Home

Homework 1: Makefiles, GDB and simple UNIX programs

This assignment will make you more familiar with how to build simple Unix programs with Makefiles, and debug them with GDB. You can do this assignment on any operating system that supports the Unix API (CADE machines, your laptop that runs Linux or Linux VM, and even MacOS, etc.). **You don't need to set up xv6 for this assignment**. Submit your programs and the shell through Gradescope (see instructions at the bottom of this page).

NOTE: YOU CANNOT PUBLICLY RELEASE SOLUTIONS TO THIS HOMEWORK. It's ok to show your work to your future employer as a private github/gitlab repo, however any public release is prohibited. For **Mac / OSX** users, the support of 32 bit applications is deprecated in the latest version. So if you already updated your system to macOS Catalina or have updated XCode then we recommend that you do the homework on CADE machines.

Download the main.c, and look it over. This is a skeleton for a simple UNIX program.

To compile `main.c`, you need a C compiler, such as gcc. On CADE machines, you can compile the skeleton with the following command:

```
$ gcc main.c
```

This will produce an `a.out` file, which you can run:

```
$ ./a.out
```

Alternatively you can pass an additional option to gcc to give a more meaningful name to the compiled binary, like

```
$ gcc main.c -o hello
```

Here gcc will compile your program as `hello`. In the rest of this part of the assignment you will explore how to automate program development with Makefiles, learn how debug your code with GDB, and disassemble the program to verify your understanding of assembly language.

Part 1: Simple Makefiles

This part of the homework is adapted from Makefiles. A tutorial by example and Makefile Tutorial (more advanced stuff). It aims to introduce you to basics of Makefiles and the `make` tool that provides a way to compile complex software projects like xv6 and Linux kernel.

If you want to run or update a task when certain files are updated, the make utility can come in handy. The make utility requires a file, Makefile (or makefile), which defines set of tasks to be executed. You may have used make to compile a program from source code. Most open source projects use make to compile a final executable binary, which can then be installed using make install.

We'll explore make and Makefile using basic and advanced examples. Before you start, ensure that make is installed in your system. Note: we will create three different makefiles (Makefile1, Makefile2, and Makefile3) in this part.

Let's start by printing the classic "Hello World" on the terminal. Create a empty directory myproject containing a file Makefile with this content:

```
say_hello:
        echo "Hello World"
```

Now run the file by typing make inside the directory myproject. The output will be:

```
$ make
echo "Hello World"
Hello World
```

In the example above, say_hello behaves like a function name, as in any programming language. This is called the target. The prerequisites or dependencies follow the target. For the sake of simplicity, we have not defined any prerequisites in this example. The command echo "Hello World" is called the recipe. The recipe uses prerequisites to make a target. The target, prerequisites, and recipes together make a rule.

To summarize, below is the syntax of a typical rule:

```
target: prerequisites
<TAB> recipe
```

As an example, a target might be a binary file that depends on prerequisites (source files). On the other hand, a prerequisite can also be a target that depends on other dependencies:

```
final_target: sub_target final_target.c
        Recipe_to_create_final_target

sub_target: sub_target.c
        Recipe_to_create_sub_target
```

It is not necessary for the target to be a file; it could be just a name for the recipe, as in our example. We call these "phony targets."

Going back to the example above, when make was executed, the entire command echo "Hello World" was displayed, followed by actual command output. We often don't want that. To suppress echoing the actual command, we need to start echo with `@`:

```
say_hello:
        @echo "Hello World"
```

Now try to run make again. The output should display only this:

```
$ make
Hello World
```

Let's add a few more phony targets: generate and clean to the Makefile:

```
say_hello:
        @echo "Hello World"

generate:
        @echo "Creating empty text files..."
        touch file-{1..10}.txt

clean:
        @echo "Cleaning up..."
        rm *.txt
```

If we try to run make after the changes, only the target say_hello will be executed. That's because only the first target in the makefile is the default target. Often called the default goal, this is the reason you will see all as the first target in most projects. It is the responsibility of all to call other targets. We can override this behavior using a special phony target called `.DEFAULT_GOAL`.

Let's include that at the beginning of our makefile:

```
.DEFAULT_GOAL := generate
```

This will run the target generate as the default:

```
$ make
Creating empty text files...
touch file-{1..10}.txt
```

As the name suggests, the phony target `.DEFAULT_GOAL` can run only one target at a time. This is why most makefiles include all as a target that can call as many targets as needed.

Let's include the phony target all and remove `.DEFAULT_GOAL`:

```
all: say_hello generate

say_hello:
        @echo "Hello World"

generate:
        @echo "Creating empty text files..."
        touch file-{1..10}.txt

clean:
        @echo "Cleaning up..."
        rm *.txt
```

Before running make, let's include another special phony target, `.PHONY`, where we define all the targets that are not files. make will run its recipe regardless of whether a file with that name exists or what its last modification time is. Here is the complete makefile:

```
.PHONY: all say_hello generate clean

all: say_hello generate

say_hello:
        @echo "Hello World"

generate:
        @echo "Creating empty text files..."
        touch file-{1..10}.txt

clean:
        @echo "Cleaning up..."
        rm *.txt
```

The make should call `say_hello` and `generate`

```
$ make
Hello World
Creating empty text files...
touch file-{1..10}.txt
```

It is a good practice not to call clean in all or put it as the first target. clean should be called manually when cleaning is needed as a first argument to make:

```
$ make clean
Cleaning up...
rm *.txt
```

Now that you have an idea of how a basic makefile works and how to write a simple makefile, let's look at some more advanced examples.

Something more real

Now lets try to create a simple Makefile that we can use to compile our programs:

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

Now you can run:

```
make
```

Instead of running GCC manually, the Makefile lets you compile `main.c` into the `hello` program.

In our example the only target in the Makefile is `all`. The `make` utility will try to resolve this target if no other targets are specified.
We also see that there are no dependencies for target `all`, so `make` safely executes the system commands specified.

Finally, make compiles the program according to the command line we gave it.

When you submit your work, rename this makefile into `Makefile1`. You can always pass a custom name to make with the `-f`

```
make -f Makefile1
```

Using dependencies

Sometimes it is useful to use different targets. It makes the Makefile more modular and allows assembling a complex project from multiple pieces.
Here is an example (in your submission name this makefile as `Makefile2`:

```
all: hello

hello: main.o
	gcc main.o -o hello

main.o: main.c
	gcc -c main.c

clean:
	rm *.o hello
```

Now we see that the target `all` has only one dependency (i.e., `hello`), but no system commands. In order for `make` to execute correctly, it has to meet all the dependencies of the called target.

Each of the dependencies are searched through all the targets available and executed if found.

In this example we see the target called `clean`. It is useful to have such target if you want to have a fast way to get rid of all the object files and executables.

```
make clean
```

Using variables and comments

You can also use variables when writing Makefiles. It comes in handy in situations where you want to change the compiler, or compiler options (create this makefile and submit it as `Makefile3`).

```
# This is how you write comments
# Use gcc as a compiler
CC=gcc
# CFLAGS will be the options we'll pass to the compiler
CFLAGS=-Wall

all: hello

hello: main.o
```

```
	$(CC) $(CFLAGS) main.o -o hello

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

clean:
	rm *.o hello
```

Variables can be very useful. To use them, just assign a value to a variable before you start writing your targets. After that, you can just use them with the dereference operator $(VAR).

If you want to know more...

With this brief introduction to Makefiles, you can create some very sophisticated mechanisms for compiling your projects. However, this is just the tip of the iceberg. More documentation is available here: Make documentation.

Here is an example of a more automated Makefile that you might use in one of your projects (not required for this homework).

```
CC=gcc
# Compiler flags
CFLAGS=-Wall
# Linker flags
LDFLAGS=
# You can add multiple source files here separated with spaces
SOURCES=main.c

# Replace .c with .o creating a list of object files
OBJECTS=$(SOURCES:.c=.o)

# Name executable
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

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

# Rule that tells make how to make an object file out of a .c file
.c.o:
	$(CC) -c $(CFLAGS) $< -o $@

clean:
	rm *.o $(EXECUTABLE)
```

What to submit

Three makefiles (Makefile1, Makefile2, and Makefile3) you created to compile your hello example

Part 2: Debugging programs with GDB

On UNIX systems the main debugger is GDB (GNU debugger). To be able to comfortably debug your code compile it with the -g option which will instruct the compiler to generate debug symbols (variable names, source lines, etc.) for the program. For example, change the command you ran to build hello to

```
$ gcc main.c -o hello -Wall -g -m32 -fno-pic
```

This will compile your hello program with debugging symbols (`-g` flag), as a 32bit x86 executable (`-m32` flag), and for simplicity avoid generating position independent code (`-fno-pic` flag). Then you can start you program under control of gdb:

```
$ gdb hello
```

This starts gdb ready to execute your `hello` program. To get it running type the `run` command in the GDB command prompt (or just `r` for short) :

```
(gdb) run
```

Now the program runs and finished printing "Hello world".

GDB is a feature-rich debugger, and it will take you some time to learn all the features. Here are a few starting points: GDB tutorial, GDB intro and GDB Cheat Sheet.

Probably, the best resource for this homework is Operating Systems from 0 to 1. Chapter 6. Runtime Inspection and Debug (it is strongly recommended to read this chapter).

At a high level you need only two main things: 1) breakpoints and 2) ability to examine data. Breakpoints can be set with the "b" command inside gdb.

Breakpoints and single stepping

Just to make debugging a bit more realistic lets add another function to our simple program. Lets change it to compute a sum of numbers from 0 to n. You can do this by implementing the sum function, and calling it from main:

```
unsigned long sum(int n) {
    int i;
    unsigned long sum = 0;
    for (i = 0; i < n; i++) {
        sum = sum + i;
    }

    return sum;
}

int main(void) {
    unsigned long s;

    s = sum(100);
    printf("Hello world, the sum:%ld\n", s);
    return 0;
```

```
}
```

Running the program on its own is not that useful. Lets try setting a breakpoint on the "main" function to examine what the program is actually doing. Type `break main` in the GDB command prompt (or `b` for short) and then run the program with `r`.

```
(gdb) break main
Breakpoint 1 at 0x56b: file main.c, line 26.
(gdb) r
Starting program: ...

Breakpoint 1, main () at main.c:26
26          s = sum(100);
(gdb)
```

The debugger stopped at the beginning of the `main` function (line 26 of `main.c`). You can examine the source code of the program by typing `list` (or `l` for short).

```
(gdb) list
21
22      int main(void) {
23
24          unsigned long s;
25
26          s = sum(100);
27          printf("Hello world, the sum:%ld\n", s);
28          return 0;
29      }
30
```

Now you can execute the program line by line by typing `next` (or `n` for short), which executes the next line. By default typing `next` will skip over functions. Type `step` (or `s` for short) to step into a function. Try stepping into the `sum` function by running `step`.

```
(gdb) s
sum (n=100) at main.c:13
13          unsigned long sum = 0;
```

We are now inside the `sum` function. Type `l` to list the source code, and then type `n` repeatedly to execute the function line by line. Note that we can also type `n` once, and then simply hit `Enter` asking GDB to execute the last command for us.

```
(gdb) l
8       #include
9       #include
10
11      unsigned long sum(int n) {
12          int i;
13          unsigned long sum = 0;
```

```
14
15          for (i = 0; i < n; i++) {
16              sum = sum + i;
17          }
(gdb) n
15          for (i = 0; i < n; i++) {
(gdb)
16              sum = sum + i;
(gdb)
15          for (i = 0; i < n; i++) {
(gdb)
16              sum = sum + i;
```

TUI: Graphical User Interface

The second most useful feature is the TUI mode that turns GDB into a real modern debugger. Here is a useful discussion about TUI.

You can switch into TUI by pressing Ctrl-X and then "1", or start gdb in TUI mode right away

```
$ gdb hello -tui
```

You can also type `tui enable` in the gdb command prompt (this command doesn't work with some terminal emulators like Kitty, try another one).

Start the program from the begginging and single step it with `n` and `s`. The source code of the program will be scrolling in the TUI window in the top part of the screen.

Examining data

You can print values of variables with "print", e.g., print the values of `i` and `sum`

```
(gdb) p i
(gdb) p sum
```

Conditional breakpoints

While debugging programs it's often useful to see what the program is doing right before it crashes. One way to do this is to step through, one at a time, every statement of the program, until we get to the point of execution where we want to examine the state of the program. This works, but sometimes you may want to just run until you reach a particular section of code based on a condition, and stop execution at that point so you can examine data at that point of execution.

For instance, in the `sum` function, you might want to examine the state of the program when the index `i` is equal to 50. You can single step until `i` increments and reaches the value 50, but this would be very tedious.

GDB allows you to set conditional breakpoints. To set a conditional breakpoint to break inside the loop of the `sum` function when the index `i` is equal to 50, we do the following: first, `list` the source code to get the exact source lines; second, set a breakpoint inside the `main.c` file at line 16 with `break main.c:16`; third, to make

the breakpoint trigger only when `i` is equal to 50 (and not trigger for every iteration of the loop) we type `condition 2 i==50`.

```
(gdb) l
11      unsigned long sum(int n) {
12          int i;
13          unsigned long sum = 0;
14
15          for (i = 0; i < n; i++) {
16              sum = sum + i;
17          }
18
19          return sum;
20      }
(gdb) break main.c:16
Breakpoint 2 at 0x56555543: file main.c, line 16.
(gdb) condition 2 i==50
```

Note that the `2` in the condition refers to the breakpoint number we were notified about when we initially set the breakpoint. We can also achieve the above in one command statement with the following:

```
(gdb) break main.c:16 if i==50
```

We now continue execution of the program with the `continue` or `c` command.

```
(gdb) c
Continuing.

Breakpoint 2, sum (n=100) at main.c:16
16              sum = sum + i;
```

When the breakpoint is hit we can check if the value of `i` is really 50:

```
(gdb) p i
$1 = 50
(gdb)
```

Exploring crashes

Now, lets take a look at how you can use GDB to debug your crashing programs. First, lets generate a program that crashes. Add a global variable `a[32]` to your program (it's an array of 32 integers), and then add a function that makes an out of bounds array access.

```
int a[32]; // the global array

unsigned long crash_array(int n) {
    int i;
    unsigned long sum = 0;

    for (i = 0; i < n; i++) {
        sum = sum + a[i];
```

```
    }

    return sum;
}
```

If you invoke this function with `n` larger than 32 it will crash. Note that you might get lucky and it will not crash: not all out of bounds accesses cause a crash in C programs. To be sure, lets invoke it with n equal to 100,000

```
s = crash_array(100000);
printf("crash array sum:%ld\n", s);
```

If you append the above lines to your main.c, compile, and run it, it will crash.

```
$ ./hello
Hello world, the sum:4950
Segmentation fault (core dumped)
$
```

Now, to understand the crash you can run it under gdb:

```
(gdb) r
Starting program: /home/aburtsev/doc/OS_Stuff/Flux/git/personal/classes/os-class/cs5460/hw/hello
Hello world, the sum:4950

Program received signal SIGSEGV, Segmentation fault.
0x56555566 in crash_array (n=100000) at main.c:18
18              sum = sum + a[i];
```

You can use the `backtrace` (`bt`) command to look at the backtrace (a chain of function invocations leading to the crash):

```
(gdb) bt
#0  0x56555566 in crash_array (n=100000) at main.c:18
#1  0x565555ec in main () at main.c:45
```

Here, the GDB tells you that `crash_array` got a segmentation fault at line 18 in `main.c`. You see that there are two stack frames available (0 for `main` and 1 for `crash_array`). You can use the `frame` (`f`) command to choose any of the frames and inspect it. For example, lets choose frame `#0` and list the crashing code with the `list` command

```
(gdb) f 0
#0  0x56555566 in crash_array (n=100000) at main.c:18
18              sum = sum + a[i];
(gdb) l
13      unsigned long crash_array(int n) {
14          int i;
15          unsigned long sum = 0;
16
17          for (i = 0; i < n; i++) {
18              sum = sum + a[i];
19          }
20
21          return sum;
22      }
```

We know that line 18 is the crashing line. We can print the values of the local variable `i`

```
(gdb) p i
$1 = 35824
```

It is equal to 35824(this number might be different on your machine). This should give you enough information for why you crashed.

Now modify the `crash_array` function to prevent the program from crashing.

What to submit

The modified main.c program.

Submit your work

Submit your solution through Gradescope [Gradescope cs5460/6460 Operating Systems](). Place each part of the assignment into folders with name part1, part2 then pack them into a zip archive and submit it. Please name the C files `main.c`, `Makefile1`, `Makefile2`, `Makefile3` for part1, and `main.c`, `Makefile` for part 2. You can resubmit as many times as you wish. If you have any problems with the structure the autograder will tell you. The structure of the zip file should be the following:

```
/
  - /part1
    - main.c
    - Makefile1
    - Makefile2
    - Makefile3
  - /part2
    - main.c
    - Makfile
```

Updated: Jan, 2023