INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A real-time multitasking kernel for the IBM personal computer

Ju, Szewei, M.S.
The University of Arizona, 1988
A REAL-TIME MULTITASKING KERNEL
FOR THE IBM PERSONAL COMPUTER

By
Szewei Ju

A Thesis Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
In Partial Fulfillment of the Requirements
For the Degree of
MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING
In the Graduate College
THE UNIVERSITY OF ARIZONA

1988
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department of the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: 

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

Theodore L. Williams
Theodore L. Williams
Department of
Electrical and Computer Engineering

December 2, 1988
DATE
ACKNOWLEDGEMENTS

The author would like to express his appreciation to his advisor, Dr. Theodore L. Williams, for his instructions, help in reviewing the drafts and providing many thoughtful comments, Dr. Fredrick J. Hill and Dr. Jerzy W. Rozenblit for their evaluation and suggestion of this thesis.
ABSTRACT

This purpose of this study is to design a simple, efficient, single-user multitasking kernel for real-time applications on the IBM Personal Computer. Since real-time applications usually consist of many small tasks and their order of execution cannot be predetermined, it is almost impossible to write a monolithic block of code that can meet the real-time requirements of all the tasks. The solution is to use multitasking. Each task is assigned a priority based on the urgency of its response time. The kernel uses a priority-based preemptive scheduling policy to schedule the CPU, so the highest-priority task gets to run as soon as it is ready. All the basic elements of an operating system are implemented to develop the multitasking kernel. A simple window display system is designed to share the PC's keyboard and screen among tasks. A user interface is also developed to allow the user to suspend, resume and terminate tasks running on the system. The Basic Input/Output System of the PC is re-written to be reentrant so that it can be shared by multiple tasks.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER 1</th>
<th>INTRODUCTION AND OVERVIEW</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multitasking</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Real-time Systems</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>The Target Machine</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Current Operating Systems For The IBM PC</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>BIOS For the Real-time Multitasking Kernel</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

| CHAPTER 2 | MULTITASKING ARCHITECTURE AND MANAGEMENT | 12 |
| Task Management | 12 |
| CPU Scheduling Strategy | 13 |
| Deadlock | 16 |
| Context Switching | 17 |
| Semaphore | 20 |
| Mutual Exclusion of Shared Resources | 23 |
| Memory Management | 25 |
| Interrupt Management | 27 |
| Task Sleeping Mechanism | 30 |

| CHAPTER 3 | BASIC INPUT AND OUTPUT SYSTEM | 31 |
| Screen Management | 31 |
| Keyboard Management | 33 |
| Serial Port Interface | 34 |

| CHAPTER 4 | USER INTERFACE | 36 |
| Command Processor task | 36 |

| CHAPTER 5 | CONCLUSION AND FUTURE WORK | 38 |

| APPENDIX A | 39 |

| REFERENCES | 66 |
CHAPTER 1

INTRODUCTION AND OVERVIEW

For years, microcomputer users have encountered single-user diskette operating systems such as CP/M from Digital Research Inc. and MS-DOS from Microsoft. Basically, these operating systems are a collection of run-time subroutines which perform device I/O operations and give application programs access to a disk-based file system. Along with these are routines to control the loading and execution of application programs. This type operating system is oriented toward user-interactive applications, such as software development, business computing and the like.

As the power of the microprocessor increases and the applications become more sophisticated, another kind of operating system -- the real-time multitasking operating system -- is steadily growing in importance. The goal of this study is to design a small, efficient, single-user multitasking kernel for real-time applications on the IBM Personal Computer.
Multitasking

Multitasking is the capability of a computer to run several programs concurrently. It is achieved by switching the CPU among programs at very high frequency, so it appears to the users that the programs are running simultaneously.

Why do we need multitasking on a personal computer? After all, a personal computer is "personal." We do not need to share our personal computer with others like we did the large-scale, expensive mainframe computer. The idea of multitasking on personal computers is still "personal," i.e. it is a single-user multitasking system. Then why does a user need to run more than one program at a time? One of the reasons is to increase the CPU utilization. In a single-task system, when a program must wait for something to happen in order to proceed, the CPU stays in a busy-waiting state, i.e. the CPU is idle. In a multitasking system, however, when a program has to wait for something, such as a command to be typed on a keyboard, or an I/O operation to complete, the CPU can switch to another program and execute it. As long as there is some program ready for execution, the CPU will never be idle. For example, when a file is sent to the printer without multitasking, the CPU must wait for the printer to finish
printing the current character before sending the next one. This is because the printer is much slower than the CPU. But, with multitasking, the CPU can switch to another program which allows the user to enter a command from the keyboard or to examine other files on the disk.

**Real-time Systems**

Multitasking is also useful in real-time applications which consist of many small tasks where the order of the execution cannot be predetermined. In real-time applications, most of the tasks must meet a specified response time after an external event happens. In a single-task system, each task is executed sequentially; it is almost impossible to write a single sequential block of code that can meet the real-time requirement of all tasks. A multitasking system, however, can assign a priority to each task based on the urgency of the task and schedule the CPU so that a high priority task can get the CPU as soon as it becomes ready.
The Target Machine

The IBM Personal Computer was chosen to be the target machine because it is a full-fledged microcomputer system with monitor, keyboard, mass storage and plenty of main memory. The PC's hardware, which includes a timer, an interrupt controller, a serial port and a parallel port, is also quite sufficient for implementing most of the operating system functions.

The Non-reentrant BIOS Code

Since the IBM PC was originally designed for a single-task system, its Basic Input/Output System (BIOS) is not reentrant. This is the IBM PC's major drawback for developing a multitasking system.

In a multitasking system, system routines such as low-level I/O routines are often invoked by multiple tasks simultaneously. They must be written so that they will not affect the results of each other; i.e. they are sharable. The code with this characteristic is called a "reentrant" code.

Current Operating Systems For The IBM PC

MS-DOS

The Microsoft's MS-DOS is the most commonly used operating system on the IBM PC. But it is not a
multitasking operating system.

Other Multitasking Systems

There are some other multitasking systems for the IBM PC. For example, Ted J. Biggerstaff designed a multitasking system in his book "Systems Software Tools." The problem with these systems is that they rely on the IBM PC's BIOS, which is not reentrant. A flag is used to prevent the kernel from switching to a second task while the first task is executing a DOS or BIOS function call. This scheme causes serious problems in real-time applications, because the response time will be quite long if the kernel happens to be within a DOS or BIOS function when an important external interrupt occurs.

BIOS For the Real-time Multitasking Kernel

Since the existing ROM BIOS code on PC is not reentrant, it cannot be used. All the Basic Input/Output routines were rewritten to ensure the reentrant property for the real-time multitasking system. No PC's BIOS routines or DOS functions are used during the system run time, except a few BIOS functions are called during the system initialization time to determine the machine's configuration. It is the user's responsibility not to invoke any BIOS or DOS functions in the user task routines;
otherwise, the results are unpredictable.
CHAPTER 2

MULTITASKING ARCHITECTURE AND MANAGEMENT

Task Management

Tasks are the basic elements of an application built on a multitasking system. Each task is an entity that consists of instructions to perform a function of the application. Each task is scheduled by the kernel for execution.

A task may be in one of the several states.
- A task is RUNNING if it has the control of the CPU.
- A task is READY if it is only waiting for the CPU.
- A task is SUSPENDED if it is waiting for a resource or an event.
- A task is SLEEPING if it is waiting for a time event to occur.
CPU Scheduling Strategy

Preemptive Scheduling

A priority-based preemptive scheduling scheme is used to schedule the CPU among tasks. Each task has a priority to indicate its importance. Each priority has a ready list associated with it. Each task is put on the ready list of its priority when it is ready to run. The scheduler always picks up the task at the front of the highest-priority ready list that is not empty. With preemptive scheduling, when a higher-priority task becomes ready during the execution of a lower-priority task, the kernel suspends the latter and transfers control to the former after the execution of a system call or an interrupt service routine.

Preemptive scheduling is an important feature of a real-time system, which must be able to respond rapidly to external events. A higher-priority task must be able to suspend the currently running task when it becomes ready, even if the time slice of the current task has not expired. The delayed execution of a high-priority task in a real-time system could have disastrous results, rather than simply making the user impatient as in a general purpose system.
Time-slice Scheduling

Time-slice scheduling is used to schedule tasks of equal priority. Tasks of equal priority are put on the same ready list. When a time slice expires, the kernel puts the currently running task back on the rear of the ready list and selects a new task to run. If there is no higher priority task ready to run, the task on the front of the same ready list will be selected to run.

The channel 0 of the PC's 8253 timer is used to generate a real-time clock interrupt at specific time intervals. The time interval can be configured during the system starting time. When a timer interrupt happens, if the time-slice scheduling is selected and time slice has expired, the current running task is put back on the ready list, and the scheduler is called to select a new task to run.

Null Task

A null task is executed when no other task is ready to run. The null task simply enables the interrupt and executes the special 8088 instruction HLT which halts the CPU but leaves the interrupt enabled. Halting the CPU when there is no task to run minimizes the interference between the CPU and other devices sharing the system bus by preventing the CPU from using the system bus to fetch
instructions from memory. The CPU is also prepared for an interrupt, which reduces the interrupt latency. Such minimization is important on systems where devices such as disks are performing a DMA transfer. Because the CPU does not contend with the DMA device for the system bus, the DMA transfer will be faster.

The kernel maintains one lowest-priority ready list which contains the null task so that the scheduler will always have at least one task to select in case that no other task is ready to run. No other task is allowed to be on the lowest-priority ready list.

There are four possible cases when the scheduler tries to select a new task to run.

1. A user task on the ready list is selected, then the kernel runs it.

2. The null task is selected, and no other task exists on the system (i.e. all tasks have completed), then the system terminates normally.

3. The null task is selected, and there are still some user tasks blocked on I/O (waiting for I/O operation to complete) or sleeping. Then, the kernel runs the null task.

4. The null task is selected, but there are no other tasks blocked on I/O or sleeping. In this case,
since all tasks are blocked on the non-I/O, non-sleeping semaphores, there is no way that any one of these tasks can be freed. So, it is a DEADLOCK. The system terminates abnormally.

Deadlock

Deadlock is the condition in which some set or sets of tasks are waiting for events which cannot happen. The most common situation is a circular wait. In such a case, a circular chain of tasks is formed and every task in the chain is waiting for something controlled by the next task in the chain. In our system, the user should not allow such situations to happen.
Context Switching

Context Switching is the process of saving the state of the current running task, loading the computer with the state of the task chosen to run, then transferring control to that task. Basically, there are two design options for context switching. One is to store all the register values of the current running task in its task table entry. The other is to store only the pointer which points to a place (usually a stack) where the register values are saved.

The first option needs to pre-allocate all the space needed to handle the maximum number of tasks. This increases the space requirements of the multitasking system. It also makes the storing and reloading of the registers somewhat awkward.

The second option usually stores the registers on the task's stack and then stores the stack pointer in the task table entry of the task. There is a risk of overflowing the stack of the user task, since after a user task invokes a system call or an interrupt happens, the kernel continues to use the stack of the invoking task. On an interrupt, the return address and the saved registers of the currently running task are pushed onto the stack. The interrupt service routine may require additional stack
space as well. So, the user task needs to allocate the maximum stack space needed for its own routines plus the amount required for any interrupt service.

We choose to place registers on the stack because it offers greater flexibility in system configuration. Although it has the risk of overflowing the stack, this is not really a serious problem if we are careful to allocate adequate stack space. When the context switch routine is finally called, either by a system call or by an interrupt service routine, all of the registers are already saved in the stack. The context switch routine needs only to save the stack pointer of the old task in its task table entry and load the stack pointer of the new task from its task table entry; then it returns.

Some C compilers automatically insert the stack checking routine in the object code when they compile a source program. The stack checking routine holds the old value of the stack pointer in a static memory. It thinks the stack overflows if the stack pointer is moved any distance from the old value. In the multitasking system, the kernel changes the stack pointer each time a context switch happens. The new value may be far distant from the old one. So, in order to prevent the stack checking routine from thinking that the stack is overflowing, we
must turn off the stack checking option or replace the stack checking routine of the compiler when the multitasking system is compiled.
Semaphore

The semaphore is used to provide intertask synchronization and mutual exclusion of shared resources.

A semaphore consists of two elements: an integer value of the semaphore count, and a pointer to the waiting list of tasks which are blocked on that semaphore.

Two operations are provided to manipulate semaphores.

The P (Wait) Operation

The semaphore's count is first decremented by 1.

Then,

1. If the semaphore value is greater than or equal to 0, the control returns immediately to the calling task.

2. If the semaphore value is less than 0, the calling task will be blocked (it is put on the waiting list of this semaphore). The system then calls the scheduler to re-schedule a task to run.

The V (Signal) Operation

The semaphore's count is first incremented by 1.

Then,

1. If the semaphore value is greater than 0, no task is waiting on this semaphore, so the control of the
CPU returns to the invoking task.

2. If the semaphore value is less than or equal to 0, one or more tasks are blocked on this semaphore. So, the first task on the waiting list of this semaphore is put back onto the ready list. Also, the calling task is put back on the ready list, then the scheduler is called to select a new task to run.

Mutual Exclusion of Semaphore Operations

Since the semaphore itself is a global data, the P and V operations of a semaphore cannot be executed at the same time by two tasks. This is guaranteed by disabling interrupts during the execution of critical sections within the P and V operations. It works because once interrupts are disabled, the control cannot transfer to another task until the critical part of the P or V operation is finished and interrupts are re-enabled.

Implementation

The semaphore manipulation routines are implemented in such a way as to eliminate the need for a central semaphore table. The key point is that the invoking task supplies the address of the semaphore data entity, rather than an index into a semaphore table. The semaphore
manipulation routines then use this pointer to access the semaphore count and the pointer to the waiting list of the semaphore.

The benefits of this implementation are:

1. Semaphore could be declared as a global data in any module or user tasks which needs to use a semaphore. Since there is no central semaphore table, the number of semaphores that could be declared is only limited by the memory space available. Also there is no need to allocate/return a semaphore entry from/to a central semaphore table.

2. Since the argument passed to the semaphore manipulation routines is the address of the semaphore, the semaphore manipulation routines can access the semaphore count and the waiting list pointer directly. Therefore, there is no need to maintain any kind of list associated with the semaphore, and therefore no extra space is wasted.
Mutual Exclusion of Shared Resources

Shared resources and global data cannot be accessed by more than one task at a time. If one task is reading a global data when another task is modifying it, the result could be wrong. So, the access to shared resources and global data by multiple tasks must be mutually exclusive in time so that they will not interfere with each other.

There are two ways to control the mutual exclusion of shared resources and global data in the system. The simplest way is to disable interrupts during the access of shared resources. Because the control will not transfer to another task, no other task will access the same resource. This method has significant disadvantages in that all the other tasks in the system cannot proceed, even those that do not need the resource. The response to external events may be delayed; so, this cannot be tolerated in a real-time system.

The second way utilizes semaphores. Besides providing intertask synchronization, semaphore offers a more flexible way to control the mutually exclusive access of shared resources. A semaphore is created for each resource with an initial count of one. Each task must execute a P operation on the semaphore before accessing the resource and then it must execute a V operation after it
finishes using the resource.

As stated above, the semaphore routines themselves access the global data -- the semaphore, so interrupts still need to be disabled briefly during the execution of the semaphore routines to guarantee the mutual exclusion access of the semaphore.

We must admit that, by using semaphore, we have not completely eliminated the need to disable interrupts to provide mutual exclusion. Rather we have moved the disabling of interrupts from the entire access of the shared resource just to the P and V operation routines in the kernel. Interrupts are disabled for a very short period of time in the P and V routines.

On the other hand, interrupts may be disabled for a long time by a user task during the access of a shared resource.
Memory Management

Memory management primarily serves two purposes. First, to manage the free space of the system, allocate it on demand, de-allocate it when a task terminates. Second, to provide memory protection and security, i.e. to prevent one task from reading or writing memory allocated to other tasks. Unlike other memory-intensive operating systems, real-time environments find less need to support a virtual memory because that will increase the system overhead and prolong the response latency. Besides, most of the memory management schemes, such as paging, segmentation and virtual memory, need special hardware supports to perform efficiently. But, unfortunately, the IBM PC hardware has no such features. So, these memory management schemes are not implemented in our design. All of the global data, the kernel codes and user task routines remain resident in the PC's memory until the system terminates. However, a dynamic memory allocation scheme is implemented to allocating and de-allocating memory for the temporary memory requirement of the user tasks.

Dynamic Memory Allocation

The dynamic memory allocation scheme is implemented to allocate memory dynamically for user tasks. The kernel keeps track of the location and size of a common memory
pool. Each task can allocate space of variable size from the common memory pool. Each task must release the space that it has allocated before it terminates.

Memory Protection and Security

As far as memory protection and security, since it is a single-user system, we assume that there is no security problem; the user must be careful when preparing user tasks not to let them read or write on others' memory space. This is the essential requirement for real-time systems.
Interrupt Management

Real-time systems must respond to external interrupts within the required response time. When an interrupt occurs, the corresponding interrupt service routine takes control of the CPU. Since interrupts are disabled during the execution of the interrupt service routine -- in order to minimize the interrupt latency -- the interrupt service routine performs only the most critical processing needed. This way, interrupts can be re-enabled as soon as possible. If more extended interrupt service is required, the interrupt service routine will signal a waiting interrupt task to perform the more complicated functions. The kernel uses the preemptive priority scheduling strategy to ensure that if a higher priority task becomes ready, it always take precedence over other system activities.

At present, the system handles three interrupts: the timer interrupt, the keyboard interrupt and the serial port interrupt.

Timer Interrupt Processing

The timer interrupt is generated by the channel 0 of the 8253 Programmable Interval Timer. The interrupt interval is configured by the user during the system starting time. The timer interrupt service routine first
updates the system time counter, then checks whether the time-slice scheduling feature is enabled or not. If the time-slice scheduling feature is enabled, the time slice is decremented and the scheduler is called to select a new task to run if the time slice has expired.

**Keyboard Interrupt Processing**

The keyboard interrupt is generated when a key is pressed. The keyboard interrupt service routine reads the key entered, puts it on the keyboard buffer, then checks to see if there is a task waiting for keyboard input. If a task is waiting for keyboard input, it is resumed.

**Serial Port Interrupt Processing**

The serial port interrupt is generated by the 8250 Synchronous Communication Controller on the serial port adapter when a character is received or the transmit buffer register is empty. The serial port interrupt service routine checks the cause of the interrupt. If data is received, it is read and put on the serial port receiving buffer. Then the interrupt service routine resumes any tasks waiting for serial port input. If the transmit buffer is empty, the interrupt service routine will check and resume any tasks waiting for transmitting on the serial port.
Disk-File System

A disk-based file system is not implemented in our system. Although disks are very important in general-purpose systems, they are used less frequently in real-time control processing. Typically, process information is collected and sent to a central file server via a serial interface.
Task Sleeping Mechanism

A task can be put to sleep for a period of time when it does not need the CPU. The sleep time unit is the timer ticks. The time between timer ticks is determined by the user at the system start-up time. Each task is associated with a sleep semaphore. When a task falls asleep, it is actually blocked on its sleep semaphore. When the sleep time of a task is over, a special kernel control task will perform a V operation on the sleep semaphore of the sleeping task to wake it up.

The kernel creates a special task to control the sleep operation of the user tasks. The sleep control task maintains a sleep task list which contains tasks that are sleeping. Tasks on the sleep list are in the order by which they will awaken. Each task keeps the number of timer ticks that it will sleep beyond the preceding one on the list. When a task invokes the kernel sleep function, it is inserted onto the sleep task list before the task that sleeps longer than it and after the task that sleeps shorter than it.

Every time a timer tick happens, the sleep control task checks to see if the sleep time of the first task on the sleep list has expired. If it has, then the sleep control task wakes up the task.
CHAPTER 3

BASIC INPUT AND OUTPUT SYSTEM

The Basic Input/Output System provides I/O services for system and user tasks to access the I/O devices of the PC. At present, the kernel supports the keyboard input, the video output and the serial port I/O. If an application requires the use of other I/O devices, users should write the I/O routines of that device and integrate them into the system, then access them as if they were part of the basic input/output system.

Screen Management

A simple window scheme was developed to make the PC's screen sharable by four tasks at the same time.

The PC's screen is divided equally into four areas. Each area is defined as a physical window with 12 rows and 40 columns. A boundary is drawn surrounding each area for easy distinction. The actual display area of a physical window is 10 rows and 38 columns. These four physical windows are non-overlapping, fixed-position, fixed-size and text-based windows.
Initially, each task has a virtual window associated with it when it is created. A virtual window is just a memory buffer with the size of 10 X 38 X 2 bytes, the same as the physical window. A virtual window must be assigned to a physical window in order to display its content on the PC's screen. During the system initialization, the first four created tasks will have a physical window assigned to their virtual windows automatically.

For those tasks without physical windows, their video output is sent to their virtual windows. When a task's virtual window is assigned to a physical window, the content of the virtual window is transferred onto the physical window so it is displayed on the screen. The user can control the assignment of the physical window to the virtual window through the command processor task of the user interface.
Keyboard Management

The keyboard is not as inherently sharable as the screen is, because only one task can own the keyboard at a time. The keyboard-sharing mechanism is to assign the keyboard to the task that owns one of the four physical windows on the screen. This window is called the "keyboard window." Only the task of the keyboard window can get input from the keyboard; i.e. the task of the keyboard window is the one which is interactively involved with the user. The keyboard window's border is blinked to distinguish it from the other windows.

When the task of the keyboard window tries to read the keyboard, it is suspended if there is no keyboard input available. When a key is pressed, an interrupt is generated. The keyboard interrupt service routine reads the key entered and puts it on the keyboard buffer, then resumes the task of the keyboard window if it is waiting for keyboard input.

If a non-keyboard window task performs a keyboard read, it will be suspended. It will be resumed when it is assigned to the keyboard window.

In order to get a quick response to typing, the task owning the keyboard window runs immediately after a key is pressed. The context switch is called directly from
the keyboard hardware interrupt service routine.

Serial Port Interface

The serial port interface provides system service routines for tasks to assign, initialize, send and receive data to/from the serial ports.

During the system initialization, the number of serial ports installed on PC will be checked by looking at PC's BIOS data area from 40:0 to 40:7.

The multitasking system uses interrupt-driven serial I/O. Although the IBM PC can have at most four RS-232C serial ports, only two of them are assigned interrupt lines by default (IRQ3 for COM2, IRQ4 for COM1). So, if more than two serial ports are required, the interrupt line assignment should be adjusted carefully, because all interrupts, except IRQ2, are reserved for other devices.

To share serial ports among tasks, each serial port will belong to a task exclusively, but a task can own more than one port at the same time. When a serial port routine is invoked, the serial port number must be supplied as an argument to specify the port. If a task gives a port number which does not belong to it, it will be suspended until the serial port is assigned to it.
The serial port sending routine first checks the 8250 transmitter holding register. If it is empty, data is written to the 8250 transmitter holding register and returned to the calling task. If the 8250 transmitter holding register is not empty, the calling task will be suspended. The suspended task will be resumed by the serial port hardware interrupt service routine when a 8250 transmitter holding register empty interrupt happens.

Receiving

In order to prevent loss of the incoming data, an 80-byte receive buffer, like the keyboard buffer, is created for each serial port to save the incoming data for later use.

The 8250 is programmed to generate an interrupt when a character is received. The interrupt service routine puts the received data in the serial port receive buffer. If there is any task waiting for data on this serial port, it will be resumed by the interrupt service routine.

When a user task invokes the serial port receive routine, if the buffer is empty, the invoking task is suspended. The suspended task will be resumed by the serial port hardware interrupt service routine when a 8250 receive data available interrupt happens.
CHAPTER 4

USER INTERFACE

The user interface allows the user to control the multitasking system via interaction with a command processor task.

Command Processor task

The command processor task is a system task which accepts and executes commands entered by the user. These commands provide the user with the capabilities to control the execution of the tasks on the system and the sharing of the system resources among tasks.

1. Change task priority.
   
   User can use this command to change the priority of the task dynamically.

2. Suspend a task.
   
   Suspend the execution of a task.

3. Resume a task.
   
   Resume the execution of a task.
4. Terminate a task.
   Force a task to be terminated.

5. Change the keyboard window.
   Re-assign the keyboard window to other physical window so tasks on the other window can get keyboard input.

6. Free the physical window.
   Disconnect a physical window on the PC's screen from a task so the physical window is available for other tasks to display their virtual window content on the screen.

7. Assign the physical window.
   Assign a free physical window to a task so its virtual window content can be displayed on the screen.

8. Quit.
   All the tasks on system are aborted and the system is terminated.
   The command processor task has the highest priority among all the tasks. If a user does not request the service of command processor, it will suspend itself.
CHAPTER 5

CONCLUSION AND FUTURE WORK

Multiprocessor Systems

For real-time applications with many urgent tasks or those that need greater throughput, a single-processor system is inherently restricted by its CPU power and sometimes just cannot make it. In such a case, a multiprocessor system can achieve the requirement. When one processor executes real-time tasks, another processor can handle new interrupts as they come through. So the addition of the multiprocessor capability will be the major enhancement for the current real-time multitasking kernel. Other improvements includes the support of more I/O devices such as disk I/O and parallel port I/O.
APPENDIX A

SYSTEM CALLS
APPENDIX A

SYSTEM CALLS

This appendix contains descriptions of the system calls supported by the multitasking kernel. User tasks can use these system calls to get system services.

1. Task Management:
   * Tcreate -- creates a new task.
   * Terminate -- terminates a task.
   * Tsuspend -- suspends a task.
   * Tresume -- resumes a suspended task.
   * Tpriority -- sets the priority of a task.

2. Memory Management:
   * Memalloc -- allocates memory space.
   * Memfree -- releases memory space.

3. Semaphore Management:
   * Seminit -- initializes the value of a semaphore.
   * SemP -- performs a P (wait) operation on a semaphore.
   * SemV -- performs a V (signal) operation on a semaphore.

4. Basic Input/Output:
* Kreset -- resets the keyboard.
* Kin   -- reads input from the keyboard.
* Wclear-- clears the window of the invoking task.
* Wsetcur-- sets the cursor position on the window of
the invoking task.
* Woutca-- writes a character and attribute on the task window.
* Woutstr-- writes a character string on the task window.
* Sinit -- initializes a serial port.
* Sin    -- get input from serial port.
* Sout   -- outputs data to serial port.

5. Miscellaneous Support:

* initIntVec -- initializes an interrupt vector.
* restIntVec -- restores the original PC interrupt vector.
* panic -- prints fatal error and shut down.
NAME

Tcreate -- creates a new task

SYNTAX

int Tcreate( taskproc, priority, stksize, nargs, args);

unsigned int (*taskproc)();
int priority;
int stksize;
int nargs;
int args;

"Taskproc" is the pointer to the starting address of the task to be created.

"Priority" is the initial priority of the task.

"Stksize" is the stack size (bytes) of the task.

DESCRIPTION

Tcreate creates a new task. The created task is left in the ready state and will begin execution when it is scheduled. The creation will fail if there is no free task table entry or there is no memory space for the stack. The address of a task termination procedure is pushed into the bottom of the stack of the task. If the task runs to the end or returns, it will return to the kernel termination procedure and be terminated.

RETURNS

If the creation is successful, the task ID of the created task is returned to the caller. Otherwise, the
value SYSERR is returned.
NAME

Terminate -- terminates a task

SYNTAX

```c
int Terminate( tid)
int tid;
"Tid" is the task ID of the task to be terminated.
```

DESCRIPTION

Terminate will stop the specified task and remove it from the system. The task table entry and stack space are returned to the system. But there is no way to recover the memory space allocated dynamically by a user task if it is terminated by another task. A user task, however, should release the space it allocates before it terminates itself.

RETURNS

If the task ID is invalid, return SYSERR.
Otherwise, return OK.
NAME

Tsuspend -- suspends a task

SYNTAX

```
int Tsuspend( tid)
int tid;

"Tid" is the task ID of the task to be suspended.
```

DESCRIPTION

Tsuspend suspends the specified task.

RETURNS

Return SYSERR, if the task ID is invalid or the referred task is not running or not on the ready list. Otherwise, return OK.
NAME

Tresume -- resumes a task

SYNTAX

int Tresume( tid)
int tid;

"Tid" is the task ID of the task to be resumed.

DESCRIPTION

Tresume resumes a suspended task.

RETURNS

If the task ID is invalid or the referenced task is not in suspended state, return SYSERR. Otherwise, return OK.
NAME

Tpriority -- sets the priority of a task

SYNTAX

int Tpriority( tid, prio)
int tid;
int prio;

"Tid" is the task ID of the task to be set priority.

"Prio" is the priority level.

DESCRIPTION

Tpriority sets the priority of the referred task with the priority supplied.

RETURNS

Return SYSERR, if the execution fails.
NAME

Memalloc -- allocates memory space

SYNTAX

char *Memalloc( size)
int size;

"Size" is the number of bytes to be allocated.

DESCRIPTION

Memalloc rounds the number of bytes to a multiple of long words, and allocates the number of long words for the caller.

RETURNS

It returns the lowest address of the memory block allocated. If the requested memory is not available, return NULL.
NAME

Memfree -- releases memory space

SYNTAX

void Memfree( memp)
char *memp;

"Memp" is the pointer to the memory block to be freed.

DESCRIPTION

Memfree returns a memory block allocated previously by "Memalloc" to the common memory pool. The argument supplied must be a pointer returned by the "Memalloc." Otherwise, the kernel will do nothing.

RETURNS

None.
NAME

Seminit -- initializes the count value of a semaphore

SYNTAX

#include "ktype.h"

void seminit( sem_p, semcnt)
sem_typ *sem_p;
int     semcnt;

The semaphore data type "sem_typ" is defined in the header file "ktypes.h."
"Sem_p" is the pointer of the semaphore.
"Semcnt" is the value of the semaphore count.

DESCRIPTION

Seminit initializes the count of the semaphore to the supplied value.

RETURNS

None.
NAME

SemP -- performs a P (wait) operation on a semaphore

SYNTAX

#include "ktype.h"

void semP(sem_p)
sem_typ *sem_p;

The semaphore data type "sem_typ" is defined in the header file "ktypes.h".

"Sem_p" is the address of the semaphore.

DESCRIPTION

SemP decrements the count of the semaphore by one. If the count become negative, the invoking task is blocked and put on the waiting list of the semaphore. The blocked task will be freed when another task signals the semaphore.

RETURNS

None.
NAME

SemV -- performs a V (signal) operation on a semaphore

SYNTAX

#include "ktype.h"

void semV( sem_p)
sem_typ *sem_p;

The semaphore data type "sem_typ" is defined in the header file "ktypes.h."

"Sem_p" is the address of the semaphore.

DESCRIPTION

SemV increments the count of the semaphore by one and resumes the first task on the waiting list of the semaphore if any are waiting.

RETURNS

None.
NAME
    Kreset -- resets the keyboard

SYNTAX
    Kreset()

DESCRIPTION
    Kreset flushes the keyboard buffer. User tasks may
    wish to invoke this service call before reading important
    data from the keyboard.

RETURNS
    None.
NAME

   Kin -- reads input from the keyboard

SYNTAX

   char Kin()

DESCRIPTION

   This is the common routine which user tasks invoke
to get a character input from the keyboard. The invoking
task is suspended if
1. It does not own the keyboard window. It will be
   resumed when the keyboard window is assigned to it.
2. It owns the keyboard window, but no key is pressed.
   When a key is pressed, the task on the keyboard
   window will be resumed if it is waiting for
   keyboard input. The data is returned to the
   invoking task.

RETURNS

   The ASCII code of the key pressed is returned to
   the caller.
NAME

Wclear -- clears the window

SYNTAX

void Wclear();

DESCRIPTION

Wclear clears the window of the invoking task. The physical window of the invoking task is cleared if a physical window is currently assigned to the virtual window of the invoking task. Otherwise, the virtual window of the invoking task is cleared.

RETURNS

None.
NAME

Wsetcur -- sets cursor position on the window

SYNTAX

Wsetcur( row, col)
int row, col;

"Row" is the row position of the cursor on the window (0 < row < 13).
"Col" is the column position of the cursor on the window (0 < col < 39).

DESCRIPTION

Wsetcur sets the cursor position of the invoking task on its window. The window output always starts at the current cursor position.

RETURNS

None.
NAME

Woutca -- writes a character and attribute on window.

SYNTAX

void Woutca( chr, attr)
unsigned char chr;
unsigned char attr;

"Chr" is the character to be displayed on the screen.

"Attr" is the attribute of the character.

DESCRIPTION

Wdispca outputs a character with the supplied attribute on the window of the invoking task at the current cursor position. The cursor is advanced to the next position after the output. The content of the window will scroll up one line if the cursor arrives at the end of the last row on the window. The carriage return and linefeed is also processed. The character and attribute will be written on the physical window if the virtual window of the invoking task is currently assigned to a physical window. Otherwise, the character and attribute will be put on the virtual window of the invoking task.

RETURNS

None.
NAME

Woutstr -- writes a character string to the window

SYNTAX

Woutstr( str_p, attr)
char *str_p;
char attr;

"Str_p" is the pointer of the character string.
"Attr" is the attribute of the character string.

DESCRIPTION

Wdispstr writes a string of characters on the window of the invoking task with the supplied attribute. The character string should be terminated either by a null character '\0' or by a linefeed '\n'. The linefeed will be converted to a newline character (linefeed + carriage return) so the cursor will appear at the beginning of the next line after the string is written on the window.

RETURNS

None.
NAME

Sinit -- initializes a serial port

SYNTAX

```c
int i;

i = Sinit( port_number, baud_rate, parity,
           stop_bit, word_len);
int port_number;
int baud_rate;
in
int parity;
int stop_bit;
in
int word_len;
```

"Port_number" is the serial port number to be initialized (COM1 = 1, COM2 = 2, ...).

"Baud_rate" is the baud rate of the serial port, only eight options are available now: 110, 150, 300, 600, 1200, 2400, 4800, 9600.

"Parity" is used to define the parity bit when serial port send or receive a character (0 = no parity, 1 = odd parity, 2 = even parity).

"Stop_bit" is the number of stop bit (0 = 1 stop bit, 1 = 2 stop bits).

"Word length" is the length of the data (0 = 7 bits, 1 = 8 bits).

DESCRIPTION

Sinit initializes the referred PC's serial port to the specified baud rate, parity, stop bit and word length.
For the sake of minimizing overhead in the serial port transmit and receive routine, the serial ports are initialized to assert two the handshaking signals, DTR (Data Terminal Ready) and RTS (Request To Send) all the time.

**RETURNS**

Return 0, if the initialization succeed, otherwise return 1.
NAME

Sout -- sends data to a serial port

SYNTAX

int Sout( port_number, character);
int port_number;
char character;

"Port_number" is the serial port number to which data is sent (COM1 = 1, COM2 = 2, ...).

"Character" is the data to be sent.

DESCRIPTION

This is the routine that user tasks can use to send a character to the serial port. The invoking task will be suspended in the following two cases:
1. It does not own the serial port.
2. It owns the serial port, but the serial port is busy, i.e. the 8250 transmit register is not empty.

When the serial port finishes sending the previous data and the 8250 transmit register become empty, an interrupt is generated. The serial port interrupt service routine will check to see if there is a task hanging on this port and resume it.

RETURNS

None.
NAME

Sin -- gets data input from a serial port

SYNTAX

char Sin( port_number)
int port_number;

"Port_number" is the serial port number from which data is received (COM1 = 1, COM2 = 2, ...).

DESCRIPTION

This is the common routine which user tasks can call to get a character input from the serial port interface. The invoking task will be suspended in the following two cases:
1. It does not own the serial port.
2. It owns the serial port, but there is no input data in the serial port receive buffer.

When a character is received, an interrupt is generated. The serial port interrupt service routine will check to see if there is task waiting for data on the port and resume it.

RETURNS

The ASCII code of the character received.
NAME

initIntVec -- initialize an interrupt vector

SYNTAX

void initIntVec(vec, newisr_p)
unsigned vec;
unsigned (far *newisr_p)();

"Vec" is the interrupt vector number.

"Newisr_p" is a far pointer to the new interrupt service routine.

DESCRIPTION

initIntVec initializes the interrupt vector of the referred interrupt and saves the original PC interrupt vector for later retrieval.

RETURNS

None.
NAME

restIntVec -- restores the original PC interrupt vector.

SYNTAX

void restIntVec(vec)
unsigned vec;

"Vec" is the interrupt vector number.

DESCRIPTION

restIntVec restores the original PC interrupt vector of the referred interrupt. The original PC interrupt vector is saved by the system when the initIntVec is called. So, restIntVec must not be called if initIntVec is not called first, because the original PC vector has not been saved yet. Otherwise, the result is unpredictable.

RETURNS

None.
NAME

panic -- prints error message and shutdown

SYNTAX

    void panic( cp)
    char *cp;

    "Cp" is the pointer of a character string of error message. The character string should end by a null character '\0'.

DESCRIPTION

    Panic is called when fatal error occurs. It prints the error message on the 25th row on the screen then shuts down the entire system.

RETURNS

    None.
REFERENCES


