Chapter 9

Measuring Program Execution Time

One common question people ask is "How fast does Program X run on Machine Y?" Such a question might be raised by a programmer trying to optimize program performance, or by a customer trying to decide which machine to buy. In our earlier discussion of performance optimization (Chapter 5), we assumed this question could be answered with perfect accuracy. We were trying to establish the cycles per element (CPE) measure for programs down to two decimal places. This requires an accuracy of 0.1% for a procedure having a CPE of 10. In this chapter, we address this problem and discover that it is surprisingly complex.

You might expect that making near-perfect timing measurements on a computer system would be straightforward. After all, for a particular combination of program and data, the machine will execute a fixed sequence of instructions. Instruction execution is controlled by a processor clock that is regulated by a precision oscillator. There are many factors, however, that can vary from one execution of a program to another. Computers do not simply execute one program at a time. They continually switch from one process to another, executing some code on behalf of one process before moving on to the next. The exact scheduling of processor resources for one program depends on such factors as the number of users sharing the system, the network traffic, and the timing of disk operations. The access patterns to the caches depend not just on the references made by the program we are trying to measure, but on those of other processes executing concurrently. Finally, the branch prediction logic tries to guess whether branches will be taken or not based on past history. This history can vary from one execution of a program to another.

In this chapter, we describe two basic mechanisms used by computers to record the passage of time—one based on a low frequency timer that periodically interrupts the processor and one based on a counter that is incremented every clock cycle. Application programmers can gain access to the first timing mechanism by calling library functions. Cycle timers can be accessed by library functions on some systems, but they require writing assembly code on others. We have deferred the discussion of program timing until now, because it requires understanding aspects of both the CPU hardware and the way the operating system manages process execution.

Using the two timing mechanisms, we investigate methods to get reliable measurements of program performance. We will see that timing variations due to context switching tend to be very large and thus must be eliminated. Variations caused by other factors such as cache and branch prediction are generally managed by evaluating program operation under carefully controlled conditions. Generally, we can get accurate measurements for durations that are either very short (less than around 10 millisecond) or very long (greater than

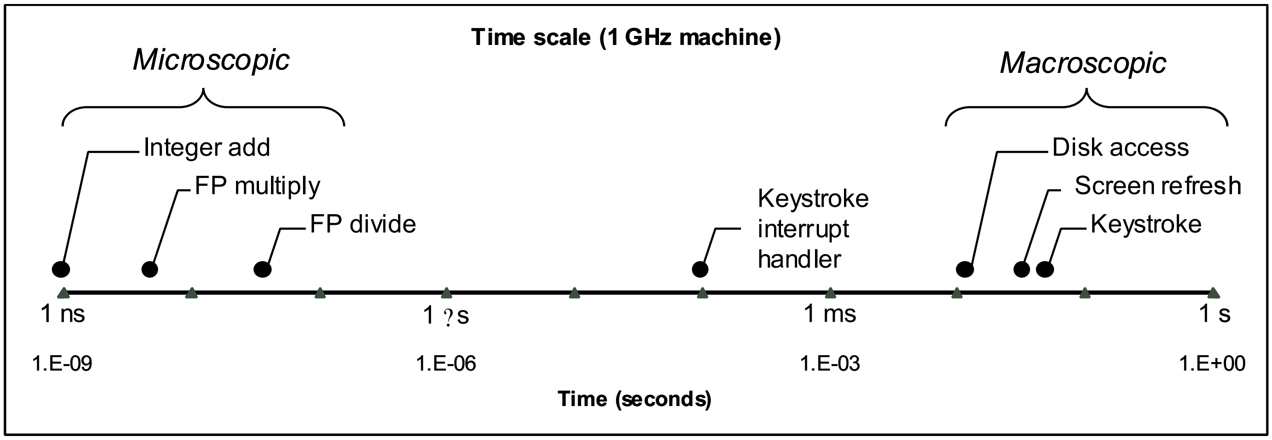


Figure 9.1: **Time scale of computer system events.** The processor hardware works at a microscopic a time scale in which events having durations on the order of a few nanoseconds (ns). The OS must deal on a macroscopic time scale with events having durations on the order of a few milliseconds (ms).

around 1 second), even on heavily loaded machines. Times between around 10 milliseconds and 1 second require special care to measure accurately.

Much of the understanding of performance measurement is part of the folklore of computer systems. Different groups and individuals have developed their own techniques for measuring program performance, but there is no widely available body of literature on the subject. Companies and research groups concerned with getting highly accurate performance measurements often set up specially configured machines that minimize any sources of timing irregularity, such as by limiting access and by disabling many OS and networking services. We want methods that application programmers can use on ordinary machines, but there are no widely available tools for this. Instead, we will develop our own.

In this presentation, we work through the issues systematically. We describe the design and evaluation of a number of experiments that helped us arrive at methods to achieve accurate measurements on a small set of systems. It is unusual to find a detailed experimental study in a book at this level. Generally, people expect the final answers, not a description of how those answers were determined. In this case, however, we cannot provide definitive answers on how to measure program execution time for an arbitrary program on an arbitrary system. There are too many variations of timing mechanisms, operating system behaviors, and run-time environment to have one single, simple solution. Instead, we anticipate that you will need to run your own experiments and develop your own performance measurement code. We hope that our case study will help you in this task. We summarize our findings in the form of a protocol that can guide your experiments.

9.1 The Flow of Time on a Computer System

Computers operate on two fundamentally different time scales. At a microscopic level, they execute instructions at a rate of one or more per clock cycle, where each clock cycle requires only around one nanosecond (abbreviated “ns”), or 10^{-9} seconds. On a macroscopic scale, the processor must respond to external events that occur on time scales measured in milliseconds (abbreviated "ms"), or 10^{-3} seconds. For example, during video playback, the graphics display for most computers must be refreshed every 33 ms. A world-record typist can only type keystrokes at a rate of around one every 50 milliseconds. Disks typically require around 10 ms to initiate a disk transfer. The processor continually switches between these many tasks on a macroscopic time scale, devoting around 5 to 20 milliseconds to each task at a time. At this rate, the user perceives the tasks as being performed simultaneously, since a human cannot discern time durations shorter than around 100 ms. Within that time the processor can execute millions of instructions.

Figure 9.1 plots the durations of different event types on a logarithmic scale, with microscopic events having durations measured in nanoseconds and macroscopic events having durations measured in milliseconds. The macroscopic events are managed by OS routines that require around 5,000 to 200,000 clock cycles. These time ranges are measured in microseconds (abbreviated μs, where μ is the Greek letter "mu"). Although that may sound like a lot of computation, it is so much faster than the macroscopic events being processed that these routines place only a small load on the processor.

Practice Problem 9.1:

When a user is editing files with a real-time editor such as EMACS, every keystroke generates an interrupt signal. The operating system must then schedule the editor process to take the appropriate action for this keystroke. Suppose we had a system with a 1 GHz clock, and we had 100 users running EMACS typing at a rate of 100 words per minute. Assume an average of 6 characters per word. Assume also that the OS routine handling keystrokes requires, on average, 100,000 clock cycles per keystroke. What fraction of the processor load is consumed by all of the keystroke processing?

Note that this is a very pessimistic analysis of the load induced by keyboard usage. It's hard to imagine a real-life scenario with so many users typing this fast.

9.1.1 Process Scheduling and Timer Interrupts

External events such as keystrokes, disk operations, and network activity generate interrupt signals that make the operating system scheduler take over and possibly switch to a different process. Even in the absence of such events, we want the processor to switch from one process to another so that it will appear to the users as if the processor is executing many programs simultaneously. For this reason, computers have an external timer that periodically generates an interrupt signal to the processor. The spacing between these interrupt signals is called the *interval time*. When a timer interrupt occurs, the operating system scheduler can choose to either resume the currently executing process or to switch to a different process. This interval must be set short enough to ensure that the processor will switch between tasks often enough to provide the illusion of performing many tasks simultaneously. On the other hand, switching from one process to another requires thousands of clock cycles to save the state of the current process and to set up the state for the next, and so setting the interval too short would cause poor performance. Typical timer intervals range between 1 and 10 milliseconds, depending on the processor and how it is configured.

Aside: Scaling of Computer Performance

It is interesting to compare the performance of a Digital Equipment Corporation VAX-11/780 computer to a modern processor. This machine was introduced in 1977 with an entry price of around $200,000. It became the first widely used machine running the Unix operating system. Note that the timer interval on this machine was typically set to 10 ms, even though its CPU was over 1000 times slower than that of a modern machine. The macroscopic time scale has not changed even though the microscopic time scale has become much faster. **End Aside.**

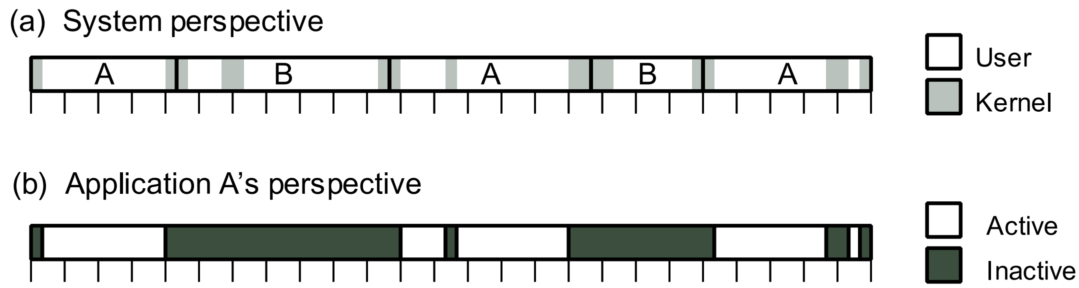


Figure 9.2: **System's vs. applications view of time.** The system switches from process to process, operating in either user or kernel mode. The application only gets useful computation done when its process is executing in user mode.

Figure 9.2(a) illustrates the system's perspective of a hypothetical 150 ms of operation on a system with a 10 ms timer interval. During this period there are two active processes: A and B. The processor alternately executes part of process A, then part of B, and so on. As it executes these processes, it operates either in *user mode*, executing the instructions of the application program; or in *kernel mode*, performing operating system functions on behalf of the program, such as, handling page faults, input, or output. Recall that kernel operation is considered part of each regular process rather than a separate process. The operating system scheduler is invoked every time there is an external event or a timer interrupt. The occurrences of timer interrupts are indicated by the tick marks in the figure. This means that there is actually some amount of kernel activity at every tick mark, but for simplicity we do not show it in the figure.

When the scheduler switches from process A to process B, it must enter kernel mode to save the state of process A (still considered part of process A) and to restore the state of process B (considered part of process B). Thus, there is kernel activity during each transition from one process to another. At other times, kernel activity can occur without switching processes, such as when a page fault can be satisfied by using a page that is already in memory.

9.1.2 Time from an Application Program's Perspective

From the perspective of an application program, the flow of time can be viewed as alternating between periods when the program is *active* (executing its instructions), and *inactive* (waiting to be scheduled by the operating system). It only performs useful computation when its process is operating in user mode. Figure 9.2(b) illustrates how program A would view the flow of time. It is active during the light-colored regions, when process A is executing in user mode; otherwise it is inactive.

As a way to quantify the alternations between active and inactive time periods, we wrote a program that continuously monitors itself and determines when there have been long periods of inactivity. It then generates a *trace* showing the alternations between periods of activity and inactivity. Details of this program are described later in the chapter. An example of such a trace is shown in Figure 9.3, generated while running on a Linux machine with a clock rate of around 550 MHz. Each period is labeled as either active ("A") or inactive "I"). The periods are numbered 0 to 9 for identification. For each period, the start time (relative to the beginning of the trace) and the duration are indicated. Times are expressed in both clock cycles and milliseconds. This trace shows a total of 20 time periods (10 active and 10 inactive) having a total duration

```
A0    Time        0    (0.00 ms),   Duration  3726508     (6.776448 ms)
I0    Time  3726508    (6.78 ms),   Duration   275025     (0.500118 ms)
A1    Time  4001533    (7.28 ms),   Duration        0     (0.000000 ms)
I1    Time  4001533    (7.28 ms),   Duration     7598     (0.013817 ms)
A2    Time  4009131    (7.29 ms),   Duration  5189247     (9.436358 ms)
I2    Time  9198378   (16.73 ms),   Duration   251609     (0.457537 ms)
A3    Time  9449987   (17.18 ms),   Duration  2250102     (4.091686 ms)
I3    Time 11700089   (21.28 ms),   Duration    14116     (0.025669 ms)
A4    Time 11714205   (21.30 ms),   Duration  2955974     (5.375275 ms)
I4    Time 14670179   (26.68 ms),   Duration   248500     (0.451883 ms)
A5    Time 14918679   (27.13 ms),   Duration  5223342     (9.498358 ms)
I5    Time 20142021   (36.63 ms),   Duration   247113     (0.449361 ms)
A6    Time 20389134   (37.08 ms),   Duration  5224777     (9.500967 ms)
I6    Time 25613911   (46.58 ms),   Duration   254340     (0.462503 ms)
A7    Time 25868251   (47.04 ms),   Duration  3678102     (6.688425 ms)
I7    Time 29546353   (53.73 ms),   Duration     8139     (0.014800 ms)
A8    Time 29554492   (53.74 ms),   Duration  1531187     (2.784379 ms)
I8    Time 31085679   (56.53 ms),   Duration   248360     (0.451629 ms)
A9    Time 31334039   (56.98 ms),   Duration  5223581     (9.498792 ms)
I9    Time 36557620   (66.48 ms),   Duration   247395     (0.449874 ms)
```

Figure 9.3: **Example trace showing activity periods.** From the perspective of an application program, processor operation alternates between periods when the program is actively executing (italicized) and when it is inactive. This trace shows a log of these periods for a program over a total duration of 66.9 ms. The program was active for 95.1% of this time.

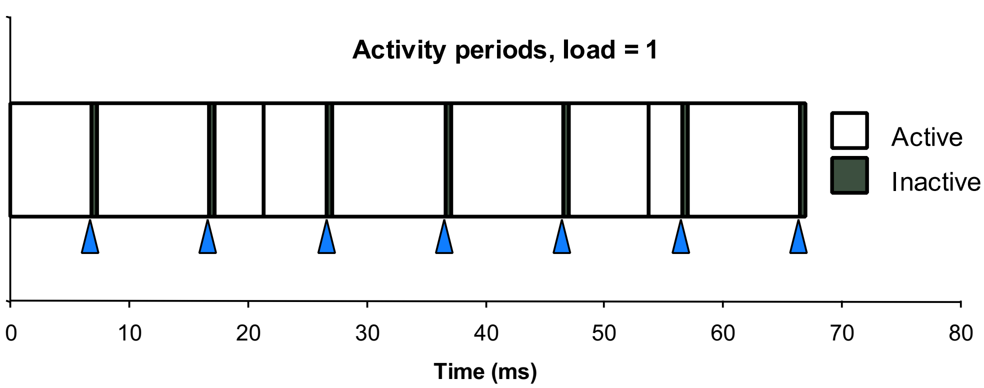


Figure 9.4: **Graphical representation of trace in Figure 9.3.** Timer interrupts are indicated with blue triangles.

```
 A48   Time 191514104   (349.40 ms),   Duration  5224961   (9.532449 ms)
I48   Time 196739065   (358.93 ms),   Duration   247557   (0.451644 ms)
 A49   Time 196986622   (359.38 ms),   Duration   858571   (1.566382 ms)
I49   Time 197845193   (360.95 ms),   Duration     8297   (0.015137 ms)
 A50   Time 197853490   (360.97 ms),   Duration  4357437   (7.949733 ms)
I50   Time 202210927   (368.91 ms),   Duration  5718758  (10.433335 ms)
 A51   Time 207929685   (379.35 ms),   Duration  2047118   (3.734774 ms)
I51   Time 209976803   (383.08 ms),   Duration     7153   (0.013050 ms)
 A52   Time 209983956   (383.10 ms),   Duration  3170650   (5.784552 ms)
I52   Time 213154606   (388.88 ms),   Duration  5726129  (10.446783 ms)
 A53   Time 218880735   (399.33 ms),   Duration  5217543   (9.518916 ms)
I53   Time 224098278   (408.85 ms),   Duration  5718135  (10.432199 ms)
 A54   Time 229816413   (419.28 ms),   Duration  2359281   (4.304286 ms)
I54   Time 232175694   (423.58 ms),   Duration     7096   (0.012946 ms)
 A55   Time 232182790   (423.60 ms),   Duration  2859227   (5.216390 ms)
I55   Time 235042017   (428.81 ms),   Duration  5718793  (10.433399 ms)
```

Figure 9.5: **Example trace showing activity periods on loaded machine.** When other active processes are present, the tracing process is inactive for longer periods of time. This trace shows a log of these periods for a program over a total duration of 89.8 ms. The process was active for 53.0% of this time.

of 66.9 ms. In this example, the periods of inactivity are fairly short, with the longest being 0.50 ms. Most of these periods of inactivity were caused by timer interrupts. The process was active for around 95.1% of the total time monitored. Figure 9.4 shows a graphical rendition of the trace shown in Figure 9.3. Observe the regular spacing of the boundaries between the activity periods indicated by the blue triangles. These boundaries are caused by timer interrupts.

Figure 9.5 shows a portion of a trace when there is one other active process sharing the processor. The graphical rendition of this trace is shown in Figure 9.6. Note that the time scales do not line up, since the portion of the trace we show in Figure 9.5 started at 349.40 ms into the tracing process. In this example we can see that while handling some of the timer interrupts, the OS also decides to switch context from one

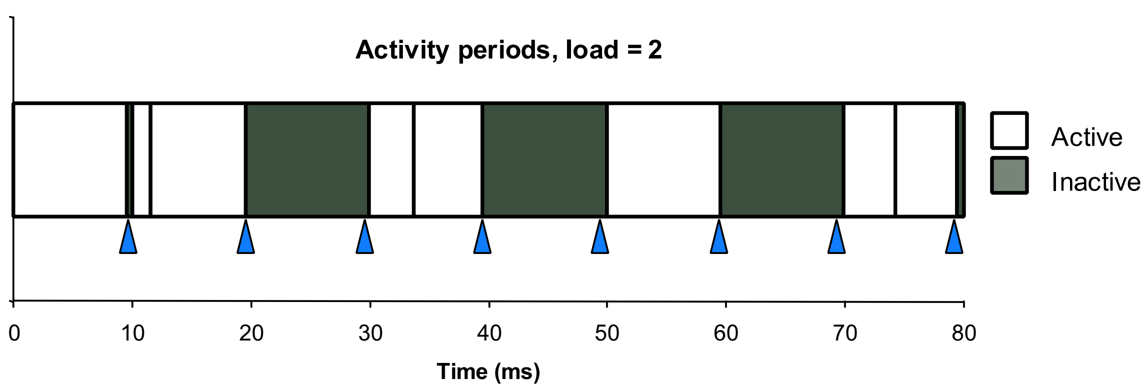


Figure 9.6: **Graphical representation of activity periods for trace in Figure 9.5.** Timer interrupts are indicated by blue triangles

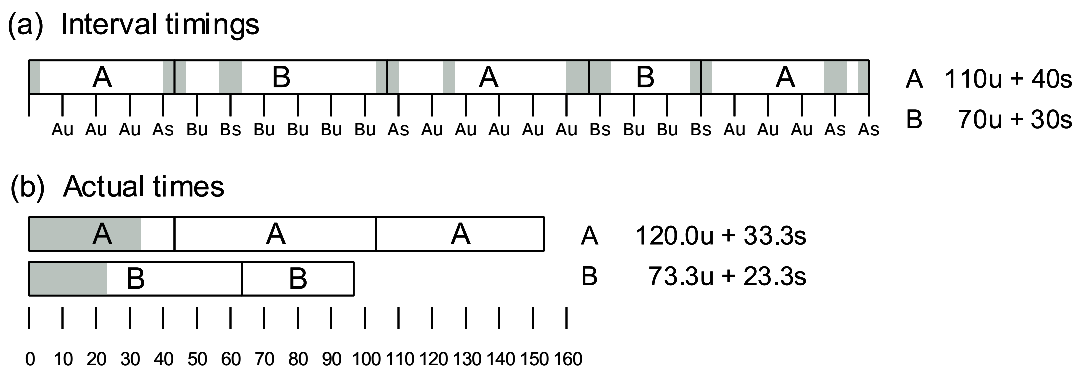


Figure 9.7: **Process timing by interval counting.** With a timer interval of 10 ms, every 10 ms segment is assigned to a process as part of either its user (u) or system (s) time. This accounting provides only an approximate measure of program execution time.

process to another. As a result, each process is only active around 50% of the time.

Practice Problem 9.2:

This problem concerns the interpretation of the section of the trace shown in Figure 9.5.

A. At what times during this portion of the trace did timer interrupts occur? (Some of these time points can be extracted directly from the trace, while others must be estimated by interpolation.)

B. Which of these occurred while the tracing process was active, and which while it was inactive?

C. Why are the longest periods of inactivity longer than the longest periods of activity?

D. Based on the pattern of active and inactive periods shown in this trace, what percent of the time would you expect the tracing process to be inactive when averaged over a longer time scale?

9.2 Measuring Time by Interval Counting

The operating system also uses the timer to record the cumulative time used by each process. This information provides a somewhat imprecise measure of program execution time. Figure 9.7 provides a graphic illustration of how this accounting works for the example of system operation shown in Figure 9.2. In this discussion, we refer to the period during which just one process executes as a *time segment*.

9.2.1 Operation

The operating system maintains counts of the amount of user time and the amount of system time used by each process. When a timer interrupt occurs, the operating system determines which process was active and increments one of the counts for that process by the timer interval. It increments the system time if the system was executing in kernel mode, and the user time otherwise. The example shown in Figure 9.7(a) indicates this accounting for the two processes. The tick marks indicate the occurrences of timer interrupts.

Each is labeled by the count that gets incremented: either `Au` or `As` for process A's user or system time, or `Bu` or `Bs` for process B's user or system time. Each tick mark is labeled according to the activity to its immediate left. The final accounting shows that process A used a total of 150 milliseconds: 110 of user time and 40 of system time. It shows that B used a total of 100 milliseconds: 70 of user time and 30 of system time.

9.2.2 Reading the Process Timers

When executing a command from the Unix shell, the user can prefix the command with the word "`time`" to measure the execution time of the command. This command uses the values computed using the accounting scheme described above. For example, to time the execution time of program `prog` with command line arguments `-n 17`, the user can simply type the command

```
unix> time prog -n 17
```

After the program has executed, the shell will print a line summarizing the run time statistics, such as the following:

```
2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w
```

The first three numbers shown in this line are times. The first two show the seconds of user and system time. Observe how both of these show a 0 in the third decimal place. With a timer interval of 10 ms, all timings are multiples of hundredths of seconds. The third number is the total elapsed time, given in minutes and seconds. Observe that the system and user time sum to 2.49 seconds, less than half of the elapsed time of 6.52 seconds, indicating that the processor was executing other processes at the same time. The percentage indicates what fraction the combined user and system times were of the elapsed time, e.g., $(2.23 + 0.26)/6.52 = 0.381$. The remaining statistics summarize the paging and I/O behavior.

Programmers can also read the process timers by calling the library function `times`, declared as follows:

```
 #include <sys/times.h>

struct tms {
   clock_t tms_utime; /* user time */
   clock_t tms_stime; /* system time */
   clock_t tms_cutime; /* user time of reaped children */
   clock_t tms_cstime; /* system time of reaped children */
};

clock_t times(struct tms *buf);
```

Returns: number of clock ticks elapsed since system started

These time measurements are expressed in terms of a unit called *clock ticks*. The defined constant `CLK_TCK` specifies the number of clock ticks per second. The data type `clock_t` is typically defined to be a long integer. The fields indicating child times give the accumulated times used by children that have terminated and have been reaped. Thus, `times` cannot be used to monitor the time used by any ongoing children. As a return value, `times` returns the total number of clock ticks that have elapsed since the system was started.

We can therefore compute the total time (in clock ticks) between two different points in a program execution by making two calls to `times` and computing the difference of the return values.

The ANSI C standard also defines a function `clock` that measures the total time used by the current process:

```
#include <time.h>

clock_t clock(void);
```

Returns: total time used by process

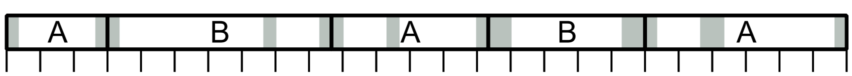
Although the return value is declared to be the same type `clock_t` used with the `times` function, the two functions do not, in general, express time in the same units. To scale the time reported by `clock` to seconds, it should be divided by the defined constant `CLOCKS_PER_SEC`. This value need not be the same as the constant `CLK_TCK`.

9.2.3 Accuracy of Process Timers

As the example illustrated in Figure 9.7 shows, this timing mechanism is only approximate. Figure 9.7(b) shows the actual times used by the two processes. Process A executed for a total of 153.3 ms, with 120.0 in user mode and 33.3 in kernel mode. Process B executed for a total of 96.7 ms, with 73.3 in user mode and 23.3 in kernel mode. The interval accounting scheme makes no attempt to resolve time more finely than the timer interval.

Practice Problem 9.3:

What would the operating system report as the user and system times for the execution sequence illustrated below. Assume a 10 ms timer interval.



Practice Problem 9.4:

On a system with a timer interval of 10 ms, some segment of process A is recorded as requiring 70 ms, combining both system and user time. What are the minimum and maximum actual times used by this segment?

Practice Problem 9.5:

What would the counters record as the system and user times for the trace shown in Figure 9.3? How does this compare with the actual time during which the process was active?

For programs that run long enough, (at least several seconds), the inaccuracies in this scheme tend to compensate for each other. The execution times of some segments are underestimated while those of others are overestimated. Averaged over a number of segments, the expected error approaches zero. From a theoretical

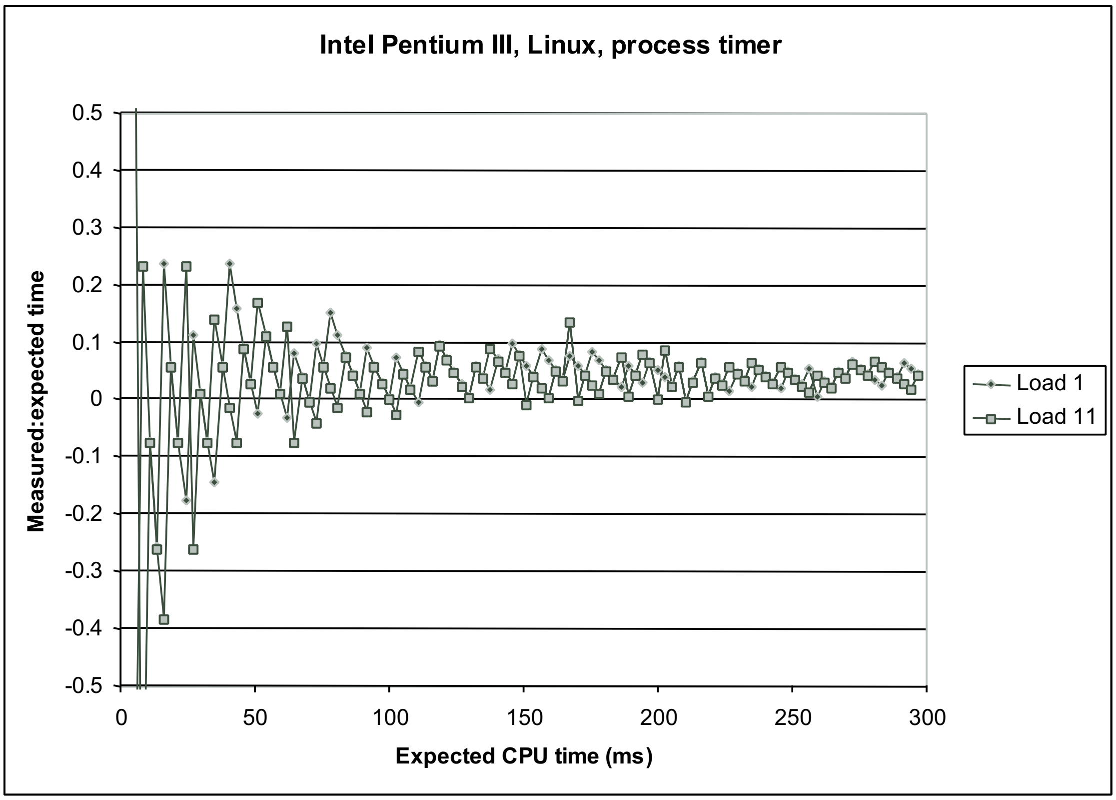


Figure 9.8: **Measuring interval counting accuracy.** The error is unacceptably high when measuring activities less than around 100 ms (10 timer intervals). Beyond this, the error rate is generally less than 10% regardless of whether running on lightly loaded (Load 1) or heavily loaded (Load 11) machine.

perspective, however, there is no guaranteed bound on how far these measurements vary from the true run times.

To test the accuracy of this timing method, we ran a series of experiments that compared the time T_m measured by the operating system for a sample computation with our estimate of what the time T_c would be if the system resources were dedicated solely to performing this computation. In general, T_c will differ from T_m for several reasons:

1. The inherent inaccuracies of the interval counting scheme can cause T_m to be either less or greater than T_c.

2. The kernel activity caused by the timer interrupt consumes 4 to 5% of the total CPU cycles, but these cycles are not accounted for properly. As can be seen in the trace illustrated in Figure 9.4, this activity finishes before the next timer interrupt and hence does not get counted explicitly. Instead, it simply reduces the number of cycles available for the process executing during the next time interval. This will tend to increase T_m relative to T_c.

3. When the processor switches from one task to another, the cache tends to perform poorly for a transient period until the instructions and data for the new task get loaded into the cache. Thus, the processor does not run as efficiently when switching between our program and other activities as it would if it executed our program continuously. This factor will tend to increase T_m relative to T_c.

We discuss how we can determine the value of T_c for our sample computation later in this chapter.

Figure 9.8 shows the results of this experiment running under two different loading conditions. The graphs show our measurements of the error rate, defined as the value of $(T_m - T_c)/T_c$ as a function of T_c. This error measure is negative when T_m underestimates T_c and is positive when T_m overestimates T_c. The two series show measurements taken under two different loading conditions. The series labeled "Load 1" shows the case where the process performing the sample computation is the only active process. The series labeled "Load 11" shows the case where 10 other processes are also attempting the same computation. The latter represents a very heavy load condition; the system is noticeably slow responding to keystrokes and other service requests. Observe the wide range of error values shown on this graph. In general, only measurements that are within $\pm 10\%$ of the true value are acceptable, and hence we want only errors ranging from around -0.1 to $+0.1$.

Below around 100 ms (10 timer intervals), the measurements are not at all accurate due to the coarseness of the timing method. Interval counting is only useful for measuring relatively long computations—100,000,000 clock cycles or more. Beyond this, we see that the error generally ranges between 0.0 and 0.1, that is, up to 10% error. There is no noticeable difference between the two different loading conditions. Notice also that the errors have a positive bias: The average error for all measurements with $T_m \geq 100$ms is 1.04, due to the fact that the timer interrupts are consuming around 4% of the CPU time.

These experiments show that the process timers are useful only for getting approximate values of program performance. They are too coarse grained to use for any measurement having duration of less than 100 ms. On this machine they have a systematic bias, overestimating computation times by an average of around 4%. The main virtue of this timing mechanism is that its accuracy does not depend strongly on the system load.

9.3 Cycle Counters

To provide greater precision for timing measurements, many processors also contain a timer that operates at the clock cycle level. This timer is a special register that gets incremented every single clock cycle. Special machine instructions can be used to read the value of the counter. Not all processors have such counters, and those that do vary in the implementation details. As a result, there is no uniform, platform-independent interface by which programmers can make use of these counters. On the other hand, with just a small amount of assembly code, it is generally easy to create a program interface for any specific machine.

9.3.1 IA32 Cycle Counters

All of the timings we have reported thus far were measured using the IA32 cycle counter. With the IA32 architecture, cycle counters were introduced in conjunction with the "P6" microarchitecture (the PentiumPro and its successors). The cycle counter is a 64-bit, unsigned number. For a processor operating with a 1 GHz clock, this counter will wrap around from $2^{64} - 1$ to 0 only once every 1.8×10^{10} seconds, or every 570 years. On the other hand, if we consider only the low order 32 bits of this counter as an unsigned integer, this value will wrap around every 4.3 seconds. One can therefore understand why the IA32 designers decided to implement a 64-bit counter.

The IA32 counter is accessed with the `rdtsc` (for "read time stamp counter") instruction. This instruction takes no arguments. It sets register `%edx` to the high-order 32 bits of the counter and register `%eax` to the low-order 32 bits. To provide a C program interface, we would like to encapsulate this instruction within a procedure:

```
void access_counter(unsigned *hi, unsigned *lo);
```

This procedure should set location `hi` to the high-order 32 bits of the counter and `lo` to the low-order 32 bits. Implementing `access_counter` is a simple exercise in using the embedded assembly feature of GCC, as described in Section 3.15. The code is shown in Figure 9.9.

Based on this routine, we can now implement a pair of functions that can be used to measure the total number of cycles that elapse between any two time points:

```
#include "clock.h"

void start_counter();
double get_counter();
```

Returns: number of cycles since last call to `start_counter`

We return the time as a `double` to avoid the possible overflow problems of using just a 32-bit integer. The code for these two routines is also shown in Figure 9.9. It builds on our understanding of unsigned arithmetic to perform the double-precision subtraction and to convert the result to a `double`.

code/perf/clock.c

```
/* Initialize the cycle counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;


/* Set *hi and *lo to the high and low order bits  of the cycle counter.
   Implementation requires assembly code to use the rdtsc instruction. */
void access_counter(unsigned *hi, unsigned *lo)
{
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"   /* Read cycle counter */
        : "=r" (*hi), "=r" (*lo)                /* and move results to */
        : /* No input */                        /* the two outputs */
        : "%edx", "%eax");
}

/* Record the current value of the cycle counter. */
void start_counter()
{
    access_counter(&cyc_hi, &cyc_lo);
}

/* Return the number of cycles since the last call to start_counter. */
double get_counter()
{
    unsigned ncyc_hi, ncyc_lo;
    unsigned hi, lo, borrow;
    double result;

    /* Get cycle counter */
    access_counter(&ncyc_hi, &ncyc_lo);

    /* Do double precision subtraction */
    lo = ncyc_lo - cyc_lo;
    borrow = lo > ncyc_lo;
    hi = ncyc_hi - cyc_hi - borrow;
    result = (double) hi * (1 << 30) * 4 + lo;
    if (result < 0) {
        fprintf(stderr, "Error: counter returns neg value: %.0f\n", result);
    }
    return result;
}
```

code/perf/clock.c

Figure 9.9: **Code implementing program interface to IA32 cycle counter.** Assembly code is required to make use of the counter reading instruction.

code/perf/clock.c

```
/* Estimate the clock rate by measuring the cycles that elapse */
/* while sleeping for sleeptime seconds */
double mhz(int verbose, int sleeptime)
{
    double rate;

    start_counter();
    sleep(sleeptime);
    rate = get_counter() / (1e6*sleeptime);
    if (verbose)
        printf("Processor clock rate ~= %.1f MHz\n", rate);
    return rate;
}
```

code/perf/clock.c

Figure 9.10: **Function `mhz`: Determines the clock rate of a processor.**

9.4 Measuring Program Execution Time with Cycle Counters

Cycle counters provide a very precise tool for measuring the time that elapses between two different points in the execution of a program. Typically, however, we are interested in measuring the time required to execute some particular piece of code. Our cycle counter routines compute the total number of cycles between a call to `start_counter` and a call to `get_counter`. They do not keep track of which process uses those cycles or whether the processor is operating in kernel or user mode. We must be careful when using such a measuring device to determine execution time. We investigate some of these difficulties and how they can be overcome.

As an example of code that uses the cycle counter, the routine in Figure 9.10 provides a way to determine the clock rate of a processor. Testing this function on several systems with parameter `sleeptime` equal to 1 shows that it reports a clock rate within 1.0% of the rated performance for the processor. This example clearly shows that our routines measure elapsed time rather than the time used by a particular process. When our program calls `sleep`, the operating system will not resume the process until the sleep time of one second has expired. The cycles that elapse during that time are spent executing other processes.

9.4.1 The Effects of Context Switching

A naive way of measuring the run time of some procedure `P` is simply to use the cycle counter to time one execution of `P`, as in the following code:

```
double time_P()
{
    start_counter();
    P();
    return get_counter();
```

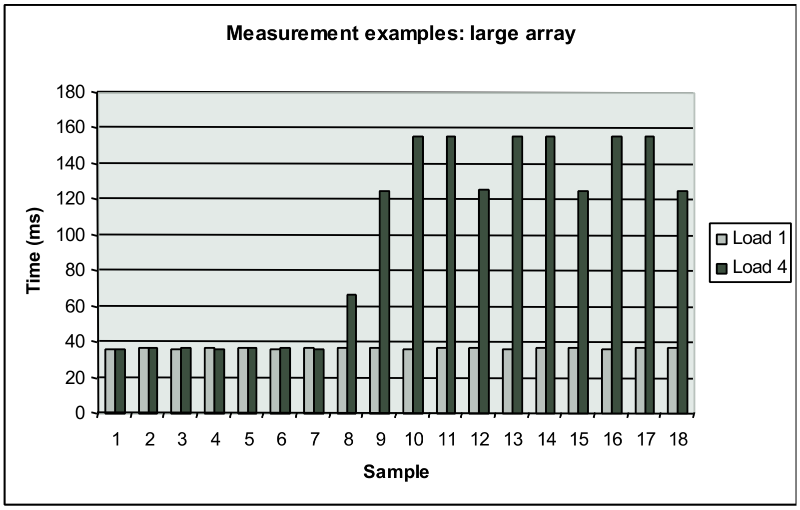


Figure 9.11: **Measurements of long duration procedure under different loading conditions.** On a lightly loaded system, the results are consistent across samples, but on a heavily loaded system, many of the measurements overestimate the true execution time.

```
}
```

This could easily yield misleading results if some other process also executes between the two calls to the counter routines. This is especially problematic if either the machine is heavily loaded, or if the run time for P is especially long. This phenomenon is illustrated in Figure 9.11. This figure shows the result of repeatedly measuring a program that computes the sum of an array of 131,072 integers. The times have been converted into milliseconds. Note that the run times are all over 36 ms, greater than the timer interval. Two trials were run, each measuring 18 executions of the exact same procedure. The series labeled “Load 1” indicates the run times on a lightly loaded machine, where this is the only process actively running. All of the measurements are within 3.4% of the minimum run time. The series labeled “Load 4” indicates the run times when three other processes making heavy use of the CPU and memory system are also running. The first seven of these samples have times within 2% of the fastest Load 1 sample, but others range as much as 4.3 times greater.

As this example illustrates, context switching causes extreme variations in execution time. If a process is swapped out for an entire time interval, it will fall behind by millions of instructions. Clearly, any scheme we devise to measure program execution times must avoid such large errors.

9.4.2 Caching and Other Effects

The effects of caching and branch prediction create smaller timing variations than does context switching. As an example, Figure 9.12 shows a series of measurements similar to those in Figure 9.11, except that the array is 4 times smaller, yielding execution times of around 8 ms. These execution times are shorter than the timer interval and therefore the executions are less likely to be affected by context switching. We see significant variations among the measurements—the slowest is 1.1 times slower than the fastest, but none of these variations are as extreme as would be caused by context switching.

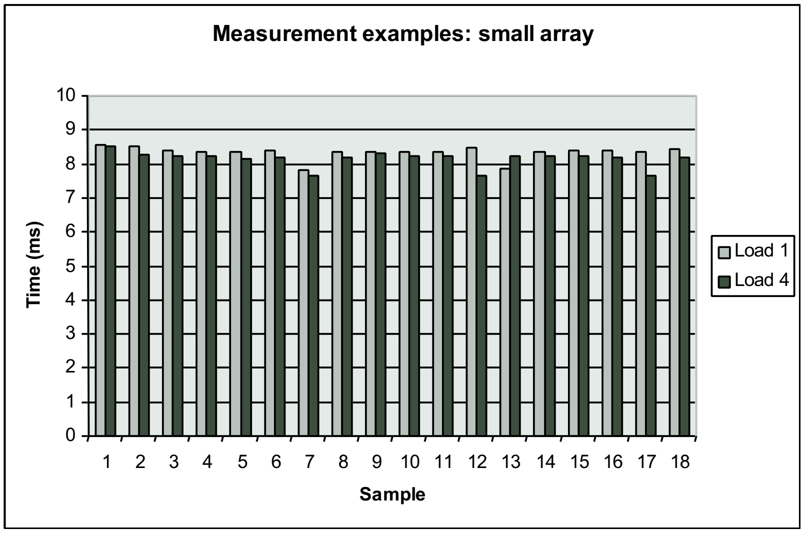


Figure 9.12: **measurements of short duration procedure under different loading conditions.** The variations are not as extreme as they were in Figure 9.11, but they are still unacceptably large.

Measurement	Call	Cycles
1	`procA(b1)`	399
2	`procA(b2)`	132
3	`procA(b3)`	134
4	`procA(b1)`	100
5	`procB(b1)`	317
6	`procB(b2)`	100

Figure 9.13: **Measurement sequence with identical procedures operating on identical data sets.** The variations in these measurements are due to different miss conditions in the instruction and data caches.

The variations shown in Figure 9.12 are due mainly to cache effects. The time to execute a block of code can depend greatly on whether or not the data and the instructions used by this code are present in the data and instruction caches at the beginning of execution.

As an example, we wrote two identical procedures, `procA` and `procB`, that are given a pointer of type `double *` and set the eight consecutive elements starting at this pointer to 0.0. We measured the number of clock cycles for various calls to these procedures with three different pointers: `b1`, `b2`, and `b3`. The call sequence and the resulting measurements are shown in Figure 9.13. The timings vary by almost a factor of 4, even though the calls perform identical computations. Because there were no conditional branches in this code, we conclude that the variations must be due to cache effects.

Practice Problem 9.6:

Let c be the number of cycles that would be required by a call to `procA` or `procB` if there were no cache misses. For each computation, the cycles wasted due to cache misses can be apportioned between the different items needing to be brought into the cache:

- The instructions implementing the measurement code (e.g., `start_counter`, `get_counter`, and so on). Let the number of cycles for this be m.
- The instructions implementing the procedure being measured (`procA` or `procB`). Let the number of cycles for this be p.
- The data locations being updated (designated by `b1`, `b2`, or `b3`). Let the number of cycles for this be d.

Based on the measurements shown in Figure 9.13, give estimates of the values of c, m, p, and d.

Given the variations shown in these measurements, a natural question to ask is "Which one is right?" Unfortunately, there is no simple answer to this question. It depends on both the conditions under which our code will actually be used, as well as the conditions under which we can obtain reliable measurements. One problem is that the measurements are not even consistent from one run to the next. The measurement table shown in Figure 9.13 show the data for just one testing run. In repeated tests, we have seen Measurement 1 range from 317 to 606, and Measurement 5 range from 301 to 326. On the other hand, the other four measurements vary only by at most a few cycles from one run to another.

Clearly Measurement 1 is an overestimate, because it includes the cost of loading the measurement code and data structures into cache. Furthermore, it is the most subject to wide variations. Measurement 5 includes the cost of loading `procB` into the cache. This is also subject to significant variations. In most real applications, the same code is executed repeatedly. As a result, the time to load the code into the instruction cache will be relatively insignificant. Our example measurements are somewhat artificial in that the effects of instruction cache misses were proportionally greater than what would occur in a real application.

To measure the time required by a procedure `P` where the effects of instruction cache misses are minimized, we can execute the following code:

```
double time_P_warm()
{
    P(); /* Warm up the cache */
    start_counter();
    P();
    return get_counter();
}
```

Executing `P` once before starting the measurement will have the effect of bringing the code used by `P` into the instruction cache.

This code also minimizes the effects of data cache misses, since the first execution of `P` will also have the effect of bringing the data accessed by `P` into the data cache. For procedures `procA` or `procB`, a measurement by `time_P_warm` would yield 100 cycles. This would be the right conditions to measure if we expect our code to access the same data repeatedly. For some applications, however, we would be more likely to access new data with each execution. For example, a procedure that copies data from one region of memory to another would most likely be called under conditions where neither block is cached. Procedure `time_P_warm` would tend to underestimate the execution time for such a procedure. For `procA` or `procB`, it would yield 100, rather than the 132 to 134 measured when the procedure is applied to uncached data.

To force the timing code to measure the performance of a procedure where none of the data is initially cached, we can flush the cache of any useful data before performing the actual measurement. The following procedure does this for a system with caches of no more than 512KB:

code/perf/time_p.c

```
/* Number of bytes in the largest cache to be cleared */
#define CBYTES (1<<19)
#define CINTS (CBYTES/sizeof(int))

/* A large array to bring into cache */
static int dummy[CINTS];
volatile int sink;

/* Evict the existing blocks from the data caches */
void clear_cache()
{
    int i;
    int sum = 0;

    for (i = 0; i < CINTS; i++)
        dummy[i] = 3;
    for (i = 0; i < CINTS; i++)
        sum += dummy[i];
    sink = sum;
}
```

code/perf/time_p.c

This procedure simply performs a computation over a very large array `dummy`, effectively evicting everything else from the cache. The code has several peculiar features to avoid common pitfalls. It both stores values into `dummy` and reads them back so that it will be cached regardless of the cache allocation policy. It performs a computation using array values and stores the result to a global integer (the declaration `volatile` indicates that any update to this variable must be performed), so that a clever optimizing compiler will not optimize away this part of the code.

With this procedure, we can get a measurement of `P` under conditions in which its instructions are cached, but its data are not, by using the following procedure:

```
double time_P_cold()
{
    P(); /* Warm up instruction cache */
    clear_cache(); /* Clear data cache */
    start_counter();
    P();
    return get_counter();
}
```

Of course, even this method has deficiencies. On a machine with a unified L2 cache, procedure `clear_cache` will cause all instructions from `P` to be evicted. Fortunately, the instructions in the L1 instruction cache will remain. Procedure `clear_cache` also evicts much of the run-time stack from the cache, leading to an overestimate of the time required by `P` under more realistic conditions.

As this discussion shows, the effects of caching pose particular difficulties for performance measurement. Programmers have little control over what instructions and data get loaded into the caches and what gets evicted when new values must be loaded. At best, we can set up measurement conditions that somewhat match the anticipated conditions of our application by some combination of cache flushing and loading.

As mentioned earlier, the branch prediction logic also influences program performance, since the time penalty caused by branch instruction is much less when the branch direction and target are correctly predicted. This logic makes its predictions based on the past history of branch instructions that have been executed. When the system switches from one process to another, it initially makes predictions about branches in the new process based on those executed in the previous process. In practice, however, these effects create only minor performance variations from one execution of a program to another. The predictions depend most strongly on recent branches, and hence the influence by one process on another is very small.

9.4.3 The K-Best Measurement Scheme

Although our measurements using cycle timers are vulnerable to errors due to context switching, cache operation, and branch prediction, one important feature is that the errors will always cause overestimates of the true execution time. Nothing done by the processor can artificially speed up a program. We can exploit this property to get reliable measurements of execution times even when there are variances due to context switching and other effects.

Suppose we repeatedly execute a procedure and measure the number of cycles using either `time_P_warm` or `time_P_cold`. We record the K (e.g., three) fastest times. If we find these measurements agree within some small tolerance ϵ (e.g., 0.1%), then it seems reasonable that the fastest of these represents the true execution time of the procedure. As an example, suppose for the runs shown in Figure 9.11 we set the tolerance to 1.0%. Then the fastest six measurements for Load 1 are within this tolerance, as are the fastest three for Load 4. We would therefore conclude that the run times are 35.98 ms and 35.89 ms, respectively. For the Load 4 case, we also see measurements clustered around 125.3 ms, and six around 155.8 ms, but we can safely discard these as overestimates.

We call this approach to measurement the "K-Best Scheme." It requires setting three parameters:

K: The number of measurements we require to be within some close range of the fastest.

ϵ: How close the measurements must be. That is, if the measurements in ascending order are labeled $v_1, v_2, \ldots, v_i, \ldots$, then we require $(1+\epsilon)v_1 \geq v_K$.

M: The maximum number of measurements before we give up.

Our implementation performs a series of trials and maintains an array of the K fastest times in sorted order. With each new measurement, it checks whether it is faster than the current one in array position K. If so, it replaces array element K and then performs a series of interchanges between adjacent array positions to move this value to the appropriate position in the array. This process continues until either the error criterion is satisfied, in which case we indicate that the measurements have "converged," or we exceed the limit M, in which case we indicate that the measurements failed to converge.