SCC.131: Digital Systems
Build automation

Ioannis Chatzigeorgiou
(i.chatzigeorgiou@lancaster.ac.uk)

Summary of the last lecture

In the last lecture, we focused on micro:bit, which uses the nRF52833 SoC built around the 32-bit ARM Cortex-M4F processor, and we discussed about:

- The different **types of memory** (flash, RAM, memory of external devices).
- The memory layout of **physical addresses**, when Bluetooth is disabled and when Bluetooth is enabled.
- The stack layout for the same **example** used in the previous lecture (but adapted for the micro:bit).
- GDB commands used within Visual Studio Code, which can help visualise the stack of the running program and reveal if functions are **inline** or not.

Building code

- So far, compiling your C code has been straightforward:

```
gcc *.c -o executable_name
```

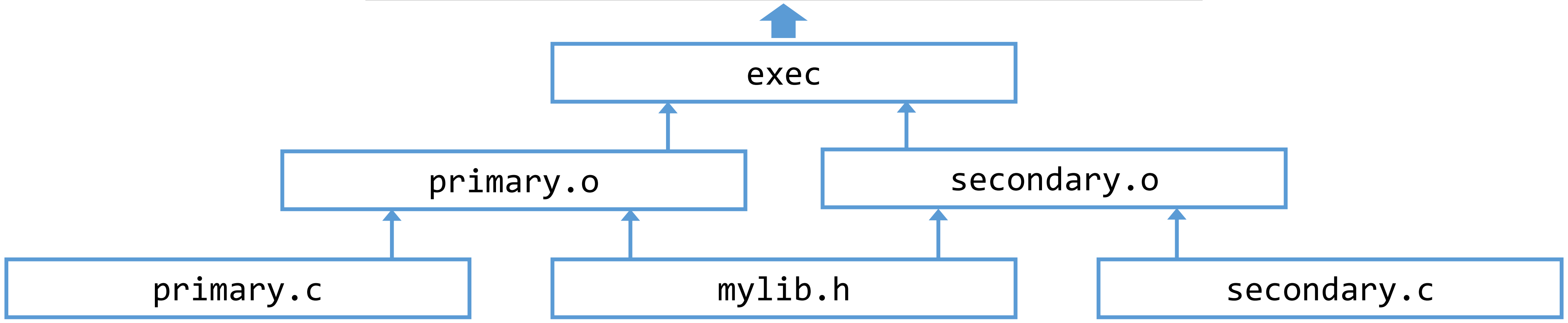
- Compiling the Linux kernel:
  - Millions of lines of code with comments included (~28 million as of 2020)
  - Thousands of source files (~36500 as of 2020)
  - Hours to compile a standard Linux kernel (~2 hours on an Intel Xeon CPU with six cores in 2022)
- Automating code compilation is crucial for large projects:
  - Collaboration
  - Faster compilation through incremental builds

Build automation and script generation

- **Build automation systems** automate the process of compiling source code (from multiple source files) into binary executable code.
  - Examples: GNU Make, MPW Make, Google Bazel, SCons, etc.
- **Build script generation tools** generate files for build automation systems, i.e., they do not build executable code directly. Users can set up a high-level configuration and generate build scripts for a specific operating system (Linus, macOS, Windows, etc.).
  - Examples: CNU Build System (autotools), CMake, Apache Meson, etc.

Example scenario: Dependencies

```
gcc primary.c secondary.c -o exec
./exec
Function func1() returns func2(5) = 50
```



- Defines `int func1(void)` in terms of `int func2(5)`.
- Prints the output of `func1()` in `main()`.

- Uses C preprocessing language: `#define IMPORTANT_CONST 10`
- Declares `int func2(int)`

- Defines `int func2(int a)` and returns `a*IMPORTANT_CONST`.

Focus on Make

- The **Make** build automation system reads a script, called **makefile**, which contains:
  - The project structure (e.g., files and dependencies between files).
  - Instructions for the creation of binary files (e.g., object files and executable files).
- A project originally developed as part of the UNIX toolchain in the 1970s:
  - Several re-implementations
  - Language-independent (works for C/C++ but for other languages too)
- In this lecture we use **GNU Make** as an example.

Version 1 of **makefile**

makefile

```
exec:
```

This is called the **TARGET**.

- It could be a filename, variable or string.
- This is the name for the ACTIONS that follow.
- To execute the ACTIONS of TARGET `exec`, type

  `make exec`
- If the `makefile` contains only one TARGET, you do not need to specify the TARGET when running `make`.
- If the `makefile` contains more than one TARGETS, running `make` without specifying a TARGET will execute the ACTIONS of TARGET `all` (if it has defined) or the first TARGET.

Version 1 of **makefile**

makefile

```
exec: primary.c secondary.c mylib.h
```

These are **DEPENDENCIES** (prerequisites):

- List of requirements for a TARGET.
- Could be filenames or other TARGETS.
- If the TARGET and the DEPENDENCIES are filenames, their timestamps are compared, and the ACTIONS are executed only if the TARGET does not exist or if it is older than the DEPENDENCIES (i.e., out of date).
- If a DEPENDENCY is the name of another TARGET, control will descend to the ACTIONS of the other TARGET.

Version 1 of **makefile**

makefile

```
exec: primary.c secondary.c mylib.h
	gcc primary.c secondary.c -o exec
```

These are the **ACTIONS**:

- Use “tab” to indent an ACTION (not spaces).
- ACTIONS are *shell* commands.
- Each line should contain one ACTION.
- If an ACTION cannot fit in one line, the backslash (\) can be used to break the ACTION into multiple lines.

Version 1 of **makefile**

makefile

```
exec: primary.c secondary.c mylib.h
	gcc primary.c secondary.c -o exec
```

The TARGET, DEPENDENCIES and ACTIONS form a **RULE**.

- A RULE contains the ACTIONS to meet a TARGET when the DEPENDENCIES are fulfilled.

Version 1 of **makefile**

makefile

```
exec: primary.c secondary.c mylib.h
	gcc primary.c secondary.c -o exec
```

Linux environment

```
$ make
gcc primary.c secondary.c -o exec
$ ls -lh
total 16K
16K Jan 22 16:44 exec
 62 Jan 22 16:44 makefile
 76 Jan 22 11:48 mylib.h
178 Jan 22 11:48 primary.c
 91 Jan 22 11:47 secondary.c
$ make
make: 'exec' is up to date.
$ touch secondary.c
$ make
gcc primary.c secondary.c -o exec
```

Version 2 of **makefile**

makefile

```
exec: primary.o secondary.o
	gcc primary.o secondary.o -o exec

primary.o: primary.c mylib.h
	gcc -c primary.c -o primary.o

secondary.o: secondary.c mylib.h
	gcc -c secondary.c -o secondary.o
```

- In **version 1** of makefile, a change in one of the two source files will trigger the creation and linking of both of their corresponding object files.
- In **version 2** of makefile, separate RULES for the object files have been introduced. The DEPENDENDENCIES and ACTIONS of TARGET exec have also been updated.
  - In this case, only the object file of the source file that has been changed will be created when we run `make`.

Version 2 of **makefile**

makefile

```
exec: primary.o secondary.o
	gcc primary.o secondary.o -o exec

primary.o: primary.c mylib.h
	gcc -c primary.c -o primary.o

secondary.o: secondary.c mylib.h
	gcc -c secondary.c -o secondary.o
```

Linux environment

```
$ make
gcc -c primary.c –o primary.o
gcc -c secondary.c –o secondary.o
gcc primary.o secondary.o -o exec
$ touch secondary.c
$ make
gcc -c secondary.c –o secondary.o
gcc primary.o secondary.o -o exec
$ touch primary.c
$ make
gcc -c primary.c –o primary.o
gcc primary.o secondary.o -o exec
```

Version 3 of **makefile**

makefile

```
# Example makefile
CC = gcc

exec: primary.o secondary.o
	$(CC) primary.o secondary.o -o exec

primary.o: primary.c mylib.h
	$(CC) -c primary.c -o primary.o

secondary.o: secondary.c mylib.h
	$(CC) -c secondary.c -o secondary.o
```

- In **version 3** of makefile, a comment and a variable have been introduced.
- A **comment** begins with a # and lasts till the end of the line.
- A **variable** used in a RULE begins with a $ and is enclosed within parentheses (...) or braces {...}. Single character variables do not need parentheses or braces, e.g., $C, $(CC).

Version 4 of **makefile**

makefile

```
# Example makefile
CC = gcc

exec: primary.o secondary.o
	$(CC) $^ -o $@

primary.o: primary.c mylib.h
	$(CC) -c $< -o $@

secondary.o: secondary.c mylib.h
	$(CC) -c $< -o $@
```

- **Version 4** of makefile uses **automatic variables** to further shorten the script:

`$@`	The filename of the TARGET.
`$*`	The filename of the TARGET without the file extension.
`$^`	The filenames of all DEPENDENCIES.
`$<`	The filename of the first DEPENDENCY.
`$?`	The filenames of all DEPENDENCIES that are newer than the TARGET.

Version 5 of **makefile**

makefile

```
# Example makefile
CC = gcc

exec: primary.o secondary.o
	$(CC) $^ -o $@

%.o: %.c mylib.h
	$(CC) -c $< -o $@
```

- The similarities between the two RULES with TARGETS primary.o and secondary.o are exploited in **version 5** of makefile.
- The two RULES can be merged into a single RULE if the **pattern matching character** % is used.
- Character % matches a DEPENDENCY filename **without the extension**.

Version 6 of **makefile**

Lancaster University

makefile

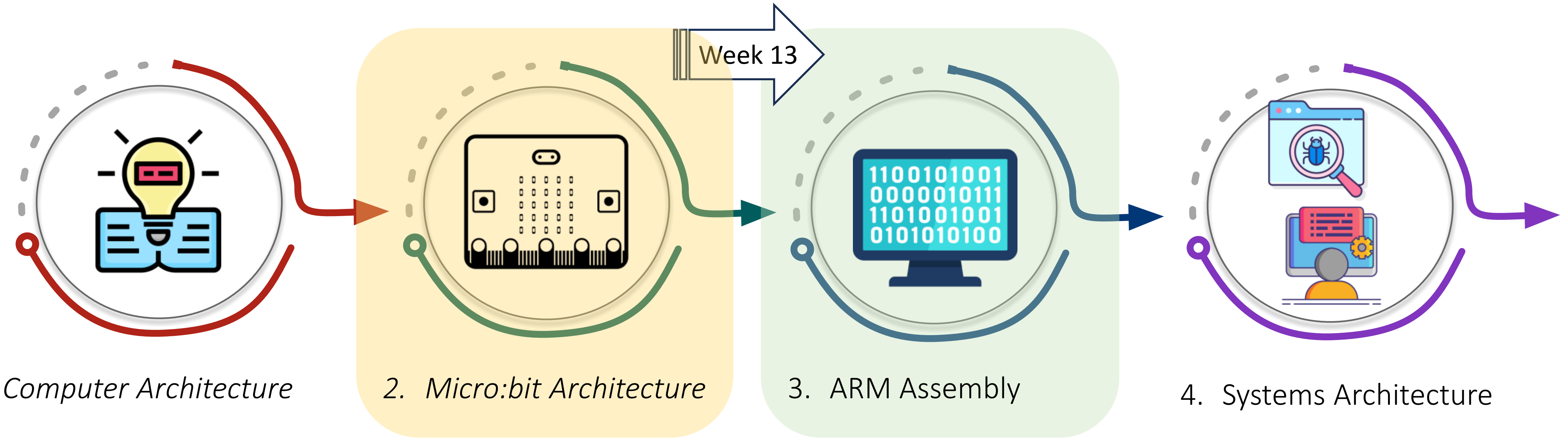
```
# Example makefile
CC = gcc

all: exec

exec: primary.o secondary.o
    $(CC) $^ -o $@

%.o: %.c mylib.h
    $(CC) -c $< -o $@

clean:
    rm primary.o secondary.o
```

- **Version 6** includes two TARGETS that do not represent filenames: `all` and `clean`.
- TARGETS of this type are often referred to as 'phony' TARGETS and they are always treated as 'out of date', i.e., ACTIONS of phony TARGETS are always executed.
- In this example, TARGET `all` establishes which RULE will be considered by default (it does not have to be at the top of the script).
- TARGET `clean` removes the two object files (when "`make clean`" is run).

Resources

- GNU Make: https://www.gnu.org/software/make/manual/make.html
- Makefile tutorials
  - https://wiki.osdev.org/Makefile
  - https://makefiletutorial.com/

Ready for the next chapter?


Week 13
1. *Computer Architecture*
2. *Micro:bit Architecture*
1100101001
0000010111
1101001001
0101010100
3. ARM Assembly
4. Systems Architecture