CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska
Winter 2007
Lecture 26 – Class Wrap-Up and Profilers

About The Final

- Date: Next Thursday, March 15th
- Time: 8:30am-10:20am
- Location: in regular classroom

- Closed books, closed laptops
- Bring 2 pieces of paper (letter size) with notes
  - You can write on both sides

Content for Our Final

- Lectures 7 through 25
  - C (some questions may require writing code)
  - Debuggers: gdb
  - Linking and makefiles (you may have to write a Makefile)
  - Version control systems: CVS
  - Testing and specifications
  - Writing robust and readable code
  - C++ (some questions may require writing code)
    - NO templates and NO STL
  - Threads (no code, just key ideas)
- There will be around 10 questions on the final

How to Study

- Go over lecture slides
- Go over the code we studied in class
- Go over the assignments
- Look over reading materials
  - They provide more details about different topics
- Try the practice finals
  - Posted on the class website
  - Ignore questions about profilers

So What Have We Learned?

- Intro to linux: model of the file system, processes, users, permissions
- Shell (bash): command line, shortcuts, variables, I/O redirection, scripts, automating
- Regular expressions
- Utilities: grep, awk, sed, and find
- C: syntax, but also pointers, manual memory management, stack & heap, dynamic data structures, arrays, strings, type casts, file I/O, const qualifier,...

So What Have We Learned?

- Tools: the C preprocessor, the linker, make, version control system (cvs), debugger (gdb)
- Software engineering: overview of software development process, specifications, testing, writing readable and robust code
- C++: basic syntax, copy constructor, const qualifier, call-by-reference, inheritance, virtual (or not) functions, abstract classes, templates, STL
- Threads and concurrency

So What Have We Learned?

- Societal and ethical implications
  - Impacs of technology on society

Where To Go From Here

- With the amount of new information and the speed of the class, you will likely forget a lot
- BUT you know what exists
- You know where to find the information
- You can re-learn things quickly when you need them
- You are now used to learning new tools, new environments, new languages, keep learning

Some Other Things To Learn

- There is plenty more to learn about each topic that we covered
- There are plenty of other topics too
- Some things that come to mind
  - Languages: perl, python, C#
  - Technologies: cgi, php, .NET, XML, RPCs, RMIs
  - Tools: autoconf, automake
    - latex... when you can compile you essays too!
  - Techniques: design patterns
    - best practices in general

Summary

- We hope 303 gave you a head start on many important topics and skills
- We hope you will continue to learn
  - When you are curious about a topic X
    - Search for “X”, “X tutorial”, “X documentation”
    - Plenty of decent tutorials out there
  - Learn from each other
    - Better ask a question today rather than tomorrow
  - Books are good too
- The more you know, the easier things get

Since We Are Talking About Continuing to Learn...

- Let's learn about another type of tool called a profiler
- A profiler monitors and reports performance information about a program execution
- Useful to understand where program consumes most time and/or space
- “90/10 rule of programs”
- Profiler helps you “find the ten”
  - Helps find bottleneck of the program

Basic Features

- Different profilers profile different things
- gprof, a profiler for code produced by gcc
- gprof is pretty typical and reports
  - Call counts
    - # times function A called function B
    - # times each function was called
  - Time samples
    - # of times program was executing function A when profiler collected a sample
    - Profiler “wakes-up” periodically and checks where program is

Profiler Implementation Basics

- Call counts
  - Add code to every function-call to update a table indexed by function pairs
  - The profiler instruments your code
    - gcc does that when you specify option “-pg”
- Time samples
  - Use the processor's timer
  - Wake-up periodically
  - Check the value of the program counter

Using gprof (1)

- Build with option -pg
  - During linking and during compiling
  - Compiler will actually "instrument" the code
    - Needed to get the call count information
    - Not needed to simply get the time samples
- Execute the program to get file `gmon.out`
- Run `gprof` to analyze results

```
gprof profile-me gmon.out > results.txt
```

Using gprof (2)

- To get line-by-line results
- Build with debug option `-g` in addition to `-pg`
- Execute the program to get file `gmon.out`
- Run `gprof` but request line-by-line results

```
gprof -l profile-me gmon.out > results.txt
```

Be Careful

- Profiler is sampling
  - Make sure there are enough samples
  - Program must run for a sufficiently long time
- Results depend on your inputs
  - For instance, imagine array is already sorted
- Several results are estimates
  - Example: The amount of time spend in children
  - “If foo used 5 seconds in all, and 2/5 of the calls to foo came from A, then foo contributes 2 seconds to A's children time, by assumption.” (from gprof doc)
  - So these estimates can sometimes be totally wrong

Be Careful (2)

- Program will be much slower than normal because of the overhead of profiling
  - Profiler will not report these times
  - Careful if you also measure time with “time”
    - Example: `time profile-me ...`
    - Returns 42s when running with profiling
    - Returns 10s otherwise

Performance Tuning

- "First make it work, then make it fast"
- Step 1, get things to work well. Write clean code
- Step 2, figure out the bottlenecks & perf. issues
  - Sometimes they are obvious
  - Otherwise, the profiler can help
  - Optimize your code only if necessary
- Step 3, optimize your overall algorithm first
- Step 4, use low-level tricks only if you need to

Compiler Optimization

- Compiler trades “compile-time” for “code-quality”
- By default, compiler optimizes for short compilation time and also ease of debugging resulting code
- But we can ask the compiler to spend more time optimizing the code that it produces
- Sometimes this is enough to get good performance
- In our example, time goes down from 10s to 2.5s
- Compile with “optimization flags”

```
-01, -02, -03
```

- Using compiler to optimize won't help a bad algorithm

Summary

- Profilers help us understand
  - Where program spends most of its time
  - What functions or chunks of code are executed most frequently
- We use gprof, but many profilers exist
  - Similar goals and similar basic functionality
- Implementation
  - Instrument the code
  - Perform extra work as program runs
- Manual for gprof:

http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html

We hope you enjoyed the class!