Compiling & Linking

Tung Yu @ 2024

Ref: Yi Kuo & William Mou & Frank Lin

Outline

- Executable
  - Compile
  - Linking
- Build System
  - Makefile
  - Automake
- Libraries
  - Static library
  - Dynamic library
- Some tools (command in Linux)
  - `readelf, ldd…`

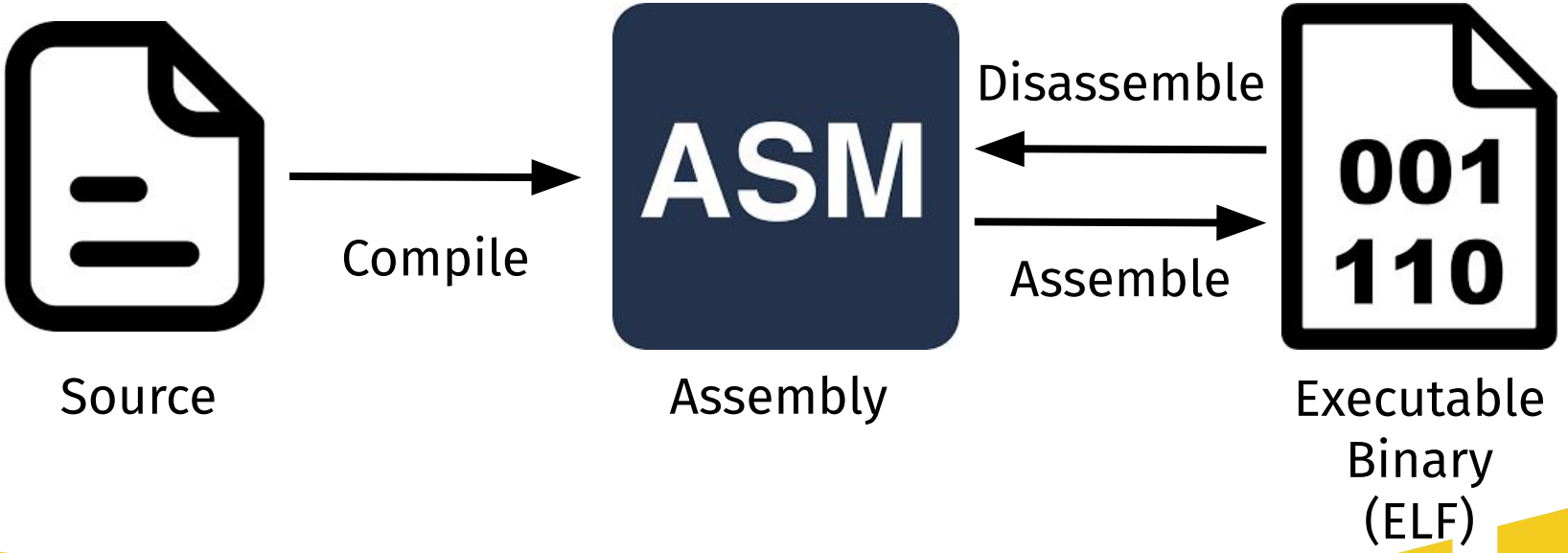
Executable

What is an executable?

- A file with **execute (x)** permission
  - An **ELF** binary
    - Compiled from source code (e.g. C, C++, Go)
    - Contains code that CPU can understand
    - e.g. ls, cp, ps, htop, ...
    - file `which cp`
  - A text file starting with **shebang**: “**#! <program name>**”
    - **<program name>** is the interpreter
    - System runs **<program name> <file>** to execute the script
    - e.g. Shell script (**#! /bin/bash**), Python (**#! /bin/python**)
    - Eventually converts to binary to run on CPU

- We focus on **ELF binary** in this course

How to make an executable?



```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
}
```

Source

Compile

```
push rbp
mov rbp, rsp
lea rdi, str.Hello_World_
call sym.imp.puts
mov eax, 0
pop rbp
ret
```

Assembly

```
push rbp
mov rbp, rsp
lea rdi, str.Hello_World_
call sym.imp.puts
mov eax, 0
pop rbp
ret
```

Assembly

Disassemble

Assemble

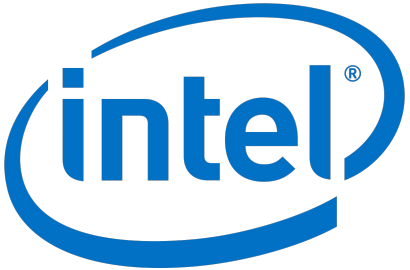
```
5548 89e5 488d 3dc0 0e00 00e8 e7fe ffff
b800 0000 005d c3f3 0f1e fa41 574c 8d3d
8b2c 0000 4156 4989 d641 5549 89f5 4154
4189 fc55 488d 2d7c 2c00 0053 4c29 fd48
83ec 08e8 7ffe ffff 48c1 fd03 741f 31db
0f1f 8000 0000 004c 89f2 4c89 ee44 89e7
41ff 14df 4883 c301 4839 dd75 ea48 83c4
085b 5d41 5c41 5d41 5e41 5fc3 6666 2e0f
1f84 0000 0000 00f3 0f1e fac3 0000 00f3
0f1e fa48 83ec 0848 83c4 08c3 ffff ffff
ffff ffff ffff ffff ffff ffff ffff ffff
```

Binary

```
55              push rbp
4889e5          mov rbp, rsp
488d3dc00e00.   lea rdi, str.Hello_World_
e8e7feffff      call sym.imp.puts
b800000000      mov eax, 0
5d              pop rbp
c3              ret
```

Common compilers

- gcc/g++ by GNU Project
- icc/icpc by Intel
- clang/clang++ by LLVM Developer group
- icx/icpx (DPC++) by Intel (based on LLVM)
- nvc/nvc++ by NVIDIA (PGI)



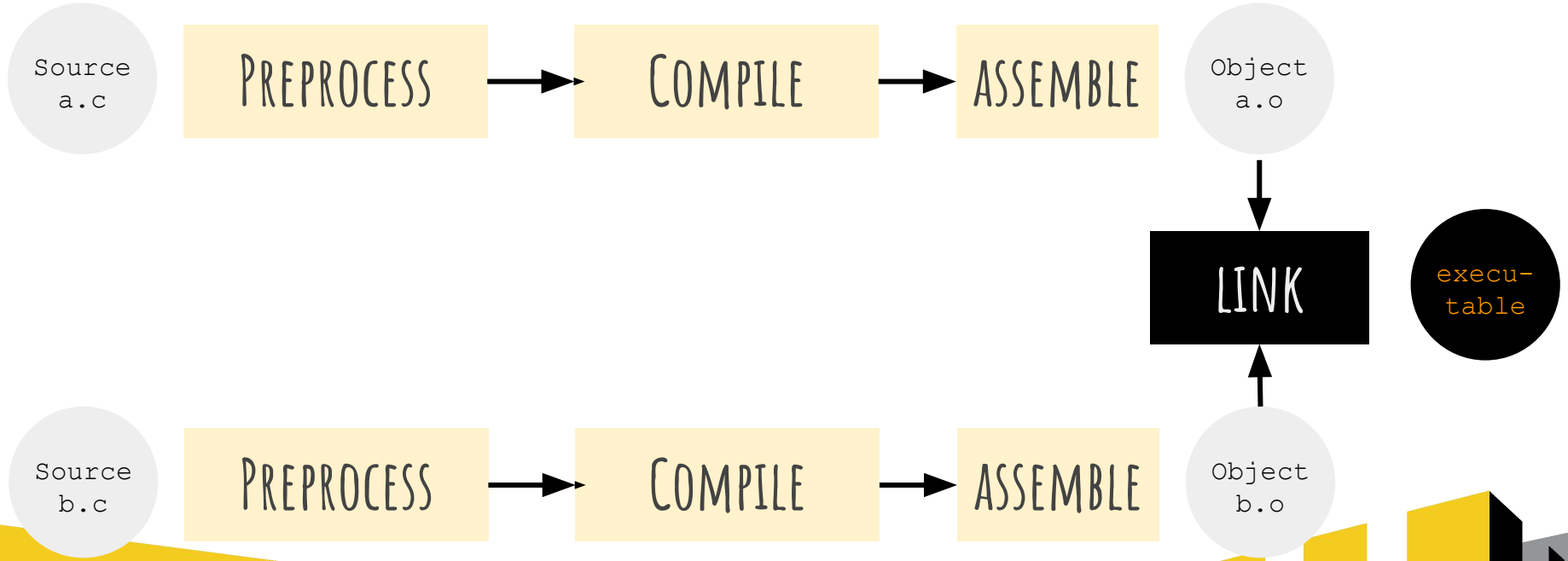
What is an ELF file?

- It is a format of executables on UNIX systems, just like pdf is kind of file format
- Derived from COFF format
  - PE: executable format on Windows (.exe)
  - ELF: executable format on UNIX(Linux/OSX)
- You can find an “ELF” magic number at the beginning of your file
  - Try `xxd <elf_file> | head`
- Obtain the information of your elf file with **readelf**
  - `readelf -a <elf_file>` read all information
- Disassemble the elf file with **objdump**
  - `objdump -d <elf_file>`

What happened during the process of a executable being built?

```
$ gcc -o hello_world.elf hello_world.c
```

Gnu toolchain



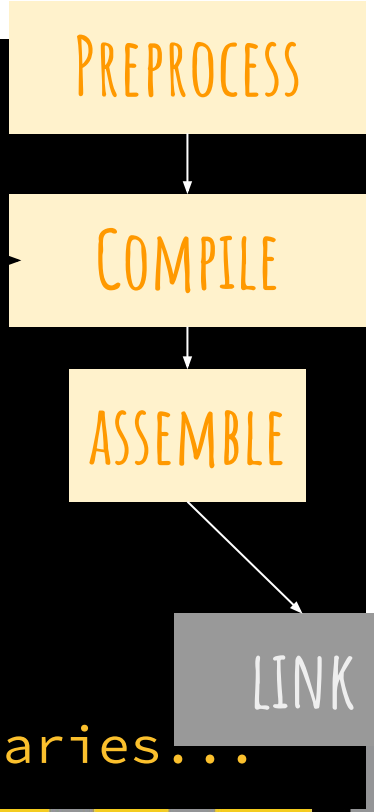
GNU toolchain

```
gcc -E hello_world.c -o hello_world.i

# /usr/libexec/gcc/x86_64-redhat-linux/4.8.5/cc1
cc1 -o hello_world.s hello_world.i

as -o hello_world.o hello_world.s
#----------------------------------------#
gcc -c -o hello_world.o hello_world.c

gcc -o hello_world.elf hello_world.o ...libraries...
```



Compiler flags : Preprocessing

- `-D<symbol>` define symbol (equal to #define)

```
#define DEBUG


#ifdef DEBUG
    printf("debug msg...");
#endif
```

Compiler flags : Preprocessing

- `-I<dirname>` specify the directory where compiler should find header files

  This is same as `C_INCLUDE_PATH` or `CPLUS_INCLUDE_PATH` in environment variable

```
gcc -I/usr/include
export C_INCLUDE_PATH=/usr/include
```

Compiler flags : COMPILATION

- `-Wall` enable all warning message
- `-g` allow debug symbol for gdb

Compiler flags : COMPILATION (optimization)

- -O optimization
  - -O1
  - -O2
  - -O3
  - -Ofast this might give rise to wrong results in gcc
- -xHost(icc), -march=native(gcc) tell compiler to do optimization according to architecture
- -fprofile-generate (PGO)

https://elinux.org/images/4/4d/Moll.pdf

https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Optimize-Options.html

Linking

單個檔案的程式

main.c

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    printf("%d\n", add(1, 2));
    return 0;
}
```

```
$ gcc -o main main.c

$ ./main
  3
```

- 全部擠在一起
- 維護困難

將部分程式拆分成檔案

main.c

```c
#include <stdio.h>

#include "add.h"

int main() {
    printf("%d\n", add(1, 2));
    return 0;
}
```

add.h

```c
int add(int a, int b);
```

add.c

```c
int add(int a, int b) {
    return a + b;
}
```

將部分程式拆分成檔案

main.c

```c
#include <stdio.h>

int add(int a, int b);

int main() {
    printf("%d\n", add(1, 2));
    return 0;
}
```

Declaration (宣告)
告訴編譯器函數長什麼樣子

add.h

```c
int add(int a, int b);
```

add.c

```c
int add(int a, int b) {
    return a + b;
}
```

Definition (定義)
函數真正的實作

- #include "xxx.h"
- 由 Preprocessor 處理
- 將 xxx.h 的內容複製貼上

將部分程式拆分成檔案

main.c

```c
#include <stdio.h>

#include "add.h"

int main() {
    printf("%d\n", add(1, 2));
    return 0;
}
```

add.h

```c
int add(int a, int b);
```

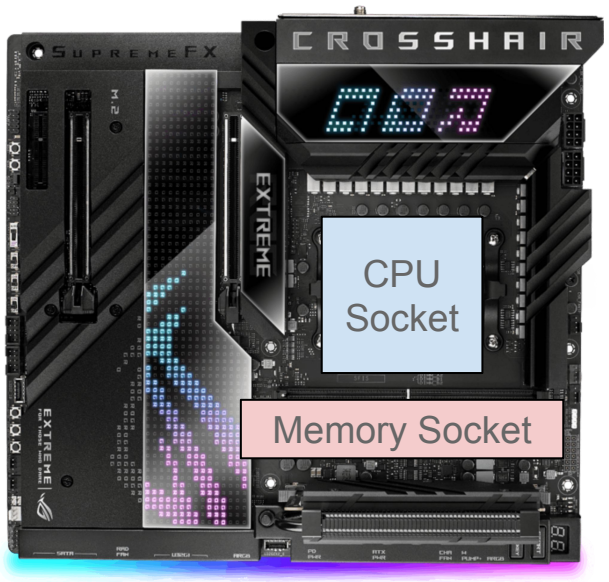
add.c

```c
int add(int a, int b) {
    return a + b;
}
```

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

- 修改部分檔案時，
  需要重新編譯整個程式

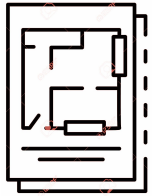
組電腦



Motherboard

Memory & CPU

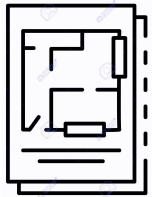
Object file (.o)
單獨一個 .c 檔的編譯輸出
不完整，無法單獨執行



memory.c

Compile

memory.o



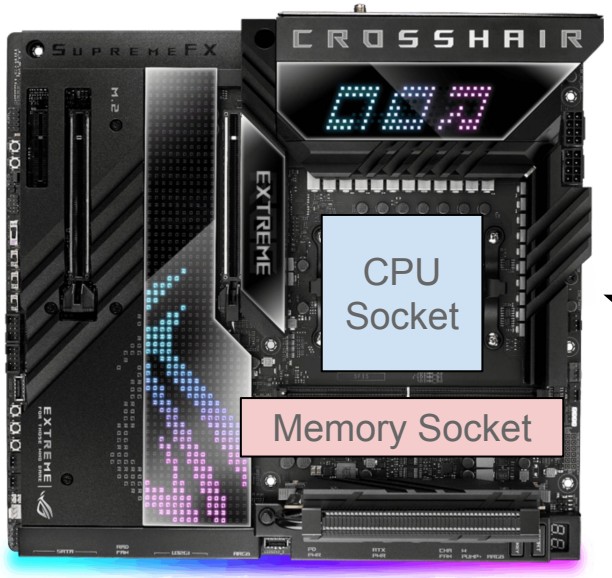
cpu.c

Compile

cpu.o

Motherboard

Header 檔
用來定義界面



motherboard.o

Memory Socket

memory.h

CPU Socket

cpu.h

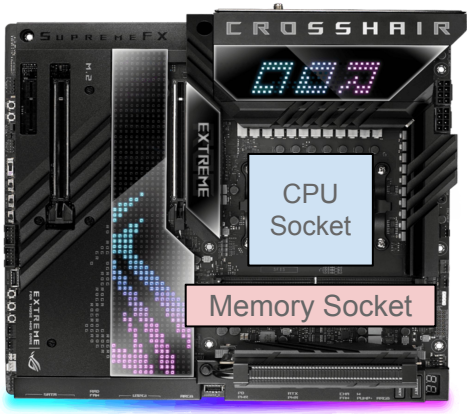
Compile

```
#include "memory.h"
#include "cpu.h"

...
```

motherboard.c

Linking

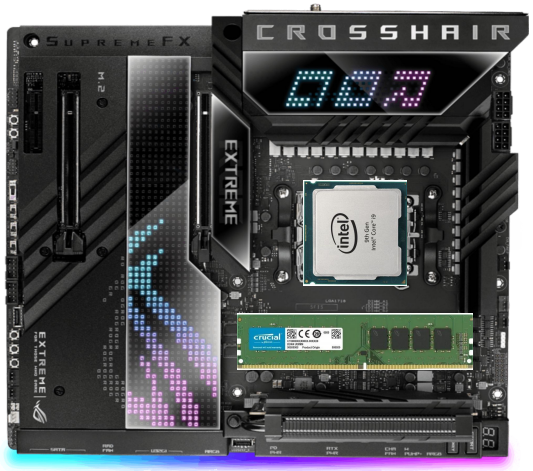


motherboard.o

memory.o

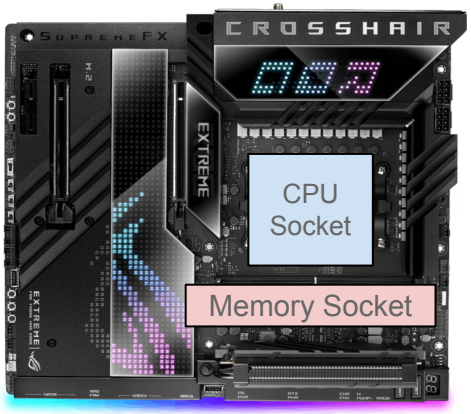
cpu.o

LINK



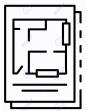
computer
（執行檔）

部分檔案更新



motherboard.o

memory.o

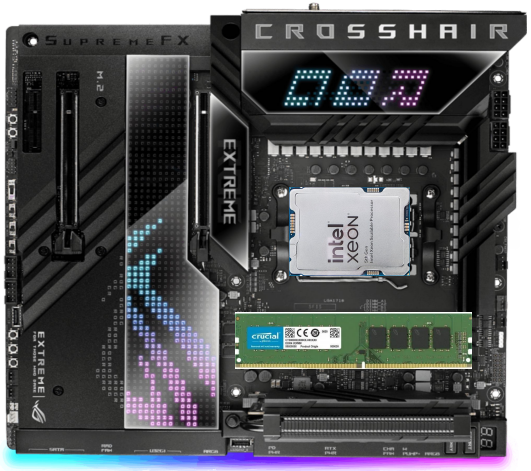


cpu.c (New)

Compile

cpu.o (New)

LINK



computer (New)

Compile 與 Linking

main.c

```c
#include <stdio.h>

#include "add.h"

int main() {
    printf("%d\n", add(1, 2));
    return 0;
}
```

add.h

```c
int add(int a, int b);
```

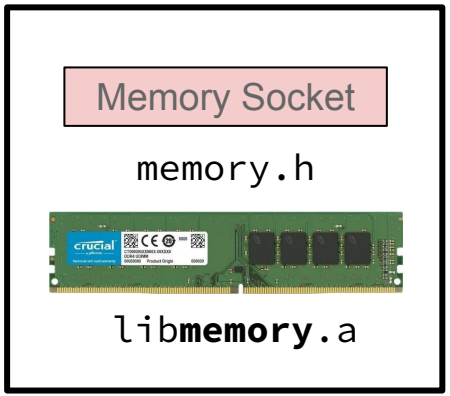
add.c

```c
int add(int a, int b) {
    return a + b;
}
```

```
$ gcc -c -o main.o main.c    # Compile
$ gcc -c -o add.o add.c      # Compile
$ gcc -o main main.o add.o   # Linking
```

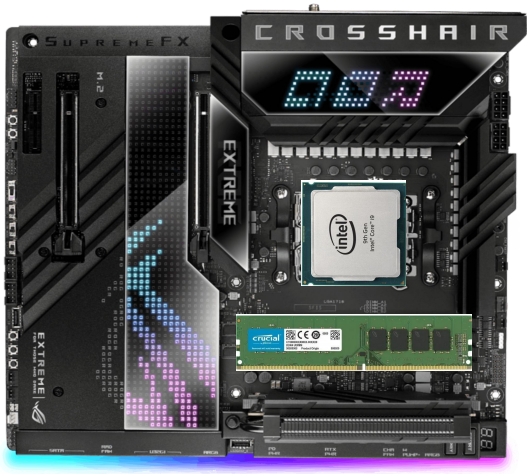
Library (函式庫)

- 由外部專案提供的程式
- 包含 **Header 檔 (.h) 與 編譯好的函式 (.a / .so)**
- 分成兩種：
  - Static Library (.a)
    - 由一或多個 Object file (.o) 組成
    - 必須被埋入執行檔中，每個執行檔都複製一份
  - Shared / Dynamic Library (.so) **較常使用**
    - 產生過程與執行檔大致相同
    - 使用 Dynamic Linking
      - 執行檔只會註記需要哪些Library
    - 執行時由 Linker 尋找需要的 Library
    - 可以被多個程序 (process) 共用

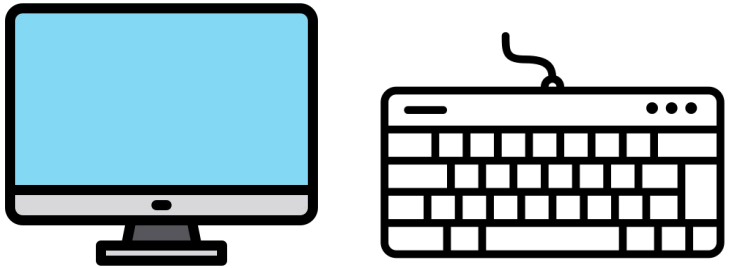


Memory Library

Libraries

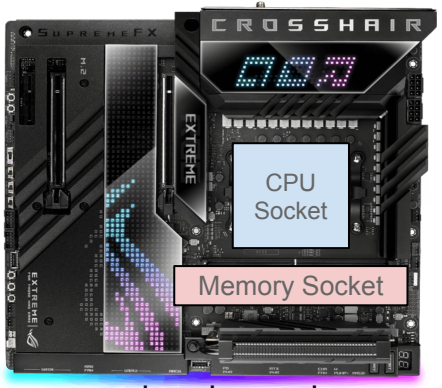


Our program

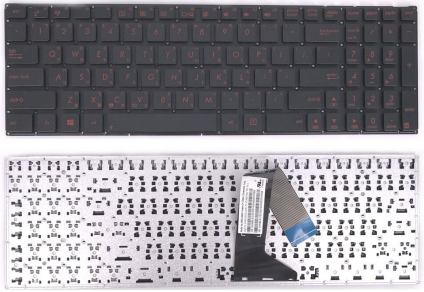


Libraries

Static Linking



motherboard.o



keyboard.a

memory.o

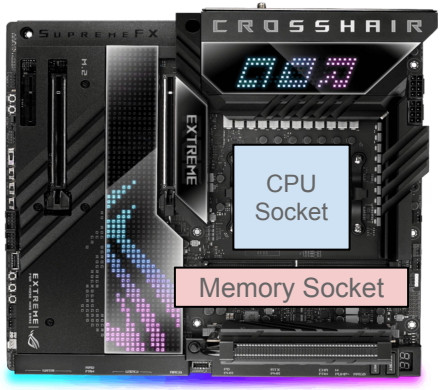
cpu.o

LINKER

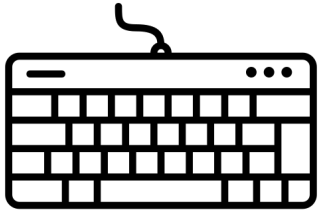
monitor.a

computer.elf

Dynamic Linking - Build Time



motherboard.o

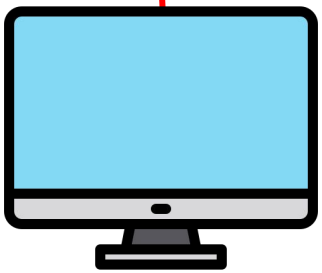


keyboard.so

LINKER

memory.o

cpu.o



monitor.so

computer.elf

Needs:
keyboard.so
monitor.so

Dynamic Linking - Runtime

monitor.so

computer.elf

Needs:
keyboard.so
monitor.so

LINKER

Finds

keyboard.so

使用 Library

main.c

```c
#include <stdio.h>

#include <add.h>

int main() {
    printf("%d\n", add(1, 2));
    return 0;
}
```

```
libadd
├── include
│   └── add.h
└── lib
    └── libadd.so
```

```
$ gcc -c -o main.o -Ilibadd/include main.c   # Compile
```

Or

```
$ export C_INCLUDE_PATH="libadd/include:$C_INCLUDE_PATH"
$ gcc -c -o main.o main.c                    # Compile
```

```
$ gcc -o main main.o -Llibadd/lib -ladd      # Linking
```

Or

```
$ export LIBRARY_PATH="libadd/lib:$LIBRARY_PATH"
$ gcc -o main main.o -ladd                   # Linking
```

- 編譯 (Compiling) 時，預處理器會從 **-I** 或是 環境變數 指定的路徑尋找 Header 檔
- 連結 (Linking) 時，需要使用 **-l** 來指定 Library 的名稱 (`-labc` 會尋找 `libabc.so` 或 `libabc.a`)
  - Linker 會從 **-L** 或是 環境變數 指定的路徑尋找指定的 Library 檔案

Make your own library - Static Library

- **.a** - Archive, a file containing multiple object files (.o)
- Pack many object files into a library as **lib*<libname>*.a**
  - `ar -rcs lib<libname>.a <object1>.o <object2>.o …`
- Show the object files in a archive
  - `ar -t lib<libname>.a`
- Show the containing symbols in a archive
  - `nm lib<libname>.a`
- Build executables with **static linking** using **.a**
  - Treat it as object files
    - `gcc <obj1>.o <obj2>.o lib<libname>.a`
  - Treat it as a library
    - `gcc -static <obj1>.o <obj2>.o -L<dirname> -l<libname>`

Make your own library - Shared Library

- **.so** - Shared object, code that can be shared by multiple processes
- Compile and link codes into a library as **lib*<libname>*.so**
  - `gcc -c -fPIC -o <codeA>.o <codeA>.c`
  - `gcc -c -fPIC -o <codeB>.o <codeB>.c`
  - `gcc -shared -o lib<libname>.so <codeA>.o <codeB>.o`
- Build executables with **dynamic linking** using **.so**
  - **Compile Time**
    - `gcc -o <exe>.elf <obj1>.o <obj2>.o -L<dirname> -l<libname>`
  - **Run Time**
    - `LD_LIBRARY_PATH=<dirname> ./<exe>.elf`

Position Independent Code (PIC)

PIC

```
100: COMPARE REG1, REG2
101: JUMP_IF_EQUAL CURRENT+10
...
111: NOP
```

Non PIC

```
100: COMPARE REG1, REG2
101: JUMP_IF_EQUAL 111
...
111: NOP
```

Can the codes still work if they are placed at line 200?

- Can PIC works?
- Can Non PIC works?

Where to find shared library at runtime?

- LD_LIBRARY_PATH tells the loader the **directory** where the libraries (.so) should be found
- LD_PRELOAD directly assign which library **file** (.so) should be loaded
- runpath: Specify where to load the library in compile time
  - `gcc -o <exe>.elf <obj>.o -Wl,-rpath,<dirname>`
  - `readelf -d <exe>.elf | grep 'R.*PATH'`
- System library paths
  - `ldconfig -v 2>/dev/null | grep -v ^$'\t'`
  - `/lib, /usr/lib`

- Priority: LD_PRELOAD > LD_LIBRARY_PATH > runpath > Others
- LD_DEBUG=all to trace all debug information

Writing Makefile

Makefile: Explicit Rules

```
#<target>: <prerequisites...>
#     <commands>

result.txt: source.txt
    cp source.txt result.txt

.PHONY: clean
clean:
    rm *.o
```

Makefile: Implicit rules

```
CXX = g++
exe = main.elf
obj = main.o print.o

$(exe): $(obj)
        $(CXX) -o $(exe) $(obj)

.PHONY: clean
clean:
        rm $(exe) *.o
```

- main.elf
  - main.o
    - main.cpp
  - print.o
    - print.cpp

- https://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html

Makefile: Variables

- $@ The file that is being made right now by this rule (aka the "target")
- $< The input file (that is, the first prerequisite in the list)
- $^ This is the list of ALL input files, not just the first one.
- $? All the input files that are newer than the target
- $$ A literal $ character inside of the rules section
- $* The "stem" part that matched in the rule definition's % bit
- % like * in shell

Makefile: Using Variables

```
CC = gcc
exe = main
obj = main.o a.o b.o c.o

$(exe): $(obj)
    $(CC) -o $(exe) $(obj)

%.o: %.c
    $(CC) -c $^ -o $@
```

- main
  - main.o
    - main.c
  - a.o
    - a.c
  - b.o
    - b.c
  - c.o
    - c.c

MAKE FILE: REFERENCE

GNU make - doc

https://www.gnu.org/software/make/manual/make.html

Github - isaacs/Makefile

https://gist.github.com/isaacs/62a2d1825d04437c6f08

簡單學 makefile：makefile 介紹與範例程式

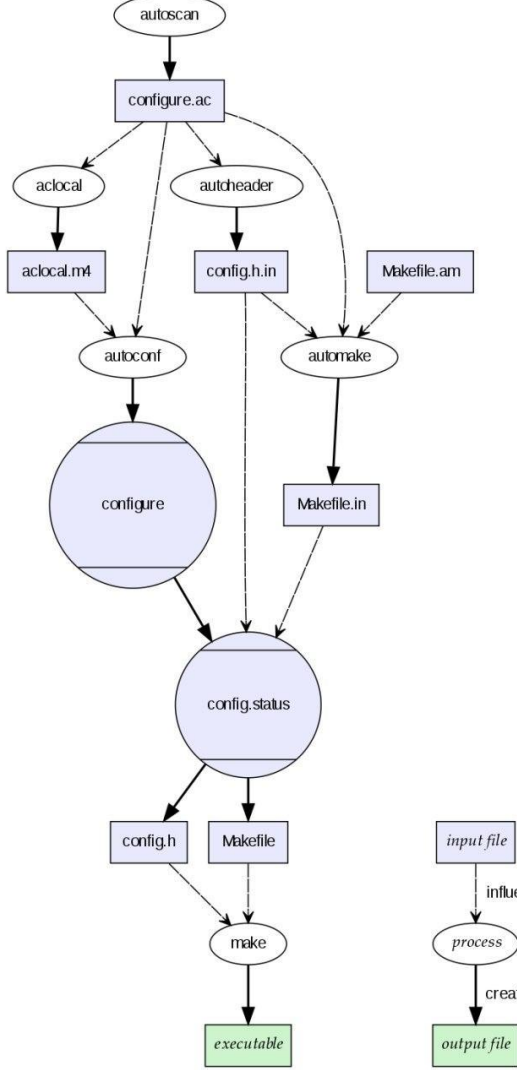
http://mropengate.blogspot.com/2018/01/makefile.html

GCC and Make

https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

Automake

- GNU's build toolchain
- `./configure --prefix <path_to_install> &&`
  `make &&`
  `make install`



After a software/library is installed

- Set the environment variables
  - **PATH**
  - **LD_LIBRARY_PATH**
  - **LIBRARY_PATH**
  - **C_INCLUDE_PATH / CPLUS_INCLUDE_PATH**
- Or, create a **modulefile**

- Review: **LIBRARY_PATH** vs **LD_LIBRARY_PATH**
  - **LIBRARY_PATH** is used to find the libraries **at linking time**
    - Same as: gcc **`-L<dir>`**
  - **LD_LIBRARY_PATH** is used to find the libraries **at runtime**
    - **`LD_LIBRARY_PATH=<dir> ./<executable>`**

Summary

- 建置 (Build) 程式有分成兩個步驟：編譯 (Compile) 與 連結 (Link)
  - Compile 將部分程式碼轉為機器碼
  - Link 將各個不完整的程式組合起來，並結合函式庫 (Library)，產生可以執行的執行檔
- 使用 Makefile 方便編譯，並可以根據檔案變更重新編譯必要的部分
- 使用 Shared Library 需要設定的編譯器參數 / 環境變數
  - `-I / C_INCLUDE_PATH`
  - `-L / LIBRARY_PATH`
  - `-l`
  - 執行時：`LD_LIBRARY_PATH`

Homework

A source code of merge sort is provided:

main.cpp
├ Merge.cpp
├ Mergesort.cpp
└ PrintArray.cpp

DO NOT copy all codes into one single file!

- Write your own Makefile to compile the executable
- Build those source files (except main) to library and execute with them
  - Shared library
  - Static library

Homework - Submission

Please submit the files in the following format:

- Makefile
- lib<your_library_name>.so
- lib<your_library_name>.a
- <your_student_id>_report.pdf (or can be .docx, ...)

Report requirements:

- What command you use to build your program? build the library?
- What command you use to run the executable? environment variables?
- Other worth mentioning (optional)
- Feedback (optional)

Thank you