CSC 526 - Principles of Compiler Design II

Debuggers and Profilers

Outline

- Debugging
  - Signals
  - How debuggers work
  - Breakpoints
- Profiling
  - Event-based
  - Statistical

What is a Debugger?

“A software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables.”
– Microsoft Developer Network

Machine-Language Debugger

- Only concerned with assembly code
- Show instructions via disassembly
- Inspect the values of registers, memory
- Key features
  - Attach to process
  - Single-stepping
  - Breakpoints
  - Conditional breakpoints
  - Watchpoints

Signals

- A signal is an asynchronous notification sent to a process about an event.
- Examples:
  - User pressed Ctrl-C
  - Exceptions (divide by zero, null pointer, etc.)
  - From the operating system (`SIGPIPE`)
- A signal handler is a procedure that executes when the signal occurs

Signal Example

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

int global = 11;

void my_handler() {
  printf("In signal handler, global = %d\n", global);
  exit(1);
}

int main() {
  int* pointer = NULL;
  signal(SIGSEGV, my_handler);
  global = 33;
  *pointer = 0;
  global = 55;
  printf("Outside, global = %d\n", global);
  return 0;
```

Attaching a Debugger

- Requires operating system support
- There is a special system call that allows one process to act as a debugger for a target
- Once this is done, the debugger can basically “catch signals” delivered to the target

Building a Debugger

- We can get breakpoints and interactive debugging:
    - Attach to target
    - Set up signal handler
    - Add in exception causing instructions
    - Inspect globals, etc.

Debugger Signal Example

```c
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

#define BREAKPOINT = *(0)=0

int global = 11;

void debugger_signal_handler() {
  // debugger code here
}

int main() {
  signal(SIGSEGV, debugger_signal_handler);
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
  return 0;
```

Reality

- We are not changing the source code
- Instead, we modify the assembly
- We cannot insert instructions because labels are already set at known constant offsets
- Instead, we can change them

Software Breakpoint Recipe

- Debugger has already attached and set up its signal handler
- User wants a breakpoint at instruction x.
- Store the instruction at x to another location, call it old_x
- Replace instruction at x with $*0=0$ (something illegal)
- The signal handler replaces the instruction at x with the stored old_x.
- Give the user an interactive debugging prompt.

Advanced Breakpoints

- Get register and local values by walking the stack
- Optimization: hardware breakpoints – a special register that can signal an exception
- Feature: conditional breakpoint – break at instruction x if $\mathit{variable} = \mathit{value}$
- In this case, the signal handler checks to see if $\mathit{variable} = \mathit{value}$

Single-Stepping

- Debuggers allow you to advance through the code one instruction at a time
- To implement this, put a breakpoint at the first instruction
- The “single step” or “next” interactive command is
    - Put a breakpoint at the next instruction
    - Resume execution

Watchpoints

- You want to know when a variable changes
- A watchpoint is like a breakpoint, but it stops execution whenever the value at location L changes.
- Software watchpoints
    - Put a breakpoint at every instruction
    - Check the current value of L against a stored value
    - If different, give the user a debugging prompt
    - Otherwise, set the next breakpoint and continue
- Hardware watchpoints
    - Special register holds L and if the value at address L changes, then the CPU raises an exception

Source-Level Debugging

- What if we want to ...
  - Put a breakpoint at a source-level location, for example, a breakpoint at line 20 in `main.c`
  - Single-step through source-level instructions, for example, from line 20 to line 21 in `main.c`
  - Inspect source-level variables, for example, `my_var`, not a register
- Here we need help from the compiler

Debugging Information

- The compiler can emit tables
  - Map every line in the program to the assembly instruction range
  - Map every line in the program to variables in scope and where they are located (registers, memory)
- Setting a breakpoint is a table lookup: set a breakpoint at the beginning of the instruction range
- Single-step is a table lookup: set next breakpoint at the end of the instruction range $+ 1$
- Inspecting a value is a table lookup
- The tables need to take up space in the executable

Replay Debugging

- Running and single-stepping are handy
- But, wouldn't it be nice to go back in time?
- That is, from the current breakpoint, undo instructions in reverse order

Time Travel

- Store the state at various times
  - time $t = 0$ at program start
  - time $t = 88$ after 88 instructions
- When the user asks you to go back one step, then you actually go back to the last stored state and run the program forward again with a breakpoint
  - for example, to go back from $t = 150$, put a breakpoint at instruction 149 and re-run from $t = 88$'s state
- Some debuggers have this power, for example `ocamldebug`

Valgrind

- Valgrind is a suite of free tools for debugging and profiling
  - finds memory errors, profiles cache times, call graphs, profiles heap space
- It does so via dynamic binary translation
  - Basically, it is an interpreter
  - There is no need to modify, recompile or relink
  - Works with any language
- Can attach `gdb` to your process, etc.
- Problem: slowdown by a factor of 5x to 100x

Profiling

- A profiler is a performance analysis tool that measures the frequency and duration of function calls as a program runs
- Flat profile
  - computes the average call times for functions, but does not break times down based on context
- Call-Graph profile
  - computes call times for functions and also the call-chains involved

Event-Based Profiling

- Interpreted languages provide special hooks for profiling
  - Java: JVM-Profile Interface
  - Python: sys.set_profile() module
- You register a function that will get called whenever the target program calls a method, loads a class, allocates an object, etc.

Statistical Profiling

- You can arrange for the operating system to send you a signal for every *n* seconds
- In the signal handler, you determine the value of the target program counter and append it to a file (sampling)
- Later, you use that debug information table to map the program counter values to procedure names and then sum up the amount of time in each procedure

Sampling Analysis

- Advantages
  - Simple and cheap – the instrumentation is unlikely to disturb the program too much
  - No big slowdown
- Disadvantages:
  - Can completely miss periodic behavior depending on sampling rate
  - High error rate: if a value is n times the sampling period, then the expected error in it is $\sqrt{n}$ sampling periods

Summary

- A debugger helps detect the source of a program error by single-stepping through the program and inspecting variable values
- Breakpoints are the fundamental building block of debuggers; breakpoints can be implemented with signals and special operating system support
- A profiler is a performance analysis tool that measures the frequency and duration of function calls as a program runs
- Profilers can be event-based or sample-based.