Linux for CS Students

A Primer

Ohad Lutzky

Technion IIT
CS Department

November 20th, 2006

Structure of this meeting

- First hour - lecture
- Second hour - Q&A

Outline

Outline

Outline

Outline

“Just like Windows”

Or rather, just like the Mac

- Icons, documents, right/double clicks
- Various desktop environments
- Web browsing
- Office suite

Your home directory

- Per-user
- Often saved on the network
- In Windows - Documents and Settings
- Hidden `.files`

Outline

What do we need this for?

- For day-to-day tasks, you don't
- Fast and efficient (i.e. tab completion)
- Excellent for working with remote servers
- Powerful shell scripting (or CSH)
- Absolutely not DOS

What do we need this for?

- For day-to-day tasks, you don't
- Fast and efficient (i.e. tab completion)
- Excellent for working with remote servers
- Powerful shell scripting (or CSH)
- Absolutely not DOS

What do we need this for?

- For day-to-day tasks, you don't
- Fast and efficient (i.e. tab completion)
- Excellent for working with remote servers
- Powerful shell scripting (or CSH)
- Absolutely not DOS

Basic usage

- Commands we'll be using here: `ls`, `cat`, `cd`, `cp`, `mv`, `rm`, `echo`
- Arrows and Tab
- `$PATH`
- Taught in the course

Basic usage

- Commands we'll be using here: `ls`, `cat`, `cd`, `cp`, `mv`, `rm`, `echo`
- Arrows and Tab
- `$PATH`
- Taught in the course

T2, SSH, etc

- From Windows: puTTY
- From Linux: `ssh smyusername@t2.technion.ac.il`
- T2's commandline is extremely similar to Linux's
- Do not use telnet!

Outline

Who cares? / Where's my IDE?

- Your most important day-to-day tool
- Important features
- No IDEs for CSH
- VIM and Emacs are installed on T2

GEdit

- Familiar and easy to use
- Syntax highlighting
- . . . but not much else.
- Recommendation: Learn VIM (`vimtutor`) or Emacs

GEdit

- Familiar and easy to use
- Syntax highlighting
- . . . but not much else.
- Recommendation: Learn VIM (`vimtutor`) or Emacs

Outline

GCC

- Gnu Compiler Collection
- Modern, full-featured compiler
- Encapsulates many language compilers and linker
- T2 has (currently) version 3.4, modern Linuxes have 4.1
- Quite different from Borland C++ or Visual C++
- Also available in Cygwin and DevCPP

Basic usage of GCC

- Compiling and linking (for programs):
  `gcc -o my_app my_app.c`
- Running your program:
  `./my_app`

Compiler flags

- `-ansi`
- `-Wall`
- `-pedantic-errors` (no space!)
- `-g` - debug symbols (GDB, DDD)

```
gcc -ansi -Wall -pedantic-errors
-o my_prog my_prog.c
```

Compiler flags

- `-ansi`
- `-Wall`
- `-pedantic-errors` (no space!)
- `-g` - debug symbols (GDB, DDD)

```
gcc -ansi -Wall -pedantic-errors
-o my_prog my_prog.c
```

Compiler flags

- `-ansi`
- `-Wall`
- `-pedantic-errors` (no space!)
- `-g` - debug symbols (GDB, DDD)

```
gcc -ansi -Wall -pedantic-errors
-o my_prog my_prog.c
```

Outline

Compiling object files

- `-c` - no linking
- Otherwise, same flags as before

```
gcc -ansi -Wall -pedantic-errors
-c -o my_lib.o my_lib.c
```

Compiling object files

- `-c` - no linking
- Otherwise, same flags as before

```
gcc -ansi -Wall -pedantic-errors
-c -o my_lib.o my_lib.c
```

Linking the application

- Linking gives you a runnable application
- We will use `gcc` for linking (like we did before)
- In reality, `ld` is called
- Avoid specifying multiple `.c` files - compile objects instead

```
gcc -o my_app main.o lib1.o lib2.o ...
```

Common caveats

- Circular dependencies
- Unit testing
- Many more!

Outline

Why makefiles?

- Recompiling happens a *lot*
- `-Wall -ansi -pedantic-errors -kimchi -...`
- Recompiles only what has changed
- Great for distributing programs
- Sometimes required by course staff

Why makefiles?

- Recompiling happens a *lot*
- `-Wall -ansi -pedantic-errors -kimchi -...`
- Recompiles only what has changed
- Great for distributing programs
- Sometimes required by course staff

In practice

1. Create your `Makefile` (or `makefile`)
2. Run `make`
3. Debug and fix your code
4. Return to step 2

Outline

A sample makefile

Or: `make` knows what you mean

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors

my_prog: my_prog.c
	$(CC) $(CFLAGS) -o my_prog my_prog.c

my_lib.o: my_lib.c
	$(CC) $(CFLAGS) -c -o my_lib.o my_lib.c
```

A sample makefile

Or: `make` knows what you mean

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors

my_prog: my_prog.c
	$(CC) $(CFLAGS) -o my_prog my_prog.c

my_lib.o: my_lib.c
	$(CC) $(CFLAGS) -c -o my_lib.o my_lib.c
```

A sample makefile

Or: `make` knows what you mean

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors

my_prog: my_prog.c
	$(CC) $(CFLAGS) -o my_prog my_prog.c

my_lib.o: my_lib.c
	$(CC) $(CFLAGS) -c -o my_lib.o my_lib.c
```

A sample makefile

Or: `make` knows what you mean

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors

my_prog: my_prog.c
	$(CC) $(CFLAGS) -o my_prog my_prog.c

my_lib.o: my_lib.c
	$(CC) $(CFLAGS) -c -o my_lib.o my_lib.c
```

A sample makefile

Or: `make` knows what you mean

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors

my_prog: my_prog.c
	$(CC) $(CFLAGS) -o my_prog my_prog.c

my_lib.o: my_lib.c
	$(CC) $(CFLAGS) -c -o my_lib.o my_lib.c
```

A sample makefile

Or: `make` knows what you mean

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors

my_prog: my_prog.c
	$(CC) $(CFLAGS) -o my_prog my_prog.c

my_lib.o: my_lib.c
	$(CC) $(CFLAGS) -c -o my_lib.o my_lib.c
```

A sample makefile

Or: `make` knows what you mean

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors

my_prog: my_prog.c
	$(CC) $(CFLAGS) -o my_prog my_prog.c

my_lib.o: my_lib.c
	$(CC) $(CFLAGS) -c -o my_lib.o my_lib.c
```

A typical multi-file sample

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors -g

all: my_prog

lib1.o: lib1.c header1.h
lib2.o: lib2.c header2.h
my_prog.o: my_prog.c header1.h header2.h # Has main
my_prog: my_prog.o lib1.o lib2.o

clean:
	rm -f my_prog my_prog.o lib1.o lib2.o
```

A typical multi-file sample

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors -g

all: my_prog

lib1.o: lib1.c header1.h
lib2.o: lib2.c header2.h
my_prog.o: my_prog.c header1.h header2.h # Has main
my_prog: my_prog.o lib1.o lib2.o

clean:
	rm -f my_prog my_prog.o lib1.o lib2.o
```

A typical multi-file sample

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors -g

all: my_prog

lib1.o: lib1.c header1.h
lib2.o: lib2.c header2.h
my_prog.o: my_prog.c header1.h header2.h # Has main
my_prog: my_prog.o lib1.o lib2.o

clean:
	rm -f my_prog my_prog.o lib1.o lib2.o
```

A typical multi-file sample

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors -g

all: my_prog

lib1.o: lib1.c header1.h
lib2.o: lib2.c header2.h
my_prog.o: my_prog.c header1.h header2.h # Has main
my_prog: my_prog.o lib1.o lib2.o

clean:
	rm -f my_prog my_prog.o lib1.o lib2.o
```

A typical multi-file sample

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors -g

all: my_prog

lib1.o: lib1.c header1.h
lib2.o: lib2.c header2.h
my_prog.o: my_prog.c header1.h header2.h # Has main
my_prog: my_prog.o lib1.o lib2.o

clean:
	rm -f my_prog my_prog.o lib1.o lib2.o
```

A typical multi-file sample

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors -g

all: my_prog

lib1.o: lib1.c header1.h
lib2.o: lib2.c header2.h
my_prog.o: my_prog.c header1.h header2.h # Has main
my_prog: my_prog.o lib1.o lib2.o

clean:
	rm -f my_prog my_prog.o lib1.o lib2.o
```

A typical multi-file sample

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors -g

all: my_prog

lib1.o: lib1.c header1.h
lib2.o: lib2.c header2.h
my_prog.o: my_prog.c header1.h header2.h # Has main
my_prog: my_prog.o lib1.o lib2.o

clean:
  rm -f my_prog my_prog.o lib1.o lib2.o
```

Better yet...

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors -g
OBJECTS=my_prog.o lib1.o lib2.o
HEADERS=header1.h header2.h
all: my_prog
lib1.o: lib1.c $(HEADERS)
lib2.o: lib2.c $(HEADERS)
my_prog.o: my_prog.c $(HEADERS)
my_prog: $(OBJECTS)
clean:
  rm -f my_prog $(OBJECTS)
run: my_prog
  ./my_prog
```

Better yet...

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic-errors -g
OBJECTS=my_prog.o lib1.o lib2.o
HEADERS=header1.h header2.h
all: my_prog
lib1.o: lib1.c $(HEADERS)
lib2.o: lib2.c $(HEADERS)
my_prog.o: my_prog.c $(HEADERS)
my_prog: $(OBJECTS)
clean:
  rm -f my_prog $(OBJECTS)
run: my_prog
  ./my_prog
```

- Many more features (recursion, automake, phony targets...)
- For C++, use `CXX` and `CXXFLAGS`
- Built-in make support in editors

Summary

- Linux is not so bad (right?)
- Compiling with the commandline - daunting, but not much voodoo
- Makefiles are a powerful timesaving tool

- Outlook
  - Debuggers
  - Valgrind
  - C without a spoon