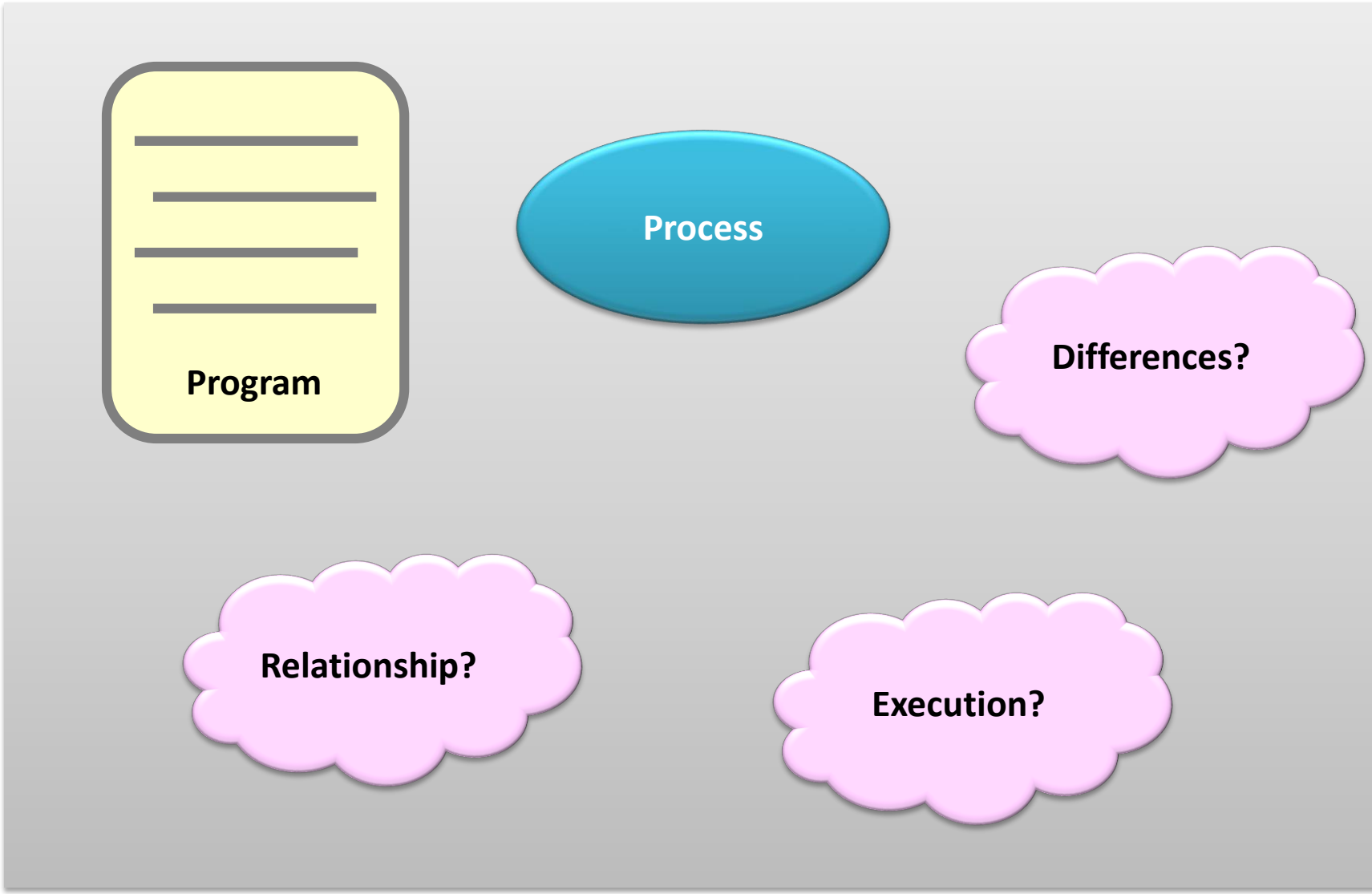
3150 - Operating Systems

Dr. WONG Tsz Yeung

Chapter 2, Part 1- Basic Process Management

- Don’t get lost from the very beginning…

Outline



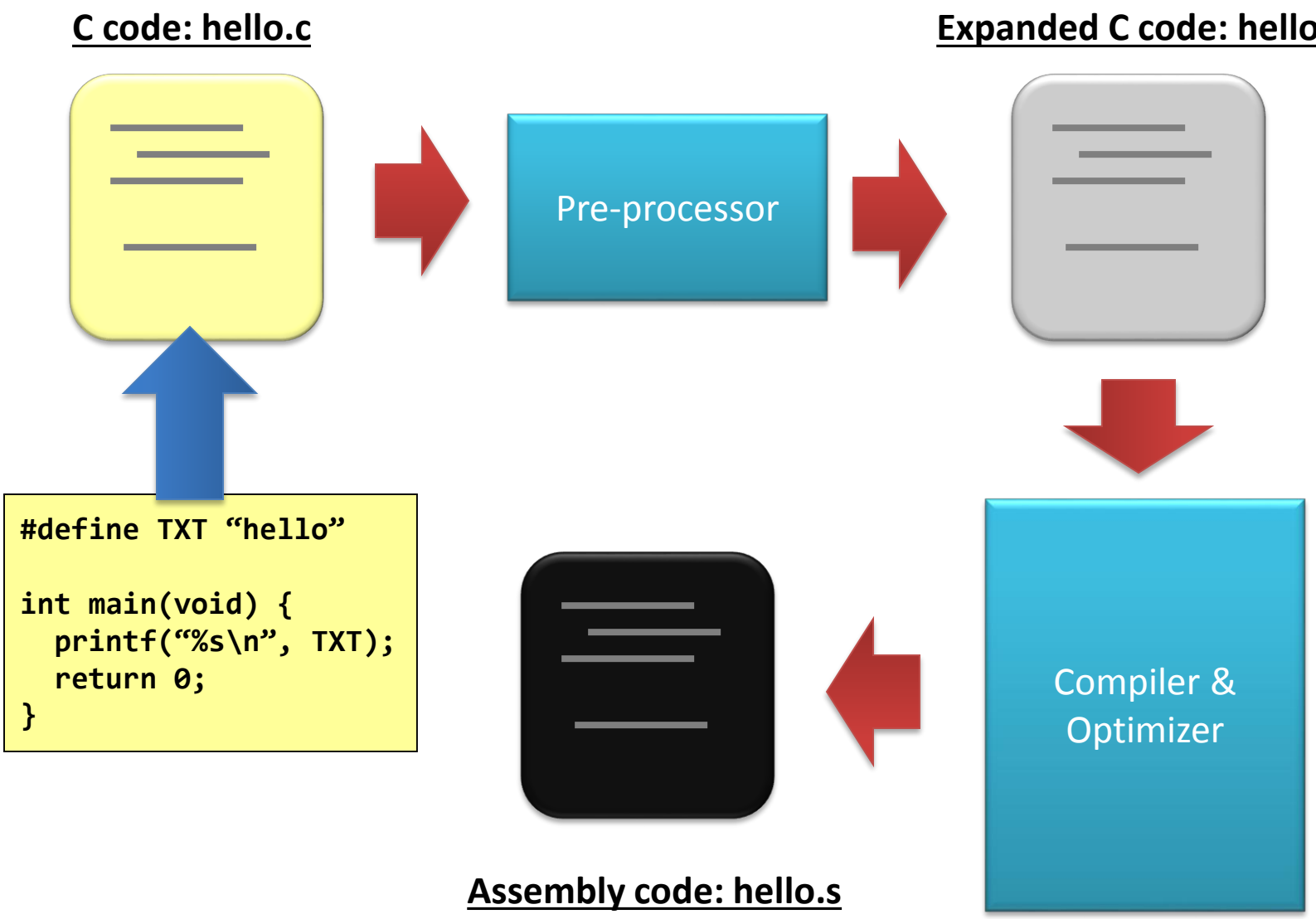
What is a program?

What is a program?

- What is a program?
  - A program is a just **a piece of code**.

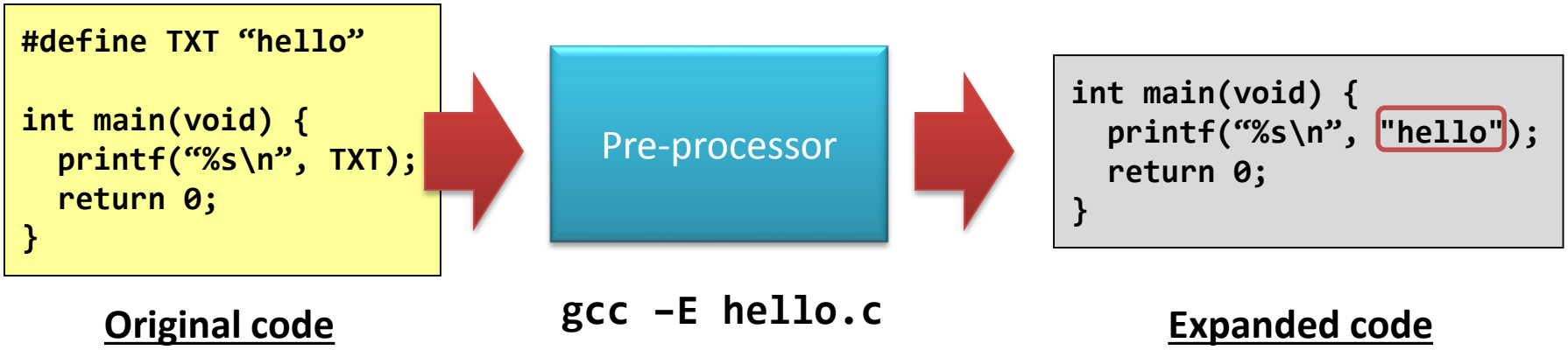
- But, which code do you mean?
  - High-level language code: C or C++?
  - Low-level language code: assembly code?
  - Not-yet an executable: object code?
  - Executable: machine code?

Flow of building a program (1 of 2)



(Still…1 of 2) Pre-processor

- The pre-processor expands:
  - **`#define`**, **`#include`**, **`#ifdef`**, **`#ifndef`**, **`#endif`**, etc.
  - Try: "**`gcc -E hello.c`**"

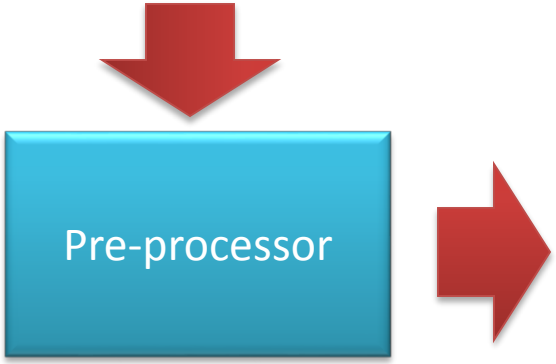



```
[example@3150]$ cat program/hello.c
```

(Still…1 of 2) Pre-processor

- Another example: **the macro**!

```
#define SWAP(a,b) { int c; c = a; a = b; b = c; }

int main(void) {
    int i = 10, j = 20;
    printf("before swap: i = %d, j = %d\n", i, j);
    SWAP(i, j);
    printf("after swap: i = %d, j = %d\n", i, j);
}
```

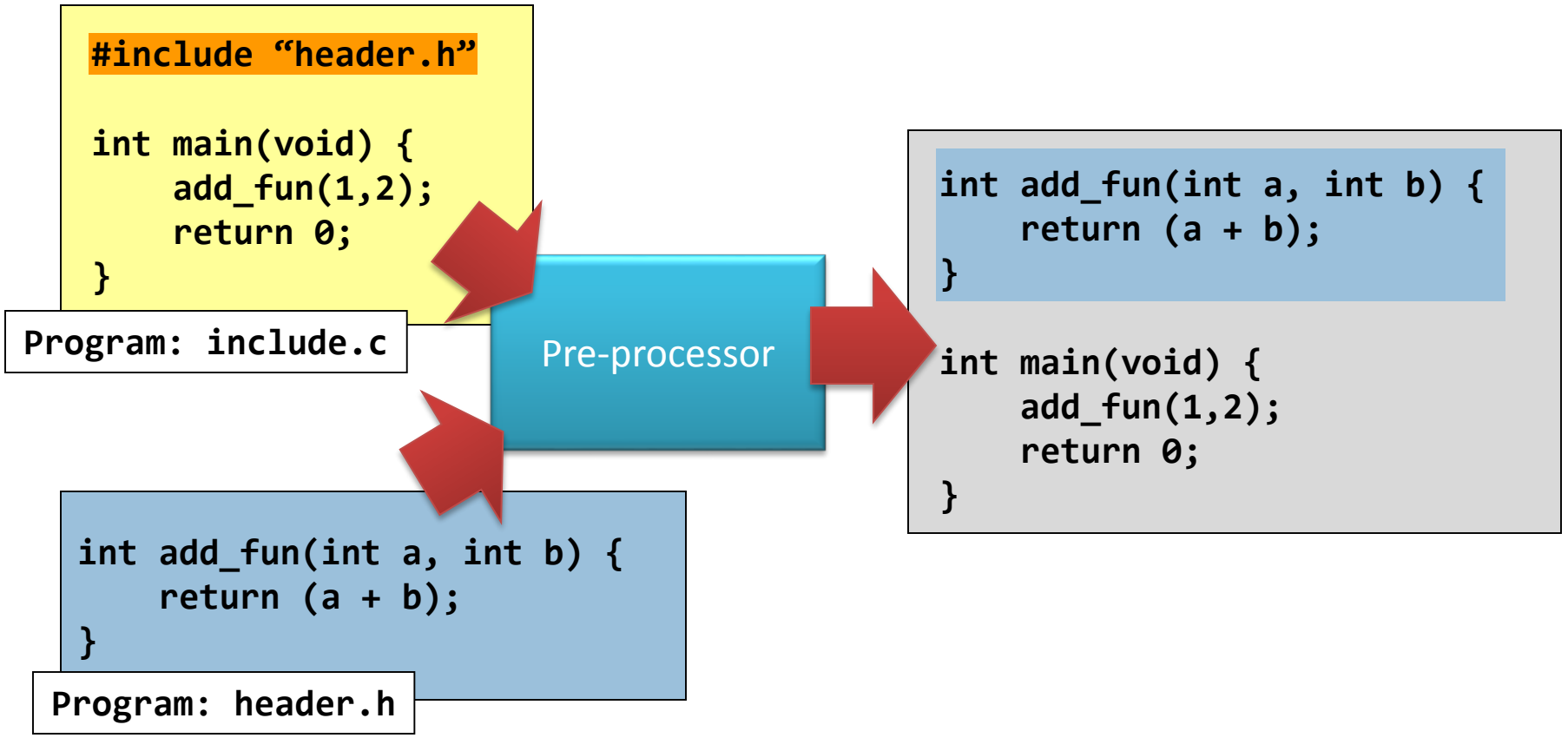



```
int main(void) {
    int i = 10, j = 20;
    printf("before swap: i = %d, j = %d\n", i, j);
    { int c; c = i; i = j; j = c; };
    printf("after swap: i = %d, j = %d\n", i, j);
}
```

```
[example@3150]$ cat program/swap.c
```

(Still…1 of 2) Pre-processor

- How about: **#include**?




```
[example@3150]$ cat program/{include.c,header.h}
```

(Still…1 of 2) Compiler and Optimizer

- The compiler performs:
  - Syntax checking and analyzing;
  - If there is no syntax error, construct intermediate codes, i.e., assembly codes;
  - For syntax-related topics, we have the courses *CSCI3130* and *CSCI3120* (It is the compiler course. But, no one is teaching this course for two years already).
  - For assembly language: we have the courses *CSCI2510* (for minor) and *CSCI3420* (for major).

(Still…1 of 2) Compiler and Optimizer

- The optimizer optimizes codes.
  - In other words: **it improves stupid codes!**
  - Try:

```
gcc –S add.c –O0 –o add_O0.s
gcc –S add.c –O1 –o add_O1.s
```

"**-O**" means to optimize.

The number followed is the optimization level.

"**-O0**": means no optimization.

Max is level 3, i.e., "**-O3**".
Default is level is "**-O1**".

- Important notice: **Don't write stupid codes because there is an optimizer!**

```
int main(void) {
  int x = 1;
  x = x + 1;
  x = x + 1;
  return x;
}
```

```
[example@3150]$ cat program/stupid_add.c
```

(Still…1 of 2) Compiler and Optimizer

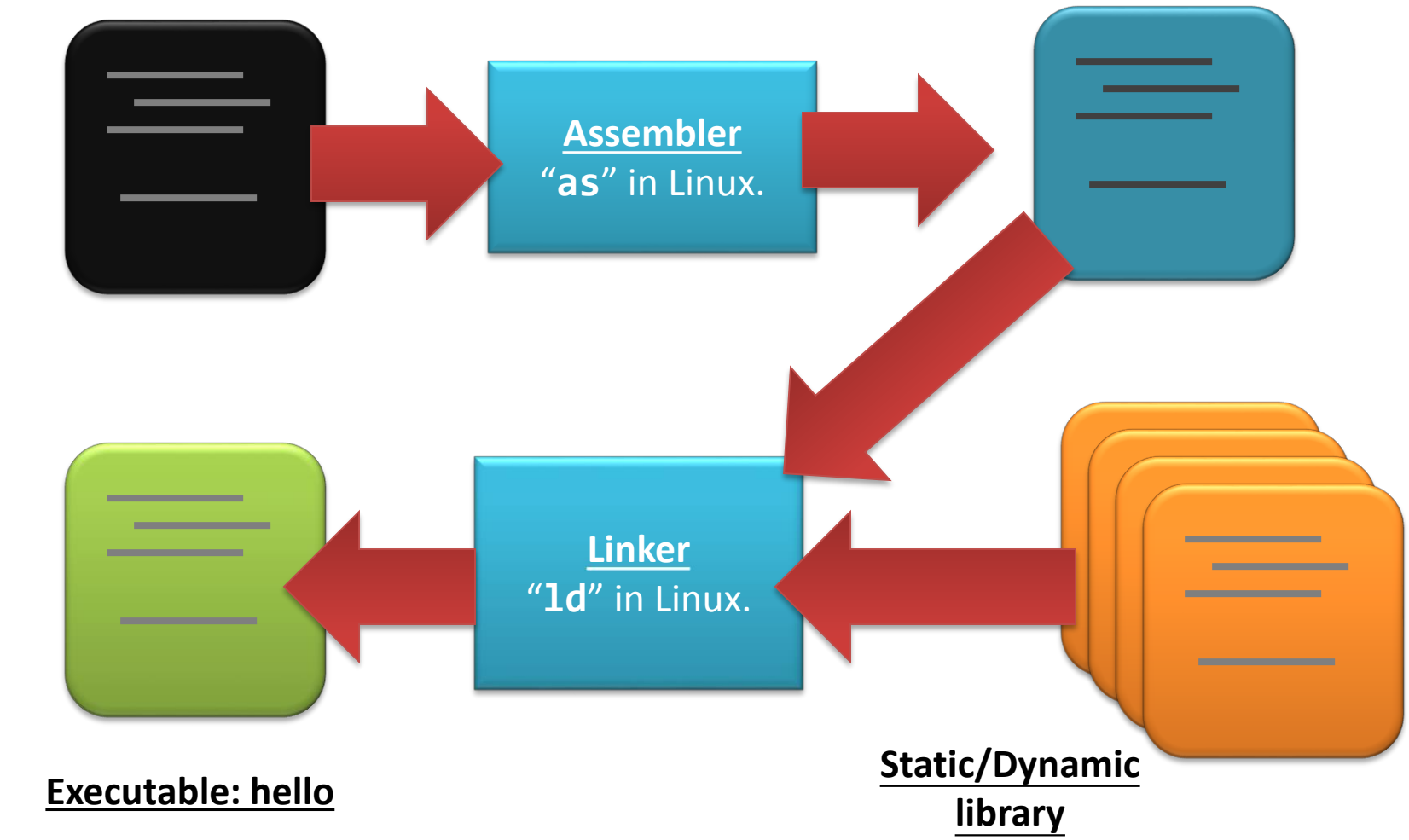
EXTRA

- Note on assembly codes:
  - It is actually a very good teacher, teaching you how C implements features.
  - E.g., I can decode the following program in assembly language:

```
int main(void) {
  char hello[20] = "hello world";
  printf("%s\n", hello);
}
```

  - Then, we can learn how the "hello array" is initialized!
    - Out of my expectation, the compiler does not involve **`strcpy()`** nor **`strncpy()`**.

Flow of building a program (2 of 2)



(Still…2 of 2) Assembler and Linker

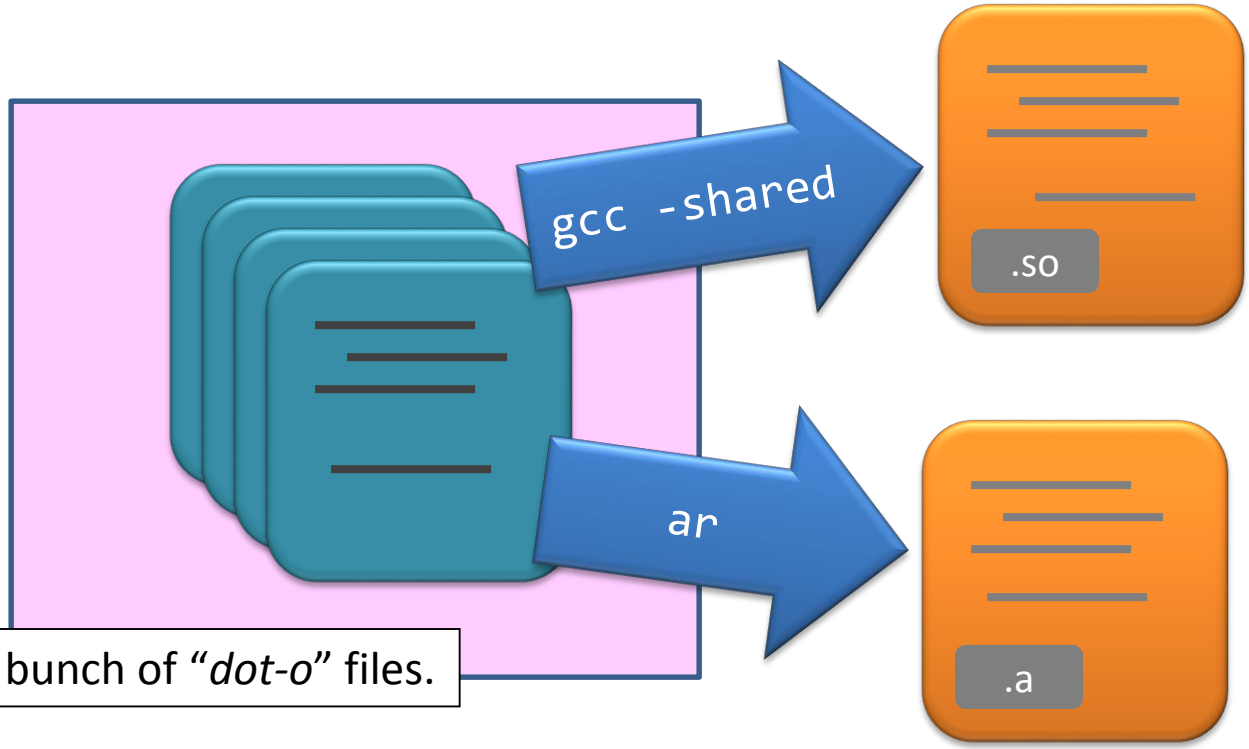
- The assembler assembles "**`hello.s`**" and generates an object code "**`hello.o`**".
  - A step closer to machine code.
  - Try: "**`as hello.s -o hello.o`**"

- The linker puts together all object files as well as the libraries.
  - There are two kinds of libraries: statically-linked and dynamically-linked ones

Sidetrack: Library files

EXTRA

- A library file is…
  - just a bunch of function implementations.
  - for the linker to look for the function(s) that the target C program needs.

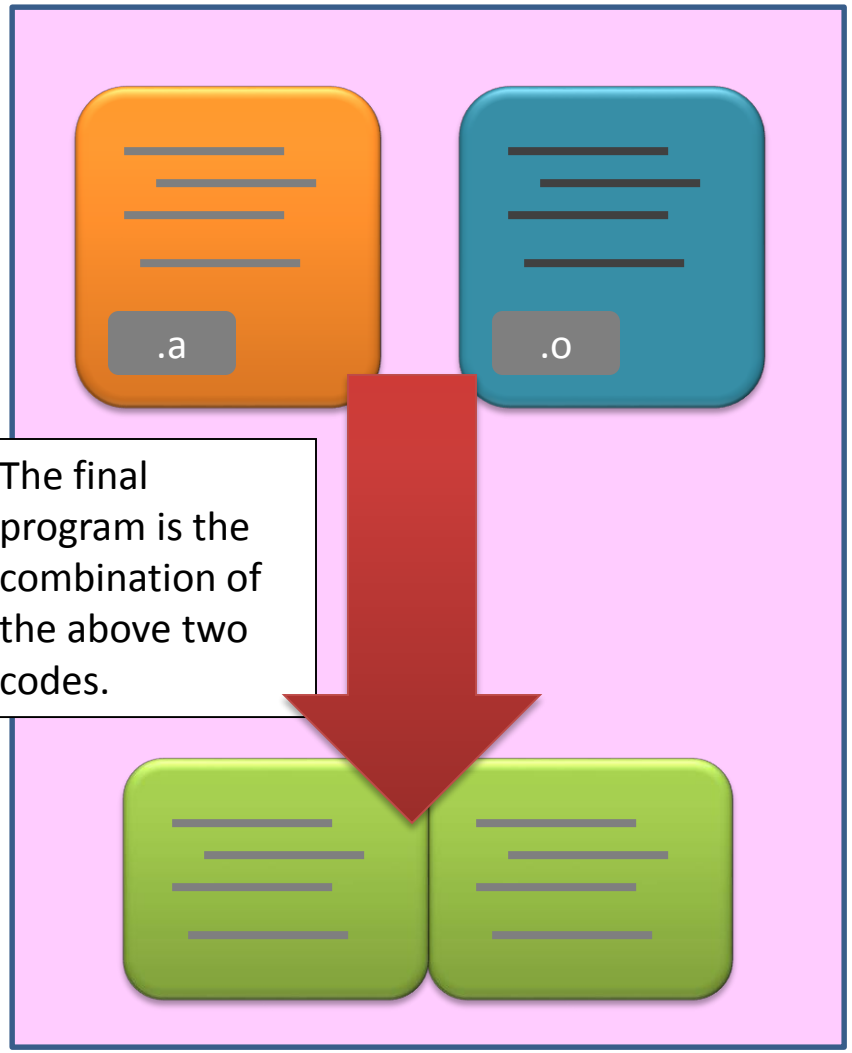


Shared library with "**.so**" file extension.

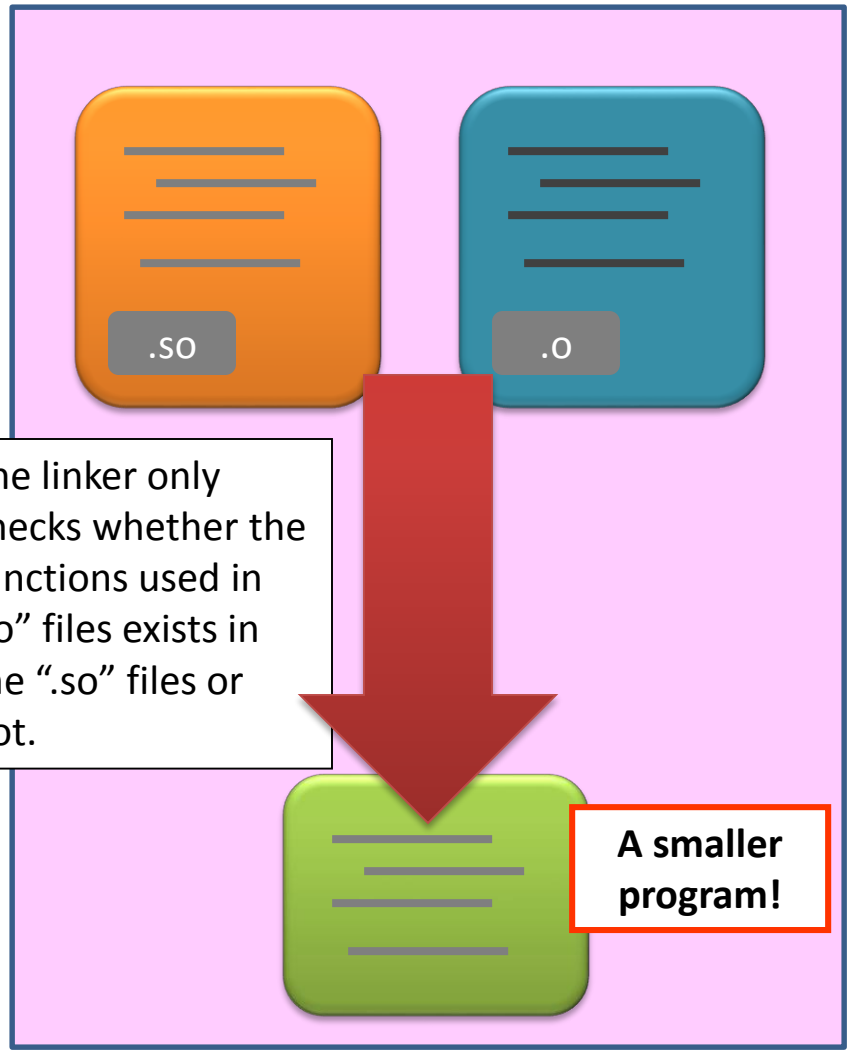
A static library with "**.a**" file extension. It is also called an archive.

Sidetrack: Library files

EXTRA



Linking with static library file.



Linking with dynamic library file.

Sidetrack: Library files

EXTRA

- The linker also verifies if the symbols (functions and variables) that are needed by the program are completely satisfied.
  - A compiled executable does not include any shared libraries.
  - E.g. "**`gcc math.c -lm`**" is to link **`libm.so.6`**, the math library.

- When the program runs, the OS will link the ".so" files for you.
  - Try: the command "**`ldd`**".

How to compile multiple files?

EXTRA

- **`gcc`** by default hides all the intermediate steps.
  - Executable: "**`gcc -o hello hello.c`**" generates "**`hello`**" directly.
  - Object code: "**`gcc -c hello.c`**" generates "**`hello.o`**" directly.

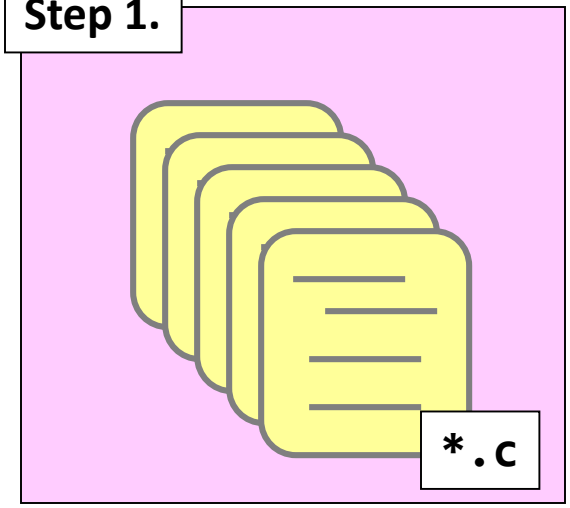
- How about working with multiple files?

How to compile multiple files?

EXTRA

Remember, below shows one of the solution.

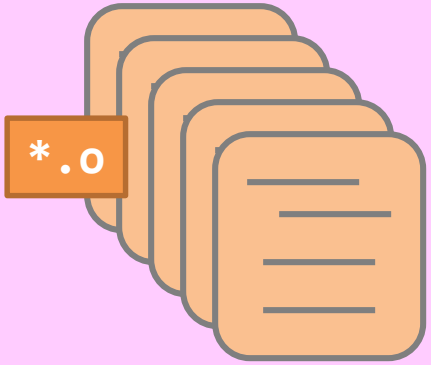
Step 1.



Prepare all the source files. **Important**: there must be one and only one file containing the main function.

Step 2.

```
$ gcc -c code.c
......
```



Compile them into object codes one by one.

Step 3.

```
$ gcc -o prog *.o
```

Construct the program together with all the object codes.

Compiling multiple files (preparation)

EXTRA

`add_int.c`

Definition of the function "**`add_int()`**" only.

`int_header.h`

Declaration of the function prototypes of "**`add_int()`**" and "**`multi_int()`**".

`multi_int.c`

Definition of the function "**`multi_int()`**" only.

`main.c`

Invoking "**`add_int()`**" and "**`multi_int()`**" in the "**`main()`**".

```
[example@3150]$ cat extra/*
```

Compiling multiple files

EXTRA

```
gcc -Wall   -c -o main.o main.c
main.c: In function ‘main’:
main.c:5: warning: implicit declaration of function ‘add_int’
main.c:6: warning: implicit declaration of function ‘multi_int’
```

main.c

```
#include <stdio.h>

int main(void) {
    printf(“3 + 10 = %d\n”, add_int  (3, 10) );
    printf(“3 * 10 = %d\n”, multi_int(3, 10) );
    return 0;
}
```

Since **add_int()** and **multi_add()** are **new vocabularies** to “**main.c**”, so we have to add the prototypes to “**main.c**”.

```
[example@3150]$ cat extra/*
```

Use of header file

EXTRA

int_header.h

```
int add_int(int a, int b);
int multi_int(int a, int b);
```

Do you still remember how the preprocessor works?

main.c

```
#include "int_header.h"
#include <stdio.h>

int main(void) {
    printf("3 + 10 = %d\n", add_int  (3, 10) );
    printf("3 * 10 = %d\n", multi_int(3, 10) );
    return 0;
}
```

```
[example@3150]$ cat extra/*
```

Function declaration VS definition

EXTRA

int_header.h

```
int add_int(int a, int b);
int multi_int(int a, int b);
```

The header file provides the declaration of a function.

It is the same as saying: "*there is a function named* ***add_int()***".

You don't have to provide the implementation in the header file.

add_int.c

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

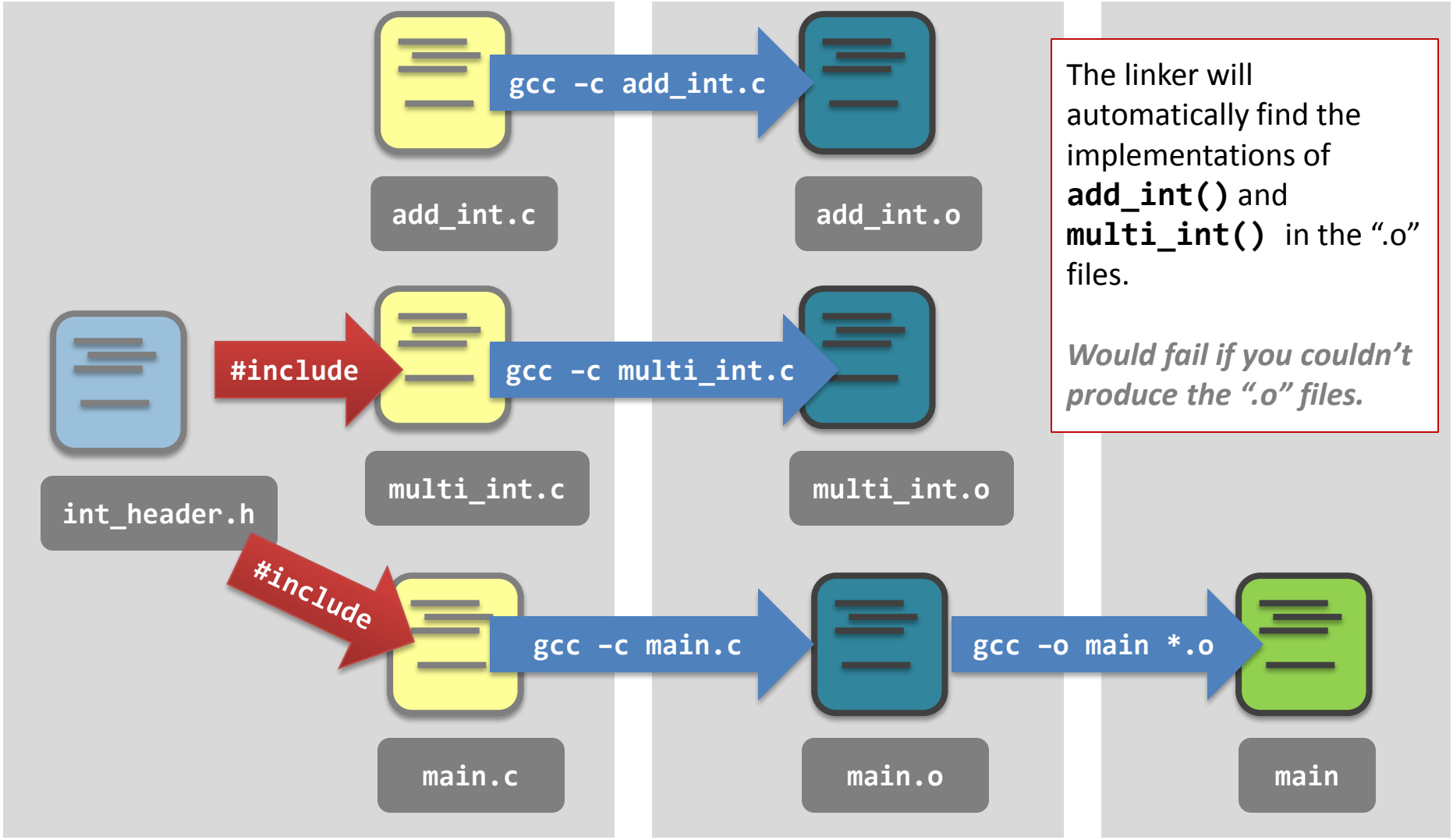
Implementation should be provided in program files.

Of course, as we know what the pre-processor will do, the declaration of the function will be written into the program file, with "**#include**".

```
[example@3150]$ cat extra/*
```

Finishing touch

EXTRA



The linker will automatically find the implementations of **`add_int()`** and **`multi_int()`** in the ".o" files.

Would fail if you couldn't produce the ".o" files.

```
[example@3150]$ cat extra/*
```

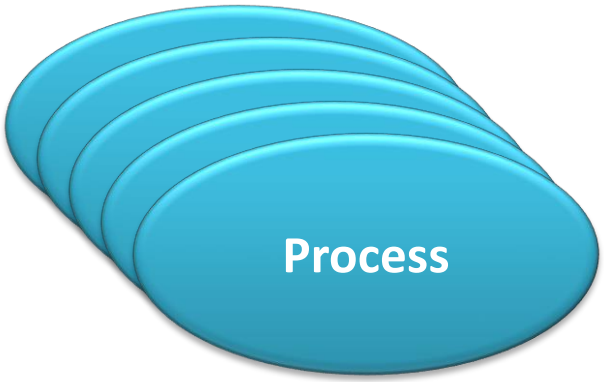
Conclusion on "*what is a program?*"

- A program is just a file!
  - It is static;
  - It may be associated with dynamically-linked files;
    - "*.so" in Linux and "*.dll" in Windows.

- It may be compiled from more than one files.

What is a process?

- process creation.



What is a process?

- Process is an irreplaceable of an OS.
  - It associates with all the files opened by that process.
  - It attaches to all the memory that is allocated for it.
  - It contains every accounting information,
    - running time, current memory usage, who owns the process, etc.
- You couldn't operate any things without processes.
  - So, this chapter is worth spending more than a month to cover.
- We start with some basic and important system calls.

What is a process?

- In order to understand the entire chapter, we will center around the following typical command:

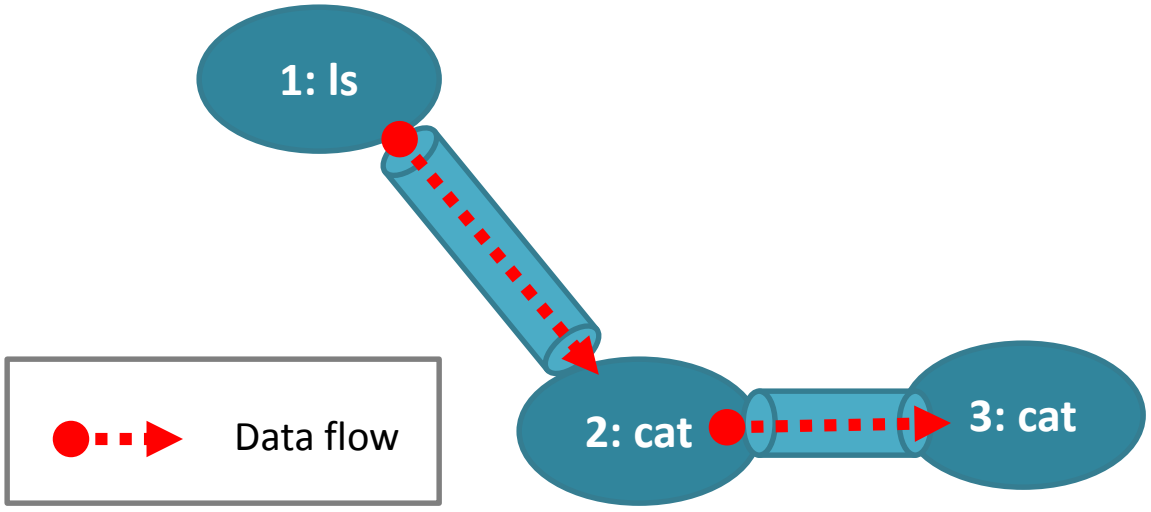
```
$ ls | cat | cat
[Ctrl + C]
$
```

- What is so special about this command:
  - The command involves **three processes**.
  - It will not stop until I send a **signal** to interrupt it.
  - Its progress is determined by the **process scheduler**.
  - The three processes **cooperate** and give useful output.

What is a process?

- What are those two “cats”!
  - 2 different processes using the same code “**`/bin/cat`**”.

```
$ ls | cat | cat
[Ctrl + C]
$
```



If you don’t know what a **`cat`** is.

```
#include <stdio.h>

int main(void) {
  int c;
  while ( 1 ) {
    c = getchar();
    if( c == EOF )
      break;
    putchar(c);
  }
}
```

What is a process?

- In this chapter, we will understand:

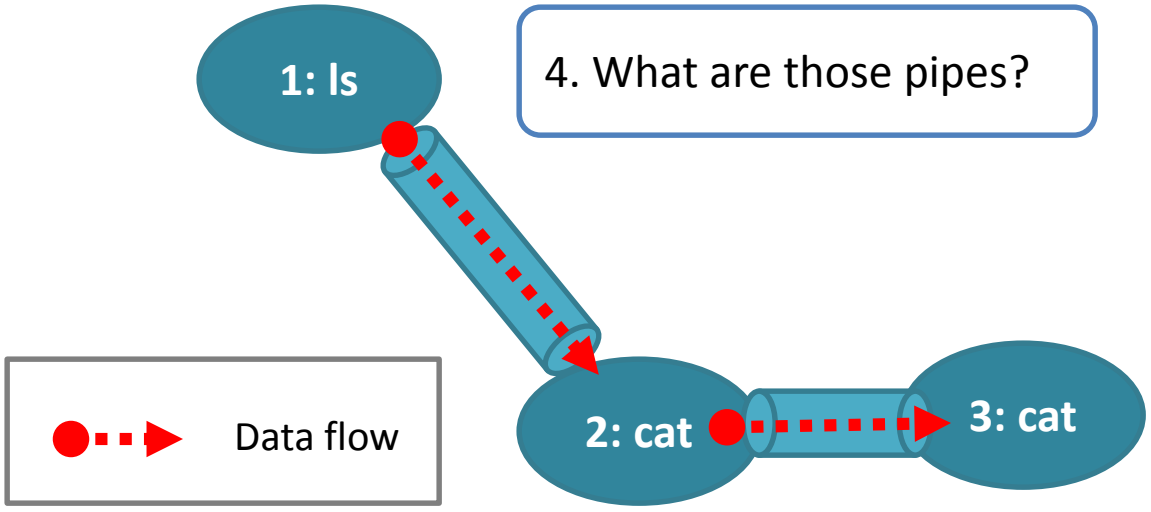
1. How to distinguish the two cats?

2. Who (and how to) create the processes?

3. Which should run first?

4. What are those pipes?

5. What if "**`ls`**" is feeding data too faster? Will the "**`cat`**" feels ***full and dies***?!



Process identification

- How can we identify processes from one to another?
  - Each process is given an unique ID number, and is called the **process ID**, or the **PID**.
  - The system call, **`getpid()`**, prints the PID of the calling process.

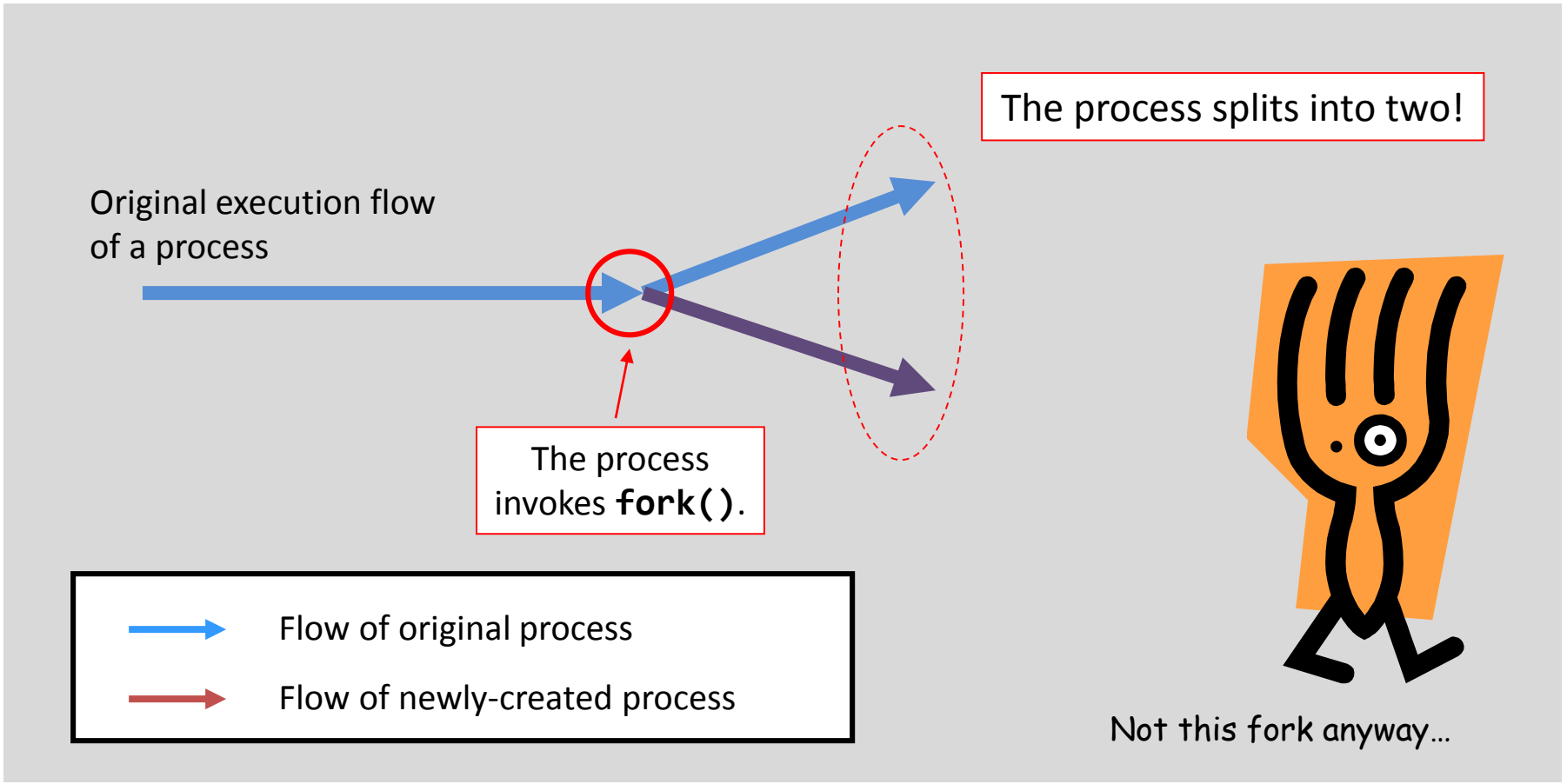
```
#include <stdio.h>   // printf()
#include <unistd.h>  // getpid()

int main(void) {
    printf("My PID is %d\n", getpid() );
}
```

```
$ ./getpid
My PID is 1234
$ ./getpid
My PID is 1235
$ ./getpid
My PID is 1237
```

```
[example@3150]$ cat process/getpid.c
```

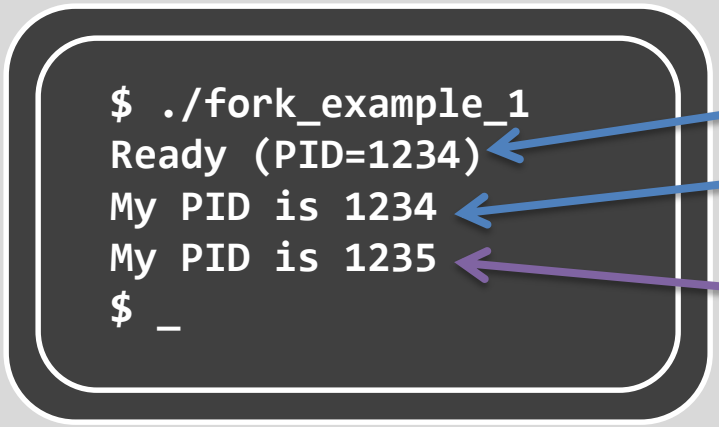
Process creation

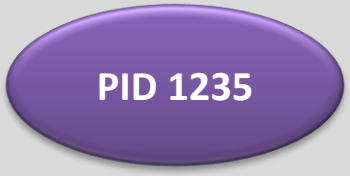
- To create a process, we use the system call **`fork()`**.
  - 分叉 in Chinese.



Process creation – **`fork()`** system call

- So, how do **`fork()`** and the processes behave?





Process 1234 is the original process, and we call it the **parent process**.

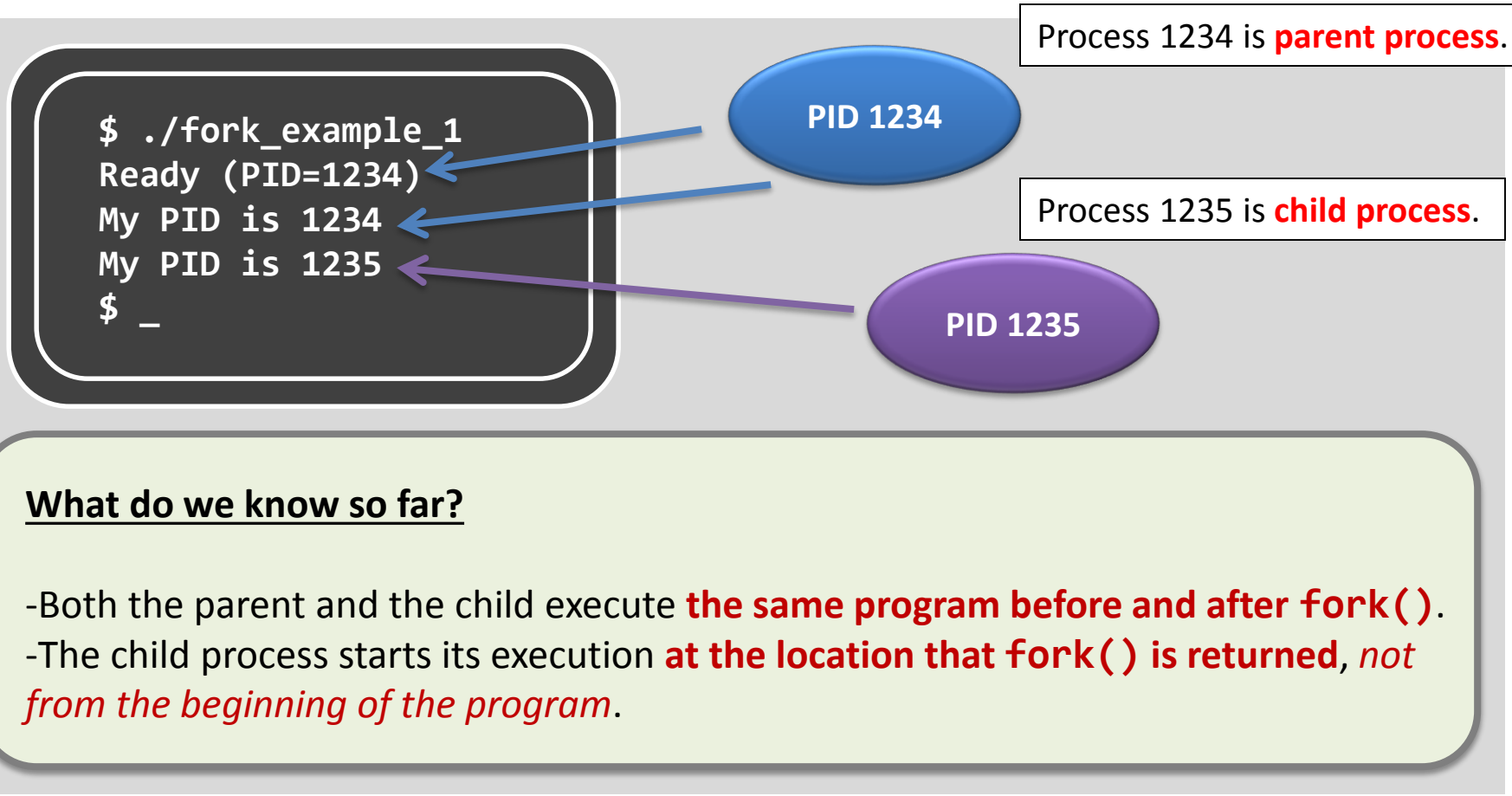
```
int main(void) {
    printf("Ready (PID = %d)\n", getpid());
    fork();
    printf("My PID is %d\n", getpid() );
    return 0;
}
```

Process 1235 is created by the **`fork()`** system call, and we call it the **child process**.

```
[example@3150]$ cat process/fork_example_1.c
```

Process creation – `fork()` system call

- So, how do `fork()` and the processes behave?



What do we know so far?

-Both the parent and the child execute **the same program before and after `fork()`**.
-The child process starts its execution **at the location that `fork()` is returned**, *not from the beginning of the program*.

```
[example@3150]$ cat process/fork_example_1.c
```

Process creation – **fork()** system call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
```

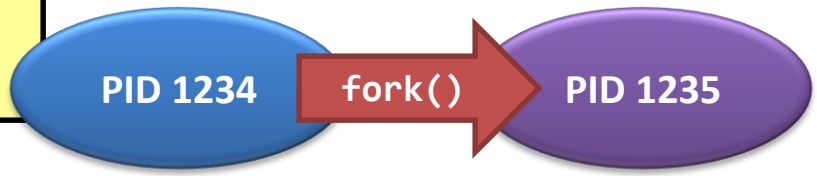
PID 1234

[example@3150]$ cat process/fork_example_2.c

Process creation – **fork()** system call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
```




```
[example@3150]$ cat process/fork_example_2.c
```

Process creation – `fork()` system call

Let there be only **ONE CPU**. Then…
- Only one process is allowed to be executed at one time.
- However, we can't predict which process will be chosen by the OS.
- By the time, this mechanism is called **process scheduling**.

Without loss of generality, can you imagine the case for multiple CPUs?

NOTE THIS

In this example, we assume that the parent, PID 1234, runs first, after the `fork()` call.

```
[example@3150]$ cat process/fork_example_2.c
```

Process creation – **fork()** system call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
```

Important

For parent, the return value of **fork()** is the PID of the created child.

PID 1234 (running)

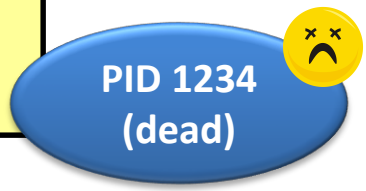
PID 1235 (waiting)

```
[example@3150]$ cat process/fork_example_2.c
```

Process creation – **fork()** system call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
```




```
[example@3150]$ cat process/fork_example_2.c
```

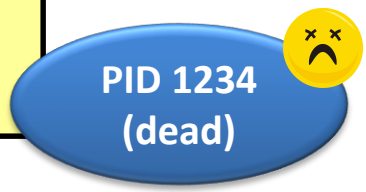
Process creation – **fork()** system call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
```

Important

For child, the return value of **fork()** is **0**.

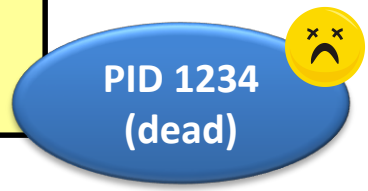


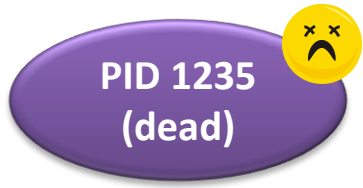

```
[example@3150]$ cat process/fork_example_2.c
```

Process creation – **fork()** system call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
I'm the child.
My PID is 1235
program terminated.
$ _
```






```
[example@3150]$ cat process/fork_example_2.c
```

Process creation – `fork()` system call

- **`fork()`** behaves like "*cell division*".
  - It creates the child process by **cloning** from the parent process, including…

Cloned items	Descriptions
Program counter [CPU register]	That's why they both execute from the same line of code after **`fork()`** returns.
Program code [File & Memory]	They are sharing the same piece of code.
Memory	Including local variables, global variables, and dynamically allocated memory.
Opened files [Kernel's internal]	If the parent has opened a file "A", then the child will also have file "A" opened automatically.

Process creation – `fork()` system call

- However…
  - `fork()` does not clone the following...
  - Note: they are all data inside the memory of kernel.

Distinct items	Parent	Child
Return value of `fork()`	PID of the child process.	0
PID	Unchanged.	Different, not necessarily be "Parent PID + 1"
Parent process	Unchanged.	Doesn't have the same parent as that of the parent process.
Running time	Cumulated.	Just created, so should be 0.
[Advanced] File locks	Unchanged.	None.

What is a process?

- process creation.
- **program execution.**

`fork()` can only duplicate…

- `fork()` is rather **boring**…
  - If a process can only duplicate itself and always runs the same program, then…
  - how can we execute other programs?

- We want **CHANGE!**
  - Meet the **`exec()`** system call family.

Program execution

- **`execl()`** – a member of the **`exec`** system call family (and the family has 6 members).

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

```
$ ./exec_example
before execl ...
```

Arguments of the `execl()` call

1st argument: the program name, "**`/bin/ls`**" in the example.
2nd argument: 1st argument to the program.
3rd argument: indicate the end of the list of arguments.

```
[example@3150]$ cat process/exec_example.c
```

Program execution

- **`execl()`** – a member of the **`exec`** system call family (and the family has 6 members).

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

```
$ ./exec_example
before execl ...
exec_example
exec_example.c
```

What is the output?

The same as **the output of running "ls" in the shell.**

```
[example@3150]$ cat process/exec_example.c
```

Program execution

- Example #1: run the command **"/bin/ls"**

```
execl("/bin/ls", "/bin/ls", NULL);
```

Argument Order	Value in above example	Description
1	**"/bin/ls"**	The file that the programmer wants to execute.
2	**"/bin/ls"**	When the process switches to **"/bin/ls"**, this string is the **first program argument**.
3	**NULL**	This states the end of the program argument list.

Program execution

- Example #2: run the command **"/bin/ls -l"**

```
execl("/bin/ls", "/bin/ls", "-l", NULL);
```

Argument Order	Value in above example	Description
1	**"/bin/ls"**	The file that the programmer wants to execute.
2	**"/bin/ls"**	When the process switches to "/bin/ls", this string is the **first program argument**.
3	**"-l"**	When the process switches to "/bin/ls", this string is the **second program argument**.
4	**NULL**	This states the end of the program argument list.

Program execution

- **`execl()`** – a member of the **`exec`** system call family (and the family has 6 me

WHAT?!
The shell prompt appears!

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

```
$ ./exec_example
before execl ...
exec_example
exec_example.c
$ _
```

The output says:
(1) The gray code block **is not reached**!
(2) The process is **terminated**!

WHY IS THAT?!

```
[example@3150]$ cat process/exec_example.c
```

Program execution

- The **exec** system call family is not simply a function that “invokes” a command.

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

Originally, the process is executing the program “**exec_example**”.


```
[example@3150]$ cat process/exec_example.c
```

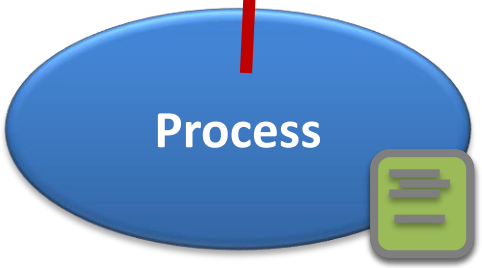
Program execution

- The **exec** system call family is not simply a function that “invokes” a command.

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

The execl() call change the execution from “**exec_example**” to “**/bin/ls**”

```
/* The program “ls” */

int main(int argc, char ** argv)
{
    ......
    exit(0);
}
```




```
[example@3150]$ cat process/exec_example.c
```

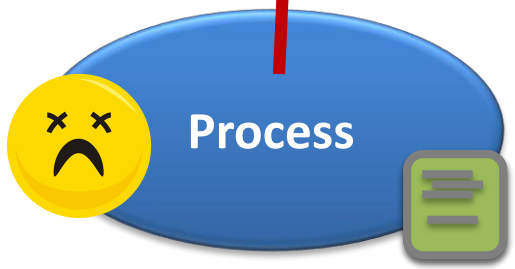
Program execution

- The **exec** system call family is not simply a function that “invokes” a command.

The “**return**” or the “**exit()**” statement in “**/bin/ls**” will terminate the process…

Therefore, it is certain that the process cannot go back to the old program!

```
/* The program “ls” */

int main(int argc, char ** argv)
{
    ......
    exit(0);
}
```




```
[example@3150]$ cat process/exec_example.c
```

Program execution - observation

- The process is changing the code that is executing and **never returns to the original code**.
  - The last two lines of codes are therefore not executed.

- The process that calls any one of the member of the exec system call family will **throw away** many things, e.g.,
  - Memory: local variables, global variables, and dynamically allocated memory;
  - Register value: e.g., the program counter;

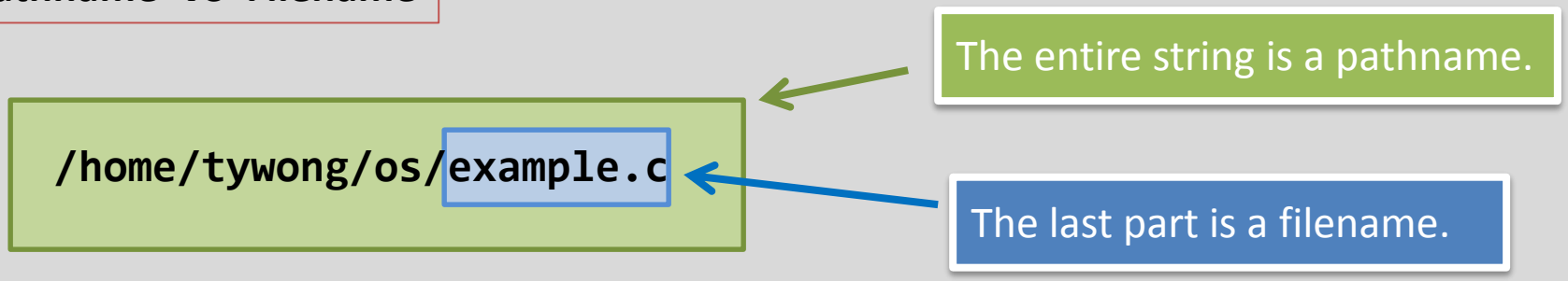
- But, the process will **preserve** something, including:
  - PID;
  - Process relationship;
  - Running time, etc.

exec system call family members

Member name	Using pathname	Using filename	Argument List	Argument Array	Original ENV	Provided ENV
`execl()`	YES		YES		YES	
`execlp()`		YES	YES		YES	
`execle()`	YES		YES			YES
`execv()`	YES			YES	YES	
`execvp()`		YES		YES	YES	
`execve()`	YES			YES		YES
Alphabet used in name		**P**	**l**	**v**		**e**

exec*() – arguments explained

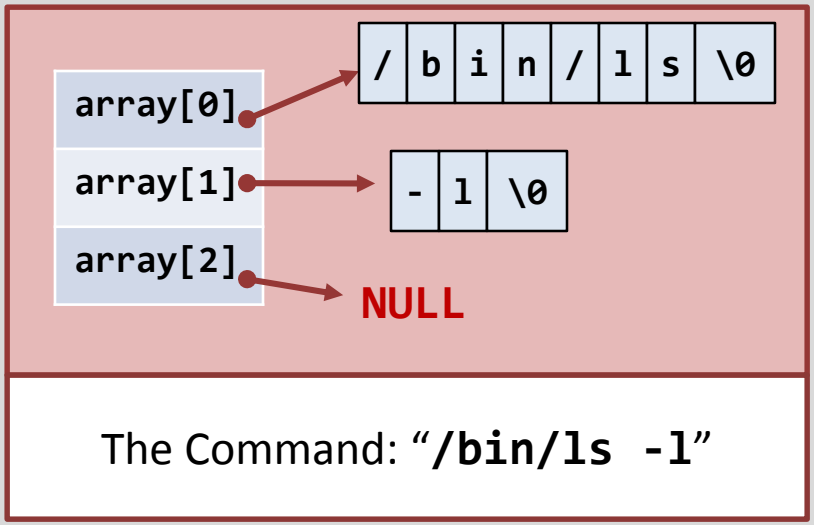
Pathname VS Filename



Argument list VS Argument array

```
execv("/bin/ls", array);
```

```
execl("/bin/ls",
"/bin/ls", "-l", NULL);
```



exec*() – arguments explained

EXTRA

- Environment variables
  - A set of strings maintained by the shell.

```
int main(int argc, char **argv, char **envp) {
    int i;
    for(i = 0; envp[i]; i++)
        printf("%s\n", envp[i]);
    return 0;
}
```

```
$ ./envp
SHELL=/bin/bash
PATH=......
......
$ _
```

The "****envp**" variable is an array, arranged in the same manner as the argument array discussed previously.

```
[example@3150]$ cat process/envp.c
```

exec*() – arguments explained

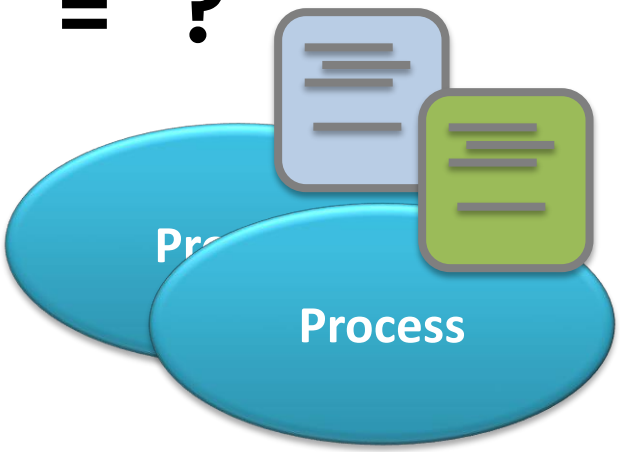
EXTRA

- Environment variables
  - A set of strings maintained by the shell.
  - Quite a number of programs will read and make use of the environment variable.

Variable name	Description
SHELL	The path to the shell that you're using.
PWD	The full path to the directory that you're currently on.
HOME	The full path to your home directory.
USER	Your login name.
EDITOR	Your default text editor.
PRINTER	Your default printer.

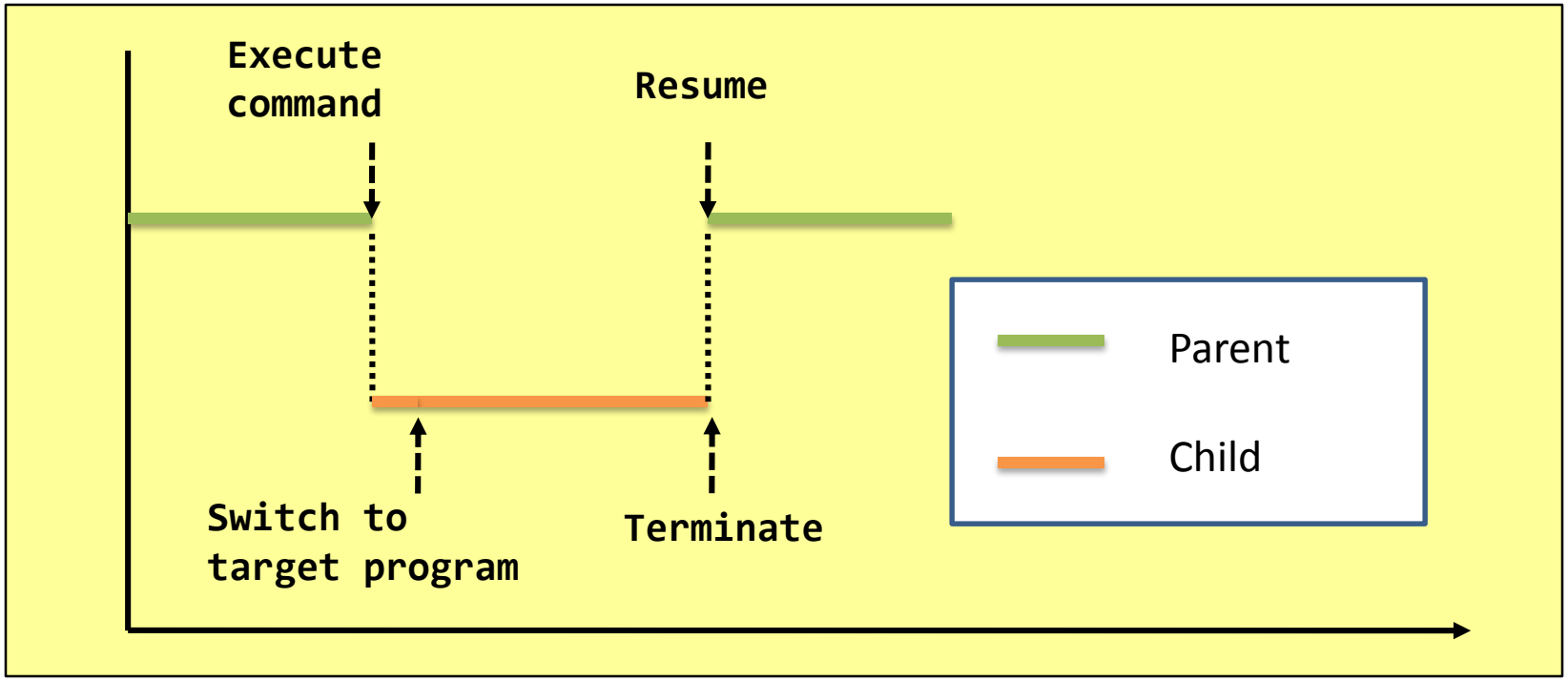
What is a process?

- process creation.
- program execution.
- `fork() + exec*() = ?`



When **fork()** meets **exec*()**…

- The mix can become:
  - A shell,
  - The **system()** library call, etc…




```
[examples@3150] cat system_example.c
```

fork() + exec*() = system()?

```
 1  int system_ver_3150(const char *cmd_str) {
 2      if(cmd_str == -1)
 3          return -1;
 4      if(fork() == 0) {
 5          execl(cmd_str, cmd_str, NULL);
 6          fprintf(stderr,
                "%s: command not found\n", cmd_str);
 7          exit(-1);
 8      }
 9      return 0;
10  }
11
12  int main(void) {
13      printf("before...\n\n");
14      system_ver_3150("/bin/ls");
15      printf("\nafter...\n");
16      return 0;
17  }
```

```
$ ./system_implement_1
before...

system_implement_1
system_implement_1.c

after...
$ _
```

[example@3150]$ cat process/system_implement_1.c

fork() + exec*() = system()?!

```
 1  int system_ver_3150(const char *cmd_str) {
 2      if(cmd_str == -1)
 3          return -1;
 4      if(fork() == 0) {
 5          execl(cmd_str, cmd_str, NULL);
 6          fprintf(stderr,
                "%s: command not found\n", cmd_str);
 7          exit(-1);
 8      }
 9      return 0;
10  }
11
12  int main(void) {
13      printf("before...\n\n");
14      system_ver_3150("/bin/ls");
15      printf("\nafter...\n");
16      return 0;
17  }
```

Some strange cases happened when the program is **executed repeatedly**!!

```
$ ./system_implement_1
before...

after...
system_implement_1
system_implement_1.c
$ _
```

[example@3150]$ cat process/system_implement_1.c

fork() + exec*() = system()...

```
 1  int system_ver_3150(const char *cmd_str) {
 2      if(cmd_str == -1)
 3          return -1;
 4      if(fork() == 0) {
 5          execl(cmd_str, cmd_str, NULL);
 6          fprintf(stderr,
                "%s: command not found\n", cmd_str);
 7          exit(-1);
 8      }
 9      return 0;
10  }
11
12  int main(void) {
13      printf("before...\n\n");
14      system_ver_3150("/bin/ls");
15      printf("\nafter...\n");
16      return 0;
17  }
```

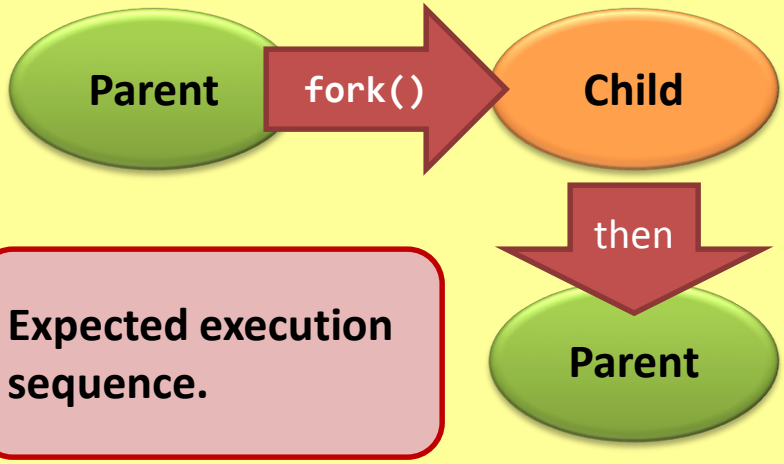
Let's re-color the program!

- Parent process
- Child process
- Both processes

```
$ ./system_implement_1
before...

after...
system_implement_1
system_implement_1.c
$ _
```

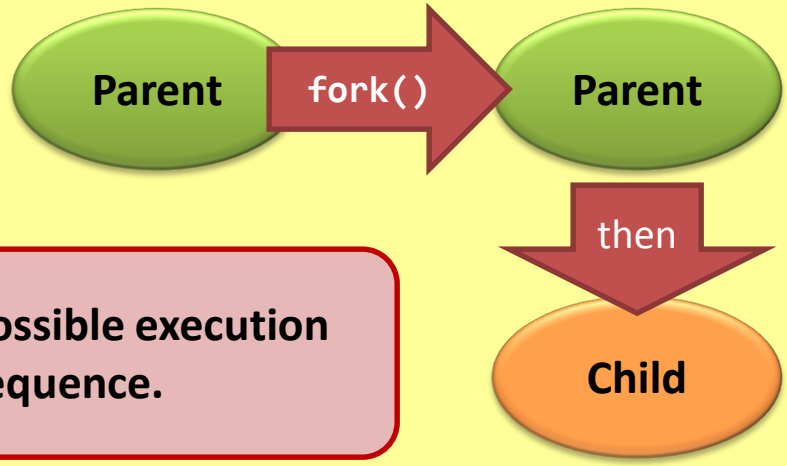
```
[example@3150]$ cat process/system_implement_1.c
```

fork() + exec*() = system()...



Expected execution sequence.

```
$ ./system_implement_1
before...

system_implement_1
System_implement_1.c

after...
$ _
```



Possible execution sequence.

```
$ ./system_implement_1
before...

after...
system_implement_1
System_implement_1.c
$ _
```

fork() + exec*() = system()...

- Don't forget that we're trying to implement a **system()**-compatible function…
  - It is very weird to allow different execution orders.

- Our problem is: how to let the child to execute first?
  - But…we can't control the **process scheduling** of the OS to this extent.

- Then, our problem becomes…
  - How to suspend the execution of the parent process?
  - How to wake the parent up after the child is terminated?

fork()+ exec*() + wait() = system()

```
 1  int system_ver_3150(const char *cmd_str) {
 2      if(cmd_str == -1)
 3          return -1;
 4      if(fork() == 0) {
 5          execl("/bin/sh", "/bin/sh",
                  "-c", cmd_str, NULL);
 6          fprintf(stderr,
                "%s: command not found\n", cmd_str);
 7          exit(-1);
 8      }
 9      wait(NULL);
10      return 0;
11  }
12
13  int main(void) {
14      printf("before...\n\n");
15      system_ver_3150("/bin/ls");
16      printf("\nafter...\n");
17      return 0;
18  }
```

kernel service, system call

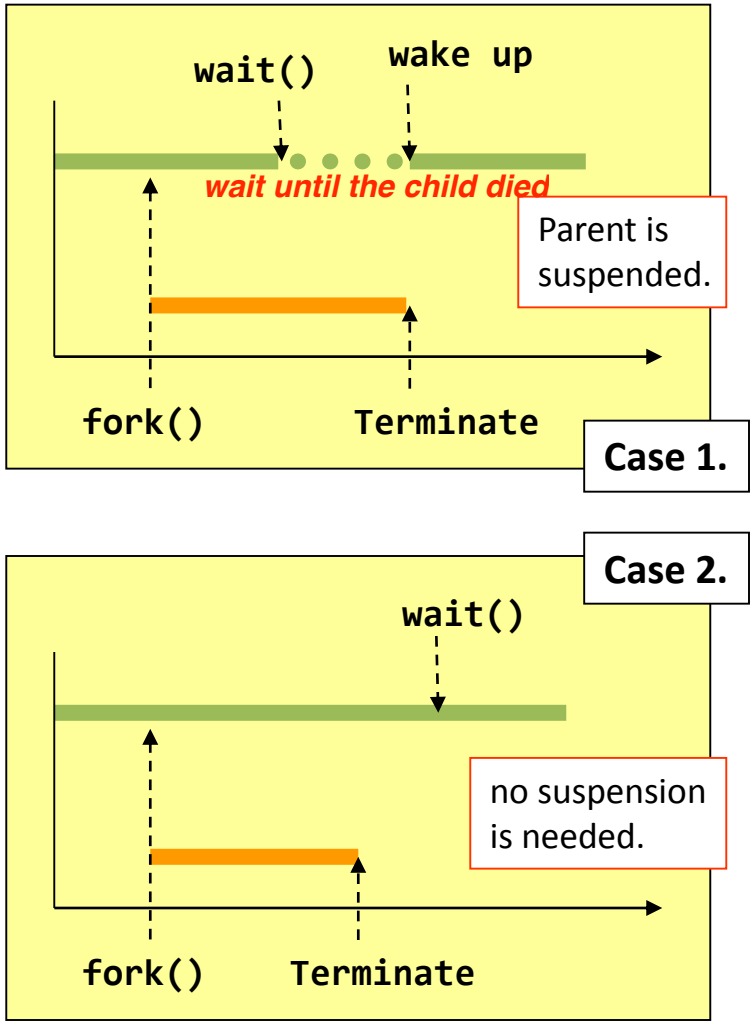
```
$ ./system_implement_2
before...

system_implement_2
System_implement_2.c

after...
$ _
```

[example@3150]$ cat process/system_implement_2.c

`wait()` – properties explained

- The **`wait()`** system call **suspend** the calling parent process (Case 1).
- **`wait()`** returns and wakes up the calling process when the one of its child processes changes from **running to terminated**.
- **`wait()`** does not suspend the calling process (Case 2) if
  - There were no running children;
  - There were no children;

  (the above two cases are different)



wait() VS waitpid()

wait()	**waitpid()**
Wait for any one of the children.	Depending on the parameters, **waitpid()** will wait for a particular child only.
Detect child termination only.	Depending on the parameters, **waitpid()** **can detect child's status changing**: -from running to suspended, and -from suspended to running.

For more details and the good of your assignment, you **must read** the man pages of **wait()** and **waitpid()**.

Reading man pages

EXTRA

- Man pages are of the vital importance to programmers working under Unix, Linux, MacOS, etc.

- It includes the following important information:
  - Header file(s) to be included;
  - Compiler flags to be added;
  - Input arguments;
  - Return values; and
  - Error conditions.

Reading man page – example 1

EXTRA

```
[tywong@linux ~]$ man pow

WAIT(3)                    Linux Programmer's Manual                    WAIT(3)

NAME
       wait, waitpid, waitid - wait for process to change state

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status);

       pid_t waitpid(pid_t pid, int *status, int options);

       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

......
DESCRIPTION
......
RETURN VALUE
......
ERRORS
......
```

The header file needed when compiling.

Reading man page – example 2

EXTRA

```
[tywong@linux ~]$ man pow

POW(3)                    Linux Programmer's Manual                    POW(3)

NAME
       pow, powf, powl - power functions

SYNOPSIS
       #include <math.h>

       double pow(double x, double y);
       float powf(float x, float y);
       long double powl(long double x, long double y);

       Link with -lm.

......
DESCRIPTION
......
RETURN VALUE
......
ERRORS
......
```

BTW, what does this number mean?

This is why we need "`-lm` " when compiling programs with math functions.

Reading man page - sections

EXTRA

- Man pages are divided into sections, and they are defined in "**`man man`**".
  - E.g., "**`man printf`**" is the same as "**`man 1 printf`**"
    - It means the *shell command* "**`printf`**", not the library call "**`printf()`**".
    - "**`man 3 printf`**" is to read the manual page of library call **`printf()`**.

Section	Description
1	Executable programs or shell commands.
2	System calls
3	Library calls
…	…
7	Miscellaneous (including macro packages and conventions)

Summary

- A process is created by cloning.
  - **`fork()`** is the system call that clones processes.
  - Cloning is copying: the new process inherits many things from the parent process.
  - (But, where is the first process?)

- Program execution is not trivial.
  - A process is the place that hosts a program and run it.
  - **exec** system call family changes the program that a process is running.
  - A process can run more than one program…
    - as long as there is a set of programs that keeps on calling the **exec** system call family.