Systems Programming
Advanced Software Development
3420ICT / 7420ICT

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2012

Outline

Teaching Team

- Lecturer
  - René Hexel (`r.hexel@griffith.edu.au`)
  - Use 3420ICT / 7420ICT Subject for eMails!
- Tutor
  - René Hexel

Teaching

- Lecture (3 hours)
  - Tuesdays 8–11am, N06_0.14
- Labs (2 hours)
  - start in week 1!
  - N44_1.17 at 11am on Tuesdays
  - demo and recap
  - assignment milestones and feedback

Labs

- Tutor Assistance
  - Ask Questions!
  - Programming Practice
- Part of the Assignments
  - Necessary skills to complete Assignments
  - Programming Environment (Compiler, Makefiles, Subversion, . . .)
  - Milestones are due each week!
  - Come prepared!
- Outside official hours
  - Check Lab closing times!
  - Dwarf is accessible via VLink from home!
  - Most people will need to spend appx. 20 hours / week on SP!

Assessment

- 2 non-trivial Assignments
  - Assignment 1 (25 %), due weeks 1-6
  - Assignment 2 (35 %), due weeks 7-11
  - Milestones due every week from day one (must be submitted by the end of your lab day)!
- End of Semester Exam
  - Worth 40 %
  - Closed Book Exam

Course Resources

- SP Nathan Web Site
  - via Learning@Griffith and
    `http://www.ict.griffith.edu.au/teaching/`*coursecode*
  - Check Notice Board regularly!
  - Read the Policies Page
- Help outside the Lab
  - Use Virgil Message Forum
  - Received your Password? – Check official Student EMail!
- Web Resources
  - Loads of Online Material via the SP Web Page!
- Books, Article, Papers
  - See the Resources Section!

Course Communication

- Notice Board
  - Important updates and changes
- Forum
  - For Student/Tutor/Lecturer communication
  - Help other students if you can
    - Good feedback for yourself to see how well you have understood a topic!
- Web Material
  - Lecture Notes, Articles, Tutorials
  - Code Examples, Model Solutions
  - Made available progressively
    - Check Web Pages regularly

Health and Safety, Policy Guidelines

- Health and Safety
  - Online Induction must be completed before the first lab!
  - Learning@Griffith -> Organisations -> Laboratory Induction
- Student Policies Web Page
  - via Portal
- Problems, Consultation, and Grievances
  - Use the Forum about SP related problems (available any time)!
  - Talk to Lecturer at Lectures, Labs, and Tutorials
  - Open Door Policy
    - Drop by my office any time the door is open!
    - EMail me for an appointment at a specified time!

Administrativa: That's It!

Any Questions?

Submitting Assignments using Subversion

What is Subversion?

- Version Control System
- Allows you manage the life cycle of a program
- Keep track of changes as you develop a program
- View and compare differences between versions
- Go back to an earlier version
- Create Milestones
  - Snapshot of your program at a given point in time
  - Won't change, even if your program keeps changes

How does Subversion work?

- Central repository for all versions of all your files
  - Logbook of changes
- Local working copy
  - Make changes as you go without losing information about earlier versions
- Track changes between versions
  - Make debugging easier
  - “Where did this error sneak into my program?”

An Example

- E.g. a source file `hello.c`

```
int main (void)
{
        printf("Hello, world!\n");

        return 0;
}
```

- Let's put these changes back into the repository:
  - `svn commit hello.c`
  - This is what we need to type on the command line

Preparation – required only once!

- Set up a repository on `dwarf.cit.griffith.edu.au`
  - Log into dwarf using `ssh` or putty
    - e.g. `ssh s1234567@dwarf.cit.griffith.edu.au`
  - Create the repository: `svn_setup sp`
- Create an (empty) assignment working copy
  - `svn checkout file://$HOME/.spsvn-2012/ass1/trunk a1`
  - `cd a1`

Adding Files – required for every new file

1. Go to your checked out working directory
   - `cd a1`
2. Create a new file with your favourite editor
   - e.g. `module1.c`
3. Add the file to Subversion
   - `svn add module1.c`
4. Commit the file to the repository
   - `svn commit -m "Log Message" module1.c`
5. Repeat the last step for any changes you make to any files
   - `svn commit -m "Log Message"`
   - *Without a file name, `svn commit` will commit all files that have changed!*

Committing Changes to Subversion

- Whenever you make any changes, commit them!
  - `svn commit -m "`*`Log Message`*`"`
- Commit early, commit often!
  - Allows you more fine grained control over your changes
  - Backup copies of earlier versions
- What happens if I forget the `-m` ?
  - An editor (usually `vi`) will open
  - In `vi` you can use the `i` key to insert text: enter the log message, then press `ESC` followed by `Shift-Z Shift-Z` to save and commit.

Submitting Assignments: Symbolic Tags

- The Problem:
  - Version numbers (1, 2, 3, ...) are not very readable!
  - Every commit gets its own version number
    - ...even if it belongs to a different project!
    - e.g. commits to Assignment 2 also changes Assignment 1
- The answer: named versions = *tags*
  - First, make sure all files are committed using `svn commit`
  - `svn copy -m "`*`Log`*`"`
    `file://$HOME/.spsvn-2012/ass1/trunk`
    `file://$HOME/.spsvn-2012/ass1/tags/milestone1`
  - (all of the above needs to be on a single line!)
  - Copies the current version to a symbolic tag

Other useful Subversion Commands

- `svn log` *`[filename]`*
  - See the history of changes you made
  - Lists your log messages (make sure they are meaningful!)
  - *filename* is optional!
- `svn diff -r` *`1:2 [filename]`*
  - Show the actual changes between versions *1* and *2*
- `svn diff`
  - Show all the changes since the last `svn commit`
- `svn status` *`[filename]`*
  - Check the current version of a file

Using Subversion over the Internet

- So far: you need to log into `dwarf` first!
  - Can be cumbersome from the labs or at home
- Simply replace the local repository URI on `dwarf`
  - `file://$HOME/.spsvn-2012`
- with the remote URI
  - `svn+ssh://sid@dwarf.cit.griffith.edu.au/export/student/sid/.spsvn-2012`
- Prefer a Graphical User Interface (GUI)?
  - GUI clients available for most Operating systems
    - TortoiseSVN for Windows
    - KSVN for Linux
    - MacSVN for Mac OS X

Multiple Working Copies

- What if you want multiple copies?
  - E.g., one at home, on in the labs
- Simply use `svn checkout` on multiple machines!
- Always commit all your changes after working on a program!
  - `svn commit -m "log message"`
- Bring your local copy up to date before working on any file!
  - `svn update`

Advanced Subversion Commands

- `svn update -r` *`version [filename]`*
  - go back to a specific *version*
- Update your local copy to the latest version
  - `svn update`
  - No `-r` means: go to the latest version (HEAD revision)
- `svn merge -r` *`version1`*`:`*`version2`*
  - merge the changes between two versions into the current working copy

What Else?

- There is a lot more to Subversion!
  - Branches, exporting, group work (outside of SP!), etc.
- Subversion Web Page
  - `http://subversion.tigris.org/`
- Subversion Book (Online and Free!)
  - `http://svnbook.red-bean.com/`

Compiling C Programs

- Integrated Development Environment (IDE)
  - Eclipse, XCode, Visual C++, Project Center, ...
  - Compiles programs at the press of a button (like BlueJ)
  - Often difficult to customise
  - Very rarely support multiple platforms and languages
- Command Line
  - Requires manual invocation
  - Requires knowledge of command line parameters
  - Can be tedious for large projects
  - Cross-platform and -language compilers (e.g. `clang`)
- Makefiles
  - Combine the best of both worlds
  - Recompile a complex project with a simple `make` command

Getting a Command Line Interface

- Via Dwarf
  - using putty (Windows)
  - `ssh dwarf.cit.griffith.edu.au`
- Via a local Terminal
  - Linux: e.g. through the Gnome program menu
  - Mac OS X: e.g. Applications / Utilities / Terminal.app
  - Windows: e.g. Start / Programs / Programming Tools / GNUstep / MSys

$\Rightarrow$ Enter commands to compile your program

- Hit *Return* (or *Enter*) after every command!

Compiling a C program using clang

- Once on the command line change to the directory (folder) your program is in
  - `cd /my/example/directory`
- Compile the source code (e.g. `Hello.c`)
  - `clang Hello.c`
  - Compiles `Hello.c` into an executable called `a.out` (or `a.exe` on Windows)
- `clang -o Hello Hello.c`
  - Compiles `Hello.c` into an executable called `Hello`
  - On Windows always use `Hello.exe` instead of just `Hello`
- `clang -Wall -std=c99 -o Hello Hello.c`
  - Prints all warnings about possible problems
  - Always use `-Wall -std=c99` when compiling your programs!
- `./Hello`
  - Run the `Hello` command from the current directory

Makefiles

- Save compile time
  - only recompile what is necessary
- Help avoiding mistakes
  - prevent outdated modules from being linked together
- Language independent
  - work with any programming language
    - C, C++, Objective-C, Java, ...

How do Makefiles work?

Example (A simple Makefile)

```
Hello:  Hello.c
        clang -Wall -std=c99 -o Hello Hello.c
```

- First Line: Dependency Tree
  - Target and Sources
  - Target: the module to be built (e.g. `Hello`)
  - Sources: pre-requisites (e.g. `Hello.c`)

Make Rules

Example (A simple Makefile)

```
Hello:  Hello.c
        clang -Wall -std=c99 -o Hello Hello.c
```

- Second Line: Make rule
  - command to execute
    - `clang -Wall -std=c99 -o Hello Hello.c`
  - requires a tab character (not spaces) for indentation

Multiple Targets

Example (Makefile for compiling multiple Modules)

```
Program:  module1.o module2.o
        clang -o Program module1.o module2.o

module1.o:  module1.c
        clang -c -Wall -std=c99 -o module1.o module1.c

module2.o:  module2.c module2.h
        clang -c -Wall -std=c99 -o module2.o module2.c
```

- Default Target: first target (`Program`)
  - link two object files (`module1.o` and `module2.o`) into one program (`Program`)

Multiple Targets (2)

Example (Makefile for compiling multiple Modules)

```
Program:  module1.o module2.o
        clang -o Program module1.o module2.o

module1.o:  module1.c
        clang -c -Wall -std=c99 -o module1.o module1.c

module2.o:  module2.c module2.h
        clang -c -Wall -std=c99 -o module2.o module2.c
```

- Second Target: `module1.o`
  - rule to compile object file `module1.o` from `module1.c`
  - `clang -c` compiles a single module (not a full executable)

Multiple Targets (3)

Example (Makefile for compiling multiple Modules)

```
Program:  module1.o module2.o
        clang -o Program module1.o module2.o

module1.o:  module1.c
        clang -c -Wall -std=c99 -o module1.o module1.c

module2.o:  module2.c module2.h
        clang -c -Wall -std=c99 -o module2.o module2.c
```

- Third Target: `module2.o`
  - compile `module2.o` from source `module2.c`
  - also depends on `module2.h` (header file)

Multiple Programs

Example (Makefile for compiling multiple Programs)

```
all:  Program1 Program2

Program1:  module1.o
        clang -o Program module1.o module2.o

Program2:  module2.o module3.o
        clang -o Program module1.o module2.o

module1.o:  module1.c
        clang -c -Wall -std=c99 -o module1.o module1.c

module2.o:  module2.c module2.h
        clang -c -Wall -std=c99 -o module2.o module2.c

module3.o:  module3.c module3.h
        clang -c -Wall -std=c99 -o module3.o module3.c
```

- 'all' target:
  - compiles all programs (`Program1` and `Program2`)
  - does not have any compiler comands itself!

Generic Rules

- Save lots of typing
  - avoid repeating the same compiler call over and over again
- Help with consistency
  - what if you want to change the compiler invocation?
- Simply list suffixes to convert from one file type to another
  - e.g. `.c.o` to compile a `.c` to a `.o` file

Generic Rule Example

Example (Makefile containing a generic rule)

```
.c.o:
        clang -c -Wall -std=c99 -o $*.o $*.c

Program:  module1.o module2.o
        clang -o Program module1.o module2.o

module2.o:  module2.c module2.h
```

- `.c.o:`
  - how to compile a `.c` into a `.o` file
  - `$*` gets replaced by the file name (without extension)

Generic Rule Example (2)

Example (Makefile containing a generic rule)

```
.c.o:
        clang -c -Wall -std=c99 -o $*.o $*.c

Program:  module1.o module2.o
        clang -o Program module1.o module2.o

module2.o:  module2.c module2.h
```

- No need for a `module1.o:` rule!
  - compiler already knows how to compile `.c` into `.o`
  - But: `module2.o` needs a rule (also depends on `.h`)

Generic Rules for Languages other than C

- The `make` utility by default only knows about C
  - "what if I want to compile a different language?"
- Suffixes can be specified
  - using the `.SUFFIXES:` command, e.g.:
  - `.SUFFIXES: .o .m`
    - "a `.o` file can also be compiled from a `.m` (Objective-C) file"

Make Variables

- Allow more flexible make files
  - "what if the compiler is not called `clang`?"
- Variables allow assigning a value, e.g:
  - `CC=clang`
- Varables can be used using `$(`*variable*`)`, e.g.:
  - `$(CC) -c -Wall -std=c99 -o $*.o $*.c`
  - will replace `$(CC)` with `clang`