CSE451 Section 2

10/4/07

Aziel Epilepsia

Goals of today

- Go over some questions from the homework
- Review GDB
  - What's available and how to use it to debug your projects

Start off

- C Review:
  - Planning on a session Friday afternoon/night.
  - Focus on basics, for those who feel lost
  - Email me at aziel@u if you are interested.

- Questions on anything?
- Feedback from project 0 and homework 1?
  - Useful? Busy work?
- How confident are you in what you learned from project 0?

Homework 1

- Return HW1
- Mean is 24.8/27.

- Each problem is graded on 0 – 3 scale.
- Worth 10% of the total course grade

Homework Review

- Some comments on the questions:
- 2.7 – Why is the command interpreter separate from the kernel?
  - Allows CI to be changed without changing the kernel

- Otherwise, students had no problem with homework

Requests from graders

- Write section#, HW#, and date on the homework
- Write legibly or type it up
  - If the grader can't read your handwriting easily, he/she will mark it wrong.
- Hand it in on time
  - Due beginning of lecture on the day stated

GDB

- GNU Debugger
- Online manual available at:
  - http://sourceware.org/gdb
- Material discussed in this section will be a condensed version of the online documentation

Goals of the debugger

- Help find sources of error in the program
- Debugger provides methods to:
  - Control program execution
  - View system state during program execution

Debugging programs in GDB

- Requirements:
  - Compiled binaries
    - Multiple object files rolled into an executable
  - Example makefile included in Backup Section

- Execution:
  - From command line: `gdb`

- Prompt should now be `<gdb>`
  - To load program, type
    `file [programName]`

Startup

- Execute GDB
  - From command line: `gdb`
  - To run gdb and load program from command line: `gdb [filename]`
- Once GDB is running, prompt will become `<gdb>`
- From the `<gdb>` prompt we can do the following:
  - View the current source being stepped through
  - Set breakpoints in program
  - Run program and step through code
  - Examine data

Viewing source

- Source can be viewed in GDB using the list command (shorthand is l)
  - `l` *`linenumber`*
    - Print lines centered around the line number in the current source file
  - `l` *`function`*
    - Print lines centered around the function
  - `l -` (l [minus])
    - Print lines just before the lines last printed
  - `l`
    - Print lines around the last instruction executed
    - OR, print more lines after the last lines printed

Setting breakpoints

- We can set breakpoints in two useful ways:
  - By function name
    - `break foo`
    - `break queue_remove`
    - `break queue_append`
  - By file name and line number
    - `break main.c:10`
    - `break queue.c:127`
- Shorthand for break is b
    - `b foo`
    - `b main.c:10`

Running into breakpoints

- After breakpoints have been set, we can type `run` to execute the program until the first breakpoint
- Once the breakpoint has been reached, you will see text such as
  - `Breakpoint 1, queue_remove (q=0x804a008, olde=0x804a008) at queue.c:62`
    `62 assert(q != NULL);`
- This provides the following information:
  - Which breakpoint was reached
  - Which function execution halted in
  - The source file and line number
  - The line of code to be executed next

Stepping through code

- After the breakpoint has been reached, we can step through code using the following commands
  - `step` or `s`
    - step through code until a new source line is reached (will step into function calls, provided source was compiled with –g flag)
  - `next` or `n`
    - step through code until a new source line in the current stack frame is reached (all function calls that occur inside that line are executed without stopping)

Examining data (1)

- Data can be examined during program execution based on scope rules
- You can use the variable names in the function currently being executed

i.e.

```
boolean_t
queue_remove(queue_t q, queue_element_t *e)
{
  queue_link_t oldHead;
….
*e = q->head->e;

  oldHead = q->head;
  q->head = q->head->next;
  return TRUE;
}
```

Examining data (2)

- Works for pointers. Example results:
  - `print q`
    - `$1 = (queue_t) 0x804a008`
  - `print *q`
    - `$2 = {head = 0x804a018}`
  - `print q->head`
    - `$3 = (queue_link_t) 0x804a018`
  - `print *q->head`
    - `$4 = {e = 0x8049aa8, next = 0x804a028}`
- When printing pointers, you can print the address, or dereference the members of the pointer.
- Shorthand for print is p
  - `p q`
  - `p *q->head`

Display formats (1)

- You can change the display format of the data
  - `p*q->head`
    - `$4 = {e = 0x8049aa8, next = 0x804a028}`
  - `p /d *q->head`
    - `$6 = {e = 134519464, next = 134520872}`
  - `p /t *q->head`
    - `$7 = {e = 10000000010010011010101000, next = 10000000010010100000101000}`
  - `p /a *q->head`
    - `$8 = {e = 0x8049aa8 <x>, next = 0x804a028}`
  - `p /c *q->head`
    - `$9 = {e = -88 '¨', next = 40 '('}`

Display formats (2)

- Display formats
  - /x, regard data as an integer, print integer as hexadecimal
  - /d, signed decimal
  - /u, unsigned decimal
  - /t, binary
  - /a, address
  - /c, regard data as integer, and print as a char

Examining memory

- Data stored in memory is accessible via the `x` command ('x' for examine)
  - `x /nfu addr`
- Memory reads can be formatted by specifying
  - `n`, the repeat count (how many units of memory to display)
  - `f`, the display format (discussed earlier)
  - `u`, the unit size
    - `b`, bytes
    - `h`, halfwords (2 bytes)
    - `w`, words (4 bytes)
    - `g`, giant words (8 bytes)

- Example:
  - `x /1ub 0x0000ffff`
    - Read the memory at 0x0000ffff, and display one byte as an unsigned integer
  - `x /2tw 0x0000ffff`
    - Read the memory at 0x0000ffff, and display 2 words as an unsigned integer

Summary of GDB slides

- Discussed commands available and necessary for basic debugging:
  - Viewing source
  - Breakpoint setting
  - Execution control
  - Examining data

Project 1 is up

- Start looking at it now!
- Write a shell

Resources

- GDB
  - http://sourceware.org/gdb/
- Make and makefile
  - http://users.actcom.co.il/~choo/lupg/tutorials/writing-makefiles/writing-makefiles.html

BACKUP

Following this slide are backup slides

Creating a Makefile

- Discuss creating a make file for local testing

Makefile from Proj0 (1)

```
CC= gcc
CFLAGS= -Wall -O  -g
SRCS= main.c queue.c
OBJS= main.o queue.o
PROGRAM= queuetest
MKDEP= gccmakedep

${PROGRAM}:     ${OBJS}
   ${CC} ${CFLAGS} ${OBJS} -o ${PROGRAM}

%.o : %.c
   $(CC) $(CFLAGS) -c $<

clean:
   rm -f ${OBJS} ${PROGRAM}

depend:
   ${MKDEP} ${CFLAGS} ${SRCS}
```

Makefile from Proj0 (2)

- CC: compiler definition (in this case gcc)
- CFLAGS: flags for the C compiler.
  - Wall: display all warnings
  - O: level 1 code optimization,
    - Higher optimization levels reduces output code size while increasing compile time and making code harder or impossible to debug
  - g: include debugging information in code
- SRCS: source files
- OBJS: output object files
- PROGRAM: the name of the program being compiled
- MKDEP: dependency list creator (in this case, gccmakedep – gcc with –M flag)

- $PROGRAM… : lists the contents of the program and the command for compiling it
- %.o -> %.c … : lists the dependencies of the object files on which source files, and the command for recompiling
  - % is a wildcard character, $< refers to a list of dependencies matching the rule (in this case, the target filename)
  - i.e. %.o must be recompiled whenever %.c is modified, using "gcc –Wall –O -g –c" [list of files matching rule]
- clean: … : defines the action to take when make –clean is called
  - remove all object and program files
- depend: … : defines the action to take to populate dependencies. Output of this is stored in Makefile unless otherwise specified.