applies- normally the operator does not directly access or alter control blocks.

Hence, it can be accurately stated that a multiprocessing kernel provides an object-oriented environment for applications. This environment consists of objects such as processes and semaphores and services such as signal() and test() provided by the kernel.. The multiprocessing paradigm encourages you to view your application as a collection of interacting objects rather than as a set of sequential operations or as a state machine.

The object model turns out to be a better model for most real-time systems. The other models simply do not deal well with the complexities of multiple, simultaneous events which are typical in real-time systems. Flow charts are good for describing sequential processes. State machines are good if there can be only one state at a time. Neither is good for describing systems of interdependent parts. Multiprocessing, on the otherhand is ideal for such systems -- just define a process for each part!

Another weakness of the flow chart and state machine approaches is their inflexibility. Usually a good programmer can create a reasonably good solution, without multiprocessing, if nothing changes. But, if requirements change, which they invariably do, initially well-designed code soon turns into spaghetti code. It's a mess. Multiprocessing, on the other hand, fosters code which is structured so that it can grow and change, as needed. Changes are accomplished merely by adding, deleting, or changing processes. Since the code is compartmentalized into processes, propogation of changes through the code is minimal.

Why Processes?

Breaking a large job into smaller processes and then performing the processes one by one is a technique we all use in our daily lives. For example, to build a fence, we first set the posts, then attach the 2x4's, and finally nail on the lats. This divide and conquer approach is equally applicable to writing real-time systems software. A multiprocessing kernel takes this one step further by allowing the final real-time system software to actually run as multipleprocesses. This has several advantages:

- Small processes are easier to code, debug, and fix than is monolithic software.

- A multiprocessing kernel provides a uniform, pre-defined interface between functions implemented as processes, thus minimizing hidden dependencies between functions.

- The uniformity provided by a kernel is especially important if processes are created by different programmers.

- A preemptive multiprocessing kernel allows processes handling urgent events to interrupt less urgent processes

- New features can easily be added by adding new processes

Basically, a preemptive, multiprocessing environment is compatible with the way real-time software is created and is a natural environment for the same software to run in. Let's consider an example: Suppose we need to control an engine using several measured parameters and a complex control algorithm. Also, assume there is a operator interface which displays information and allows operator control and assume that the system must communicate with a remote host computer. Clearly there are at least three major functions:

engine control

operator interface

host interface

Each of the above is sufficiently complex, that it is necessary to work on it individually. Wouldn't it be nice if an environment already existed so that the three functions could be created and operated independently? This can be done by using a multiprocessing kernel and by making each function a process.Note that the processes are not of equal urgency: The operator can be kept waiting for long periods of time relative to microprocessor speeds, but the engine control process may need to respond quickly to input changes in order to maintain smooth engine operation. The host probably falls somewhere in between in urgency. With a preemptive multiprocessing kernel, this can be easily accomplished merely by giving the engine control process a higher priority than the other two processes. The host process requires an in-between priority to do its job well, and the operator process can operate satisfactorily at low priority.

4.4. Single processor kernel

Any kernel contains data structures that represent processes and three basic kinds of routines: interrupt handlers, the primitives themselves, and a dispatcher. the kernel may also contain other data structures and functionality, e.g., file descriptors and file access routines. We focus here just on the parts of a kernel that implement processes.

There are three basic ways to organize a kernel:

- As a monlithic unit in which each kernel primitive executes as an atomic action;

- as a collection of specialised processes that interact to implement kernel primitives; e.g. one may handle file I/O, and another, memory management; or

- as a concurrent program in which more than one user process may be executing a kernel primitive at the same time.

We use the first approach here since that is the simplest for a small, single processor kernel. We will use the third later in the multiprocessor kernel.

Each process is represented in the kernel by a process descriptor. When a process is idle, its descriptor contains all the information needed to execute the process. This includes the address of the next instruction the process will execute and the contects of processor registers.

The kernel starts executing when an interrupt occurs. Interrupts can be divided into two boad categories: external interrupts from peripheral devices and internal interrupts (also called traps) triggered by the executing process. When an interrupt occurs, the processor automatically saves enough state information so that the interrupted process can be resumed. Then the processor enters an interrupt handler; there is typically one handler for each kind of interrupt.

To invoke a kernel primitive, a process causes an internal interrupt by executing a machine instruction variously named a supervisor call (SVC) or trap. the process passes an argument with the SVC instruction to indicate what primitice is to be executed and passes other arguments in registers. The SVC interrupt handler first saves the full state of the executing process. Then it calls the appropriate primitive, which is implemented within the kernel by a procedure. When a primitive completes, it in turn calls the dispatcher (processor scheduler). The scheduler selects a process for execution and then loads its state.

To ensure that the primitives are executed atomically, the first action of an interrupt handler is to inhibit further interrupts; the last action of the scheduler is to enable interrupts. When an interrupt occurs, further interrupts are inhibited automatically by the hardware; the kernel reenables interrupts as a side-effect of loading a process state.

4.5. Multiprocessor kernel

A shared-memory multiprocessor has two or more processors and at least some memory that is accessible to every processor. It is relatively straightforward to extend the single processor kernel to a multiprocessor kernel. The main changes we need to make are to store kernel procedures and data structures in shared memory, to access the data structures with mutual exclusion when that is needed to avoid interference and to change scheduler to exploit the multiple processors. There arew some subtleties, however, that result from the special characteristics of multiprocessors.

We assume that internal interrupts (traps) are serviced by the processor that was executing the process that caused the trap and assume that every processor ahs an interval timer. For now, we also assume that each kernel operation and process can be executed by any processor.

When a processor is interrupted, it enters the kernel and inhibits further interrupts on that processor. This makes execution in the kernel indivisible on that processor, but it does not prevent

other processors from simultaneously executing in the kernel. To preclude interference between the processors we could make the entire kernel as a critical section. However, this is a poor choice for two reasons. First, it unnecessarily precludes some safe cuncurrent execution. In particular, only access to shared data structures such as the free and ready lists is critical. Second, making the entire kernel into a critical section results in unnecessarily long critical sections. This decreases performance because it delays processors trying to enter the kernel, and it increases memory contention for the variables that implement the kernel critical section protocol.

The performance potential of multiprocessors has proven to be elusive in practice. One important reason is that current multiprocessor operating system structures and facilities tend to be warmed-over rehashes of their uniprocessor predecessors. **Multiprocessors create the opportunity and the need to re-think various aspects of operating system design.** Topics have included:

- Lock management techniques for shared-memory multiprocessors.
- Data structures for high-performance threads.
- Kernel organization for high-performance interprocess communication on uniprocessors and multiprocessors.
- Kernel abstractions providing effective support for the user-level management of parallelism.
- The relationship of processor architecture and operating system design.

At the heart of much of this work is the observation that **the basic role of the kernel must change as we move to multiprocessor systems**.

- On uniprocessors, performance is enhanced by *moving functionality into the kernel*.

On multiprocessors, the opposite is true: threads, thread scheduling, and interprocess communication should be managed entirely at user-level, with the *kernel merely providing address spaces and occasionally reallocating processers* among them.

4.6. Distributed kernel

If all process interaction is by measn of message passing, it is possible to execute a concurrent program on a distributed system such as a multicomputer or network of workstations. Here we show how to extend the shared memory kernel to support distributed execution. The basic idea is to replicate the kernel - placing one copy on each machine - and to have the different kernels communicate with each other using network communication primitives.

In a distributed program, each channel is stored on some one machine. For now, assume, that a channel can have any number of senders but that it has only one receiver. Then the logical place to put a channel's descriptor is on the machine on which the receiver executes. A process executing on that machine accesses the channel as in the shared-memory kernel. However, a process executing on another machine cannot access the channel directly. Instead, the kernels on the two machines need to interact.

The kernel on each machine contains descriptors for the channels stored on that machine and for the processes that execute on that machine. As before, each kernel has local interrupt handlers for supervisor calls (internal traps) timers and input-output devices. The comminucation network is a special kind of input-output device. Thus each kernel has network interrupt handlers and each kernel contains routines that write to and read from the network.

When a kernel primitive - executing on behalf on an application process - needs to send a message to another machine, it calls kernel procedure net-write. This procedure has three arguments: a destination processor, a message kind and the message itself. First, net-write acquires a buffer, formats the message, and stores it in the buffer. Then, if the writing half of the network controller is free, an actual write is initiated. Otherwise the buffer is inserted on a queue of write requests. In either case, net-write returns. Later, when a write interrupt occurs, the associated interrupt handler frees the buffer containing the message that was just written; if its write queue is not empty, the interrupt handler then initiates another network write.

Input from the network is typically hadled in a reciprocal fashion. When a message arrives at a kernel, the network read interrupt handler is entered. It first saves the state of the executing process. Then it allocates a new buffer for the next network input message. Finally the read handler unpacks the first field of the input message to determine the kind and then calls the appropriate kernel primitive.

The net-read-handler services three kinds of messages: SEND, CREATE-CHAN and CHAN-DONE. These are sent by one kernel and serviced by another. The kernel's dispatcher routine is called at the end of net-write-handler to resume execution of the interrupted process. However, the dispatcher is not called at the end of net-read-handler since that routine calls a kernel primitive - depending on the kind of input message - and that primitive in turn calls dispatcher.

For simplicity we assume that network transmission is error free and hence that messages do not need to be acknowledged or retransmitted.

5. KERNEL STRUCTURE

5.1. Monolithic

An operating system is a very large piece of software; for example, Sun's UNIX kernel (the basic functions) occupies over a megabyte. Figure xx gave a simple picture of an operating system as a single module with an interface comprising system calls. The system calls can be partitioned into those which invoke file storage services, those which request terminal I/O, those which request netwrok communication, and so on. The term subsystem is often used for the major internal modules of operating systems.

As operating systems grow in size a great deal of research effort was devoted to considering how best they should be structured. One proposal was to impose a strictly layered structure on these functions introduced above.

5.2. Layered architecture

The idea is that each layer in the hierarchy ceates a single abstraction. Any layer uses the services provided by lower layers and provides a service for the layers above it. An advantage is that a change at a given level does not affect lower levels. The research system THE (1972) explored this design idea. In practice, systems have not been designed with their functions in a single strict hierarchy. This is because of the difficulty of choosing one specific layering of functions. In THE, for example, a memory fault could not be reported to the operator's console because the console manager was placed above memory in the hierarchical structure.

Certain modules fall naturally into a hierarchical relationship, and this is exploited in system design. For example, a module concerned with driving a disk comes naturally "below" a module concerned with providing a file service. A layered structure is used for communication software.

If a concurrent system is to be built on distributed hardware many of the component computers may have very limited functionality. A component of a process control system may be dedicated to monitoring the process; a component of s distributed system may be dedicated to providing a file storage service or to function as a gateway between networks, concerned only with communications handling. Such components do not need to run a complete operating system, neither would the many programming support utilities provided for human users be needed there. For this reason an open operating system structure is advocated.

5.3. Microkernel-based

The idea here is that those functions which are needed universally, by every component of a system, form a minimal kernel, sometimes called a lightweight kernel or microkernel. the phrase "kernelisation of an operating system" is also coming into use. These names tend to be used now rather than "open operating system", partly to avoid confusion with the term "open system" which was also coined in the 1970's for use in communications standards. In the latter case, the term means that a system is open to communication in a heterogeneous world.

To give an example, if a computer has no local disks because storage is provided across the network, then modules concerned with storage need not be present in that component of the system..

A small kernel has the advantage of being easier to engineer and maintain. Also, the services that are implemented above rather than within the kernel can easily be recompiled and loaded and may therefore be changed (or omitted) in certain configurations.

Processes that are executing kernel functions run at a higher priority than user-level processes. They made be privileged to carry out certain functions that are forbidden to user-level processes.

There has been a good deal of experimental work on what should be included in a minimal kernel. Every computer has at least one processor, some memory and at least one device, for example a terminal or network connection. The aim is to implement efficient mechanisms in the kernel and as much as is reasonable at user level. If you are sure that you always want a particular function to

run as fast as possible, whatever applications the kernel is to support, you should include it in the kernel. You then give it rpiority over every function you might ever implement at user level. This is a difficult decision to make. There is a trade-off involved between making a service part of the kernel, and therefore fast but inflexible, and keeping the kernel as small as possible for maintainability and low overhead. Any particular function may provide a slower service when it is located at user level than when it is provided as part of the kernel.

The kernel overhead is the time spent in the kernel between an event occurring and the user level process that it is interested in it being scheduled. In real-time systems it is essential to be able to bound this time; the requirement is to be able to express the maximum possible delay rather than to obtain high speed as such. A minimal kernel makes this feasible. The requirements of the applications that run above a minimal kernel can be expressed by indicating the relative priorities of these user-level processes.

With good design, the minimal functionality of the kernel should lead to low overhead and the potential for high performance which is essential for many network-based server computers. Any inefficiency in a kernel design affects every application, however high its priority.

To summarize, a microkernel-based system has the following potential advantages:

- The system configuration can be tailored for dedicated servers.
- A small kernel is easier to engineer and maintain.
- The services which now run above the kernel are easier to engineer, tune, update and maintain.
- The time spent executing the kernel can be bounded.

The kernel provides efficient basic mechanisms; policies are expressed flexibly at user level. Programmers can control the relative priorities of user-level software.

A potential disadvantage is that a given service will respond more slowly to its clients if it is implemented at user level instead of inside the operating system.

A microkernel should be able to form the basis for a traditional operating system which provides a programming support environment. A problem in moving to a new operating system design is that a vast amount of software has been written to run above the old operating system. Ideally, a microkernel should provide a set of abstractions from which higher-level services can be built. It should be possible to build a conventional operating system interface above a microkernel and run existing software above that. This approach has been taken by several microkernel projects. The UNIX emulation can take UNIX system calls and translate them into calls on the microkernels services.

Finally, the microkernel itself split into separate modules, each with its own interface. We shall discuss alternative approaches to system structuring and alternative methods of invoking the services offered by modules through their interface operations.

5.4. Microkernels-macrokernels-and-the-in-betweens

A recurrent topic of discussion has been the comparison between microkernel (for example Mach and QNX) and`macrokernel' (traditional Unix) operating systems. The basic notion of a microkernel consists of devolving as much functionality as possible into processes rather than the kernel itself; different systems take different approaches to implementing this.

For example, some systems (such as Mach) leave device drivers in the kernel, and place higher-level services (such as file systems) outside; others (such as QNX) move device drivers outside of the kernel.

However, anecdotal evidence suggests that the distinction between microkernel and monolithic architectures is becoming more blurred as time goes on, as the two advance. For example, most modern monolithic kernels now implement multiple threads of execution and fine-grained parallelism. Architecturally, this approach begins to appear similar to a microkernel with several kernel-space processes working from shared memory.

As an aside, people often complain that the Mach system can't be a `real' microkernel, because it is so large (at least, this is the argument most frequently cited). However, the automatically-generated code stubs contribute very significantly to the size of the kernel, and that some size reduction would be likely if MIG (the stub generator) produced better code. As mentioned above, the leaving of device drivers in the kernel also contributes to Mach's size.

Debating microkernels versus monolithic kernels on the basis of kernel size misses the central, architectural point. In the same way as the point of a RISC processor is not to minimise the instruction count, but rather to make a different tradeoff between what is implemented in the processor instruction set and what is implemented in other ways, the microkernel architectural issue is to determine which services are implemented in the microkernel, and which services are implemented external to that microkernel. By making appropriate choices here, the goal is to enhance various OS attributes in a manner that might not be addressable with a monolithic kernel OS. System attributes such as performance, flexibility, realtime, etc. are all variables which are taken into

5.5. Adaptable kernel

How it is possible to increase the performance for resource-intensive applications, such as multimedia and databases? The key idea of an adaptable kernel architecture: an extensible microkernel, in which application-specific code can be embedded. An application can define new resources, which are split into a user-level component and a kernel-level component. The kernel contains mechanisms to permit the embedding of application-specific code (the kernel-level components) in a way that is completely protected.

5.6. Implementation of the kernel

The exact implementation of the kernel and the details of the internal algorithms used can have a profound bearing on the resources the kernel needs to run. A traditional assemblylanguage

implementation produces a monolithic object that must be loaded into the target system in its entirety. A more modern approach involves writing as much of portability and maintainability of the kernel and provides performance advantages, since the overhead of a trap interface to the kernel is avoided. The disadvantage of this approach is that, if all kernel services are used, more code space will be required. But note that since the kernel is structured as a library, only the system calls you actually use get linked into your target image. So if your application does not use all of services - and very few applications use all of the services provided by a kernel - the version of kernel that gets linked into an application will be smaller.

Evaluating the RAM required to support a kernel is less straightforward since it is highly application dependent. For instance, if you have six tasks in your system, the kernel must create six Process Control Blocks (PCBs). These PCBs hold information such as the processes' priorities, current states, and stack pointers. Depending on the implementation, the PCB may also include space for pointers that will link it into the process ready list or other lists maintained by the kernel. Other kernel objects such as partitions and message queues also require control blocks.

Each process must also have its own stack. The stack is used primarily by the application but also by the kernel to process system calls and to save the registers of the task when the task is not executing.

6. PROCESS MANAGEMENT

A process has always been the abstraction of a program being executed. The state of a process would include the contents of its address space, its registers, including program counter and stack pointer, and its state with respect to the operating system and file system - system-call state and open-file state.

With the advent of multiprocessors and parallelism, the process abstraction has become confused. Within a single address space we now have several threads of control and thus several program counters and stack pointers and the state of several system calls. The confusion resulted in different research groups using different and sometimes conflicting names for address spaces and threads of control.

Mechanisms for process management

Executing processes in distributed systems requires the collaboration of several entities in the system. First, there is the entity issuing the request to create and run a process. This can be a user or system process. In a UNIX system, this process is referred to as the parent process. Next there is, of course, the process to be created, which, as in UNIX, we shall refer to as the child process.

Then, there is the child process' operating system kernel which will have to carry out the creation and execution of a child process. We will refer to this as the child kernel.

Finally, there is some system entity that either selects a target node to run the process, or merely checks that the target node indicated by the parent is an appropriate one. Typically, this entity is a

distributed system service that assigns processes to processors. This service may also do load balancing by migrating processes from one processor to another. We shall refer to it as the process management service. In the most general case, it will be implemented as a distributed service that collaborates with the operating system kernels in the nodes to gather the information it needs to make process-allocation decisions.

6.1. Goals

The traditional UNIX-style notion of processes was found in the 1980s to be unequal to the requirements of more sophisticated single-computer applications that require internal concurrency. The problem, as we shall see, stems from the fact that UNIX associates just one processing activity with each process, even though the process is an expensive resource to create and manage. This makes sharing between related activities awkward and expensive.

The solution reached was to generalise the notion of process so that it could be associated with multiple activities. Henceforth, a process consists of an execution environment together with one or more threads. A thread is the operating system abstraction of an activity. The term derives from the phrase ' thread of execution'. An *execution environment* is the unit of resource management; a collection of local kernel-managed resources to which its threads have access. An execution environment primarily consists of:

- an address space;
- hread synchronisation and communication resources such as semaphores and communication interfaces (for example ports).

Note that, in a microkernel architecture, higher-level resources such as open files and windows do not appear as part of the collection of resources of an execution environment. *Higher-level resources are implemented by servers*, and access to them is via the servers' ports.

Execution environments are normally expensive to create and manage, but several threads can share them - that is, they can share all resources accessible within them. Threads can be created and destroyed dynamically as needed. The central aim of having multiple threads of execution is to maximize the degree of concurrent execution between closely-related operations. This can be particularly helpful within servers, where cuncurrent processing of clients' requests can reduce the tendency for servers to become bottlenecks.

An execution environment provides *protection*, so that the data and other resources contained in it are by default inaccessible to threads residing in other execution environment. But certain kernels allow the controlled sharing of resources such as physical memory between execution environments residing at the same computer.

As many older operating systems allow only one thread per process, we shall sometimes use the term 'multi-threaded process' for emphasis. Confusingly, in some prograwmming models and operating system designs the term 'process' means what we have called a thread. The reader may encounter in the literature the terms 'heavyweight process', where an execution environment is taken to be included, and 'lightweight process', where it is not.

Figure 7. gives the names used in four major microkernel designs for the execution environment and thread abstractions. Despite the varied and conflicting terminology, these designs provide similar semantics for threads and execution environments. The above definitions are based on them.

6.2. Address spaces

An addresss space, introduced in the previous section, is the most expensive component of an execution environment to create and manage. It is large and consists of one or more regions, separated by inaccessible areas of virtual memory. A region is an area of continuous virtual memory that is accessible by the threads of the owning process. Regions do not overlap. Note that we distinguish between the regions and their contents. Each region is specified by the following properties:

- its extent (lowest virtual address and size);
- read/write/execute permissions for the process' threads;
- whether it can be grown upwards or downwards.

Sharing between processes or between processes and the kernel in a single computer is another factor. A shared memory region (or shared region for short) is one thaqt is backed by the same physical memory as one or more regions belonging to other address spaces. Processes therefore access identical memory contents in the regions that are shared. The uses of shared regions include the following:

6.2.1. *Libraries:*

Library code can be very large, and would waste considerable memory if it was loaded separately into every process that used it. Instead, a single copy of the library code can be shared by being mapped as a region in the address spaces of processes that require it.

6.2.2. *Kernel:*

Often the kernel code and data are mapped into every address space at the same location. When process makes a system call or an exception occurs, there is no need to set up a new address space.

6.2.3. *Data sharing and communication:*

Two processes, or a process and a kernel may need to share data in order to cooperate on some task. It can be considerably more efficient for the data to be shared by being mapped as regions in both address spaces, then by being passed in messages between them.

6.3. Creation of a new process

The creation of a new process has traditionally been an indivisible operation provided by the operating system. For example, the UNIX fork system call creates a process with an execution

environment copied from the caller (except for the return value from fork). The UNIX exec system call transforms the calling process into one executing the code of a named program. For a distributed operating system, the design of process creation has to take account of two new requirements. the first is the utilisation of multiple computers; the second is the division of the process support infrastructure into separate system services.

The creation of a new process can be separated into three independent aspects:

- the choice of a target host,
- the creation of an execution environment, and
- the creation of a thread within it.

The initial thread is created with a default stack pointer and program counter, in much the same way as a UNIX process is initialised.

6.3.1. *Choice of a target host*

The choice of a computer at which the new process will reside is entirely a matter of policy. For example, the V system provides a command for users to execute a program at a particular computer, or at an idle workstation chosen by the operating system. In the Amoeba system, the 'run server' chooses a host from the processor pool for each process, and the choice is transparent both to the programmer and the user. Those programming for explicit parallelism or fault tolerance, however, may require a means of specifying process location. In general, processor allocation policies range from localisation of processes to their originator's workstation to balancing the load between a set of shared computers.

6.3.2. *Creation of a new execution environment*

A process requires an execution environment consisting of an address space with initialised contents and an initial set of port capabilities for communicating with services. A kernel creates the new execution environment at the local computer. If necessary, a system service can request a remote server process to create an environment at a remote computer.

There are two approaches to defining and initialising the adddress space of a newly created process. The first approach is used where the address space is of statically defined format.

Alternatively, the address space can be defined with respect to an existing execution environment. In the case of UNIX fork semantics, for example, the newly created child process physically shares the parents' text region and has heap and stack regions which are copies of parents' in extent (as well as in initial contents). This scheme has been generalised so that each region of the parent process may be inherited by (or omitted from) the child process. An inherited region may either be shared with or logically copied from the parents' region. When parent and child share a region, the page frames (units of physical memory corresponding to virtual memory pages) belonging to the parents' region are mapped simultaneously into the corresponding child region.

Mach and Chorus apply an important optimisation called 'copy-on-write' when an inherited region is copied from the parent. The region is logically copied, but no physical copying takes place by default. The page frames that make up the inherited region are shared between the two address

spaces. A page in the region is only physically copied when one or other process attempts to modify it.

7. THREAD MANAGEMENT

Traditionally, a proces is a program being executed. A process consists of an address space, and an execution state. The execution state typically consisted of the values of program counter, stack pointer, processor status word, a number of registers, and system-call state.

The traditional process model can be executed for parallel processing in two ways. One is to use several processes simultaneously that share memory; the other is to extend the execution state to incorporate more than one execution. the first model is referred to as the shared-memory model, the second as the thread model.

In functionality the two models are not very different; the main difference between the two models is the efficiency of the implementation. The overhead of context switching in the traditional process model is high. the virtual-to-physical address mapping has to be changed (whereby the contents of the TLB are usually lost) and the protection domain has to be changed.

The thread model does not have these overheads. Context switching from one thread to another within the same address space essentially consists of saving one thread's registers and restoring those of another.

7.1. Threads versus multiple processes

We can see from the above examples the utility of threads, which allow computation to be overlapped with input-output and, in the case of a multiptocessor, with other computation . The reader may have noted, however, that the same overlap could be achieved through the use of multiple single-threaded processes. Why, then, should the multi-threaded process model be preferred? The answer is twofold: threads are cheaper to create and manage than processes, and, resource sharing can be achieved more efficiently between threads than between processes because threads share an execution environment. Figure 9. shows some of the main state components thjat must be maintained for execution environments and threads, respectively.

An execution encironment has an address space, communicfation interfaces such as ports, and thread synchronisation objects, such as semaphores; it also lists the threads associated with it. A thread has a scheduling priority, an execution state (such as BLOCKED or RUNABLE), saved processor register values when the thread is blocked, and state concerning the threads' software interrupt handling. A software interrupt is an event that causes a thread to be interrupted (similarly to the case of a hardware interrupt). If the thread has assigned a handler procedure, control is transferred to it. UNIX signals are examples of software interrupts.

Execution environment Thread

Address space tables	Saved processor registers
Ports or other communication interfaces	Priority and execution state (such as BLOCKED)
Semaphores, other synchronisation objects	Software interrupt handling info.
List of thread identifiers	Execution environment identifier

Figure 9.

The figure shows that an execution environment and the threads belonging to it are both associated with pages belonging to the address space held in main memory, and data and instructions held in hardware caches.

We can summarize a comparison of processes and threads as follows:

- A new thread within an existing process is cheaper than creating a process.
- More importantly, switching to a different thread within the same process is cheaper than switching between threads belonging to different processes.
- Threads within a process may share data and other resources conveniently and efficiently compared to separate processes.
- But, by the same token, threads within a process are not protected from one another.

Consider the cost of creating a new thread in an existing execution environment. The main tasks are to allocate a region for its stack and to provide initial values of the processor registers and the threads' execution state (it may initially be SUSPENDED or RUNABLE) and priority. Since the execution environment exists, only an identifier for this has to be placed in the thread's descriptor record (which contains data necessary to manage the threads' execution).

The overheads associated with creating a process are in general considerably greater than those of creating a new thread. A new execution environment must first be created, including address space tables and some initial ports.

When the new entity performs some useful work rather than calling a null procedure, there are also long-term costs, which are liable to be greater for a new process than a new thread within an existing process. In a kernel supporting virtual memory, the new process will incur page faults as data and instructions are referenced for the first time; hardware caches will initially contain no data values for the new process, and it must acquire cache entries as it executes. In the case of thread creation, on the other hand, these long-term overheads may also occur, but they are liable to be less. When the thread accesses code and data that have recently been accessed by other threads within the process, it automatically takes advantage of any hardware or main memory caching that has taken place.

The second performance advantage of threads concerns switching between threads - that is, running one thread instead of another at a given processor. This cost is the most important because it may be incurred many times in the lifetime of a thread. Switching between threads sharing the same execution environment is considerably cheaper than switching between threads belonging to different processes. The overheads associated with thread switching are scheduling (choosing the next thread to run) and context switching.

A processor context comprises the values of the processor registers such as the program counter, and the current hardware protection environment: the address space and the processor protection mode (supervisor or user). A context switch is the transition between contexts that take place when switching between threads, or when a single thread makes a system call. It involves the following:

- the saving of original processor state, and the loading of the new state,
- when switching between execution environments, or to or from the kernel, a transfer has to be made to a new protection environment - this is known as a domain transition.

When switching betweeen threads belonging to the same process on a kernel which supports threads directly, context switching only occurs to and from the kernel. If the kernel is mapped into the process' address space, the cost of this may be relatively low. When switching between threads belonging to different processes, however, there is also context switch to a separate user-level execution environment.

In our example above of the client process with two threads, the first thread generates data and passes it to the second thread which makes an RPC. Since the threads share an address space, there is no need to use message passing to pass the data. Both threads may access the data via a pointer in a common variable. Herein lies both the advantage and the danger of using multi-threaded processes. The convenience and efficiency of access to shared data is an advantage. This is particularly so for servers, as the example of caching file data given above showed. However, threads that share an address space are not protected from one another. An errant thread can arbitrarily alter data used by another thread, causing a fault. If protection is required, it can be preferable to use multiple processes, instead of multiple threads.

7.2. Multi-threaded processes

The exact meaning of the term `thread' is not generally agreed upon. One of the more common usages denotes a `lightweight' process (sequential flow of control) which shares an address space and some other resources with others, and for which context switching time is lower than for `heavyweight' (i.e. kernel-supported) processes.

Throughout the following material, when we refer to `processes', this can be taken as meaning heavyweight processes.

Distinguishing features

Some of the features which distinguish different approaches to threading are listed below:

- Number of concurrent flows of control: generally, threads may potentially make use of multiple processors in order to allow several to execute concurrently. That is, the model usually takes into consideration the possibility that there may be more than one flow of control active at any time.
- Scheduling policy: a thread scheduler may be pre-emptive, in which case a thread is put to sleep either when it waits upon some resource or runs for the full duration of its time quantum, or non-pre-emptive, in which case individual threads continue to run until they relinquish the processor themselves (either through waiting on a resource or calling the analogue of a sleep() function).

Systems which are non-pre-emptive and may only ever have a single active flow of control (regardless of the number of processors available) are referred to as coroutine systems. Coroutine programming requires quite a different approach from threads-based programming, as many of the synchronisation and resource-sharing problems which occur in threaded environments need never trouble the coroutines programmer.

Characterising implementations of multithreading

An important distinction may be made between user-level threads and **kernel-supported threads**. A **user-level thread** maintains all its state in user space. A consequence of this is that no kernel resources need to be allocated per thread, and switching between threads can be done without changing address space. A disadvantage is that user level threads cannot execute while the kernel is busy, for instance, with paging or I/O. This would require some knowledge and participation on the part of the kernel.

It is possible to combine both methods, as is done in SunOS 5.x (Solaris 2.x). Here, one or more light weight processes (LWPs) multitask one or more user-level threads, which in turn are implemented using user-space libraries.

Some issues which characterise thread implementations, and which determine the uses to which a threads package may be put, include:

- *How much* by way *of kernel resources does a thread require*? This will typically limit the number of threads that can be started by a process.
- *Under what circumstances will the entire process hang*? For instance, if some thread gets a page fault, may another thread in that process be dispatched?
- *Does switching threads require a full system call* (as on the SPARC), or may context switches between threads be performed entirely at user level?
- *How are signals handled*? Can signals be masked individually per thread? Is there a `broadcast' signal?
- *How are stacks handled*? Will the stacks shrink/grow dynamically on a per thread basis?

Several systems today (QNX and Plan 9, for instance) take the stance that **threads `fix the symptom, but not the problem'.** Rather than using threads, because the OS context switch time is too slow, a better approach, according to the architects of these systems, is to fix the OS. It's ironic, now that even PC-hosted desktop OSes provide MMU-protected multitasking, the fashionable programming model has become multiple threads running in a common address space,

making debugging difficult, and also making it more difficult to generate reliable code. With fast context switching, existing OS services like explicitly allocated shared memory between a team of cooperating processes can create a `threaded' environment, without opening the box of problems that a fully shared memory space entails.

The history of threads

The notion of a thread, as a sequential flow of control, dates back to 1965, at least, with the Berkeley Timesharing System. Only they weren't called threads at that time, but processes [Dijkstra, 65]. Processes interacted through shared variables, semaphores, and similar means. Max Smith did a prototype threads implementation on Multics around 1970; it used multiple stacks in a single heavyweight process to support background compilations.

Perhaps the most important progenitor of threads is the programming language PL/I, from about the 1965 time frame. The language as defined by IBM provided a `CALL XXX (A, B) TASK;' construct, which forked a thread for XXX. It is not clear whether any IBM compiler ever implemented this feature, but it was examined closely while Multics was being designed; it was decided that the TASK call as defined didn't map onto processes, since there was no protection between the threads of control. So Multics took a different direction, and the TASK feature was removed from PL/I by IBM in any case, along with the ABNORMAL attribute and lots of other weird stuff.

Then came Unix, in the early 1970s. The Unix notion of a `process' became a sequential thread of control *plus* a virtual address space (incidentally, the Unix notion of a process derived directly from the Multics process design [Saltzer, 66]). So `processes', in the Unix sense, are quite heavyweight machines. Since they cannot share memory (each has its own address space), they interact through pipes, signals, etc). Shared memory (also a rather ponderous mechanism) was added much later.

After some time, Unix users started to miss the old processes that could share memory. This led to the `invention' of threads: old-style processes that shared the address space of a single Unix process. They also were called `lightweight', by way of contrast with `heavyweight' Unix processes. This distinction dates back to the very late 70s or early 80s, i.e. to the first `microkernels' (Thoth (precursor of the V-kernel and QNX), Amoeba, Chorus, the RIG-Accent-Mach family, etc).

On a side note, threads have been in continuous use in telecommunications applications for quite a long time.

Advantages of threads:

- Improve performance of a program.
- Useful in driving slow devices such as disks, printers, etc.
- Improve interaction with users.(For example, in a window system
- where multiple actions occur.)
- Especially useful when building a distributed system.

7.3. Software Models for Multithreaded Programming:

Boss/Worker Model: In this type of model, one thread functions as the boss and assigns tasks to the worker threads. The worker thread signals the boss on completion of its task. Alternatively, the boss polls the workers periodically and assigns tasks when a worker has completed its task.

Work Crew Model: In this model, multiple threads work together on a single task. The task is divided horizontally into pieces that are performed in parallel, and each thread performs one piece.

Pipelining Model: In this model, a task is divided into steps. The steps must be performed in sequence. However, the program is designed to produce multiple instances of the output, and the steps are designed to operate in parallel.

Combinations of Models: One or several of the above models are combined in a single program.

7.4. Thread scheduling

Threads form the structuring tool for parallel activities within an address space. Determining which thread or threads run when, is the process of a thread scheduler. The scheduler has to select threads to run from a collection of runable threads.

Threads can be blocked, runable or running. Blocking can be viewed as the mechanism that sychronizes a thread with some event outside it. Threads can wait for events in other threads - the release of a mutex, a V operation on a semaphore, or all other threads in a group reaching some point in their execution (a barrier sychronization event) - or they can wait for some completely external event which is alost invariably modelled as the completion of a system call.

Thread scheduling on unirpocessors is not a particularly difficult task, especially if most of the blocking is of threads waiting for other threads. It is only on multiprocessors that thread scheduling becomes an interesting problem.

On multiprocessors the possibility exists to choose groups of threads to run cuncurrently on different processors. A producer/consumer pair of threads, for instance, never blocks as long as the queue of items produced and still to be consumed remains approximately half full. As long as these threads do not block, no context switches are needed, so throughput is increased. As another example, consider a parallelised computation where a number of threads independently compute for a while, then synchronize to exchange new values before proceeding independently until the next sychronization point. Suppose nine out of ten threads are allocated to processors and one remains on the runable queue. When the nine threads reach the barrier, they all have to wait for the tenth thread which still has to be started. If the tenth theads could be scheduled together, the application would finish in half the time.

This scheduling strategy that allocates threads together in order to make the ensemble of threads run faster is called gang scheduling. Optimal - or even good - gang scheduling requires knowledges of the application, so its exploitation often depends on giving the application programmer access to the scheduling algorithms.

In many applications, threads block only for very short periods. If the blocking period is short enough, the cost of the context switch to another thread can be greater than blocking time so that time is saved by letting the thread wait until the unblocking event occurs. Typically, a thread instead of blocking, will execute a tight loop in which it waits until the blocking condition has gone away. We call this spinning.

On uniprocessors, spinning is only useful, of course, when waiting for external events, such as completion of I/O. It is, therefore, only really used by kernel threads. On multiprocessors spinning is also useful for waiting for sychronisation events by other threads. Spinning is always useful when there are no other runable threads.

If one had perfect advance knowledge of blocking times, one could decide in advance whether to block or to spin. Lacking predicitve powers, one must resort to heuristics to determine the optimal spinning times under the circumstances. The extreme possibilities are never spin or always spin. Other possibilities are some fixed time before blocking or spinning some empirically determined time. A well-known algorithm is to spin for a fixed time that is equal to the time to do a context switch.

Selecting which threads to run is not the only issue on multiprocessors, selecting where to run them is also an issue. When a thread has run on a particular processor, that processor's cache tends to fill up with information relevant to that thread. If another thread is then assigned to that processor, the cached information is gradually replaced. If the new process runs only for a short time and the original thread is scheduled again, assigning it to its original processor has the advantage of providing that thread with a cache that is still "warm". We call this cache affinity.

7.5. Threads versus multiple processes

We can see from the above examples the utility of threads, which allow computation to be overlapped with input-output and, in the case of a multiptocessor, with other computation . The reader may have noted, however, that the same overlap could be achieved through the use of multiple single-threaded processes. Why, then, should the multi-threaded process model be preferred? The answer is twofold: threads are cheaper to create and manage than processes, and, resource sharing can be achieved more efficiently between threads than between processes because threads share an execution environment. Figure 9. shows some of the main state components thjat must be maintained for execution environments and threads, respectively.

An execution encironment has an address space, communicfation interfaces such as ports, and thread synchronisation objects, such as semaphores; it also lists the threads associated with it. A thread has a scheduling priority, an execution state (such as BLOCKED or RUNABLE), saved processor register values when the thread is blocked, and state concerning the threads' software interrupt handling. A software interrupt is an event that causes a thread to be interrupted (similarly to the case of a hardware interrupt). If the thread has assigned a handler procedure, control is transferred to it. UNIX signals are examples of software interrupts.

Execution environment	Thread
Address space tables	Saved processor registers
Ports or other communication interfaces	Priority and execution state (such as BLOCKED)
Semaphores, other synchronisation objects	Software interrupt handling info.
List of thread identifiers	Execution environment identifier

Figure 9.

The figure shows that an execution environment and the threads belonging to it are both associated with pages belonging to the address space held in main memory, and data and instructions held in hardware caches.

We can summarize a comparison of processes and threads as follows:

- A new thread within an existing process is cheaper than creating a process.
- More importantly, switching to a different thread within the same process is cheaper than switching between threads belonging to different processes.
- Threads within a process may share data and other resources conveniently and efficiently compared to separate processes.
- But, by the same token, threads within a process are not protected from one another.

Consider the cost of creating a new thread in an existing execution environment. The main tasks are to allocate a region for its stack and to provide initial values of the processor registers and the threads' execution state (it may initially be SUSPENDED or RUNABLE) and priority. Since the execution environment exists, only an identifier for this has to be placed in the thread's descriptor record (which contains data necessary to manage the threads' execution).

The overheads associated with creating a process are in general considerably greater than those of creating a new thread. A new execution environment must first be created, including address space tables and some initial ports.

When the new entity performs some useful work rather than calling a null procedure, there are also long-term costs, which are liable to be greater for a new process than a new thread within an existing process. In a kernel supporting virtual memory, the new process will incur page faults as data and instructions are referenced for the first time; hardware caches will initially contain no data values for the new process, and it must acquire cache entries as it executes. In the case of thread creation, on the other hand, these long-term overheads may also occur, but they are liable to be

less. When the thread accesses code and data that have recently been accessed by other threads within the process, it automatically takes advantage of any hardware or main memory caching that has taken place.

The second performance advantage of threads concerns switching between threads - that is, running one thread instead of another at a given processor. This cost is the most important because it may be incurred many times in the lifetime of a thread. Switching between threads sharing the same execution environment is considerably cheaper than switching between threads belonging to different processes. The overheads associated with thread switching are scheduling (choosing the next thread to run) and context switching.

A processor context comprises the values of the processor registers such as the program counter, and the current hardware protection environment: the address space and the processor protection mode (supervisor or user). A context switch is the transition between contexts that take place when switching between threads, or when a single thread makes a system call. It involves the following:

- the saving of original processor state, and the loading of the new state,
- when switching between execution environments, or to or from the kernel, a transfer has to be made to a new protection environment - this is known as a domain transition.

When switching betweeen threads belonging to the same process on a kernel which supports threads directly, context switching only occurs to and from the kernel. If the kernel is mapped into the process' address space, the cost of this may be relatively low. When switching between threads belonging to different processes, however, there is also context switch to a separate user-level execution environment.

In our example above of the client process with two threads, the first thread generates data and passes it to the second thread which makes an RPC. Since the threads share an address space, there is no need to use message passing to pass the data. Both threads may access the data via a pointer in a common variable. Herein lies both the advantage and the danger of using multi-threaded processes. The convenience and efficiency of access to shared data is an advantage. This is particularly so for servers, as the example of caching file data given above showed. However, threads that share an address space are not protected from one another. An errant thread can arbitrarily alter data used by another thread, causing a fault. If protection is required, it can be preferable to use multiple processes, instead of multiple threads.

8. SCHEDULING

In a multiprogramming system, mutiple processes are maintained in the main memory. Each process alternates between using the processor and waiting for I/O to be performed or for some other event to occur. The processor keeps busy by executing one process while the others wait.

The key to multipprogramming is scheduling. In fact, four types of scheduling are typically involved. One of these, I/O scheduling, is a special problem. The remaining three types of scheduling, which are types of processor scheduling, are addressed in this chapter.

8.1. The scheduling problem

When a free processor is available the operating system must select a runable process (if one exists) to run on it. The scheduling policy of the system determines whioch process is selected.

The mechanisms to effect this policy must be as efficient as possible since they are an overhead on every process, however urgent. the design of a data stuctures representing processes and the algorithms for inserting and removing processes into and out of them is therefore important.

8.2. Types of sceduling

The aim of processor scheduling is to assign processes to be executed by the processor or processors over time in a way that meets system objectives such as response time, throughput, and processor efficiency. In many systems this scheduling activity is broken down inro three separate functions: long-, medium-, and short-term scheduling. The names suggest the relative frequency with which these functions are performed. Figure xx relates the scheduling fuctions to the process state transition diagram. Long-term scheduling is performed when a new process is created. The creation of a new process results from a decision to add a new process to the set of currently active processes. Medium-term scheduling is a part of the swopping function and results from a decision to add a process to those that are at least partially in main memory and therefore available for execution. Short-term scheduling is the actual decision of which Ready process to execute next. Figure xx reorganizes the state transition diagram to suggest the nesting of scheduling fuctions.

Scheduling affects the performance of the system because it determines which processes will wait and which will progress.

Long-term scheduling

The long-term scheduler determines which programs are admitted to the system for processing. Thus it controls the degree of multiprogramming. Once admitted, a program becomes a process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swopped-out condition in which case it is added to a queue for the mediium-term scheduler.

The decision about when to create a new process is generally driven by the degree of multiprogramming. The more processes that are created, the smaller is the percentage of time that each process can be executed. Thus the long-term scheduler may limit the degree of multiprogramming to provide satisfactory service to the current set of processes. Each time a program terminates, the scheduler may make the decision to add one or more new programs. Additionally, if the fraction of time that the processor is idle exceeds a certain threshold, the long-term scheduler may be invoked.

Medium-term scheduling

meduim-term scheduling is part of the swopping function. Typically the swopping-in decision is based on the need to manage the degree of multiprogramming. On a system that does not use virtual memory, memory management is also an issue. Thus the swopping-in decision will consider the memory requirements of the swopped-out processes.

Short-term scheduling

The long-term scheduler executes relatively infrequently and makes the coarse-grained decision of whether or not to take on a new process, and which one to take. The medium-term scheduler is executed somewhat more frequently to make swopping decision. The short-term scheduler, also known as the dispatcher executes most frequently and makes the fine-grained decision of which process to execute next.

The short-term scheduler is invoked whenever an event occurs that may lead to the suspension of the current process or that may provide an opportunity to preempt to the currently running process in favour of another. Examples of such events include the following:

- Clock interrupts
- I/O interrupts
- Operating-system calls
- Signals

8.3. Sceduling algorithms

The main objective of short-term scheduling is to allocate processor time in such a way as to optimize one or more aspects of sytem behaviour. Generally a set of criteria is established against which various scheduling policies may be evaluated.

The commonly use d criteria can be categorized along two dimensions. First, we can make a distinction between user-oriented and system-oriented criteria. User-oriented criteria relate to the behaviour of the system as perceived by the individual user or process. An example is response time in an interactive or real-time system. Response time is the elapsed time between the submission of a request until the response begins to appear as the output. This quantity is visible to the user and is naturally of interest to the user. We would like a scheduling policy that provides “good” service to various users. The goal of the scheduling mechanism should be to maximize the number of users who experience an average response time of 2 sec or less.

Other criteria are system-oriented. That is, the focus is on effective and efficient use of the processor.An example is throughput, which is the rate at which processes are completed. Throughput is certainly a worthwhile measure of system performance and one that we would like to maximize. However, it focuses on system performance rather than on service provided to the user. Thus it is of concern to a system administrator but not to the user population.

Other dimensions along which criteria can be classified are those that are performance-related and those that are not directly performance-related. Performance-related critera are quantitative and generally can be readily measured. Examples include response time and throughput. Criteria that are not performance-related are either qualitative in nature or cannot readily be measured and analysed. An example of such a criterion is predictability. We would like for the service provided to users to exhibit the same characteristics over time independent of other work being performed by the system. To some extent, this criterion can be measured by calculating variances as a function of workload. However, this method is not merely so straightforward as measuring throughput or response time as a function of workload.

The use of priorities

One important aspect of scheduling is the use of priorities. In many systems, each process is assigned a priority, and the scheduler will always choose a process of higher priority over one of lower priority.

Figure xx illustrates the use of priorities. For clarity, the queuing diagram is simplified, ignoring the existence of mutiple-blocked queues and of suspended states. Instead of a single Ready queue we provide a set of queues in descending order of priority: RQ0, RQ1, RQ2 ... Rqn.

One problem with a pure priority scheduling scheme is that lower-priority processes may suffer starvation. This problem will happen if there is always a steady supply of higher-priority ready processes. To overcome this problem, the priority of a process can change with its age or execution history.

8.4. Multiprocessor scheduling

When a computer system contains more than a single processor several new issues are introduced into the design of the scheduling function.

Design issues

Three interrelated issues are related to the multiprocessor:

- The assignment of processes to processors
- The use of multiprogramming on individual processors
- The actual dispatching of a process

In looking at these three issues, it is important to keep in mind that the approach taken will depend in general on the degree of granularity of the applications and on the number of processors available.

Assignment of a process to processors

If we assume that the architecture of the multiporcessor is uniform, in the sense that no processor has a particular pgysical advantage with respect to access to main memory or to I/O devices then the simplest approach is to treat the processors as a pooled resource and assign processes to processors on demand. The question then arises as to whether the assignment should be static or dynamic.

If a process is permanently assigned to one processor from activation until its completion, then a dedicated short-term queue is maintained fro each processor. An advantage of this approach is that there may be less overhead in the scheduling function because the processor assignment is made once and for all. Also, the use of dedicated processors allows a strategy known as group or gang scheduling which is discussed later.

A disadvantage of static assignment is that one processor can be idle, with an empty queue while another processor has a backlog. To prevent this situation a common queue can be used. All processes go into one global queue and are scheduled to any available processor. Thus, over the life of a process, the process may be executed on different processors at different times. In a tightly coupled, shared-memory architecture, the context information for all processes will be available to all processors, and therefore the cost of scheduling a process will be independent of the identity of the processor on which it is scheduled.

Regardless of whether processes are dedicated to processors some means is needed to assign processes to processors. Two approaches have been used: master/slave and peer. In master/slave architecture key kernel functions of the operating system always run on a particular processor. The other processors may execute only user programmes. The master is responsible for scheduling jobs. Once a process is active, if the slave needs service, it must send a request to the master and wait for the service to be performed. This approach is quite simple and requires little enhancement to a unifprocessor multiprogramming operating system. Conflict resolution is simplified because one processor has control of all memory and I/O resources. The disadvantages of this approach are two:

- A failure of the master brings down the whole system, and
- the master can become a performance bottleneck.

In peer architecture, the operating system can execute on any processor, and each processor does self-scheduling from the pool of available processes. This approach complicates the operating system. The operating system must ensure that two processors do not choose the same process and that the processes are not somehow lost from the queue. Techniques must be employed to resolve and sychronize competing claims to resources.

There are of course a spectrum of approaches between these two extremes. One can provide a subset of processors dedicated to kernel processing instead of just one. Another approach is to simply manage the difference between the needs of kernel processes and other processes on the basis of priority and execution history.

8.5. Scheduling for shared-memory multiprocessors

In a shared-memory multiprocessor, each processor has a separate cache and address translation unit.

A given processor may have a local run queue for the device handlers of any dedicated devices it may have. There are also likely to be one or more global run queues. A possible approach is that a processor, on becoming free, should execute code such that it first examines its local run queue and, if that contains no runable process, should take a process from the global run queue(s). Note that the global run queues are writable data structures and adding or removing a process will involve more than one read or write of memory. It must be ensured that these data structures remain correct in the presence of simultaneous attempts to update them. This is important in all systems but requires particular attention in a multiprocessor.

A general point is that an application should be able to indicate to the operating system that it has a number of processes and to give their relative priorities for scheduling purposes. It is then possible to run them simultaneously on the processors of a multiprocessor.

Scheduling algorithms for multiprocessors form an active research area. One approach to process scheduling for shared-memory multiprocessors is to allocate the process at the top of the run queue to the first processor that becomes free. This policy ignores the fact, that a process may have run very recently on a particular processor and may have built up a useful state in both the cache and the address translation unit. It may have blocked for a high-priority event and be runable again in a very short time. If a process has been blocked for some time then its state is likely to have been superseded by that of more recent processes. In the case of a page fault, for example, the process should clearly continue to run on the same processor.

It may be appropriate to allow a process to “busy wait” on a page fault. The idea is to keep the process scheduled and running instead of blocking it and doing a context switch to another process. The waiting process might execute in a tight loop until it is intererupted by the arrival of the required page.

Another factor to take into account is the relationship between groups of processes. If certain processes are designed to work closely together then the ideal arrangement may be for them to be scheduled simultaneously on the multiple processors of the multiprocessor. Bad performance could result from ignoring this kind of relationship; for example, a process runs, then tries to communicate with a partner. The partner is runable but not scheduled. The first process is blocked and the partner is scheduled on the same processor. It picks up the pertinent information from the first process, acts on it, then tries to send back an answer. Its partner is not scheduled.

The counter-argument is that if the processes are sharing data it is better for them to run on the same processor so that the same cache is used. If they run on separate processors and are sharing writable data the overhead of keeping the caches coherent will be introduced.

Thread scheduling

Threads have become increasingly common in new languages and operating sytems. In the use of threads the concept of execution is separated from the rest of the definition of a process. An application can be implemented as a set of threads that cooperate and execute concurrently in the same address space.

On a uniprocessor, threads can be used as a program structuring aid and to overlap I/O with processing. Because of the minimal penalty in doing a thread switch compared to a process switch, these benefits are realised at a little cost. However, the full power of threads becomes evident in a multiprocessor system where threads can be used to exploit true parallelism in an application. If the various threads of an application are simlutaneously run on separate processors, dramatic gains in performance are possible. However it can be shown that fro applications that require siginficant interaction among threads (medium-grained parallelism) small differences in thread management and scheduling can have a significant performance impact. There is active research in the area of thread scheduling on multiprocessors. The discussion here provides an overview of key issues and approaches.

Among the many proposals for multiprocessor thread scheduling and processor assignment, the following four general approaches stand out:

Self-scheduling: Processes are not assigned to a particular processor. A global queue of Ready threads is maintained, and each processor when idle selects a thread from the queue. This is sometimes called load sharing to distinguish it from load balancing schemes in which work is allocated on a more permanent basis.

Gang scheduling: A set of related threads is scheduled to run on a set of processors at the same time, on a one-to-one basis.

Dedicated processor assignment: This is the opposite of the self-scheduling approach and provides implicit scheduling defined by the assignment of threads to processors. For the duration of the program execution each program is allocated a number of processors equal to the number of threads in the program. When the program terminates the processors return to the general pool for possible allocation to another program.

Dynamic scheduling: The number of threads in a program can be altered during the course of execution.

Scheduling in distributed systems

There is not really a lot to say about scheduling in a distributed system. Normally, each processor does its own local scheduling (assuming that it has multiple processes running on it), without regard to what the other processors are doing. Usually, this approach works fine. However, when a group of related, heavily interacting processes are all running on different processors, independent scheduling is not always the most efficient way.

The basic difficulty can be illustrated by an example in which processes A and B run on one processor and processes C and D run on another. Each processor is time-shared in, say, 100 msec time slices, with A and C running in the even slices and B and D running in the odd ones. Suppose

that A sends many messages or makes many remote procedure calls to D. During time slice 0, A starts up and immediately calls D, which unfortunate is not running because it is now C's turn. After 100 msec, process switching takes place, and D gets A's message, carries out the work, and quickly replies. Because B is now running, it will be another 100 msec before A gets the reply and can proceed. The net result is one message exchange every 100 msec. What is needed is a way to ensure that processes that communicate frequently run simultaneously.

Although it is difficult to determine dynamically the interprocess communication patterns, in many cases, a group of related processes will be started off together. For example, it is usually a good bet that the filters in a UNIX pipeline will communicate with each other more than they will with other, previously started processes. Let us assume that processes are created in groups and that intragroup communication is much more prevalent than intergroup communication.

8.6. Real-time scheduling

As we know, real-time computing may be defined as that type of computing in which the correctness of the system dependes not only on the logical result of computation but also on the time at which the results are produced. We can define a real-time system by defining what is meant by a rwal-time process, or task. In general, in a real-time system, some of the processes are real-time processes, and they have a ceratin degree of urgency to them. Such processes are attempting to control or react to events that take place in the outside world. because these events occur in "real time", a real-time process must be able to keep up with the events with which it is concerned. Thus it is usually possible to associate a deadline with a particular process when the deadline specifies either a start time or a completion time. Such a process may be classified as hard or soft. A hard real-time process must meet its deadline; otherwise it will cause undesirable damage or a fatal error to the system. A soft real-time process has an associated deadline, which is desirable but not mandatory; it still makes sense to schedule and complete the process even if it has passed its deadline.

Another characteristic of realt-me processes is whether they are periodic or aperiodic. An aperiodic process has a deadline by which it must finish or start or it may have a constraint on both start and finish time. In the case of a periodic task, the requirement mauy be stated as "once per period T" or "exactly T units apart".

Real-time preemptation

The basic characteristics of current real-time operating systems include the following:

- Fast context switch
- Small size (with its associated minimal functionality)
- Ability to respond to external interrupts quickly
- Multiprocessing with interprocess communication tools such as semaphores, signals, and events.

- Use of special sequential files that can accumulate data at a fast rate.
- Preemptive scheduling based on priority
- Minimization of intervals during which interrupts are disabled
- Primitives to delay processes for a fixed amount of time and to pose/resume processes
- Special alarms and timeouts

The heart of a real-time system is the short-term task scheduler. In designing such a scheduler, fairness and minimizing average response time are not important. What is important is that all hard real-time tasks complete (or start) by their deadline and that as many as possible soft real-time processes also complete (or start) by their deadline.

Most contemporary real-time operating systems are unable to deal directly with deadlines. Instead, they are designed to be as responsive as possible to real-time processes so that when a deadline approaches, a process can be quickly scheduled. from this point of view, real-time applications typically require deterministic response times in the several-millisecond to submillisecond span under a broad set of conditions; leading edge applications often have constraints in the range of 10-100 microseconds. Figure illustrates a spectrum of possibilities. In a preeemptive scheduler that uses simple round robin scheduling, a real-time process would be added to the Ready queue to await its next time slice as illustrated in Figure xx. The scheduling time would generally be unacceptable for real-time applications. Alternatively, in a non-preemptive scheduler, we could use a priority scheduling mechanism giving real-time processes higher priority. A real-time process that is ready would be scheduled as soon as the current process blocks or runs to completion. This could lead to a delay of several seconds if a slow low-priority process were executing at a critical time. Again, this approach is not acceptable. A more rpomising approach is to combine priorities with clock-based interrupts. Preemptation points occur at regular intervals. When a preemptation point occurs, the currently running process is preempted if a higher priority process is waiting, including the preemptation of processes that are part of the operating system kernel. Such a delay may be on the order of several milliseconds. Althoug this last approach may be adequate for some real-time applications, it does not suffice for more demanding applications, in which cases, the approach that is taken is sometimes referred to as immediate preemptation. In immediated preemptation, the operating system responds to an interrupt almost immediately unless the system is in a critical-code lockout section. Scheduling delays for a real-time process can then be reduced to 100 microseconds or less.

Deadline scheduling

Most contemporary real-time operating systems are designed with the objective of starting real-time processes as rapidly as possible, and hence they emphasize rapid interrupt handling and process dispatching. In fact, this is not a particularly useful metric in evaluating real-time operating sytems. real-time applications are generally concerned with sheer speed but with completing (or starting) processes at the msot valuable times, neither too early or too late despite dynamic resource demands and conflicts, processing overloads, and hardware or software faults. It follows

that priorities provide a crude tool and do not capture the requirement, completion (or initiation) at the most valuable time.

In recent years, there have been a number of proposals for more powerful and appropriate approaches to real-time process scheduling. All these are based on having additional information about each process. In its most general form, the following information about each process may be used.

Ready time: Time at which process becomes ready for execution. In the case of a repetitive or periodic process, this is actually a sequence of times that is known in advance. In the case of an aperiodic process, this time may be known in advance, or the operating system may be aware only when the process is actually ready.

Starting deadline: Time by which a process must begin.

Completion deadline: Time by which a process must be completed. The typical real-time application has either starting deadlines or completion deadlines but not both.

Processing time: The time required to execute the process to completion. In some cases, this time is supplied. In others, the operating system measures an exponential avegare. For still other scheduling systems, this information is not used.

Resource requirements: The set os resources (other than the processor) required by the process while it is executing.

Priority: Measures relative importance of the process. Hard real-time processes may have an "absolute" priority, with the system failing if a deadline is missed. If the system is to continue to run no matter what, then both hard and soft real-time processes may be assigned relative priorities as a guide to the scheduler.

Subprocess structure: A process may be decomposed into a mandatory subprocess and an optional subprocess. Only the mandatory subprocess has a hard deadline.

There are several dimensions to the real-time scheduling function when deadlines are taken into account: which process to schedule next, and what sort of preemptation is allowed. It can be shown for a given preemptation strategy, and using either starting or completion deadlines, that a policy of scheduling the process with the earliest deadline minimizes the fraction of processes that miss their deadlines. This conclusion holds both for single processor and multiprocessor configurations.

The other critical design issue is that of preemptation. When starting deadlines are specified, then a non-preemptive scheduler makes sense. In this case, it would be the responsibility of the real-time process to block itself after completing the mandatory or critical portion of its execution, allowing other real-time starting deadlines to be satisfied. This fits the pattern of Figure xx. For a system with completion deadlines, a preemptive strategy (Figure c or d) is most appropriate. For example, if process X is running and process Y is ready, there may be circumstances in which the only way

to allow both X and Y to meet their completion deadlines is to preempt X, execute Y to completion, and then resume X to completion.

As an example of scheduling processes with completion deadlines, consider a system that collects and processes data from two sensors, A and B. The deadline for collecting data from sensor A must be met every 20 msec, and that for B every 50 msec. It takes 10 msec, including operating system overhead, to processw each sample of data from A and 25 msec to process each sample of data from B. The computer is capable of making a scheduling decision every 10 msec. Suppose, under these circumstances, we attempted to use a priority scheduling scheme. The first two timing diagrams in Figure xx show the result. If A has higher priority, B is given only 20 msec of processing time by the time its deadline is reached, and thus fails. If B is given higher priority, then A will miss its first deadline. The final timing diagram shows the use of earliest deadline scheduling. By scheduling to give priority at any preemptation point to the process with the nearest deadline, all system requirements can be met.

Now consider a scheme for dealing with starting deadlines. A straightforward scheme is always to schedule the Ready process with the earliest deadline and let that process run to completion. An example of how this policy schedules a set of aperiodic processes is shown in the first diagram of Figure xx. Note that although process 2 requires immediate service, the service is denied. This is the risk in dealing with aperiodic processes, especially with starting deadlines. A refinement of the policy will improve performance if deadlines can be known in advance of the time that a process is ready. This policy, referred to as earliest deadline with unforced idle times, operates as follows: always schedule the eligible process with the earliest deadline and let that process run to completion. An eligible process may not be ready, and this may result in the processor's remaining idle even though there are Ready processes. This behaviour is illustrated in the second diagram of Figure. Note that the system refrains from scheduling process 1 even though that is the only Ready process. The result is that, even though the processor is not used to maximum efficiency, all scheduling requirements are met. Finally, for comparison, the FCFS policy is shown. In this case, processes 2 and 3 do not meet their deadlines.

9. APPLICATION INTERFACE

9.1. Standard interfaces

Estimates are that some 60 percent of real-systems designers are moving from roll-your-own real-time operating systems to commercial offerings. But talks with designers reveal that off-the-shelf RTOS products still fall short in several significant ways. Designers complain about the lack of a common means of interfacing with kernels, restrictive CPU support, excessive memory requirements and difficulties with multithreading.

Two areas where designers still rely on in-house kernels are low-end 8-, 16- and 32-bit applications with limited memory space, and high-performance 32-bit applications such as networking.

According to some estimates, as many as 80 percent of real-time designers writing 8-, 16- and low-end 32-bit applications still build their own kernels in house. And in the high-performance

area, one Intel Corp. estimate shows that as many as 70 percent of i960 applications, especially in networking, make use of in-house operating systems.

Off-the-shelf RTOS vendors face a number of challenges. They must find some way of providing OS services to an increasingly diverse population of CPU architectures, both RISC and CISC, as well as increasingly application-specific real-time computer systems. They must also deal with the fact that many real-time applications are not isolated, and need to communicate with other real-time processors.

Moreover, vendors must find ways to solve these problems without further balkanizing a market already full of proprietary, closed solutions. They must respond not only to a demand for more features and capabilities, but also to a need for open systems, or at least a common way of communicating and moving applications across diverse CPU architectures and RTOS implementations.

Perhaps the most prevalent limitation of commercial RTOSes noted by users is that, until relatively recently, there was no common way of interfacing to the kernels. This limits applications even where the development support is rather extensive.

For example, even though [development tool suites] allow you to write and rewrite code for a number of different hardware platforms, it can only be done in the context of a common RTOS. Because of differences in such things as procedural calls and the way interrupts are serviced, there is no way to get from one environment to the other, unless you have the same underlying operating system

While many RTOS vendors have ported their offerings to multiple-arget CPUs, choices are still somewhat confining, Unfortunately, you are often forced to make a choice between the RTOS that is most effective in a particular application and on a particular target, or one which is slightly less effective but has been ported to a number of different CPUs. Occasionally, if you are lucky, you will find an OS that fits both requirements.

In an attempt to make their products more adaptable and portable, many real-time RTOS vendors are rallying behind a common applications-programming-interface standard called Posix. This move is especially compelling because the technical committees responsible for the standard are now in the process of approving a set of extensions specific to the needs of real-time systems.

A derivative of Unix designed mainly for workstations, Posix with its forthcoming real-time extensions is attractive to users. That's because it specifies only the interface, and not how the functions are carried out.

What that means to me is that I still have a wide range of options in terms of picking an RTOS that fits my needs in terms of features, responsiveness and compatibility There is no standard as to the way this Posix compliance is built in, which could lead to unforeseen problems of compatibility.

This concern arises from the fact that in their rush to provide Posix compliance, RTOS vendors are implementing strategies that could conceivably be mutually exclusive. Some companies, such

as Venturcom (Cambridge, Mass.) and Modular Computer Systems, licensed the original AT&T Unix code, rewrote the kernel to be preemptible and deterministic, and are not adding features to make it Posix compliant. Others, such as Lynx Real-Time Systems have taken a "clean room" approach and written a kernel that is aimed at compliance with the Unix/Posix interface.

A third approach, taken by companies such as Wind River Systems and JMI Software Systems, is to add a Posix interface to their real-time kernels via library modules that use the Posix interface. In its new 5.1 version of VxWorks, for example, Wind River supports full 1003.1 Posix calls that map to the RTOS interface. It incorporates such real-time extensions as priority scheduling, clock and timer facilities and real-time files. It also incorporates some real-time extensions not included in the draft standards, such as interrupt control, device control and memory management.

This third alternative is attractive, because that gives the option of linking into the customer's Unix-based system if that is what he desires, or using the proprietary modules.

On the other hand, other experienced developers are concerned that the different approaches to implementing Posix could have varying effects on performance.

It is not so much the possible performance degradation, per se, but the differences in performance. There could be some serious discontinuities in applications that run on different Posix implementations.

In spite of improvements such as Posix, designers at the low end of the microcontroller market are still creating in-house real-time kernels and OSes. That's because memory space and addressability rule out the use of what have, until recently, been the memory-hogging requirements of commercial RTOSes.

Memory limitations in the 8- and 16-bit microcontroller world impact the sophistication of the application. Most of the time you are working in RAM space limited to 1 to 3 kbytes and ROM space not very much larger. At that level, it is very hard to separate the application from the operating environment. So more often than not the kernel is the application and vice versa.

In the past, designers have developed the kernel in the machine or assembly code of the CPU on which they were working. But recent improvements in C compilers have resulted in quite impressive improvements in code density that make it possible to write kernel code that fits into relatively tight space.

Meanwhile, companies such as Integrated Systems Inc. with its recently introduced pSOSelect, and Microtec Technology with its VRTXmc, are looking to address the microcontroller segment with space-efficient, scaled-down versions of their RTOSes.

Taking another approach, Wind River Systems has customized a version of its VxWorks RTOS for General Motors for use in memory-constrained microcontroller applications that require no more than 1.5 kbytes for the run-time size of the kernel. With extensions for tools and I/O, memory requirements expand only to 8 kbytes of ROM and 2 kbytes of RAM. This scaled version of VxWorks will be introduced early next year as a standard product.

Ideally, what we would really like is one of those off-the-shelf RTOSes, such as pSOS, VRTX or VxWorks, with not only a standard set of kernel features and services, but all of the tool support that goes with it. It would save a lot of time and money if engineers could work with the same RTOS and the same tools no matter at what level in the real-time-control hierarchy they are working.

High-end hesitation

At the 32-bit end of the real-time-control spectrum, meanwhile, a number of users continue to rely on operating systems and kernels developed in house. That's because the existing commercial RTOSes are neither sufficiently flexible to allow companies to adapt to changes in the market segments they are targeting, nor do they provide sufficient response time or predictability.

In the network environment, things are changing so fast, with new hardware solutions and networking protocols proposed and tried, that an off-the-shelf operating system with standardized services and features seldom fits very neatly.

And while some new standard RTOSes, such as ISI's pSOS and Microtec's VRTX, use a modularized approach that allows the user to add or subtract features, that there just is not sufficient granularity in such offerings. To address such needs, ISI has proposed the concept of scalable kernels. *Unlike traditional microkernels that work by distributing kernel facilities to a memory space outside the kernel*, a scalable microkernel lets the user balance kernel resources against application requirements and memory constraints.

Such an architecture should also be object-oriented internally, so that the individual, predefined kernel objects could be recombined in various ways to accommodate the needs of the application environment..With any such standard OS or kernel, we are limited to manipulating the the kernel objects defined by the vendor.

Another area where roll-your-own enthusiasts fault commercial RTOSes is in the lack of multithreading, which can improve real-time response. Extensive use is made of multithreading, both for improved performance and for enhanced communications between nodes in a system. Unlike traditional processes such as those used in Unix and many off-the shelf RTOSes, threads are not self-contained, independently schedulable entities that hold less context information about the application environment in which they operate. Because they contain less context information, threads can be started, switched and terminated much faster than processes, On the other hand, one reason they are faster is that they contain less information.

The context information in a process typically contains data relating to registers, address translation tables, paths and I/O descriptors. A thread contains little of this. As a consequence, threads are limited to including information about relatively simple objects, such as a program counter. The trade-off is that this more limited context provides finer granularity.

While a number of off-the-shelf RTOS vendors-including Lynx, Microware, Microtec Research and Wind River-are moving to multithreading, One sign of hope is that among the real-time extensions to Posix now being considered is one that supports multithreading. That may enforce some commonality among the various RTOS vendors.

9.2. What is POSIX?

The POSIX specification is a standard developed by the Institute of Electrical and Electronics Engineers (IEEE) to define the interface of the operating system at a high level to provide program compatibility at the source code level and to promote platform independent code. POSIX stands for: Portable Operating System Interface for Computer Environments and is loosely based on the UNIX operating system. The specification is part of the ISO/IEC 9945 and defines the Systems Application Program Interface (API) for the development of "open" or portable code in UNIX-like environments. There are various parts to the complete POSIX specification the most applicable sections are:

- POSIX.1. Operating system interface
- POSIX1a Extensions for real-time
- POSIX.4a Threads Extensions

9.3. What is the difference between POSIX Conformance and POSIX Compliance?

The difference between POSIX Conformance and Compliance is based in the definitions of the terms. Conformance has a precise and reliable legal definition describing the complete adherence to a set, precise, agreed-upon standard. Compliance, on the other hand has a more vague definition; to yield to ones wish, to agree a requirement. The term "Compliance" is often used by marketing to imply the same set of functionality when in fact this not the case. If a product has not been verified and certified by NIST is should be assumed NOT to conform to the POSIX specifications. For more information about POSIX conformance please consult the article in the February 21, 1994 issue of Embedded Systems by Gurjot Rajpal entitled, "POSIX: Conformance Vs. Compliance".

9.4. What is POSIX.1 (spec. 1003.1)?

The POSIX standard has evolved out of the basic pressures from the industry to have some form of compatibility; as historically applications could not be easily moved to different operating systems or hardware architectures. As the result of POSIX conformance, applications can now run on open architectures which provide the following direct advantages for users and developers:

- Vendor independence
- Protection against technological obsolescence

- Interoperability between dissimilar architectures
- Increased availability of off-the-shelf software
- A larger pool of trained programmers and users already familiar with a standard environment.
- Reduced overall costs for project development.

The POSIX.1 specification group is the Systems Application Program Interface (API) which is language independent. The specification provides the definition of the operating system interface at a high level to provide program compatibility at the source code level and to promote platform independent code. The POSIX.1 specification provides the POSIX standardized interface for the following functions:

- File system organization
- System configuration
- Error messages
- Application environment
- Process creation, management, and termination
- Process environment
- Exception conditions
- Timer operations
- File and directory operations
- Record and file locking
- Device specific functions
- C Language specific routines
- User and group database information
- Data interchange formats

9.5. What is POSIX.4 (spec. 1003.4)?

The POSIX.4 specification group is the Realtime Extensions to the base POSIX.1 specification. The POSIX.4 specification provides the POSIX standardized interface for the following functions:

- Asynchronous events
- Priority interrupts and scheduling
- Preemptive scheduling
- Memory locking
- High-performance file system (Contiguous Files) (POSIX Realtime Files)
- Realtime timers
- Shared memory
- Semaphores
- Message passing
- Synchronous I/O

An OS must first be POSIX.1 conformant before POSIX.4 conformance can be achieved. The integration of the POSIX.4 specification into the POSIX.1 specification is known as POSIX.1b.

9.6. What is POSIX.4a (spec. 1003.4a)?

The POSIX.4a specification group is the Threads Extensions to the POSIX.4 Realtime Extensions specification. The POSIX.4a specification provides the POSIX standardized interface for multiple flows of control in the same POSIX.1 process. A process can be divided into tasks of execution which are independent of each other but work in the same process space.

9.7. What is POSIX.1b (spec. 1003.1b)?

The POSIX.1b specification is the application of the POSIX.4 specification to the POSIX.1 specification. The combined specification is referred to as POSIX.1b.

A.