cse30 discussion 1

Ibrahim Awwal

June 29, 2015

raspberry pi setup

creating your sd card

- Download the image file, unzip it, and use the appropriate tool for your platform to burn the contents to your SD card
- Linux: dd
- OS X: Apple-Pi Baker
- Windows: Win32 Disk Imager
- More details from the Raspbery Pi website

direct ethernet connection

- Plug an ethernet cable between your Raspberry Pi and your computer
- Set a static IP address on your computer on the 192.168.2.x *subnet*

ssh

- Secure SHell: A command and protocol for secure remote login
- Generally, use a command of the form `ssh username@server`
- `ssh pi@rpi.local` or `ssh pi@192.168.2.2`
- To avoid having to type a password, look into ssh key generation
- Use `ssh-copy-id` or copy your *public* key to `~/.ssh/authorized_keys`
- Store frequently used host configurations in `~/.ssh/config`

connecting to wifi

- Easier if you have a monitor connected via HDMI
- Run the command `wpa_gui` to select a network and authenticate
- Might be able to run this over SSH if you enable X forwarding (`ssh -X` or `ssh -Y`)and have an X server installed (see Xming on Windows)

basic unix commands

- `ls`: List files in directory
- `pwd`: Print (current) working directory
- `cd *arg*`: Change directory
- `cp *source* *dest*`: Copy file from source to dest
- `mv *source* *dest*`: Move file from source to dest
- `scp user@host:/path/to/file .`: Secure copy, copy file over ssh from remote host to local machine
  - the last `.` says put it in the current directory
- editing files: `vim` and `emacs` are some advanced editors, a more simple one is `nano`
  - `emacs` can transparently edit files over SSH (called TRAMP mode)

getting help

- manpages: Built in manuals for most Unix commands
  - eg. `man ssh`
- Google
- Ask on Piazza or in office hours

bonus: dynamic dns and remote ssh

- It would be convenient to stick your RPi somewhere and never have to carry it around
- Problems:
  1. We don't know the IP address
  2. The SSH port may be closed by our router's firewall (not the case on UCSD-PROTECTED)

bonus: dynamic dns and remote ssh

- Solutions:
  1. Dynamic DNS: Our RPi can tell a DNS server online what its IP address is, so that we can have a nice name like `my-rpi.duckdns.org`
     - Some free services: DuckDNS, No-IP
  2. Port forwarding: Open a port on your router for ssh, forward incoming traffic on that port to your Raspberry Pi
     - Eg. forward port 10022 on your router to port 22 on your Raspberry Pi
     - Instructions for doing this are router specific
     - Need to add an argument to ssh to use the correct port
     - If you don't control your router, you can set up a reverse SSH tunnel (complicated and you need some other publicly accessible ssh server)

c programming

```c
#include <stdio.h>

int main(int argc, char** argv)
{
    if(argc > 1){
        printf("Hello, %sn", argv[1]);
    }
    else{
        printf("Hello worldn");
    }
    return 0;
}

```

gcc

- Simple example: `gcc hello.c -o hello`
- Some useful options:
  - `-g`: Enable debugging symbols
  - `-Wall`: Enable warnings (can often catch basic errors)
  - `-O0, -O1, -O2`: Different levels of optimization
  - See the manpage for more

compilation process

- Each source file is translated to an object file by the compiler
- The *linker* finds references to libraries or other shared object files and replaces abstract references with actual addresses (for instance, standard library functions like from `<stdio.h>`) and produces the executable

other useful utilities

- `objdump`: Lets you inspect an object file (including executables)
  - `objdump -D` will let you disassemble your binary and look at the assembly code the compiler produced
- `readelf`: Gives information about an ELF format executable (default for Linux)
  - Eg. `readelf -H *executable*` tells you the architecture the executable is compiled for

a note on architectures

- CPUs implement different Instruction Set Architectures (ISAs)
  - **ISA**: The instruction format your CPU understands
- Your desktop/laptop/server is most likely x86 (Intel compatible)
- Raspberry Pi, your phone, other embedded systems are usually ARM
- Other ISAs include POWER, SPARC, MIPS, Itanium, etc
- **Binaries compiled for one ISA will *not* run on another ISA**
- In particular, binaries compiled for your RPi won't run on PC
- Hence, you must compile your ARM code on your RPi or use a **cross compiler**

make

- System for describing how to build a project
- Input file is (usually) named `Makefile`
- Don't have to type all those gcc arguments every time!
- Has simple dependency management
- Tries to recompile only files that have changed
- Can become unwieldy for more complex projects (hence, more complicated build tools such as CMake, Automake, etc)
- **Whitespace sensitive, so be careful!**

make

- A Makefile defines one or more *targets* and how to build them
- Basic syntax:

```
target: dependencies
    command to build target
```

- **Common gotcha**: Commands to build a target must be preceded by a **TAB** character
- Can use multiple commands to build a target
- Each command must be preceded by a tab
- The default target when you run `make` with no arguments is `all`

make

- You can define variables to be used in commands

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

all:
    $(CC) $(CFLAGS) hello.c -o $(OUTFILE)
```

make

- Targets can depend on previous targets

```
all: hello.o util.o
    gcc hello.o util.o hello.c -o hello
```

- hello.o is produced from hello.c and util.o is produced from util.c
- gcc combines these two files to make the executable
- This example takes advantage of the fact that Make knows how to compile *object files* (.o) from C files (.c) implicitly
- You can also define your own rules for automatically transforming inputs of one type to outputs of another

example makefiles

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

all: hello

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

example makefiles

```
all: hello
```

- How does this work?
- `make` implicitly knows how to compile c files
- You can even run `make bob` and Make will know to run `gcc bob.c -o bob` if bob.c exists

generic makefile

- Taken from `http://mrbook.org/blog/tutorials/make/`

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LDFLAGS) $(OBJECTS) -o $@
```

gdb

gdb

- `gdb` is the GNU debugger
- Allows you to debug your programs in a more sophisticated way than inserting print statements into your code
- `gdb *executable*` starts gdb and loads your executable (eg. `gdb hello`)
- Main features: breakpoints, step by step execution, inspect variables, handle errors
- Commands have intuitive names, and also can be shortened (eg. `b` instead of `break`)

gdb commands

- Once you've loaded a file, `run` or `r` will start execution
- Add breakpoints by using `break linenumber`
  - Then, when the program hits that line, it will pause
  - For multi-file projects, `break filename:linenumber`
  - Can also add breakpoint on a function to stop at the beginning of that function
- `info breakpoints` lists breakpoints and their numbers
- `delete` removes all breakpoints, `delete *number*` deletes numbered breakpoints

gdb commands

- `continue` resumes execution after a breakpoint
- `step` runs one line of code and then stops
- `list *linenumber*` prints the code around the line number or at the start of a function
- `print *expression*` prints the value of an expression
  - Can print variables, arrays, memory addresses, 2+2, etc.

turning in homework

turnin

- (optional) `scp` your files onto `ieng6.ucsd.edu` if you worked elsewhere (necessary for PA2 and PA3)
  - For PA1, it's easiest to just do it from ieng6
- `ssh` into `ieng6.ucsd.edu`
- Create a tar file containing all your homework files
  - `tar czf hw1.tar.gz hw1/`
- Submit using the turnin command: `turnin hw1.tar.gz`
- Submitting again will override the previous submission