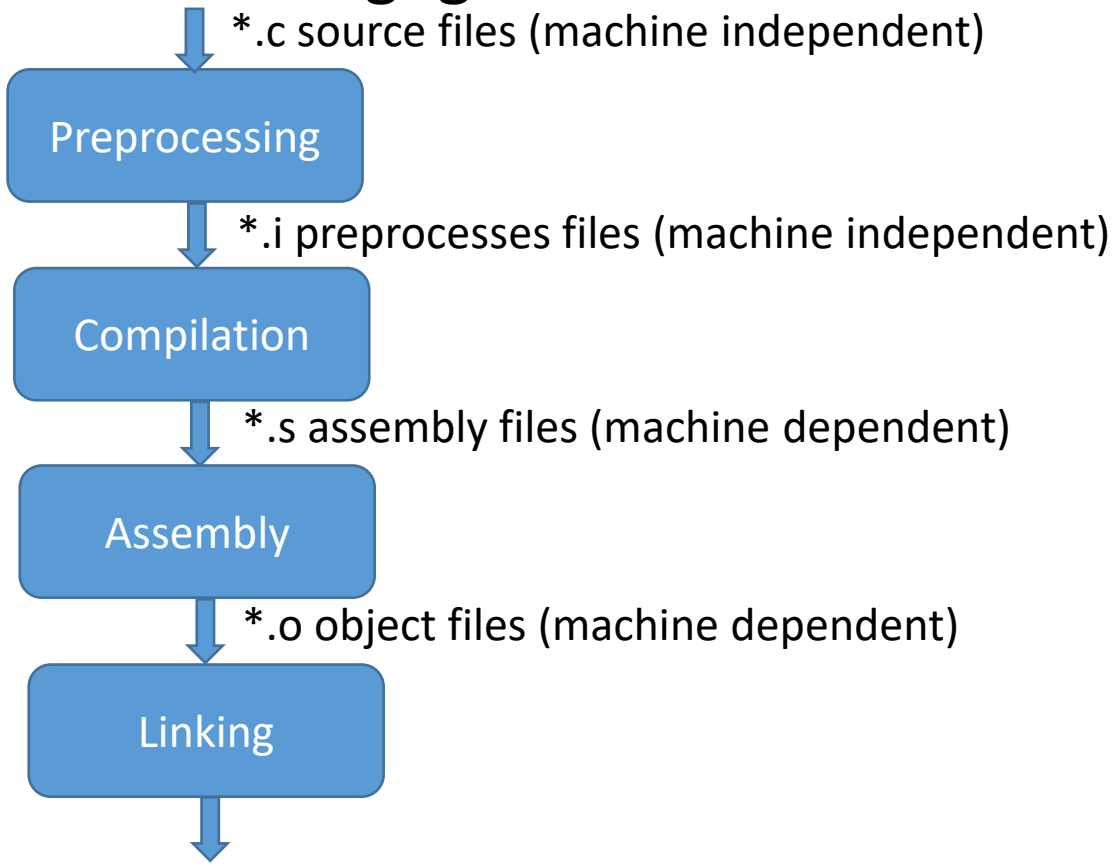
Systems Software/Programming

Preprocessing, Compilation, Assembly, Linking, Loading

GCC GNU compiler for Unix (and Unix like OS)

- Documentation available at: https://gcc.gnu.org/onlinedocs/
- Steps followed by gcc when executing: gcc file.c
  - Preprocessing
  - Compilation
  - Assembly
  - Linking



Preprocessing

- cpp is preprocessor for gcc which is invoked automatically by gcc

  cpp helloworld.c > helloworld.i

The same can be done with –E switch in gcc

  gcc –E helloworld.c > helloworld.i

- Handles preprocessor directives such as #define, #include, #ifdef, #endif etc
- E.g. #include <stdio.h> tells preprocessor to read the content of stdio.h and insert into program text. Generally creates new file with .i extension which is not saved unless –save-temps option is used with gcc

- C_revision\helloworld.c, C_revision\helloworld.h, C_revision\helloworld.i

Compilation

- Compiler cc translates .i file which contains preprocessed program text to .s file an assembly code

```
cc -S helloworld.i
```

OR

```
gcc -S helloworld.i
```

- -S option with gcc is used to create assembly code
- What is assembly code?

C_revision\helloworld.s

Assembly

- Assembler (as) translates .s file containing assembly code into object file (.o) that has code in machine language

as helloworld.s -o helloworld.o

Note: Generally we don't follow the process of preprocessing, compilation and assembly separately. Instead we can use following command directly

gcc -c helloworld.c –o helloworld.o

-c option will perform all 3 steps (preprocessing, compilation and assembly) and create object file directly

Object Files are binary files so cannot be viewed without binary editor

For more readable assembly code you can use

gcc –g –c helloworld.c → -g option puts debugging information in object file

objdump -d -M intel -S helloworld.o > helloworld.assembly

C_revision\helloworld.assembly

Linking: ld linker program used within gcc

- C program uses many function declared as user defined (fib, fact etc) or system defined library functions (scanf, printf etc)
- Linking is the process of linking multiple object files to generate a single executable

e.g. progmain.c contains main() function and uses user defined function fib() from fib.c

After first 3 steps 2 object files are created: progmain.o, fib.o

Linking will take the reference of fib() function from fib.o object file and update in progmain.o so that the fib function can be called

gcc helloword.o –o helloworld

Will generate helloworld as executable (default name of executable in unix/linux is a.out when –o option is not used)

C_revision\helloworld.out.assembly

Use of multiple C files

- file1_main.c → only compile using –c flag
  - Compile with “gcc –c file1_main.c” will create file1_main.o object file
- file2_other.c → only compile using –c flag
  - Compile with “gcc –c file2_other.c” will create file2_other.o object file
- myheader.h – user defined header file with function declaration and external variables
- Finally run linker to build the executable
  - “gcc file1_main.o file2_other.o –o file1_main.out”

Linking Static or Shared(Dynamic) Library

- Shared library (unix extension .so, windows extension .dll)
- Static library (unix extension .a, windows extension .lib)

Static Library	Shared Library
Referenced function code is part of program code so loaded with program code	Referenced function code is loaded only on needed basis (when function is called)
Object code of referenced function is embedded in program object code from static library	Object code of referenced function is NOT embedded in program object code from shared library
Object code of referenced function is linked at compile time (static linking)	Object code of referenced function is linked at runtime (dynamic linking)
Increases the size of executable	Increase the runtime of running program

Create Static Library

- Create Library File using User Defined Functions

Create 2 source files add.c and sub.c which has functions “int add(int, int)” and “int sub(int, int)” for performing addition and subtraction.

Step1: Compile sources into object files (use –c option of gcc):

gcc –c add.c and gcc –c sub.c (or gcc –c add.c sub.c)

This will create add.o and sub.o object files

Step2: Create static library using object files:

ar rs libmymath.a add.o sub.o

Create Static Library Continue…

- Use Static Library functions add() and sub() in Application Program

Step3: Include functions add() and sub() by declaring them in a user-defined header file or declaring them in application program source.

Step4: Create application program source file to call library function and compile

gcc –c call_lib_function.c

Step5: Now create an executable using library function

gcc –o call_lib_func.out call_lib_func.o libmymath.a OR

gcc –o call_lib_func.out –L . call_lib_func.o –lmymath

Create Shared(Dynamic) Library

Consider you have 2 source files add.c and sub.c which has functions for performing addition and subtraction.

Step1: Compile using –fPIC option for “Position Independent Code”

gcc –fPIC –c add.c sub.c

Create new object file with “Position Independent Code”

Step2: Create shared library:

gcc –shared -o libmymath.so add.o sub.o

This will create shared library

Step3: Create a proprietary header file e.g. myheader.h to declare functions add() and sub()

Step4: Create c file to call library function and compile

gcc –c call_lib_function.c

Create Shared Library Cont…

Step5: Now you can create an executable using shared library

gcc –o call_lib_func call_lib_func.o libmymath.so OR

gcc –o call_lib_func –L . Call_lib_func.o –lmymath

Compare Disassembly

C_revision\call_lib_func_static.assembly

```
689:        e8 80 00 00 00          call   70e <add>

000000000000070e <add>:

70e:        55                      push   rbp

70f:        48 89 e5                mov    rbp,rsp

712:        f2 0f 11 45 f8          movsd  QWORD PTR [rbp-0x8],xmm0

717:        f2 0f 11 4d f0          movsd  QWORD PTR [rbp-0x10],xmm1

71c:        f2 0f 10 45 f8          movsd  xmm0,QWORD PTR [rbp-0x8]

721:        f2 0f 58 45 f0          addsd  xmm0,QWORD PTR [rbp-0x10]

726:        5d                      pop    rbp

727:        c3                      ret
```

C_revision\call_lib_func_dynamic.assembly

```
7d9:        e8 72 fe ff ff          call   650 <add@plt>

0000000000000650 <add@plt>:

650:        ff 25 6a 09 20 00       jmp    QWORD PTR [rip+0x20096a]        # 200fc0 <add>

656:        68 00 00 00 00          push   0x0

65b:        e9 e0 ff ff ff          jmp    640 <.plt>

0000000000000640 <.plt>:

640:        ff 35 6a 09 20 00       push   QWORD PTR [rip+0x20096a]        # 200fb0 <_GLOBAL_OFFSET_TABLE_+0x8> 646:  ff 25 6c 09 20 00          jmp    QWORD PTR [rip+0x20096c]        # 200fb8 <_GLOBAL_OFFSET_TABLE_+0x10> 64c:    0f 1f 40 00                     nop    DWORD PTR [rax+0x0]
```

Executing code using static library

- Just run the executable C_revision\call_lib_func_static.out since all the library function are part of binary executable code

Executing code using dynamic library

- ldd utility shows which library is not reachable
  - ldd call_lib_func_dynamic.out

    linux-vdso.so.1 (0x00007ffe308d7000)

    libmymath.so => not found

    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff535154000)

    /lib64/ld-linux-x86-64.so.2 (0x00007ff535747000)

- Perform Option1, 2 or 3 shown next…

Executing code using dynamic library Cont...

Set search path for Shared Library file you have created (why do we need this?)

Option1: Set LD_LIBRARY_PATH environment variable (You can modify .bashrc file if this path needs to added at the start of the terminal)

export LD_LIBRARY_PATH=<path to library file>:$LD_LIBRARY_PATH

This will look for .so file in the <path to library file>

Executing code using dynamic library Cont...

Option2 (not good option):

1. Copy the .so file to folder which is the default locations where OS search for library file
2. run ldconfig to add the shared directory in ld (dynamic linker). This will register the .so file with ld to look in /usr/lib folder

Note: default paths for shared library in *.conf files under /etc/ld.so.conf.d

Executing code using dynamic library Cont...

Option3:

1. Create your own conf file in /etc/ld.so.conf.d/myapplib.conf with path to *.so library.
2. Run ldconfig with sudo access to register the .so file with ld to look in your own library folder

Loading

- When executable is used as command on shell prompt, it first need to be loaded from disk to memory
- Executable code is loaded (except shared library functions used in program code) plus data memory (heap) and stack is allocated
- As the function is called from program to shared library they are loaded from library files containing object code of the referenced function

gcc optimization levels

- -O0 → No optimization (the default); generates unoptimized code but has the fastest compilation time
- -O1 → Moderate optimization; optimizes reasonably well but does not degrade compilation time significantly
- -O2 → Full optimization; generates highly optimized code and has the slowest compilation time
- -O3 → Full optimization as in -O2; also uses more aggressive automatic inlining of subprograms within a unit (Inlining of Subprograms) and attempts to vectorize loops
- -Os → Optimize space usage (code and data) of resulting program.

GNU Make (https://www.gnu.org/software/make/)

- Assuming we have following 3 files:
- hellomake.c

```
#include <hellomake.h>
int main() {
        // call function in another file
        myPrintHelloMake();
        return(0);
}
```

- hellofunc.c

```
#include <stdio.h>
#include <hellomake.h>
void myPrintHelloMake(void) {        printf("Hello makefiles!\n");
        return;
}
```

- hellomake.h

```
/* example include file */
void myPrintHelloMake(void);
```

How do compile and link using gcc on command line

gcc -o hellomake hellomake.c hellofunc.c -I.

-I. Indicates look for header file in current directory not only the default location of header files /usr/include

What are the problem with this approach?

1. If you type this command on one computer and also need to run on another computer you have to retype
2. Even if one c file is modified all (both) the files will be recompiled waisting precious CPU time

Make

- 1st problem is solved by creating Makefile or makefile with following command and then just type make on command prompt which will run this makefile : C_revision\makefile1

```
hellomake: hellomake.c hellofunc.c
        gcc -o hellomake hellomake.c hellofunc.c -I.
```

Shows that hellomake.c and hellofunc.c are the dependent files to generate executable hellomake

Using makefile you can run this command on any computer but problem 2 is not yet solved i.e. if any of the source file is changed both will be recompiled

Make

To solve problem 2 we modify makefile as below : C_revision\makefile2

```
CC=gcc
CFLAGS=-I.
hellomake: hellomake.o hellofunc.o
        $(CC) -o hellomake hellomake.o hellofunc.o –I.
```

Make uses some standard Make variables; CC indicates which C compiler to use and CFLAGS indicates the compiler flags

By specifying object files as dependency of executable rather then source files it informs make that source files need to compiled individually which means if the source file has not been modified don't recompile that source file

What is the problem now? If the only the hellomake.h header file is modified then make will not compile any of the source files

Make

makefile3

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h
OBJ = hellomake.o hellofunc.o
%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)
hellomake: $(OBJ)
        $(CC) -o $@ $^ $(CFLAGS)
```

→ only compile with source and dependent header file

→ only generate executable using object files

Now we add DEPS as another variable and mention all the header files which on which source file depends

New rule has to be added for object file creation.

-o $@ says to put the output of the compilation in the file named on the left side of the :

$< is the first item in the dependencies list

$^ indicates take the values from left side of the previous command i.e. all the object files in this case

Make

- what if we want to start putting our .h files in an include directory, our source code in a src directory, and some local libraries in a lib directory? Also, can we somehow hide those annoying .o files that hang around all over the place?

Make – Format generally used for professional software

Folder Structure for Project : make_proj

$ tree

```
.
├── include
│   └── hellomake.h
└── src
    ├── hellofunc.c
    ├── hellomake
    ├── hellomake.c
    ├── makefile  → need to run this makefile
    └── obj
        ├── hellofunc.o
        └── hellomake.o
```

Make – Format generally used for professional software

IDIR =../include → you can put include files in this directory

CC=gcc

CFLAGS=-I$(IDIR)

ODIR=obj → directory where object files will be created

LDIR =../lib → you can put library files in this directory

LIBS=-lm → looks for library libm.so or libm.a

_DEPS = hellomake.h

DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS)) → create string with ../include/<header filename> in our case ../include/hellomake.h

_OBJ = hellomake.o hellofunc.o

OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ)) → create string with obj/<object filename> in our case obj/hellomake.o and obj/hellofunc.o (obj folder will be created under current i.e. source directory)

Make – Format generally used for professional software

```
$(ODIR)/%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)
hellomake: $(OBJ)
	gcc -o $@ $^ $(CFLAGS) $(LIBS)
.PHONY: clean
clean:
	rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

→ removes object files and temporary files created with ~ at the end of file name in current i.e. source and include directory

Quiz

- Consider an example of creating a dynamic library from previous slides.
- Assume directory structure as

project

src

obj → all object file should be created here

lib → dynamic library should be created here

include

bin → application executable should be created here

- Develop “makefile” to:
  - Build dynamic library in lib folder using using add.c and sub.c functions
  - Build application binary using library in bin folder