CS 740, Fall 1998
Homework Assignment 1

Assigned: Tuesday, September 15
Due: Thursday, September 24, 1:00PM

The purpose of this assignment is to develop techniques for measuring code performance, to practice reasoning about low-level code optimization, and to better understand Alpha procedure calling conventions.

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/15740-f98/public`

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

You may hand in your assignment either as a hard copy or else by emailing it to Prof. Mowry at `tcm@cs`. If you hand it in as a hard copy, formatted text is preferred over hand-written text.

Using Interval Timers

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.

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.

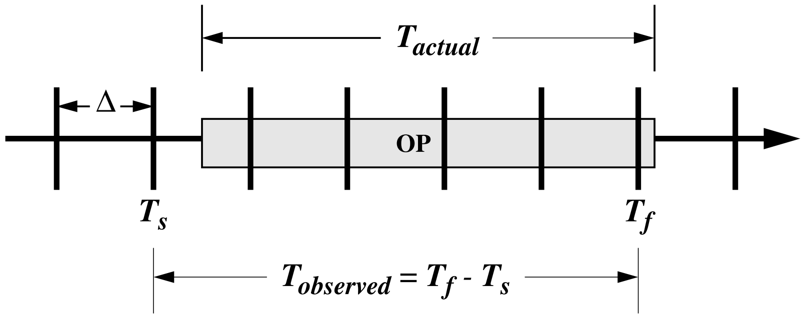

Figure 1: Time Measurement with an Interval Timer

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: Bounded Measurement Error

Consider a processor with a 600 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 2%? As always, show all of your work.

Problem 2: Measuring Δ for Your Timer

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: Implementing a Function Timer

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: Testing Your Function Timer

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: Alternative Timer Algorithms

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.

Optimizing the `memcpy()` Routine

The purpose of these next problems is to get hands-on experience with machine-level programming. Our interest is in being able to understand, measure, and optimize the machine code generated by a compiler. This is a far more useful skill than being able to churn out pages of assembly code by hand. Parts of this assignment involve compiling, disassembling, and running code on one of our Alpha-based machines. For information on how to access these machines, please refer to the link labeled "Information about our Alpha systems" on the class WWW page.

In the next several problems, we will be focusing on the performance of the `memcpy()` routine, which is part of the C library. The following paraphrased excerpts from the `memcpy()` man page describe its interface and behavior:

```
void *memcpy(void *s1, const void *s2, size_t n);
```

- The `memcpy()` function operates on strings in memory areas. A memory area is a group of contiguous characters bound by a count and not terminated by a null character. This function does not check for overflow of the receiving memory area. This function is declared in the `string.h` header file.
- The `memcpy()` function copies `n` bytes from the string pointed to by the `s2` parameter into the location pointed to by the `s1` parameter. When copying overlapping strings, the behavior of this function is unreliable.
- The `memcpy()` function returns the string pointed to by the `s1` parameter. No return value is reserved to indicate an error.

The file *ASSTDIR*/`memcpy_naive.c` contains a straightforward (but naive, from a performance perspective) implementation of `memcpy()` in C called "`my_memcpy()`". The file *ASSTDIR*/`memcpy_naive.s` contains the Alpha assembly code generated using the command: `gcc -O -S memcpy_naive.c`

The file *ASSTDIR*/`memcpy.dis` contains a disassembled version of the `memcpy()` routine taken from the Unix library `/usr/lib/libc.a` on one of our Alpha machines.

Problem 6: Understanding the `memcpy()` Assembly Code

Generate an "annotated" version of both *ASSTDIR*/`memcpy_naive.s` and *ASSTDIR*/`memcpy.dis` using the following conventions:

- Put comments at the top of a code segment describing register usage and initial conditions.

- Put comments along the right hand side describing what each instruction does.

NOTE: Comments of the form:

```
# I won't tell you anything about the registers.
s8addq  r1, r2, r2   # r2 = 8*r1 + r2
ldq     r3, 0(r2)    # r3 = Mem[r2]
```

are useless and will receive little (if any) credit. Instead, we would like to see comments like the following:

```
# Throughout the loop: r1 holds i, r7 holds n
# At the beginning of the loop: r2 = &v[0]
s8addq  r1, r2, r2   # r2 = 8*i + &v[0]
ldq     r3, 0(r2)    # r3 = v[i]
```

In other words, your comments should convey semantic information from the source code, and not simply reiterate what would be obvious to anyone who can read Alpha assembly code.

Problem 7: Measuring the Performance of the `memcpy()` Routines

Use your interval timer code to measure the performance of both the `my_memcpy()` routine in *ASSTDIR*`/memcpy_naive.c` and DEC C library implementation of `memcpy()` on the various `memcpy()` calls contained in *ASSTDIR*`/memcpy_test.c`. Note that you should produce separate timing numbers for each these individual calls to `memcpy()`, and be sure to call the initialization routine in this file before you start timing things to ensure that the cache is warm.

Discuss the relative performance differences between the two versions of the routine, and whether they make sense given your analysis of the assembly code.

Problem 8: Implementing a Better Version of `memcpy()` in C

Write your own version of `memcpy()` in C. Your code must behave correctly, but at the same time it should be as efficient as possible. You should create a version of your code which only uses C constructs (i.e. no explicit assembly code). In addition, you may optionally create a second version of your code which uses the GCC assembly code directives (i.e. "`ASM`") if it further enhances performance. For further information on how to use assembly code directives in gcc, see the "info" pages on gcc (under "C extensions"). These info pages are reproduced on the *Assignment and exam information* class web page under Assignment 1. Use a minimal number of `ASM` statements—do not simply reproduce large amounts of hand-coded assembly in your C code. Be sure to compile your code using the "`-O`" optimization flag.

Measure the performance of your C-only code and your assembly-augmented code (if applicable). If your assembly-augmented code achieves better performance than your C-only code, discuss why you are not able to achieve comparable performance using only normal C constructs. Also, compare your code with both the naive and UNIX library versions of `memcpy()`. If your performance falls short of the UNIX library version, explain why.

Problem 9: Measuring `memcpy()` on a Different Architecture

Using your interval timer code, measure and compare the performance of both your C-only version of `memcpy()` and the native (i.e. UNIX C library) version of `memcpy()` on a machine other than an Alpha machine. Discuss whether these results are what you expected, or whether they are surprising.

Problem 10: Stack Frames and Procedure Calls

The file *ASSTDIR*/`structure.c` shows the C code for a function that demonstrates many interesting features of implementing C on an Alpha machine. The code generated by GCC with the `-O` flag is shown in the file *ASSTDIR*/`structure.s`

Document the following:

A. Show the layout of the data structure `list_ele`.

B. Explain how the program implements the returning of a structure by `sum_list`.

C. Describe the register allocation used in `sum_list`.

D. Show the layout of the stack frame used by `sum_list`.

E. Create an annotated version of the assembly code for `sum_list` using our usual formatting conventions.

Long Jumps

(NOTE: this last section is for extra credit only.)

In this section, you will enhance your understanding of Alpha calling and returning conventions by examining the Unix `setjmp` library, which in effect cheats on the standard procedure calling conventions.

`setjmp` provides a mechanism for nonlocal procedure exits. By using `setjmp` and `longjmp`, a procedure can exit to a function a long way up the procedure chain, bypassing its calling procedure.

This is typically used for error handling, in the following style:

```
jmp_buf env;

main()
{
   int jval;
   if ((jval = setjmp(env)) != 0) {
     /* Do error recovery and cleanup. */
     /* And possibly try to execute again. */
   } else {
     /* Do normal case. */
     ...
   }
}
```

The first time `setjmp()` is called, it sets up the buffer env with enough information to reconstruct the state of the registers and the stack. It returns 0 to the calling function.

The calling function then does other code, and deep within some complicated hierarchy of other functions, it may find some error:

```
deep_within_some_complicated_hierarchy ()
{
   if (error_detected) {
      longjmp(env, ERROR_CONSTANT);
   }
   ...
}
```

`longjmp()` restores the state that was saved in `setjmp()` and resets the stack pointer to the position it was in when `setjmp()` was called.

The file `ljdemo.c` contains a somewhat contrived example of the use of `setjmp()` and `longjmp()`. Study this code to better understand the behavior of these constructs. Try compiling and running it (it should work on any Unix machine).

Problem 11: ***(For Extra Credit)*** **Understanding the Implementation of Long Jumps**

The file *ASSTDIR*`/longjmp.dis` contains a disassembly of the code for `___setjmp()` and `_longjmp()`, which are simplified versions of `setjmp()` and `longjmp()`. In addition, the file also includes `__longjump_resume()` which is called by `_longjmp()` This code was extracted from the Unix library `/usr/lib/libc.a` on one of our Alpha machines. Your job is to create annotated versions of these three procedures. Show clearly what state is being saved and restored, and how longjmp subverts the Alpha calling routines to appear to 'return' to a place other than that from which it was called.

It may help you understand the code to write down what data is stored in which places in the `jmp_buf`. If you write this information down, turn that in too.

Hints:

- The `ldt` and `stt` instructions load and store 64-bit numbers into/from the *floating-point* register file. Their behavior is analogous to `ldq` and `stq` for integer data.
- The value `0xacedbade` is used as a "magic token" to identify a buffer that has (probably) been set by `setjmp()` (A moment's thought should convince you that doing a `longjmp()` to a buffer that has not been set up by `setjmp()` is likely to result in errors that are very difficult to track down.) This value is generated by a combination of three instructions:

  ```
  ldah r1, 22135(r31)
  lda  r1, -8849(r1)
  addq r1, r1, r1
  ```

- The `ldah` instruction, which stands for Load Address High, multiplies a signed 16-bit constant by 65536, and then adds this value into the destination register. The `lda` instruction, which stands for Load Address, generates the effective address address in the destination register (in this case, it subtracts 8849 from `r1`).