**Lab Requirements**

This lab assumes that you are using Visual C++ under a 32-bit version of Windows, using Intel Pentium or Pentium-equivalent machines. The exact compiler and platform are mostly unimportant, as the lab is reasonably portable, but the lab specifically uses Pentium features. A list of compatible processors follows; if you don't have access to such a machine, please talk to your instructor.

* Intel Pentium
* Intel Pentium Pro
* Intel Celeron
* Intel Pentium II
* Intel Pentium III
* Intel Pentium 4
* Intel Xeon
* AMD K6-2
* AMD K6-III
* AMD Athlon

The specific criterion for compatibility is an IA-32 processor with the RDTSC instruction; note that, in particular, the AMD K6 lacks this instruction, as do all processors prior to the Pentium. If you aren't certain whether your processor supports this instruction, the lab does include a function to check this, and will do so automatically each time you run the *Timestamp.exe* program. If your processor does not support the RDTSC instruction, you will be notified with the message "processor does not support timestamp register".

**Introduction**

You may find [these notes](http://swjx.scu.edu.cn/moodle/file.php/61/notes.html) (see the Appendix) useful.

The hardware used in most computers today times internal functions in terms of a hardware clock, not to be confused with the system clock. The system clock simply keeps track of the current time, while, without going too in-depth, a hardware clock sends a regular electrical pulse along a wire, allowing different pieces of hardware to coordinate their activities. In particular, processors receive clock signals at a specific frequency and use these signal to coordinate the execution of instructions. Higher frequencies translate to faster execution. All processors have a maximum frequency at which they can operate; exceeding this frequency can cause the processor to not work properly, and sometimes can permanently damage the processor. Therefore, processors are rated by their manufacturers to run at frequencies up to a certain limit; this limit is the number often seen next to a processor's name. Thus, a 1.6GHz Pentium 4 processor is rated to safely receive 1.6 billion clock cycles per second. Configuring a computer to exceed this amount (called overclocking) is sometimes possible, but always dangerous.

The number of clock cycles required to run a given instruction varies from processor to processor; some processors take multiple cycles to execute instructions that another processor might execute in even less than a full clock cycle. For these and other reasons, clock speed is far from giving a complete idea of the speed with which a processor can execute a program, even a CPU-intensive program.

Beginning with the Pentium, Intel processors feature a special internal 64-bit register called the **timestamp counter**. On power-on, the value of this counter is zero. Every clock cycle, the processor increments the value in this counter by one. Since it's a 64-bit register, 264 cycles can pass before the register loops around to zero. 264 is 18,446,744,073,709,551,616. Dividing this number by your current clock frequency yields how many seconds it will take your computer to wrap around. For example, at exactly 2.0GHz, the processor will have to be on for 9223372037 seconds, or over 292 years.

When a preemptively multitasking operating system like Windows schedules a process, it arranges to have a signal sent to the kernel after a certain amount of time. On this signal, which is known as the **timer interrupt**, the kernel decides whether to switch out the current thread, or leave the current thread running. This decision is made by the scheduling policy of the operating system, and can take into account various factors such as thread priority, the time a thread has spent processing, and the time other threads have spent waiting to process. The length of time between timer interrupts is known as the **timer interval**. The timer interval is an important factor in the effectiveness of a preemptive multitasking operating system, since a timer interval that is too short will create a large amount of overhead and reduce overall system performance. On the other hand, a timer interval that is too long will allow programs to run much longer without interruption, thus reducing the illusion of many programs running simultaneously.

Through the power of the timestamp counter, it's possible to detect when the program has been preempted. As the first part of this exercise, you will both determine the timer interval used by your operating system and analyze data given to you to determine properties of your operating system.

**Getting Started**

[Here](http://swjx.scu.edu.cn/moodle/file.php/61/timestamp_v1.zip) is one of the programs you will be using for this section. The archive should contain the following files; you will want to create a new directory to unzip them in.

Timestamp.dsw

The Visual C++ Workspace for the project

Timestamp.dsp

The Visual C++ Project for this project

Timestamp.opt

Timestamp.plg

Timestamp.ncb

Timestamp.exe

A precompiled version of the code in this directory

tsc.h

An interface to code, which reads the timestamp counter

tsc.c

Assembly-language routines to read the timestamp counter

test.cpp

Contains function main and the routines for recording and reporting the timestamp information

Definitely take a look at *test.cpp*, specifically procedure spin. Procedure spin contains a for-loop that, upon each iteration, reads and stores the value of the timestamp register. Actually, there are a few for-loops in function spin; the for-loop that reads the timestamp register is labeled "Reading Timestamp". Run the precompiled executable and examine this source code to gain a good understanding of how this program operates. As you examine the source code, particularly function spin and the labeled "Reading Timestamp" for-loop, ask yourself the following questions:

* What is being stored in array table?
* What size is array table initialized to?
* At its peak, how many different instances of array table will be there when 4 threads are running?

Do not spend a long time studying this source code. Just read through it to get an idea of how the program works. Note that the file *Timestamp.exe* is precompiled directly from the source in this archive. If you decide to recompile for some reason, make sure you're compiling for "Release" version (as opposed to the "Debug" version); the debugger adds strange latencies and inefficiencies to the code that can be difficult to analyze.

**The Timestamp program - How it works**

You can run the program from inside Visual C++, but you won't be able to specify options or clearly read the output. Instead, you should start a command prompt, change to the project directory, and run the program (Timestamp.exe) from there. If you type "Timestamp -h", that will tell you about the options available to you.

This program will create a user specified number of threads that each executes a simple for-loop. For each execution of this for-loop, the value of the timestamp register is read and stored into an array. Each thread stores these values into a separate array. By examining the timestamp values in one of these arrays, the program can determine when a specific thread was running (and when it wasn't running). If a thread was running, that means it was executing the for-loop and storing consecutive values into its array that differ by very little. These values would differ only slightly simply because it does not take the processor very long to complete a loop iteration. For our purposes, a difference of 1 to 200 or so cycles is considered normal for a loop iteration. A difference between consecutive values in one of the thread arrays that is very large, say 20,000,000 cycles, means that the thread was not running during that timespan. The values in all of the arrays (from all the threads) can be processed into a timeline of when and for how long each thread in the program ran. We will use the timestamp program to help generate this timeline.

The basic usage of this program is straightforward. The program fills a table (an array named table) of a given size (this size can be set with the '-i' option, as in, "Timestamp -i 5000") with readings from the timestamp counter, one per iteration of the for-loop in procedure spin. After the program finishes the for-loop, a report is generated of when it thinks the thread was switched out. It does this by looking at the differences between consecutive timestamps; anything that exceeds a given threshold is considered a switch. By default, this threshold is the minimum -- that is, it is set to the minimum number of cycles elapsed during one iteration of the loop in function spin. Any difference between consecutive timestamps that is longer than the minimum threshold the program views as an interrupt. Sometimes, for various reasons, an iteration of the loop simply takes longer, and unfortunately the timestamp program will incorrectly view these instances as interrupts. Using the -t command-line option we can override the default minimum threshold and manually set the threshold that the timestamp program will use. Setting the threshold to a much higher value than the minimum will allow the timestamp program to better depict real interrupts by ignoring the instances where it just took the loop longer to execute. For example, on my 700Mhz machine running Windows NT 4.0, a threshold value of 1000 cycles (as opposed to my minimum calculated threshold of 42 cycles) will allow the timestamp program to only report actual interrupts. A final option of the timestamp program that we will use in this exercise is the ability to have the timestamp program create more than one thread. Using the -p command-line option, we can specify how many threads we would like to have continually reading the timestamp register. The default for this option is 1, but that is not very interesting, so in practice, we will set this option to usually around 4 or 5.

The results of this program are clearest when being executed on a computer with very little else running. Remember, the program picks up on any context switch, not just those caused by timer interrupts. Disk activity causes interrupts (to signal the end of specific transfers) which can make the data much more confusing.

**The graph program - How it works**

[Here](http://swjx.scu.edu.cn/moodle/file.php/61/graph.zip) is the second program that you will use for this section. This program will graphically depict the results of the Timestamp.exe program. This file should contain the following files; you will want to create a new directory to unzip them in.

graph.exe

The compiled version of the following source code files

graph.vbp

A Visual Basic 6.0 project file

frmGraph.frm

A Visual Basic 6.0 form file

clsData.cls

A Visual Basic 6.0 class module

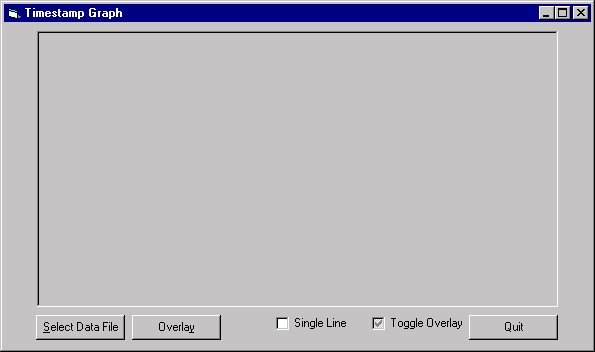
clsThread.cls

A Visual Basic 6.0 class module

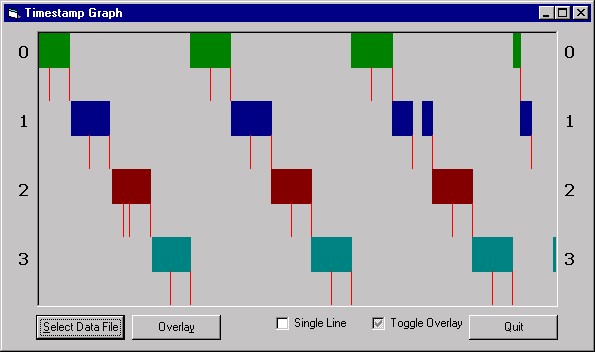
Feel free to browse the source code if you wish. To use the graph program, you will need the data generated from the timestamp program to be saved in a file with a .out extension. Use redirection to do this. For example, the following line will redirect the output of the timestamp program to a file named output.out:

Timestamp.exe -t 1000 -i 1000000 -p 4 > output.out

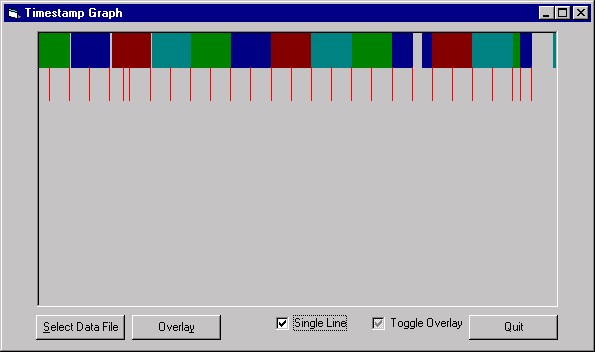
After saving the output of the timestamp program to a file, start the graph program by double-clicking on the file from Explorer, or using the Start-Run menu command. When you initially start the graph program you will see the following:

[](http://swjx.scu.edu.cn/moodle/file.php/49/systemlevelprogramming/week9/handout/graph1.jpg)

Clicking on the *Select Data File* button will bring up a dialog where you can browse to a .out timestamp program generated data file. After selecting a file, you will see (depending on the data in the actual file) something like this:

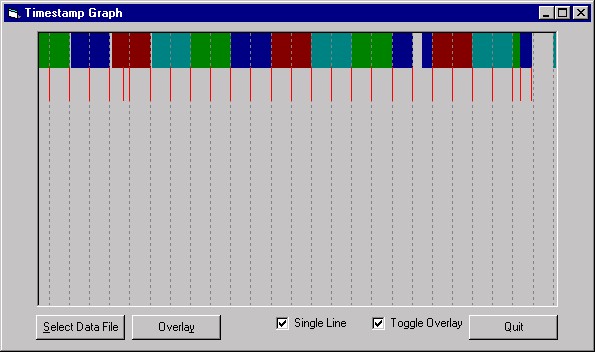


In the above screen shot of the graph program, we can easily see that data from four threads is depicted, since each thread is color coded, and we see four colors (green, blue, red, and cyan). Also, in the default viewing mode of the graph program each thread is depicted on a separate horizontal line, with thread numbers on the left and right of the graph. The graph, which simply shows when each thread was running, reads from left to right, so the green thread (thread 0) was the first thread to begin execution. The green thread encountered two interrupts, denoted by the red tick marks. Note that during the first interrupt the green thread was no longer running, but that since the length of this interrupt was so short (and the resolution of the monitor is limited), we see no visible gap in the display of the green thread execution. For this reason, the red tick marks are included, otherwise it would be difficult to see these very short times when the operating system code (and not the thread) is being run during the interrupt. During the second interrupt that occurs while the green thread is running, the operating system switches execution to the blue thread. The green thread does not run again until after the other three threads each run. To get a more linear view of the four threads, check the *Single Line* check box at the bottom of the interface. Something similar to this will result:



**Finding the timer interval**

Looking at the linear view of the threads in the screen shot above, we can see that most of the interrupts (the red ticks) are occurring at a regular interval. It is probably the case that these are timer interrupts. We can get a good guess at the timer interval by finding the difference between when these interrupts occur. To do this, hover your mouse pointer above an interrupt. A message will appear that gives information about the interrupt, namely, when the interrupt occurred and how long it lasted. By subtracting the starting points of two suspected timer interrupts, we can get a multiple of the timer interval. If these are two adjacent interrupts, then our result is the timer interval, and not a multiple of the timer interval. Performing this on the graph listed above, I got an estimate of 6,988,500 for my timer interval. We can test this hypothesis by overlaying our guess of the timer interval onto the graph. To do this, click the *Overlay* button. This option will first ask us to select an interrupt that we believe is a timer interrupt. Using the mouse, we point to any one of the several interrupts we believe is a timer interrupt. After clicking on an interrupt, we are then asked to input our calculated estimate of the timer interval. After typing in 6,988,500 the program then displays the following:



It is important to understand what the program did for us. The program drew the dashed gray lines at an interval of every 6,988,500 cycles (our guess of the interval), from the starting point of the interrupt (the red tick) that we picked. If we had picked an interrupt that was not a timer interrupt, the overlaid dashed gray lines would not match up to very many (if any) other interrupts.

If the timer interval is correct, and a timer interrupt is indeed picked as the starting point for the overlay, the overlaid dashed gray lines will match timer interrupts in **all** threads, not just one or two. If your overlaid estimate seems to match interrupts for only one thread and not the others, then your interrupts are not timer interrupts. In this case, you may have to increase the threshold (with the -t option) from the timestamp program, since you may be viewing periodic events (but not interrupts) in the course of the execution of a specific thread. Remember, timer interrupts are hardware generated and are periodic across all threads. This means that a timer interrupt will always occur every so many cycles (or seconds), and also, this period never shifts. In the screen shot listed above, we can easily see that our guess of the timer interval and our choice of an interrupt were correct, since our overlaid guess matches the periodic interrupts in all threads. Also, timestamp data (not shown here) with much smaller thresholds did not show any shorter periodic interrupts.

Now we can also see some other interesting facts. For instance, notice that the right most full timer interval contains none of our four threads. What happened here? Put simply enough, the operating system scheduled another process. After that process ran one full time interval, the OS switched back to the one of our four threads (thread 3, the cyan one), which needed only a very short time to complete execution. Something else that is interesting is the presence of non-timer interrupts. We see a few here. What are these? Most likely the first (which is the 5th re-tick from left) is an I/O interrupt, but the final two (the last two red-ticks on the right) actually are the green and blue threads completing execution before the end of a timer interval. During these interrupts, the operating system has to decide what to schedule next. In the case of the green thread finishing, the blue thread (thread 1) was scheduled next.

You will also need to know the approximate clock speed of your processor, since, to determine the timer interval in seconds, the length of the timer interval in cycles must be divided by the number of clock cycles the processor receives in one second. Make sure to remember the relationship between cycles and Hertz (Hz). Hertz is a measurement of cycles per second. So a 200MHz processor, by definition of Hertz (Hz), receives 200000000 (200 million) clock cycles in one second. Likewise, a 2GHz processor can safely receive 2000000000 (2 billion) clock cycles in one second. To get an estimate of the clock rate of your machine, execute the following: Timestamp -s. The program will determine how long it takes (in milliseconds and cycles) to execute a series of instructions. From this data it will output an estimate of the clock rate of your processor.

Also, typical timer intervals range between 1 and 10 milliseconds. So if you know your processor's clock rate, you can get an idea of what range to look in for your timer interval. For example, my processor is rated to receive 700 million cycles per second. So my timer interval should be between 700,000 and 7,000,000 cycles. This may seem like a large range of values, but in reality, 7,000,000 cycles go by very quickly (actually in 10ms on my 700Mhz machine).

**Your Task**

For this part of the exercise, run the timestamp program with 4 threads (the -p 4 command-line option) executing. Save the output to a file named output.out. Then, open this data file with the graph program and answer the following questions. You may have to run the timestamp program many times, with different options for the threshold and number of iterations to get data that clearly depicts the timer interval. If your machine is faster than 700Mhz, you may need to increase the number of iterations and/or decrease the threshold. For the record, the command line used on my 700Mhz machine to generate the data that the graph program screen shots display was as follows:

Timestamp -t 1000 -i 1000000 -p 4 > output.out

Hand in your answers to these questions in a file named *answers.txt*. **Also, make sure you submit the *output.out* file that you based the answers to the questions on.**

1. Using the timestamp program, what is the estimated clock rate of your processor in cycles per second? (2 points)
2. What is the approximate timer interval of your operating system? Report your answer in both cycles and in seconds. (14 points)
3. What are the minimum and maximum lengths of the timer interrupts that you observed? Do not count timer interrupts that switched to a different thread or different process. (2 points)
4. How many total interrupts did you record? (2 points)
5. How many of these were timer interrupts? (2 points)
6. How many timer interrupts resulted in context switches to another thread that belongs to the timestamp program? (3 points)
7. Did a timer interrupt resulted in a context switch to a different process? If so, how many times did this occur, and how many timer intervals did these other processes run? (3 points)
8. Did a thread finish executing before the end of a timer interval? If so, how many times did this happen and what did the operating system schedule for the rest of that timer interval? Was it one of the other timestamp threads, or another process? (3 points)
   1. How many timer intervals elapse from the time your program started to the time your program ended? (1 point)
   2. How many of those intervals did your program not run? (1 point)
9. Expressed as a hyphen delimited string of the label numbers, what was the execution order of the threads? (For example, from the data in the screen shots above, the execution order was 0-1-2-3-0-1-2-3-0-1-2-3-0-1-3) (2 points)

### Appendix： more information

### Introduction

There's a lot that has to happen when a computer is first turned on. The physical configuration of the computer has to be determined. Devices need to be checked to make sure they're working properly. All of the onboard hardware needs to be enabled and initialized. The hard drives have to be spun up and examined. The initial program has to be loaded into memory from the boot device. The processor has to configured and initialized -- oh, and on an IBM-compatible PC the BIOS initializes the RS bits of Status Register A on the MC146818 RTC to read 0110.

The MC146818 RTC -- that is, the Real Time Clock chip by Motorola, first included in the IBM AT computer and ever since a standard part of Intel-based systems -- is a reasonably simple example of the gritty details of our computers that we users rarely have to worry about. The RTC chip has two chief functions. The first is to generate periodic interrupts, if it's instructed to do so; this is useful for many reasons, some of which we'll get to later. The second function is maintaining the system clock. The chip stores the time in what's known as BCD format; that is, a single byte is divided into two 4-bit segments, each of which holds a decimal digit: so, for example, 58 (decimal) would be encoded as 01011000 (that is, 0x58). This is inefficient with space, but can easily be converted to a human-readable form. Also, the original MC146818 only stored two digits (one byte) for the year, which I seem to remember being relevant as the year 1999 came to its end. Newer chipsets use two bytes. Now, while you may or may not find all that interesting, there's a more important point here: the people in this world who have used a computer with a MC146818-compatible chipset vastly outnumber the people who know that they have used a computer with a MC146818-compatible chipset.

### Towards modern operating systems

Consider a closed box on a table. On the side of the box is a clock, which keeps perfect time, and a small dial, which can be used to set the hands on the clock. As long as the clock is ticking properly, the inner workings could be anything: it could run on batteries, or a pendulum, or lab rats, for that matter. This is the principle of abstraction at work: an interface between two objects should be as general as is possible, while allowing both objects to achieve what they require. A well-designed abstraction can be easily and completely applied anywhere where it can conceivably fit. A poorly-designed abstraction is bulky, or worse, impedes function. The whole point of abstraction is to make complicated things easier to use.

Operating systems are just abstractions between the hardware of a computer and the software running on that hardware. Now, let's build up what we mean by that.

A running process uses memory primarily in three ways. The first is its program code, which is loaded into memory just like any other data. The second is its global data, called the heap; this is where memory allocated by C's malloc() and C++ new() functions is found. The third is the stack, which holds local variables and the "stack trace" -- the nest of what functions are currently being executed by what functions.

A software program is just a set of instructions to be executed on a processor. The processor runs through these machine instructions, one by one (actually, modern processors are considerably more complex then this, but they still present the abstraction of running through the instructions one by one). These instructions generally fall into one of three categories:

1. Computation. Most processors operate on both integer types (short, int, long int, etc.) and floating-point types (float, double). These instructions range from additions and comparisons to more esoteric calculations like the Pentium MMX's RCPPS instruction, which takes the reciprocals of the four packed source operands, and REPNE CMPSB, which is basically the C strncmp() implemented in hardware.
2. Flow control. The processor checks to see if certain conditions are met, and if so, resumes executing instructions at the target address. Processors usually also have instructions specifically meant for calling other functions (which preserve the calling address) and the operating system (usually by creating a trap of some sort; more on this later).
3. Input/output. Processors can usually operate directly on memory, but it's considerably more efficient to work in a special in-processor memory area called the registers, and so most programs keep as much data as they can in registers. Input/output instructions copy data between the registers, main memory, and special I/O addresses which hardware devices listen to.

In addition, processors have a small set of control instructions which allow software to configure the processor.

To the processor, a program is just a set of these instructions. The organization of programs into functions and classes happens independently of the processor [footnote: at least, for standard processors; there are chips made to model the behavior of the Java Virtual Machine, and these do have to know something of the nature of functions and classes]. When a function calls a function, it stores the current instruction pointer (that is, the address of the next instruction to be executed) on the stack and jumps to the address of the function. When the function is ready to return, it pops the stored address off the stack and jumps to it.

The processor communicates with the hardware by way of a wire called the **system bus**; the various components of a computer send out signals on the bus, and the targets of the signals listen and respond. There are two possible ways for software to find out that something is happening with a hardware device. First, the software could constantly poll all the devices it's interested in; it sends a signal to the keyboard, say, and the keyboard responds with whether there have been any keystrokes recently. The problem is that all of those signals would completely tie up the bandwidth of the system bus -- and for nothing, since devices rarely have anything important to report. What would be nice is a way for the hardware to tell the software that something interesting is happening.

Fortunately, this is possible. When a piece of hardware has something to report, it sends the processor a signal called an **interrupt** or a **hardware trap**. The processor checks whether it's currently accepting interrupts from that device, and if so, it saves where it was in the program and jumps to an address in memory called an interrupt handler. The handler does its business and returns; the processor then picks up again with the program. In a primitive model, if a program wants to receive a certain type of event, it just installs its own handler for that interrupt -- and in fact, under MS-DOS, this is exactly what many programs do.

There a lot of problems with that primitive model, though. First and foremost, it's tricky business -- the programmer has to be familiar with the hardware in question, enough so to be able to interpret all of its signals and know what sort of responses it expects. If it could be one of several different devices, the programmer has to able to communicate with any of them, and this functionality has to be duplicated across every program which wants to use that device. As an example, on an IBM machine, the first parallel printer port signals on IRQ 7 (that is, it requests a processor interrupt of type 7). Printers are standardized, but only enough to print plain unformatted text without special instruction; special formatting, like graphical fonts, require commands that vary from printer to printer. DOS word processors like WordPerfect were distributed on dozens of disks, a large proportion of which were printer drivers. One of the innovations of Windows (at least, innovative in the PC world) was having a universal printing API so that users only needed a single Windows printer driver, rather than needing a special driver for each application.

There's another problem with programs having this sort of low-level access to the hardware. With a printer, a faulty program can hardly do worse than sending absolute garbage to the printer, but with a filesystem, a broken program can potentially destroy all of the data on the disk. It would be nice to have some sort of protection against misbehaving programs.

A third problem touches on a new topic: this sort of low-level access doesn't work with multiple programs running at once. Only one program can have its interrupt handler installed at once; one handler can certainly call the other, but there's no way of making sure that they aren't counteracting each other, or even overwriting each other's data. In fact, there are all sorts of issues involved in having multiple programs running at once: ideally, each should be independent of the other as much as it desires to be. No program should be able to monopolize the hardware; no program should be able to overwrite the data in the others; no program should be able to take down the computer.

### Multitasking operating systems

An operating system is called **multitasking** if it allows multiple threads of execution to overlap in their execution: that is, if it is capable of maintaining the illusion to the user that several threads of execution are being run at the same time, even if there is only one processor installed in the system.

Let's go over some terminology for a second. A **program** is a set of instructions to run on a processor. A **process** is a collection of threads of execution running a single program, and the data shared between those threads. A **thread of execution** is a processor context.

A processor context includes everything that is necessary for a program to run. For instance, each processor context has its own execution history and local data (stack), its own registers, and its own instruction pointer. It may have its own thread-local data, but more likely it just shares the same global data space (heap) as the rest of the process. The program code, and certain types of data, are read-only and thus can shared among multiple processes running the same program. All running processes have at least one thread.

So, for example, in Windows a user may have multiple Notepads open. Each Notepad is its own process, and probably keeps the document text on the heap. Notepad is a simple enough program that it may only use one thread, but a more complicated version might use multiple threads -- for example, if it were saving a document, it might spawn a thread to do that while leaving another to run the user interface. That thread would run in its own context, with its own stack, until it finished its task.

In a modern operating system, different threads in a process are protected from each other, just as different processes are protected from each other. This is accomplished by virtual memory, and there are several different types of it -- but for here, suffice it to say that when a processor in virtual-memory mode accesses a certain address, it's not directly addressing physical memory. Instead, the processor converts this virtual address into a physical address using special tables in memory, and each thread has its own copies of parts of these tables. Thus, one thread's stack isn't even addressable by another thread -- those addresses refer to its own stack.

Threads can't change these tables on their own -- if they could, that would effectively break the entire protection scheme. In order to make a change -- usually to allocate more memory -- they have to ask the operating system to do it for them. The action of a user process making a procedure call to the operating system is called a **system call**, or **syscall**; the exact method of making a system call depends on the underlying architecture and the operating system being considered, but the divide is common to all modern operating systems.

The reason why threads can't modify the virtual-memory tables is simple: modern operating systems execute in a mode of the processor called **protected mode**. The first Intel processor with a protected mode was the 80386, but the idea predates that considerably. In protected mode, the processor executes instructions at one of several different privilege levels. At the lowest privilege level, called **user mode**, the hardware and configuration instructions are usually restricted, and attempting to execute one will result in a processor fault, which generates a special interrupt. At the highest privilege level, called **supervisor mode** or **kernel mode**, no instructions are restricted; the part of the operating system which runs in supervisor mode -- that is, the part which handles syscalls and interrupts -- is known as the **kernel** of the operating system.

The services provided by the kernel vary greatly from one operating system to the next. There are widely-accepted specifications for operating systems called **POSIX**, and most operating systems have POSIX layers. UNIX systems fully implement POSIX. Windows versions descended from Windows NT (like Windows 2000 and XP, but not like Windows 95, 98, and ME) implement an early version of POSIX which existed when NT when first written, but there are significant compatibility problems. Versions of MacOS prior to OS X are not at all POSIX; the MacOS X kernel is a variant of the BSD kernel, and as such is fully POSIX. Whether or not an OS is POSIX-compliant impacts how easy it is to port applications to and from it; it is far more difficult to port an application from a UNIX computer to a Windows one than to port an application from one UNIX variant to another.

A word of caution about system calls. Switching from a user thread to the kernel imposes a fair amount of overhead. There's a lot of thread state that has to be saved and restored on each kernel entry. Worse, the kernel can't make assumptions about the arguments it's passed, so it has to ask questions like "Is this a valid string?", "Is this file actually open?", and "Does this pointer point to as much mapped memory as it claims to?". It's not something that will make a huge difference on a small scale, but a thread which makes a lot of system calls may notice a significant performance penalty. This isn't always a solvable problem; just keep it in mind, and try to accomplish as much as possible on each call.

Multitasking support is one of the more interesting parts of a kernel. In **cooperative multitasking**, threads must signal the operating system that they're willing to be switched out; MacOS versions prior to 10 cooperatively multitasked. Most operating systems, including UNIX, Windows, and Mac OS X, use **preemptive multitasking**. Usually, the kernel arranges to have signals sent to it at a certain interval (in the Intel architecture, conceivably by the MC146818 -- although the Pentium did add a more full-featured equivalent) and, whenever it gets such a signal ("pre-empting" the current thread), decides whether or not the active process should be changed and if so, what it should be changed to. The simplest algorithm for making this decision is to always answer "yes" and then switch to whatever thread was last executed the furthest time ago -- in effect, cycling through all active threads. Actual operating systems use more sophisticated algorithms, as do multi-processor systems. Changing from one thread to another is called a **context switch**.

There are several advantages to a multitasking system. Theoretically, a multitasking system running several independent processor-intensive tasks will be slightly slower than a comparable system running the tasks one at a time, due to the overhead of the timer interrupts. In reality, outside of scientific computing and CGI animation, very few tasks rely mostly on the processor. Most user applications spend an overwhelming percentage of their time waiting for input. Disk utilities wait for the disk. Network utilities wait for the network. Multitasking operating systems can take advantage of these latencies to schedule other threads for execution, resulting in a superior performance overall.

If you're interested, you can check what processes are currently running on your system. There's an optional tool that comes with Windows 95, 98, and ME called System Information which has this information. In Windows NT, 2000, and XP, the Task Manager has a process listing in one of its tabs. The command 'ps -A' will work the same way on a UNIX box.

Threads are a useful part of every programmer's toolbox, if for no other reason than that they allow a technique known as **event-driven programming**. Imagine, if you will, a finger program. finger is a protocol, used mostly on UNIX servers, which allows users to remotely gain specific, public information about the server's users. A finger utility opens a TCP/IP connection to the remote server, makes a query, and then reports back the response -- usually the user's name, office, and phone number. Normally, a finger utility isn't the sort of thing you'd need to thread, but let's imagine you want to have a finger utility which can finger more than one address. In pseudocode, there's a simple nonthreaded solution:

foreach address:

open connection

send query

get response

show response

However, this is inefficient. It could be that the second server will respond much faster than the first, or that one of the addresses will time out -- but even if they all take about the same time, waiting for the servers to respond one at a time will take longer than waiting for all of them at the same time. A threaded solution follows:

foreach address:

create thread:

open connection

send query

get response

show response

Now the program will complete in as much time as it takes for the slowest server to respond, rather than as much time as it takes all the servers altogether. However, there's an important problem. If two servers take about the same time to respond, then their threads will get the responses at about the same time -- and then start displaying the results at the same time. Now, the operating system can handle that, but the user will see something strange: the two responses, mingled it with each other! You could argue that this wouldn't happen with a single processor, since one would be guaranteed to start showing the output before the other, but you would be wrong. If one thread is printing the results line by line, then it will make a system call to print out that line. The system will start printing the line and block the thread until it's finished printing. Now, the second thread can be scheduled and send a line to the output; it'll block, but that line will still be printed before the next line from the first thread. You can see the problem.

Worse, subtle concurrency problems can exist in software that will only rarely be triggered. For instance, there's a subtle bug in the following C code (assume that thread1() and thread2() are, in fact, being run by two different threads).

int i = 10;

void thread1() {

while (i) {

printf( "%d\n", i );

i--;

}

}

void thread2() {

i--;

}

There's a miniscule chance that thread1() will loop around instead of stopping at 0. If, sometime during the "final" iteration (the one where i started out 1), thread1() is switched out for thread2(), and thread2() hasn't run at all yet, then i will be decremented twice to become -1, so that thread1() will have to run all the way through the integers before it finally ends.

Fortunately, operating systems provide services to solve issues like these, which are as a whole called **concurrency problems**.

[...]

Threads are not the solution to every problem, but they can elegantly solve many timing issues. If you ever find yourself polling some kind of state, waiting for it to change, you're better off spawning a thread to block until it changes -- this will significantly cut down on the number of system calls made, and will also allow other threads to run on the processor.