Princeton University
COS 217: Introduction to Programming Systems
Execution Profiler: Development Stages

Stage 1: Utility Functions for the *instrument* Program

Design a function that accepts a string which is an assembly language instruction. The function should determine whether the given instruction is a .type directive. If the given instruction is a .type directive, then the function should determine the function name that is embedded within the instruction.

Design a function that accepts a string which is an assembly language instruction. The function should determine whether the given instruction is a label definition whose label is the name of a function. If it is such an instruction, the function should determine the function name that is embedded within the instruction.

Testing:

Design a driver program that reads an assembly language program from a file, calls your utility functions to identify those lines that are .type directives and function label definitions, and writes those lines to another file.

Design a suite of testing C programs. Use "gcc –S" to create assembly language programs from those C programs. Test your driver program (and thus your utility functions) using those assembly language programs.

Stage 2: A Simplified *instrument* Program

Using the utility functions from Stage 1, design a simplified version of the *instrument* program. The simplified *instrument* program should add code at the beginning of each assembly language function as appropriate. The simplified *instrument* program should also add code at the end of the assembly language program that creates function name strings in the rodata section.

Testing:

Enhance your suite of testing C programs as appropriate. Use "gcc –S" to create assembly language programs from those C programs. Instrument each of the assembly language programs. Examine the results.

Stage 3: A Simplified *__count* Function

Design a simplified *__count* function (and subordinate functions) that uses a SymTable object to accumulate function call counts, and that writes those counts to a "stats" file immediately before process exit.

Testing:

Enhance your suite of testing C programs as appropriate. Compile, instrument, assemble, and link those programs. Run those programs. Examine the results in the stats file. The results should be similar to those produced by gprof (but they may not be identical).

Stage 4: The Complete *instrument* Program

Enhance your simplified *instrument* program from Stage 2, thus designing the complete *instrument* program. Specifically, enhance your simplified *instrument* program so it also adds code at the end of each function as appropriate, and so it adds code to the end of the assembly language program that creates a "function name / function start address / function end address" table in the rodata section.

Testing:

Enhance your suite of testing C programs as appropriate. Use "gcc –S" to create assembly language programs from those C programs. Instrument each of the assembly language programs. Examine the results.

Stage 5: The Complete __count function

Enhance your simplified *__count* function (and subordinate functions) from Stage 3, thus designing the complete *__count* function. Specifically, enhance your simplified *__count* function so it uses signals and interval timers to accumulate function signal counts, and writes those counts to a "stats" file immediately before process exit.

Testing:

Enhance your suite of testing C programs as appropriate. Compile, instrument, assemble, and link those programs. Run those programs. Examine the results in the stats file. The results should be similar to those produced by gprof (but they may not be identical).

Stage 6: Testing Strategy

Describe your strategy for testing the complete execution profiler in your readme file.