Profilers and Debuggers

Introductory Material

• First, who doesn’t know assembly language?
  - You’ll get to answer all the assembly questions. Yes, really.

• Lecture Style:
  - “Sit on the table” and pose questions. So, wake up!

• Lecture Goal:
  - After the lecture you’ll think, “Wow, that was all really obvious. I could have done that.”

One-Slide Summary

• A debugger helps to 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 OS 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- or sampling-based.
<table>
<thead>
<tr>
<th>What is a Debugger?</th>
</tr>
</thead>
<tbody>
<tr>
<td>“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.”</td>
</tr>
<tr>
<td>MSDN</td>
</tr>
</tbody>
</table>

### Machine-Language Debugger

- Only concerned with **assembly code**
- Show instructions via **disassembly**
- Inspect the values of registers, memory
- Key Features (we’ll explain all of them)
  - Attach to process
  - Single-stepping
  - Breakpoints
  - Conditional Breakpoints
  - Watchpoints
Signals

- A **signal** is an asynchronous notification sent to a process about an event:
  - User pressed Ctrl-C (or did `kill %pid`)
  - Exceptions (divide by zero, null pointer)
  - From the OS (`SIGPIPE`)
- You can install a **signal handler** - a procedure that will be executed when the signal occurs.
  - Signal handlers are vulnerable to race conditions. Why?

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

int global = 11;

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

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

Signal Example

- What does this program print?

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
  - What are the security concerns?
- Once this is done, the debugger can basically “catch signals” delivered to the target
  - This isn’t really what happens, but it’s a good explanation...
Building a Debugger

```c
#include <stdio.h>
#include <signal.h>
#define BREAKPOINT *(0)=0

int global = 11;

int debugger_signal_handler() {
    printf("debugger prompt: 
");
    // debugger code goes here!
}

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

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

Reality

• We’re not really changing the source code
• Instead, we modify the assembly
• We can’t insert instructions
  - Because labels are already set at known constant offsets
• Instead we change them

Adding A Breakpoint

• Add a breakpoint just after “global = 33;”

Storage Cell: movl $55, _global
  _main + 15

```c
.global _global
.data
.align 4
_global:
.long   11
.def    ___main
.section .rdata,"dr"
LC0:
.ascii "Outside, global = %d\12\0"
.text
.globl _main
.def    _main
_main:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
andl $-16, %esp
movl $0, %eax
addl $15, %eax
addl $15, %eax
shrl $4, %eax
sall $4, %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
ret
```
Software Breakpoint Recipe

- Debugger has already attached and set up its signal handler
- User wants a breakpoint at instruction X
- Store \( (X, \text{old_instruction_at}_X) \)
- Replace instruction at X with “**0=0”
  - Pick something illegal that’s 1-byte long
- Signal handler replaces instruction at X with stored \( \text{old_instruction_at}_X \)
- Give user interactive debugging prompt

Advanced Breakpoints

- Get register and local values by walking the stack
- Optimization: hardware breakpoints
  - Special register: if PC value = HBP register value, signal an exception
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: condition breakpoint: “break at instruction \( X \) if \( \text{some_variable} = \text{some_value} \)”
  - As before, but signal handler checks to see if \( \text{some_variable} = \text{some_value} \)
  - If so, present interactive debugging prompt
  - If not, return to program immediately

Single-Stepping

- Debuggers allow you to advance through code on instruction at a time
- To implement this, put a breakpoint at the first instruction (= at program start)
- The “single step” or “next” interactive command is equal to:
  - Put a breakpoint at the next instruction
    - +4 bytes for RISC, +X bytes for CISC, etc.
  - Resume execution
Watchpoints

- Sometimes you want to know when a variable in memory changes
  - Perhaps because you have tricky aliasing problems
- A watchpoint is like a breakpoint, but it signals when the value at location L changes, regardless of what instruction is being executed
- How could we implement this?

Watchpoint Implementation

- Software Watchpoints
  - Put a breakpoint at every instruction (ouch!)
  - Check the current value of L against a stored value
  - If different, give interactive debugging prompt
  - If not, set next breakpoint and continue
- Hardware Watchpoints
  - Special register holds L: if the value at L ever changes, the CPU raises an exception

Source-Level Debugging

- What if we want to ...
  - Put a breakpoint at a source-level location (e.g., breakpoint at main.c line 20)
  - Single-step through source-level instructions (e.g., from main.c:20 to main.c:21)
  - Inspect source-level variables (e.g., inspect local_var, not register AX)
- We’ll need the compiler’s help
- How can we do it?
Debugging Information

- The compiler will emit tables
  - For every line in the program (e.g., main.c:20), what assembly instruction range does it map to?
  - For every line in the program, what variables are in scope and where do they live (registers, memory)?
- Put a breakpoint = table lookup
  - Put breakpoint at beginning of instruction range
- Single-step = table lookup
  - Put next breakpoint at end of instruction range +1
- Inspect value = table lookup
- Where do we put these tables?

How Big Are Those Tables?

```c
/* example.c */
#include <stdio.h>
#include <signal.h>

int my_global_var = 11;

void main() {
    int my_local_var = 22;
    my_local_var += my_global_var;
    printf("Outside, my_local_var = %d\n", my_local_var);
}
```

```
gcc example.c" 9418 bytes
"gcc -g example.c" 23790 bytes
```

Debugging vs. Optimizing

- We said: the compiler will emit tables
  - For every line in the program (e.g., main.c:20), what assembly instruction range does it map to?
  - For every line in the program, what variables are in scope and where do they live (registers, memory)?
- What can go wrong if we optimize the program?
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
- Intuition: functional + single assignment
  - \[ x = 11; \quad \text{let } x_0 = 11 \text{ in} \]
  - \[ x = x + 22; \quad \text{let } x_1 = x_0 + 22 \text{ in} \]
  - breakpoint ;
  - \[ x = x + 33; \quad \text{let } x_2 = x_1 + 33 \text{ in} \]
  - print x

Time Travel

- **Store the state** at various times
  - time t=0 at program start
  - time t=88 after 88 instructions
  - ... why does this work?
- When the user asks you to go back one step, you actually **go back to the last stored state** and run the program forward again with a breakpoint
  - e.g., to go back from t=150, put breakpoint at instruction 149 and re-run from t=88’s state
- ocamldebug has this power - try it!

Valgrind

- **Valgrind** is a suite of tools for debugging and profiling Linux programs
  - Finds memory errors, profiles cache times, profiles call graphs, profiles heap space
- It does so via **dynamic binary translation**
  - Fancy words for “is an interpreter”
  - No need to modify, recompile or relink
  - Works with any language
- Can attach gdb to your process, etc.
- Problem: slowdown of 5x-100x
  - Rational Purify (commercial) is similar
  - PIN (Kim Hazelwood) is >3x faster (local research!)
Valgrind Example

```c
int main() {
    int some_var = 55;
    int array[10];
    int i;
    for (i=0;i<=10;i++)
        array[i] = i;
    printf("some_var = %d\n", some_var);
}
```

What's the output?

```
[weimer@weimer-laptop ~]$ ./a.out
some_var = 10
```

Valgrind Example

```c
int main() {
    int some_var = 55;
    int array[10];
    int i;
    for (i=0;i<=10;i++)
        array[i] = i;
    printf("some_var = %d\n", some_var);
}
```

Sad to say, valgrind won't help you here. Psyche!

```
[weimer@weimer-laptop ~]$ ./a.out
some_var = 10
```

DDD

- Gnu Data Display Debugger
  - Similar in spirit to Visual Studio's built-in debugger
  - But for gdb, the Java debugger, the perl debugger, the python debugger, etc.
- How does this work?
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, JVM API
  - Python: sys.set_profile() module
  - Ruby: profile.rb, etc.
- You **register a function** that will get called whenever the target program calls a method, loads a class, allocates an object, etc.
  - You could do this for PA5

JVM Profiling Interface

- VM notifies profiler agent of various **events** (heap allocation, thread start, method invocation, etc.)
- Profiler agent issues control commands to the JVM and communicates with a GUI
Statistical Profiling

• You can arrange for the operating system to send you a signal (just like before) every X seconds (see alarm(2))
• In the signal handler you determine the value of the target program counter
  - And append it to a growing list file
  - This is called sampling
• Later, you use that debug information table to map the PC values to procedure names
  - Sum up to get 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 (e.g., you sample every k seconds but do a network send at times 0.5 + nk seconds)
  - High error rate: if a value is n times the sampling period, the expected error in it is sqrt(n) sampling periods
• Read the gprof paper for midterm2

One-Slide Summary

• Real-world programs must have error-handling code. Errors can be handled where they are detected or the error can be propagated to a caller.
• Passing special error return codes is itself error-prone.
• Exceptions are a formal and automated way of reporting and handling errors. Exceptions can be implemented efficiently and described formally.
Homework

- **Midterm 2** - Thursday April 12 (2 days)
  - Covers Lectures 10 - 21 and all reading, WA’s and PA’s done during that time
  - Everything *after* LR parsing

- **Midterm 2 Review Session**
  - Olsson 228E, 5pm - 6pm