Tools

- Introduction
- Introduction Continued
- Using `gcc`
- `gcc` phases
- Common `gcc` Options
- Common `gcc` Options Continued
- Common `gcc` Options Continued
- Using `make`
- `make` Gotchas
- `make` Gotchas Continued
- Using the Gnu Debugger `gdb`
- `gdb` Commands
- `gdb` Commands Continued
- Debugging Using `ddd`
- strace
- Memory Debugging
- Example Use of valgrind
- Running valgrind
- Another Run of valgrind
- Running valgrind without leaks
- Using `emacs` as an IDE
- Shell Families

- Simple Command Facilities
- Simple Command Facilities Continued
- Simple Command Facilities Continued
- Programming Features of Unix Shells
- Programming Features of Unix Shells Continued
- Environmental Variables
- LD_LIBRARY_PATH
- The X Window System
- The X Window System Continued
- The X Window System Continued
- Using VNC for Light-Weight Remote X-Access
- Using VNC Continued
- References

Introduction

- A high-level overview of the programming tools which are being used in this course. Tools are not the main emphasis of this course and the material covered here is more in the nature of a very basic survival guide.
- Compiling using `gcc`.
- Automating compilation using `make`.
- Debugging using `gdb`.
- Using `ddd` as a GUI for `gdb`.
- System call tracing.
- Memory debugging.

Introduction Continued

- Using `emacs` as an *Integrated Development Environment*.
- Shells.
- Environmental variables.
- The X-window System.
- Light-weight remote access using VNC.

Using gcc

- The GNU C compiler is the most popular open-source compiler.
- gcc is not only used for compiling C. Different front-ends allow it to compile C++ and Objective-C (among other languages). Different backends allow it to be widely retargeted to different architectures. It can also be used as a cross-compiler.

gcc phases

- The `gcc` program really is simply a driver program which invokes other programs which do the real work:

  `cpp`
  : The C preprocessor.

  `cc1`
  : Compiles the preprocessed source to assembly language.

  `as`
  : Assembles assembly language program to a relocatable object file.

  `ld`
  : Links with libraries to produce an executable.

Common gcc Options

`-o` *filename*

Specify name of output file as *filename*. Usually not needed for object files, but desirable for executables which have the default name `a.out` (for historical reasons).

`-c`

Compile without linking. By default object file names formed from source file names by replacing `.c` extension with `.o`.

`-D`*macro-definition*

Define a C preprocessor macro using *macro-definition*. Examples: `-DNDEBUG`, `-DDO_TEST=1`.

`-g -ggdb`

Include debugging information. `-ggdb` includes macros and `enum`'s in a format only understood by the `gdb` debugger.

Common gcc Options Continued

-I*dir*

Prepend *dir* to list of directories used to seach for `include` files.

-L*dir*

Prepend *dir* to list of directories used to seach for libraries.

-l*foo*

Link against library `lib`*foo*. Prefers shared libraries to static libraries. Searches for outstanding functions in libraries in order in which libraries are specified on the command lines. It is imperative that the object files be specified before the libraries.

Common gcc Options Continued

`-O`*n*

Optimize at level *n* for *n* < 3. Can be combined with `-g`.

`-static`

Link agains static libraries only.

`-std=c99` or `-std=c11`

Enable C99 or C11 features.

`-Wall`

Produce warning messages for dubious constructs. Similar to using `lint`.

Using `make`

- Maintains dependencies betweens files and automatically rebuilds a dependent file (called a *target*) when any of its prerequisite files changes.

- A `Makefile` basically consists of rules which give the prerequisites for each target file followed by commands to rebuild the target.

```
CC=        gcc
CFLAGS=    -g -Wall

LIBS=      -L $$HOME/lib -lmylib

OFILES= \
  main.o \
  util.o

all:       foo

foo:       $(OFILES)
           $(CC) $(OFILES) $(LIBS) -o $@
```

`make` Gotchas

- Commands **must** start with a tab character as the first character on the line.
- `$` is used to signal the expansion of `make` *macros*. Other occurrences of `$` must be quoted by repeating the `$` twice.
- Older `make`'s do not handle the transitive closure of dependencies correctly: i.e., if there is a rule telling `make` how to build `foo` from `bar` and another rule telling `make` how to build `bar` from `fie`, then given the existence of file `fie`, those `make`'s cannot figure out how to make `foo`.

`make` Gotchas Continued

- Successive commands are executed in **separate** processes. Hence

```
foo:    bar
        cd $$HOME/foodir
        wc bar >foo
```

  the `cd` has no effect. If the `wc` command is to be run in the `~/foodir` directory, then the above should be written as:

```
foo:    bar
        cd $$HOME/foodir; \
        wc bar >foo
```

- Linux's gnu `make` is very full-featured and does not have the problems of other `make`'s.

Using the Gnu Debugger gdb

- Command-line driven.
- Can be used to debug already running programs (by `attach`ing to a process) or do a post-mortem analysis of a `core` dump.
- Can be used to insert breakpoints and examine data.
- Can step thru the program both at the source level and at the machine instruction level.
- For source level debugging to work, it is necessary that the program be compiled using the `-g` or `-ggdb` option.

gdb Commands

`attach`, `at`

Attach to a process specified by its pid. Stops the process and allows the debugger to control it.

`break`, `b`

Set a breakpoint on a function or particular line.

`backtrace`

Print a stack trace.

`clear`

Clear previously set breakpoint.

`continue`, `c`

Continue execution till the next breakpoint.

`detach`

Detach from the current process.

`display`

Display the value of an expression each time execution stops.

gdb Commands Continued

`list`

List source lines.

`next`, `n`

Step to next line in current function without stepping over any functions.

`print`, `p`

Print the value of a expression.

`run`, `r`

Start execution of the current program from the beginning. Can specify arguments as on the command-line.

`step`

Step to next line, stepping into functions if any.

Debugging Using ddd

- A GUI frontend to gdb.
- Allows examining graphs of complex data structures.
- Allows accessing most gdb commands via menus.
- Shows gdb command-line trace.

strace

A useful tool available on many Unix families which will trace all system-calls made by when executing a program.

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 30 vars */]) = 0
brk(0)                                  = 0xbe9000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = \
  0x7fb3ec6af000
[LOTS OF STUFF DELETED]
write(1, "tools_004.html\ttools_013.html\tto"..., 60tools_004.html      \
  tools_013.html        tools_022.html  tools_031.html
) = 60
close(1)                                = 0
munmap(0x7fb3ec6ae000, 4096)            = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
$
```

Memory Debugging

When writing C or C++ programs, it is a good idea to ensure freeing of all dynamically allocated memory. Various tools can help:

- The gnu library's memory debugger: All allocation calls between calls to `mtrace()` and `muntrace()` are logged in file specified by environment variable `MALLOC_TRACE`.
- Electric-Fence: Uses VM hardware to setup a red-zone around dynamically allocated buffers to allow detecting buffer overflows.
- Memwatch: Detects erroneous allocation patterns.
- Valgrind: Suite of tools which can detect memory, cache, and threading problems.

Example Use of valgrind

Consider following leaky program with 3 errors:

```
/** Leaky program which checks for existence of file
 *  specified by "argv[1]/argv[2]"
 */
int
main(int argc, const char *argv[])
{
  if (argc != 3) {
    fprintf(stderr, "usage: %s DIR FILE_NAME\n", argv[0]);
    exit(1);
  }
  char *path =
    malloc(strlen(argv[1]) + 1 + strlen(argv[2]));
  strcpy(path, argv[1]); strcat(path, "/");
  strcat(path, argv[2]);
  if (!fopen(path, "r")) {
    fprintf(stderr, "file \"%s\" does not exist\n", path);
    exit(1);
  }
  return 0;
}
```

Running valgrind

Edited output:

```
$ make leaky-file-check
gcc -g -Wall -std=c11    leaky-file-check.c   -o leaky-file-check
$ ./leaky-file-check . no-such-file
file "./no-such-file" does not exist
$ valgrind --leak-check=full ./leaky-file-check . no-such-file
==30205== Invalid write of size 1
...
==30205==    by 0x40080A: main (leaky-file-check.c:18)
file "./no-such-file" does not exist
==30205==
==30205== HEAP SUMMARY:
==30205==     in use at exit: 14 bytes in 1 blocks
==30205==   total heap usage: 2 allocs, 1 frees, 582 bytes allocated
==30205==
==30205== LEAK SUMMARY:
==30205==    definitely lost: 0 bytes in 0 blocks
==30205==    indirectly lost: 0 bytes in 0 blocks
==30205==      possibly lost: 0 bytes in 0 blocks
==30205==    still reachable: 14 bytes in 1 blocks
==30205==         suppressed: 0 bytes in 0 blocks
...
$
```

Another Run of valgrind

```
$ valgrind --leak-check=full ./leaky-file-check . leaky-file-check.c
...
==30212== Invalid write of size 1
...
==30212==    by 0x40080A: main (leaky-file-check.c:18)
...
==30212== 20 bytes in 1 blocks are definitely lost in loss record 1 of 2
...
==30212==    by 0x4007A4: main (leaky-file-check.c:15)
==30212==
==30212== LEAK SUMMARY:
==30212==    definitely lost: 20 bytes in 1 blocks
==30212==    indirectly lost: 0 bytes in 0 blocks
==30212==      possibly lost: 0 bytes in 0 blocks
==30212==    still reachable: 568 bytes in 1 blocks
==30212==         suppressed: 0 bytes in 0 blocks
...
$
```

Running valgrind without leaks

```
$ valgrind --leak-check=full ./file-check . leaky-file-check.c
==30244== Memcheck, a memory error detector
==30244== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==30244== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==30244== Command: ./file-check . leaky-file-check.c
==30244==
==30244==
==30244== HEAP SUMMARY:
==30244==     in use at exit: 0 bytes in 0 blocks
==30244==   total heap usage: 2 allocs, 2 frees, 589 bytes allocated
==30244==
==30244== All heap blocks were freed -- no leaks are possible
==30244==
==30244== For counts of detected and suppressed errors, rerun with: -v
==30244== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

Using `emacs` as an IDE

- `M-x compile` will compile a program using a specified compilation command.
- `C-x ‘` can be used to position cursor at line causing compiler error.
- `M-x gdb` can be used to run `gdb` inside emacs.
- `M-x font-lock-mode` can be used to add lexical coloring to buffers containing program source.
- `M-x shell` allows running a shell within emacs.
- Complete violation of Unix philosophy of *doing one thing only* (xkcd comic).

Shell Families

- Two main families of shells: shells based on the original Bourne-shell `sh` and shells based on the `csh`.
- The `sh` family includes the Bourne-SHell `sh`, the Korn-SHell `ksh`, the Z SHell `zsh` and GNU's Bourne-Again SHell `bash`. The latter includes some `csh` features as well.
- The `csh` family includes the original Berkeley Unix C Shell `csh`, and GNU's `tcsh`.
- Typically, the command prompt (which is customizable using shell-variable `PS1`), includes a character which depends on the shell family and user type: `$` for `sh`-family shells for non-root users, `%` for `csh`-family shells for non-root users and `#` for root users in all shells.

Simple Command Facilities

- Most commands are executed by the shell launching a separate process corresponding to the command. A small number of commands (like `cd`) are built-in to the shell and execute within the same process as the shell.

- Globbing replaces *patterns* in command-line with a sorted list of matching file names. `*` matches any string (including null), `?` matches any single character and `[...]` matches any one of the enclosed characters.

```
$ ls foo/*.c *.? src/*.[ch] src/[a-f]*
```

- Distinguish globbing from *regular-expressions* used by many tools like `grep`.

Simple Command Facilities Continued

- Simple I/O redirection uses > to redirect standard output to a file, '>>' to append standard output to a file and < to redirect standard input from a file. Use | between two commands to redirect the standard output of one command to the standard input of another.

```
$ ls *.c | wc -l > ncfiles.count
```

- Special characters can be quoted by enclosing within single quotes ' or double-quotes " (the latter allows expansion of shell and environmental variables), or by being escaped using a \.

```
$ echo abc > \>\*; ls -l ">*"; rm '>*'
-rw-rw-r--    1 umrigar  umrigar         4 Jan 30 00:05 >*
$
```

Simple Command Facilities Continued

- If a word begins with a unquoted ~, then following characters upto the first / treated as a *login-name*. This sequence of characters is replaced by the home directory of the user corresponding to *login-name* (the current user if *login-name* is empty).

```
cd ~/projects; ls ~joe/projects
```

- Words enclosed within back-quotes ‘ are replaced by the output which results from executing the contents within the ‘ as a command.

```
$ wc -l ‘find . -name ’*.c’ -print‘
```

- Modern shells like `tcsh` and `bash` provide autocompletion of commands and filenames, typically using the TAB-key.

Programming Features of Unix Shells

- Shell variables denoted as words starting with `$` like `$var` or `${var}`. Note that the `$` is used only when reading the value of the variable as in `$var`, but not when assigning to the variable as in `var=1`.

- Control constructs like `&&`, `||`, `if-then-else-fi`, `for-do-done` and `case`. Exact syntax varies between shell families.

- `if-then-else-fi` condition based on successful/unsuccessful execution of a command which can be the `test` command (which many shells allow to be abbreviated as `[ ]`).

Programming Features of Unix Shells Continued

- Modern shells allow subroutines.
- For maximum portability, I write simple shell scripts using `sh`. For more involved scripts, I use a scripting language like `perl` or `ruby`.

```
for f in *.c; do \
  if [ ! -r ‘basename $f .c‘.h ]; \
  then \
    echo "$f has no header file"; \
  fi; \
done
```

Environmental Variables

- Shell variables are visible within the shell. Specifically, they are not visible to external programs which are launched by the shell.

- To pass variables to external programs, they must be set in the environment. This can be done using `export` (sh-family) or `setenv` (csh-family).

- A important environmental variable is the `PATH` variable which is a `:` delimited set of directories which are searched by the system for the program to be executed for a specified command.

```
$ export PATH=$HOME/bin:$PATH
```

- Certain terminal-based applications require setting the `TERM` environmental variable:

```
$ export TERM=vt100
```

LD_LIBRARY_PATH

- Another important environmental variable is `LD_LIBRARY_PATH` which is a `:` delimited set of directories which are searched for dynamic libraries when a program is loaded.

```
% setenv LD_LIBRARY_PATH $HOME/lib:$LD_LIBRARY_PATH
```

- Use `ldd` to see a program's dynamic dependencies:

```
$ ldd ‘which ls‘
        libc.so.1 =>     /usr/lib/libc.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
        /usr/platform/SUNW,Sun-Fire-V440/lib/libc_psr.so.1
$
```

The X Window System

- X is a network transparent graphical user interface.
- Developed in the mid-80s.
- The user interacts with a computer (or X-terminal) running a X-server.
- The application (which may be on a remote computer) is referred to as a X-client and uses the X-protocol to interact with the user on the *screen* of a *display* using the X-server.
- The application must be told where the display/screen is located by a environmental variable `DISPLAY` which has a value of the form *hostname*`:`*display-number.screen-number*.

```
[client-machine] $ export DISPLAY=serverHostName.serverDomain:1
```

The X Window System Continued

- The X-server must be explicitly started. Usually done using some sort of script like `startx`.
- The X-server must be told to allow the remote client to use the display. Can be done using `xauth` or `xhost`.

```
$ xhost client-machine.clientDomain
```

- X provides `xlib` as an interface to the X-protocol. `xt` provides very primitive widgets. Other GUI toolkits like Motif, Gnome and KDE provide high-level facilities.

The X Window System Continued

- The use of the display by multiple applications is mediated by a window-manager like `twm`, `fvwm`, `sawfish` and numerous others.
- The use of multiple GUI toolkits and multiple window managers gives users a lot of flexibility. The flip-side is the lack of a common look-and-feel across applications.
- The X-protocol is quite heavy-weight and remote use is best accomplished over a high-speed network like a LAN. Use over a modem or WAN can sometimes be painful.
- There are signs that X will be replaced in the coming years with more local display more suited to modern graphics hardware.

Using VNC for Light-Weight Remote X-Access

- VNC stands for *Virtual Network Computing*.
- It uses algorithms which are optimized to typical uses of GUIs to allow viewing a computer display over a network.
- A VNC server is run on the computer which controls the display which is to to be viewed remotely.
- A VNC client is run on the computer which is used to view the remote display.
- Can be used to interact with a Windows machine on a Unix box or interact with a Unix box via a Windows machine.

Using VNC Continued

- When a VNC server is run on a Unix machine, it starts up a new X server. Hence we can have multiple VNC servers running on the same machine with multiple displays.
- The VNC protocol is quite lightweight compared to the X-protocol. A VNC viewer is extremely lightweight compared to a X-server.
- Allows remote collaboration.

References

- Mark Mitchell, Jeffrey Oldham and Alex Samuel, *Advanced Linux Programming*, Chapter 1, New Riders, 2001. Available at http://www.advancedlinuxprogramming.com/.
- Michael K. Johnson and Erik W. Troan, *Linux Application Development*, Part 2: *Development Tools and Environment*, Addison-Wesley, 1998.
- Brian W. Kernighan and Rob Pike *The UNIX Programming Environment*, Chapters 1 - 5, Prentice-Hall, 1984.
- VNC.
- Online manuals.
- Valgrind.