CS/ECE 347, Spring 1998
Homework 1: Measuring Performance

Assigned: Thursday, January 15
Due: Thursday, January 22, 3:00PM

Introduction

Measuring performance is fundamental to the study of computer systems. When comparing machines, or when optimizing code, it is often useful to measure the amount of time that it takes (preferably at the resolution of processor clock cycles) to execute a particular operation or procedure. Some machines have special facilities to assist in measuring performance. Even without such facilities, almost all machines provide *interval timers*—a relatively crude method of computing elapsed times. In this assignment, you will investigate how to reason about and control the accuracy of timing information that can be gathered using interval timers. One of the goals is to develop a *function timer* which accurately measures the execution time of any function on any machine.

Policy

You may work in a group of up to 3 people in solving the problems for this assignment. Turn in a single writeup per group, indicating all group members.

Logistics

Any clarifications and revisions to the assignment will be posted on Web page `assigns.html` in the class WWW directory.

In the following, *HOMEDIR* refers to the directory:

```
/afs/cs.cmu.edu/academic/class/15347-s98/public
```

and *ASSTDIR* refers to the subdirectory *HOMEDIR*`/asst/asst1`.

You should hand in a hard copy document for this assignment, with a cover sheet identifying the names and the Andrew ID's of your group members. Formatted text is preferred over hand-written text. Start each problem on a separate sheet of paper.

Using Interval Timers

The overall operation of an interval timer is illustrated in Figure 1. The system maintains a (user-settable) counter value which is updated periodically. That is, once every Δ time units, the counter is incremented by Δ. Using the Unix library routine `getitimer`, the user can poll the value of this counter. Thus, to measure the elapsed time of some operation `Op`, the user can poll the counter to get a starting value T_s, perform the operation, and poll the counter to get a final value T_f. The elapsed time for the operation can be *approximated* as $T_{observed} = T_f - T_s$. As the figure illustrates, however, the actual elapsed time T_{actual} may differ from $T_{observed}$ significantly, due to the coarseness of the timer resolution. Since the value of Δ is around 10 *milliseconds* for most systems, this error can be very significant.

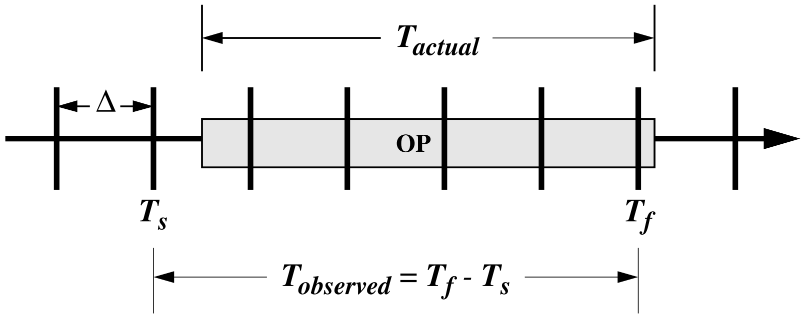

Figure 1: Time Measurement with an Interval Timer

We have encapsulated the Unix interval timer routines for you in a handy timer package called *ASSTDIR*/`etime.c`. You should use this package for all measurements in the assignment. See *ASSTDIR*/`example.c` for a simple example of how to use the package. One notable feature is that it converts the measurements to units of seconds, expressed as a C `double`. The procedure for timing operation `Op` is then:

```
init_etime();
Ts = get_etime();
Op;
Tf = get_etime()
T_observed = Tf - Ts;
```

Problem 1:

Consider a processor with a 500 MHz clock rate where precisely one addition operation can be performed every clock cycle, and where the value of Δ for the interval timer is 10 milliseconds. You would like to time a section of code (`Op`) consisting purely of a sequence of back-to-back additions.

What is the minimum number of additions within `Op` such that the relative measurement error (using the technique shown above) of $T_{observed}$ with respect to T_{actual} is guaranteed to be less than 3%? As always, show all of your work.

Problem 2:

Write a C procedure that uses measurements to estimate (as accurately as possible) the value of Δ on any UNIX machine. Provide a listing of your code along with a brief description of your scheme.

We can improve the accuracy of the measurements by making sure that the activity we measure has sufficient duration to overcome the imprecision of interval timers. That is, we can accurately measure the time required by `Op` by executing it n times for a sufficiently large value of n:

```
init_etime();
Ts = get_etime();
for (i=0; i<n; i++) {
    Op;
}
```

```
Tf = get_etime()
T_aggregate = Tf - Ts;
T_average = T_aggregate/n;
```

How do we choose a large enough value of n? The idea is that n must be large enough such that $T_{aggregate}$ is larger than the minimum value ($T_{threshold}$) which guarantees a relative measurement error less than the desired upper bound of E. The value of $T_{threshold}$ can be computed based on Δ and E. However, since the elapsed time for `Op` is unknown, we cannot compute the minimum value of n ahead of time.

One approach is to start with $n = 1$, and continue doubling it until the observed $T_{aggregate}$ is large enough to guarantee sufficient accuracy (i.e. it is larger than $T_{threshold}$).

Problem 3:

Implement a function timer in C that uses the doubling scheme outlined above to accurately measure the running time of any function on any system. Your function timer should have the following interface

```
typdef void (*test_funct)(void);
double ftime(test_funct P, double E);
```

where P is the function to be timed and E is the maximum relative measurement error. These prototypes are already defined for you in *ASSTDIR*/`ftime.h`. Implement your `ftime()` function in a separate file called `ftime.c`.

Your function timer should: (1) determine the timer period Δ using the scheme from the previous problem; (2) calculate $T_{threshold}$ as a function of Δ and E; and then (3) repeatedly double n until $T_{aggregate} \geq T_{threshold}$. It should work for any function on any system, regardless of the running time of the function or the timer period of the system.

Problem 4:

Test your function timer using the program *ASSTDIR*/`freq.c`, which uses `ftime()` to estimate the clock frequency of your machine. This routine assumes that your machine executes an integer addition in one clock cycle. This is a safe assumption for most modern processors.

Turn in the output string from freq.c and the type of system you ran it on (e.g., Sparc 5).

Problem 5:

Recall that in Problem 3, you repeatedly *doubled* the value of n until it was sufficiently large (i.e. until $T_{aggregate} \geq T_{threshold}$). Now consider the following three algorithms for increasing the value of n:

Algorithm 1: Set $n = 1$ initially, and repeatedly multiply n by a factor of **2** until n is sufficiently large (i.e. the algorithm used in Problem 3).

Algorithm 2: Set $n = 1$ initially, and repeatedly multiply n by a factor of **10** until n is sufficiently large.

Algorithm 3: Set $n = 1$ initially, and repeatedly **add** (*not multiply!*) **100** to n until n is sufficiently large.

Your goal is to minimize the total amount of time that your timing routine takes to accurately time a function. In this problem, you will evaluate the three algorithms described above based on this criteria.

Part 1: Assuming that $T_{threshold} = 100$ milliseconds, and assuming that the timing loop surrounding `Op` involves zero overhead, compute how long it would take for each of the three algorithms for increasing n to accurately time `Op` for each of the following three cases: (i) $T_{actual} = 12.0$ milliseconds, (ii) $T_{actual} = 99.0$ microseconds, and (iii) $T_{actual} = 110.0$ microseconds.

Part 2: Based on a quantitative analysis of their *worst-case* behaviors, evaluate which of the three algorithms for increasing n is most desirable for measuring functions where T_{actual} is an arbitrary value no greater than a microsecond and where Δ for the interval timer is at least 10 milliseconds.

Problem 6:

The total amount of time spent executing a procedure includes not only the time spent executing the *body* of the procedure, but also a certain amount of bookkeeping overhead associated with procedure entry and exit. (We will discuss the details of exactly what takes place at procedure entry and exit later on in the course, but these details are not relevant to solving this problem.) When you measure the elapsed time of a procedure using your function timer from Problem 3, your measurement includes this procedure entry and exit overhead.

In an effort to isolate the amount of time spent only within the procedure *body* itself, you have devised the following scheme. You have created an empty procedure called "`dummy`" which contains no body, but which has the same entry and exit overhead as the procedure you would like to measure (which is called "`foo`"). Therefore to estimate the amount of time spent in the body of `foo`, you will subtract the execution time of `dummy` from the execution time of `foo`.

Please answer the following question. Assuming that the upper bound on the relative measurement error (E) for the total execution time of both `dummy` and `foo` is known to be less than 10%, what can you say about the upper bound on the relative measurement error for your estimate (using the scheme described above) of the time spent in the *body* of `foo`? Explain your answer.
