**Lab 09:Performance Measurement**

作业提交：文档id\_name.docx

包含practice1.profileing问题的回答

practice2. 你的参数Timestamp.exe -t ？-i ？-p 4 > output.out

类似图4的截图(10个问题的回答不需要提交，只需要思考)

**Practice 1: VC++ Profiling**

Read [profiling\_lab.doc](http://swjx.scu.edu.cn/moodle/file.php/49/systemlevelprogramming/week9/profiling_lab.doc) and answer the questions by writing them on a word document.

**Practice 2: The kernel timer interval**

Instructor's little [explaination](http://swjx.scu.edu.cn/moodle/file.php/49/systemlevelprogramming/week9/lab09-2ä¸­æç®è¿°.doc). 关闭多核

多核或者双核电脑关单核程序.

按开始菜单--运行--输入“msconfig”(无引号)后回车调出系统配置实用程序--选择“BOOT.INI引导“选项卡--点击“高级选项”--把“/NUMPROC(N)=”前面的方框打钩(此时默认数字是1)--确定后系统提示重新启动计算机 重启后即只开一个核

原理:很简单从系统启动时就自动关闭一个核心 也就是在加载系统时以一个核心来运行系统 BOOT.INI中设置的/NUMPROC是设置cpu核心数量的设置 双核cpu可以设置成1(单核) 设置成2或者把前面对钩去掉都是设回双核 每次设置确定后必须重新启动计算机......

**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/49/systemlevelprogramming/week9/handout/notes.html) 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, 264cycles 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/49/systemlevelprogramming/week9/handout/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/49/systemlevelprogramming/week9/handout/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

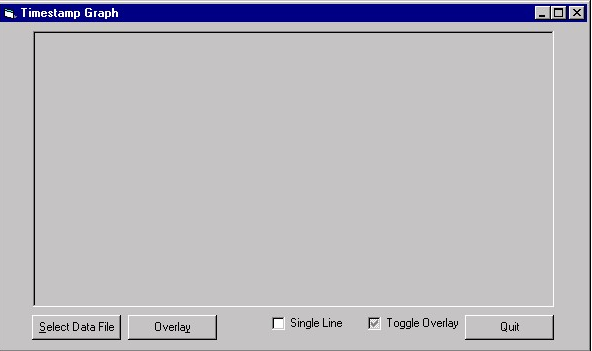
clsThread.cls

A Visual Basic 6.0 class module

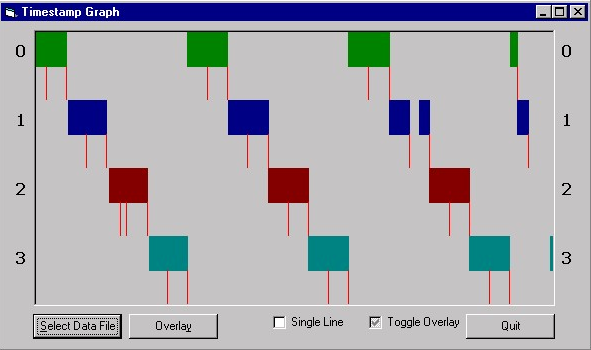
Feel free to browse the source code if you wish. To use the graph program, you will need the data generated from the timestamp program to be saved in a file with a .out extension. Use redirection to do this. For example, the following line will redirect the output of the timestamp program to a file named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

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

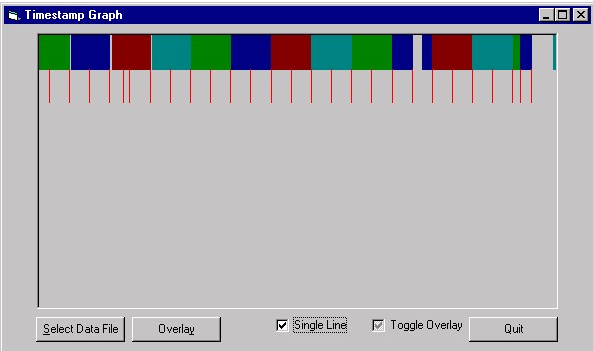
After saving the output of the timestamp program to a file, start the graph program by double-clicking on the file from Explorer, or using the Start-Run menu command. When you initially start the graph program you will see the following:



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:

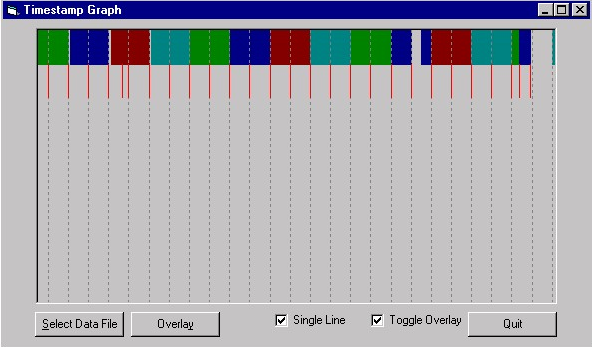


In the above screen shot of the graph program, we can easily see that data from four threads is depicted, since each thread is color coded, and we see four colors (green, blue, red, and cyan). Also, in the default viewing mode of the graph program each thread is depicted on a separate horizontal line, 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) 2962681000
2. What is the approximate timer interval of your operating system? Report your answer in both cycles and in seconds. (14 points)

2303994 7.77\*10^-4

1. 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)

2303994 738972

1. How many total interrupts did you record? (2 points) 10
2. How many of these were timer interrupts? (2 points) 9
3. How many timer interrupts resulted in context switches to another thread that belongs to the timestamp program? (3 points) 12
4. Did a timer interrupt resulted in a context switch to a different process?

Yes

If so, how many times did this occur, and how many timer intervals did these other processes run? (3 points) 2 1

1. Did a thread finish executing before the end of a timer interval?

Yes

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 another process.

* 1. How many timer intervals elapse from the time your program started to the time your program ended? (1 point) 16
  2. How many of those intervals did your program not run? (1 point) 8

1. 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)

0-1-2-3-0-1-2-3-0-1-2-3-0-1-2-3