

# CME 212: Development tools

## Lecture 5

January 23, 2012

## Environment variables

- A set of key value pairs that the operating system “exports” to each process
- Environment variables from a parent process are normally passes to a child process
  - For example: any program you run from a shell is a child process of that shell
  - The shell will pass the environment variables flagged for export to the child process
- Important environment variables
  - HOME: users home directory
  - PATH: where shell looks for commands
  - SHELL: path to shell
- `env` command lists environment variables

\$ env

```
[nwh@nwh-mbair]$ env
```

```
TERM_PROGRAM=iTerm.app
```

```
TERM=xterm-256color
```

```
SHELL=/bin/bash
```

```
HISTSIZE=10000
```

```
TMPDIR=/var/folders/im/imRSVrAxHf8i7LSdcDfMYk+++TI/
```

```
Apple_PubSub_Socket_Render=/tmp/launch-9agPKy/Rende
```

```
USER=nwh
```

```
COMMAND_MODE=unix2003
```

```
SSH_AUTH_SOCK=/tmp/launch-cl5yBT/Listeners
```

```
__CF_USER_TEXT_ENCODING=0x1F5:0:0
```

```
PAGER=less
```

```
LSCOLORS=gxfxcxdxbxegedabagacad
```

```
PATH=../bin:/Users/nwh/bin:/Users/nwh/.cabal/bin:/
```

```
PWD=/Users/nwh
```

```
EDITOR=emacs -nw
```

```
LANG=en_US.UTF-8
```

```
ITERM_PROFILE=Default
```

## Variables in BASH

Looking at a variable:

```
$ echo $PATH  
../../../../bin:/Users/nwh/bin:/Users/nwh/.cabal/bin:/Appli
```

Setting a variable (no spaces around =):

```
$ A=5  
$ echo $A  
5
```

Flagging variable for export:

```
$ export A
```

Set and export at same time:

```
$ export B=5
```

## Some notes on environment variables

- All major operating systems have environment variables
- Programming languages have methods to get and set environment variables
  - C: `getenv()` and `setenv()` in `stdlib.h`
  - Matlab: `getenv()` and `setenv()`
  - Python: `os.getenv()` and `os.putenv()`
- Variables need to be explicitly flagged for export to be passed to a child process
- Variables set or changed in a child process ARE NOT passed to the parent (or anything else on the system)

## Some important environment variables

Table: always exist

variable	function
PATH	tells shell where to look for programs
HOME	user's home directory
PWD	present working directory
SHLVL	shell level

Table: set by user in special circumstances

variable	function
LD_LIBRARY_PATH	where to look for shared libraries
MANPATH	where to look for man pages

## man pages

- Almost all UNIX-like OS are documented in so-called manual pages
- Or man pages, for short
- Looked up using the `man` tool
  - Try `$ man man`
- Path to man pages can be modified by environment variable `MANPATH`. However, today this is often not recommended.
- Source of geek jokes: `$ man kill`

## man pages, sections

section	type
1	Executable programs or shell commands
2	System calls (kernel stuff)
3	Library calls (user-level code)
4	Special files
5	File formats
6	Games!
7	Misc
8	System administration commands (root)
9	Kernel routines (non-standard)

## Try these

```
$ man 3 malloc
```

```
$ man 3 stdio
```

```
$ man top
```

```
$ man bash
```

```
$ man 5 utf8
```

# Man Tools

- By default man pages are shown using the **more** or **less** tool
- `$ man less`
- To trigger search press `/` and then a phrase
  - `/` + `SEE ALSO`
- In **less** hit `h` or `H` to access help system
- Raw format of man pages is device independent
  - You can generate ASCII, PS, DVI
  - Format: `troff/nroff` or `groff` (GNU variants)
  - I would use markdown, then convert to `groff` using `pandoc`

## Man Tools, cont'd

You can search the man database using the commands **whatis** and **apropos**

```
$ whatis malloc
```

```
$ apropos compression
```

```
$ apropos errno
```

Some interesting pages/commands

```
$ man intro (try all man sections)
```

# The compilation process

There are 4 basic steps in the compilation process. It goes in the following order:

**preprocessing** all preprocessor statements are executed, header files are included

**compilation** translates C code into assembly language

**assembly** translates assembly code text to binary format

**linking** links together object files, resolves symbols

Compilers do all of this automatically for you. But, you can break it down. . .

## The compilation process breakdown

```
$ make helloworld
# preprocess
gcc -E -o helloworld.i helloworld.c
# compile
gcc -S -o helloworld.s helloworld.i
# assemble
gcc -c -o helloworld.o helloworld.s
# link
gcc -o helloworld helloworld.o
```

# Preprocessing

helloworld.c

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("hello_world!\n");
5     return 0;
6 }
```

helloworld.i (853 lines)

```
1 # 1 "helloworld.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 1 "helloworld.c"
5 # 1 "/usr/include/stdio.h" 1
6 # 28 "/usr/include/stdio.h" 3
7 # 1 "/usr/include/features.h"
8 # 322 "/usr/include/features."
9 # 1 "/usr/include/bits/predef
10 # 323 "/usr/include/features.
```

...

```
1 extern void flockfile (FILE *)
2 extern int ftrylockfile (FILE
3 extern void funlockfile (FILE
4 # 936 "/usr/include/stdio.h"
5 # 2 "helloworld.c" 2
6
7 int main(void) {
```

## Compilation proper

helloworld.c

```
1 | #include <stdio.h>
2 |
3 | int main(void) {
4 |     printf("hello_world!\n");
5 |     return 0;
6 | }
```

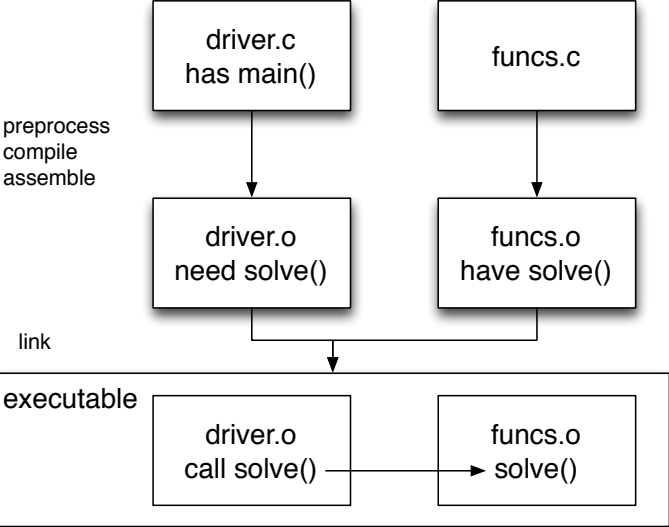
helloworld.s

```
1 | .file "helloworld.c"
2 | .section .rodata
3 | .LC0:
4 | .string "hello_world!"
5 | .text
6 | .globl main
7 | .type main, @function
8 | main:
9 | .LFB0:
10 | .cfi_startproc
11 | pushq %rbp
12 | .cfi_def_cfa_offset 16
13 | movq %rsp, %rbp
14 | .cfi_offset 6, -16
15 | .cfi_def_cfa_register 6
16 | movl $.LC0, %edi
17 | call puts
18 | movl $0, %eax
19 | leave
```

## Assembling and linking

These two steps produce binary files, which cannot be easily displayed. Assembly takes assembly code and translates it to binary machine code or object file. The linking process builds the executable. Say your code is split between **driver.c** and **funcs.c**. The `main` function calls function `solve` which is defined in **funcs.c**. It is the linker's job to match the function call in **driver.o** to the function code in **funcs.o** before outputting the executable.

# Compilation process diagram



## Some gcc options

- Just compile, don't link

```
$ gcc -c
```

- Just preprocess, print out source

```
$ gcc -E
```

- Produce assembly code

```
$ gcc -S
```

- Assemble, produce object file

```
$ gcc -o object_file.o
```

- Search for include files in <ipath> and libraries in <lpath>

```
$ gcc -I<ipath> -L<lpath>
```

- Define macro, like #define

```
$ gcc -D
```

# Documentation on gcc

- `$ man gcc`
- [gcc.gnu.org/onlinedocs/](http://gcc.gnu.org/onlinedocs/)
- Typical manual topics
  - Introduction
  - Programming Languages Supported by GCC
  - Language Standards Supported by GCC
  - GCC Command Options
  - C Implementation-defined behavior
  - Extensions to the C Language Family

## The Link Editor (`ld`)

- The compiler produces machine code and a table of names (symbol table) of variables and procedures
  - This is called an object file (`.o`)
- The link editor (or linker) takes several object files and merges them, by associating the symbols with machine code addresses
- `gcc` acts as a front-end for `ld`. One rarely needs to use `ld` directly.

## Symbol table example

```
1 #include <stdio.h>
2 #include <math.h>
3 int im_a_global;
4 int main(void) {
5     double pi;
6     pi = 4.0*atan(1.0);
7     printf("Pi_is_%lf\n",pi);
8     return 0;
9 }
```

```
$ gcc -c pi.c
```

```
$ nm pi.o
```

```
000000000000000004 C im_a_global
```

```
000000000000000000 T main
```

```
U printf
```

```
$ gcc -o pi pi.o
```

```
$ ./pi
```

```
Pi is 3.141593
```

## Executable files

- The link editor produces an executable file which is compatible with the program loader
- Format: **ELF** (Executable and Linking Format)
- When a program is executed, the program loader copies the sections of the executable file into the virtual address space and execution can begin

# Executable files, example

```
1 #include <stdio.h>
2 #include <math.h>
3 int im_a_global;
4 int main(void) {
5     double pi;
6     pi = 4.0*atan(1.0);
7     printf("Pi_is_%lf\n",pi);
8     return 0;
9 }
```

```
$ gcc pi.c -o pi
$ nm ./pi
0000000000600e50 d __DYNAMIC 0000000000600fe8 d __GLOBAL_OFFSET_TABLE__
0000000000400618 R __IO_stdin_used
                                w __Jv_RegisterClasses
... some lines removed ...
0000000000601020 A __bss_start
0000000000601010 D __data_start
00000000004005d0 t __do_global_ctors_aux
0000000000400460 t __do_global_dtors_aux
0000000000601018 D __dso_handle
                                w __gmon_start__
0000000000600e24 d __init_array_end
0000000000600e24 d __init_array_start
0000000000400530 T __libc_csu_fini
0000000000400540 T __libc_csu_init
                                U __libc_start_main@@GLIBC_2.2.5
0000000000601020 A _edata
0000000000601038 A _end
0000000000400608 T _fini
00000000004003c8 T _init
0000000000400410 T _start
000000000040043c t call_gmon_start
0000000000601020 b completed.7424
0000000000601010 W data_start
0000000000601028 b dtor_idx.7426
00000000004004d0 t frame_dummy
0000000000601030 B im_a_global
00000000004004f4 T main
                                U printf@@GLIBC_2.2.5
```

# Libraries

- It is very common to use a library of object files instead of explicitly stating every object file to use when linking
- Libraries can be either **static** or **dynamic**
- In **static libraries (.a)**, the object files are copied into the executable file
- In **dynamic libraries (.so, .dll, .dylib)**, only the symbols are associated and the linking takes place at runtime by the runtime linker
- Most dynamic libraries can also be shared between many programs to save memory

## Creating static libraries (linux)

- A static library is put together by the `ar` command
- Static libraries are sometimes referred to as an archive (.a)
- When using libraries the “lib” part of the name is removed at link time

```
$ gcc -c triangle.c
```

```
$ ar cru libtriangle.a triangle.o
```

```
$ gcc -c program.c
```

```
$ gcc -o program program.o -L<path> -ltriangle
```

## Creating a Dynamic Library

- Dynamic shared libraries are created by the compiler where each object file should be compiled with `-fPIC`.

```
$ gcc -c -fPIC triangle.c
$ gcc -shared -fPIC -Wl,-soname,libtriangle.so \
>      -o libtriangle.so triangle.o
$ gcc -c program.c
$ gcc -o program program.o -L<PATH> -ltriangle
```

## Checking dependencies (ldd)

```
1 #include <stdio.h>
2 #include <math.h>
3 int im_a_global;
4 int main(void) {
5     double pi;
6     pi = 4.0*atan(1.0);
7     printf("Pi_is_%.1f\n",pi);
8     return 0;
9 }
```

```
$ gcc -o pi pi.c -lm
```

```
$ ldd ./pi
```

```
linux-vdso.so.1 => (0x00007fff215ff000)
libm.so.6 => /lib/libm.so.6 (0x00007fb431ce5000)
libc.so.6 => /lib/libc.so.6 (0x00007fb431962000)
/lib64/ld-linux-x86-64.so.2 (0x00007fb431f89000)
```

# Tracing system calls

```
$ strace ./pi
execve("./pi", [ "./pi" ], [ /* 43 vars */ ]) = 0
brk(0) = 0x21a8000
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) = 0x7f88250c7000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=124711, ...}) = 0
mmap(NULL, 124711, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f88250a8000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/libm.so.6", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\360>\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=534832, ...}) = 0
mmap(NULL, 2629864, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8824c26000
mprotect(0x7f8824ca8000, 2093056, PROT_NONE) = 0
mmap(0x7f8824ea7000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x8192000) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\240\356\1\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1572232, ...}) = 0
mmap(NULL, 3680296, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f88248a3000
mprotect(0x7f8824a1d000, 2093056, PROT_NONE) = 0
mmap(0x7f8824c1c000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x8192000) = 0
mmap(0x7f8824c21000, 18472, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0
close(3) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f88250a7000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f88250a6000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f88250a5000
arch_prctl(ARCH_SET_FS, 0x7f88250a6700) = 0
mprotect(0x7f8824c1c000, 16384, PROT_READ) = 0
mprotect(0x7f8824ea7000, 4096, PROT_READ) = 0
mprotect(0x6000000, 4096, PROT_READ) = 0
mprotect(0x7f88250c9000, 4096, PROT_READ) = 0
```

## Summary: dev tools

- **gcc**: The GNU C compilers
- GNU Binutils
  - **ld**: the linker
  - **as**: the assembler
  - **ar**: tool for archives
  - **nm**: list symbols from an object file
- **ldd**: list shared library dependencies
- **strace**: trace system calls

## make, a maintenance tool

- `make` is usually used to compile and link programs consisting of several files
- A program often uses, or depends on, several other modules or libraries
- A Makefile specifies the dependencies between the program and the modules
- If a file changes, `make` recompiles only those files which depend on the changed file
- For large projects can compile pieces of code in parallel (see `-j` option)

## make concepts

- **target:** a file that depends on other files
- **dependencies:** a list of files that a target depends on
- **rule:** command(s) to build a target. must be indented with a tab

```
prog: module1.o prog.o
    gcc -o prog module1.o prog.o
```

```
prog.o: prog.c module1.o
    gcc -c prog.c
```

```
module1.o: module1.c module1.h
    gcc -c module1.c
```

## make, advanced usage

Make has:

- variables (some have default values)
- automatic variables
- implicit rules
- conditionals
- and more!

## make, some automatic variables

- `$*`: base name of current target, no extension
- `$@`: full name of current target
- `$<`: name of first prerequisite
- `$?`: names of all the prerequisites that are newer than the target

## make, example

```
CFLAGS = -g -Wall
CC = gcc
LIBS = -lm
OBJS = a.o b.o c.o
SRCS = a.c b.c c.c prog1.c prog2.c
HDRS = abc.h

all: prog1 prog2

prog1: prog1.o ${OBJS}
    ${CC} ${CFLAGS} -o $@ prog1.o ${OBJS} ${LIBS}
prog2: prog2.o ${OBJS}
    ${CC} ${CFLAGS} -o $@ prog2.o ${OBJS} ${LIBS}

# make knows what to do for prog1.o and prog2.o
```