5DV022 Software Exercise 1 Fall 2008

Solutions to this assignment are due on September 22, 2008 at 5pm (1700). This is a *laboratory exercise*, and the amount of collaboration allowed is as defined in the documents *Riktlinjer vid labgenomförande* (*Rules for the Preparation of Laboratory Exercises*) and *Hederskodex* (the *Honour Code*), which may be found on the course home page.

Late submissions are subject to penalty as described in the course syllabus.

This exercise may be carried out in groups of at most three individuals.

The report **must** be written in English. Reports written in any other language will be returned, ungraded, unless prior written arrangement has been made with the course instructor.

Place your report in the instructor's mailbox or give them to the instructor during class. Do not place it in the mailbox for laboratory reports.

If you completed this exercise for the course offering in a previous year, and are resubmitting the same solution, you must abide by the restrictions which are identified in the course syllabus for such submissions.

1. General Description

The purpose of this assignment is to develop a basic clock-based fixed-position logging profiling package for C programs which run under the UNIX operating system. The package which you develop in this assignment will be used in the solution of Software Exercises 2 and 3, so it is worth your while to develop useful, friendly software.

2. The Components

Profiling using the package to be developed consists of three steps. In the first step, the original program is *augmented* with instructions which generate timing data when the program is run. In the second step, the augmented program is run to generate the timing data. In the third step, an analysis program is run, with the timing data as input, to generate a performance report for the algorithm.

2.1 Augmentation

Development of software which performs automatic augmentation of a program is a formidable task; at the very least it must include a parser with a substantial knowledge of the syntax of the host

language. Therefore, in this exercise, augmentation will be performed manually. However, some basic support will be provided by the package to be developed.

Figure 3 shows a simple (nonsense) C program to be profiled, while Figure 4 shows the same program with augmentation statements included. The heart of the augmentation package is an include file which you are to write, to be named `profile.h`. This file provides support for all of the other statements which have been added to the program in Figure 4. In evaluating the following particulars, remember that it is essential that the profiling code consume as little run time as possible.

1. The profiling process generates a timing log which consists of records (structures in C) containing three fields. A typical declaration for such a record is as follows.

```
struct  profile_log_pt
{
  unsigned char proc_index;
  unsigned char direction;
  struct timeval clock_value;
};
```

   The first entry is the *procedure index*. Each procedure to be profiled is assigned an index between 0 and 255, which is recorded as a `char` in C. This encoding is done strictly to enhance efficiency, since recording such a number in a timing record is more efficient than recording the entire name of a procedure. In this example, procedure `main` has been assigned the number `0`, procedure `A` the number `1`, and procedure `B` the number `2`. The second entry is the *direction*. Again, integer encoding is used, with `0` indicating the beginning of a procedure, and `1` the end. The final entry is the *clock value*, which is the value of the profiling clock at that instant.

2. The `write_log` statements are used to write the timing log. The first argument is `0` for the beginning of the procedure, and `1` for the end of the procedure. The second argument is the number of the procedure. Thus, `write_log(1,2)` means write the timing log with a record indicating that procedure `B` has terminated execution, together with the value of the profiling clock.

3. To keep profiling overhead to a minimum, the timing log entries are written to an array (of records) during execution of the program. Upon completion, this array is written to a file. The parameter `log_size` specifies how large this array is to be. The command `write_log_to_file`, which is generally executed at the end of the program, forces the array to be written to a file.

4. The command `start_log`, which is executed once at the beginning of the profiling session, performs whatever initialization is necessary.

5. For efficiency considerations, the commands `start_log`, `write_log`, and `write_log_to_file` must be realized as macros (using `#define`), and not as functions. Although such code will not be pretty, it will execute with much less overhead than will function calls.

6. The name of the file to which the timing data is written may be fixed to be `timing.dat`. Alternately, if you wish, you may place a `#define` command in the augmented program which identifies the file.

7. For the package to function properly, it is essential that the proper clock be accessed, and that the data which it provides be interpreted properly. The appropriate timer is the `ITIMER_PROF` selection of the `getitimer` system call. The latter has a UNIX man page, which you should read carefully. You will also need to look at some system include files (start with `/usr/include/sys/time.h`) to determine the format of a `timeval` structure. Note further that this clock counts *downwards* towards zero. Make sure that you initialize it properly.

2.2 The Analyzer

The rôle of the analyzer program is to process the raw data generated by the augmented program. For the program of Figure 3, a report of the form shown in Figure 1 should be generated.

```
-------------------------------------------------------------------------
This is a sample run with all three procedures profiled.

                      Calls                 Execution Time (ms.)
Procedure           --------                --------------------
  Name            Total    Pct.        Total     Avg./Call      Pct.
_________       _______  _______   _________   _________   _________
Main                1    0.11%        0.000       0.000       0.00%
A                 303   33.52%     1420.000       4.686      34.47%
B1                600   66.37%     2700.000       4.500      65.53%

Total running time in milliseconds:  4120.000
-------------------------------------------------------------------------


-------------------------------------------------------------------------
This is a sample run with two procedures profiled.

                      Calls                 Execution Time (ms.)
Procedure           --------                --------------------
  Name            Total    Pct.        Total     Avg./Call      Pct.
_________       _______  _______   _________   _________   _________
Main                1    0.17%        0.000       0.000       0.00%
A                   -        -            -           -           -
B1                600   99.83%     4120.000       6.867     100.00%

Total running time in milliseconds:  4120.000
-------------------------------------------------------------------------
```

Figure 1: Sample report of the analyze program.

The details of what is to be reported are communicated to the analyze program via the contents of another file, which is named `analyze.indat`. For this example, the contents of this file are shown in Figure 2.

The contents of `analyze.indat` is simple ASCII text, and consists of a sequence of logical records, with each logical record consisting of a sequence of lines. Each logical record corresponds to one report of the analyzer. The record separator is a blank line. The first line of each record contains the name of the file which contains the timing data. The second line contains the heading which

```
timing.dat
This is a sample run with all three procedures profiled.
0 Main 1
1 A 1
2 B 1

timing.dat
This is a sample run with two procedures profiled.
0 Main 1
1 A 0
2 B 1
```

Figure 2: Contents of the file `analyze.indat` for the example.

is to be used for the report. The rest of the lines identify the procedures whose performance is to be summarized in the report. The first word on each line is the procedure number (which must be the same as the number used in by the augmented program). The second word is the name of the procedure. The third word is a `1` if the performance of the procedure is to be summarized, and a `0` if it is not. In generating the summary report, note the following:

1. In the summary, an instant of time may be assigned to only one procedure. Thus, if `main` calls `A`, which calls `B`, only `B` gets credited with the time during which it executes. The total execution time in the report is thus the total running time of the program. (Note that the utility `gprof` does not follow this convention.)
2. If a procedure is excluded in a report (as is `A` in the second report of Figure 1), then its running time is credited to its caller. (If the top-level procedure is excluded, this will result in an incorrect total running time, but it is seldom the case that the top-level is excluded.)
3. The analyzer program may assume that the data in `analyze.indat` and in `timing.dat` are properly formatted. It need not perform any error checking.
4. For subsequent exercises, the analyzer program may be required to write timing data to a file. Keep this in mind, although it should not be a significant design issue.

3. A Further Experiment

For proper profiling, it is essential to be aware of the granularity of the clock which is used. As part of this exercise, you are also required to devise and implement a test which will determine the granularity of the `ITIMER_PROF` clock. (The granularity of this clock is on the order of milliseconds, not microseconds.)

4. What to Submit

At the level of this course, all students should be able to develop the associated software without undue difficulty. The grading of this exercise will therefore be based greatly upon the quality of the presentation, which must include at least the following.

1. A user manual for your package. This manual must be comprehensive enough to allow someone to use your package without undue difficulty, and without having to study your source code.
2. Well-documented source code for `profile.h` and the analyzer program, `analyze.c`.
3. The results of test runs on the program of Figure 3, using the `analyze.indat` file of Figure 2.
4. A run of the program of Figure 3 using the `gprof` package, together with a write-up discussing and explaining the differences between the results produced by your profiling program and those of `gprof`.
5. A description of your experiment to determine the granularity of the system clock, together with the results.
6. For all experimental runs, a clear statement indicating the architecture (i386 or Sparc) and operating system (Linux or Solaris) of the machine which was used. For consistency, the final runs of all experiments should be on the same type of machine.

It goes without saying that the report for this exercise must be prepared using a document-processing system (such as LaTeX or Openoffice.org). Be sure to check the spelling with a spellcheck program, such as *Ispell*.

```
#include <stdio.h>

int n;
int mloops = 2;
int aloops = 500000;
int bloops = 500000;

int main ()
{
  int i;
  for (i=0; i<=mloops; i++)
    {
      n = 0;
      A();
    }
  return 0;
};

int A ()
{
  int i;
  n++;
  for (i=0; i<=aloops; i++);
  if (n<=200)
    B();
  return 0;
};

int B ()
{
  int i;
  for (i=0; i<=bloops; i++);
  if ((n % 2) == 0)
    A();
  else
    {
      n++;
      B();
    }
}
```

Figure 3: The test program to be profiled.

```
#include <stdio.h>
#define log_size 5000
#include "profile.h"

int n;
int mloops = 2;
int aloops = 500000;
int bloops = 500000;

int main ()
{
  int i;
  start_log;
  write_log(0,0);
  for (i=0; i<=mloops; i++)
    {
      n = 0;
      A();
    }
  write_log(1,0);
  write_log_to_file();
  return 0;
};

int A ()
{
  int i;
  write_log(0,1);
  n++;
  for (i=0; i<=aloops; i++);
  if (n<=200)
    B();
  write_log(1,1);
  return 0;
};

int B ()
{
  int i;
  write_log(0,2);
  for (i=0; i<=bloops; i++);
  if ((n % 2) == 0)
    A();
  else
    {
      n++;
      B();
    }
  write_log(1,2);
  return 0;
}
```

Figure 4: The test program with profiling statements included.