

OS Lecture 3

OS Lecture 3

OS Course Structures (Cont.)

Security and Protection

What will we learn?

Process Management

Process \neq program

What is a program?

Life of a C program

Preprocessor

Compiler

Assembler

Linker

Processes

Process identification: `getpid()` and `getppid()`

Process creation: `fork()`

`fork()` Behaves like "cloning."

OS Course Structures (Cont.)

OS abstractions

i.e., course structure

Processes

- Part 2: Process management
- Part 3: Process synchronization

Files

- Part 5: Storage management
- Part 6: File system

Address spaces

- Part 4: Memory management

Security and protection

- Part 7: Security and protection

• Security and Protection

The OS needs to **control the access** of processes or users **to the resources** defined by the system.

It must provide means to...

- **Specify** the controls to be imposed.
- **Enforce** the controls.

Have you heard of the following **attacks**?

- Viruses, worms, denial-of-service (DOS) attacks, identity theft, theft of service, ...

– What will we learn?

- **Protection-related functions and syscalls**
 - How does the OS control access to resources?
 - What do file permissions mean?
- **Security best practices**
 - How to store and verify the user password?
- **Hacking**
 - How to gain the root privilege legally and illegally?
 - How to protect yourself from being hacked?

Process Management

• Process ≠ program

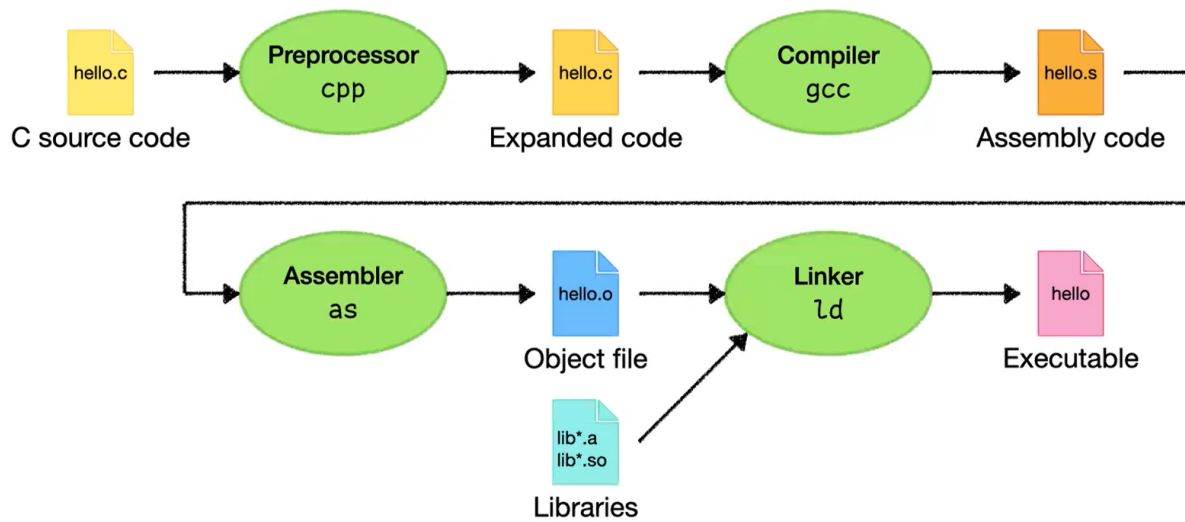
– What is a program?

A **program** is just a piece of **code**.

But... *which* code do you mean?

- **High-level language** (C, C++, Java...)
- **Intermediate language** (Java bytecode, LLVM IR, .NET CIL...)
- **Low-level language** (assembly)
- **Machine code**

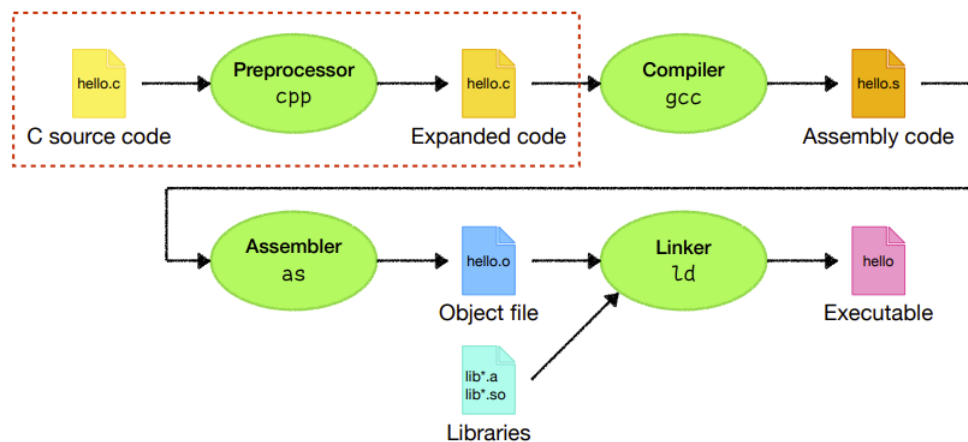
Life of a C program



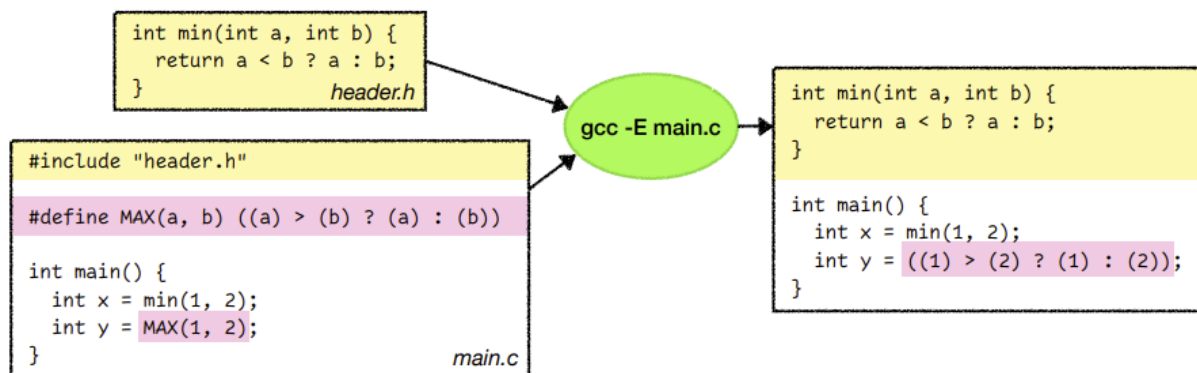
“

When you run the gcc, it compiles the code and also invokes the cpp, assembler and linker

- **Preprocessor**



- The **preprocessor** expands **directives** such as `#include` , `#define` , `#ifdef` ...



- Example:

```
[yt2475@linserv1 proc]$ cat main.c
#include "header.h"

#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {
    int x = min(1, 2);
    int y = MAX(1, 2);
}

[yt2475@linserv1 proc]$ cat header.h
int min(int a, int b) {
    return a < b ? a : b;
}
```

Preprocessor:

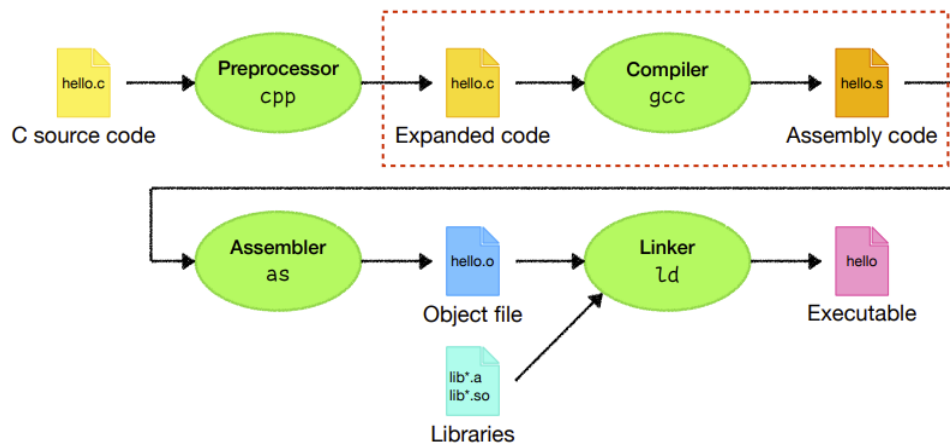
```
1 gcc -E main.C
```

Result:

```
[yt2475@linserv1 proc]$ gcc -E main.c
# 0 "main.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "main.c"
# 1 "header.h" 1
int min(int a, int b) {
    return a < b ? a : b;
}
# 2 "main.c" 2

int main() {
    int x = min(1, 2);
    int y = ((1) > (2) ? (1) : (2));
}
```

- Compiler



The **compiler** takes the expanded C code, checks the syntax, and generates...

- Assembly code (gcc)
- LLVM IR (clang)

In the meantime, it also **optimizes** the code.



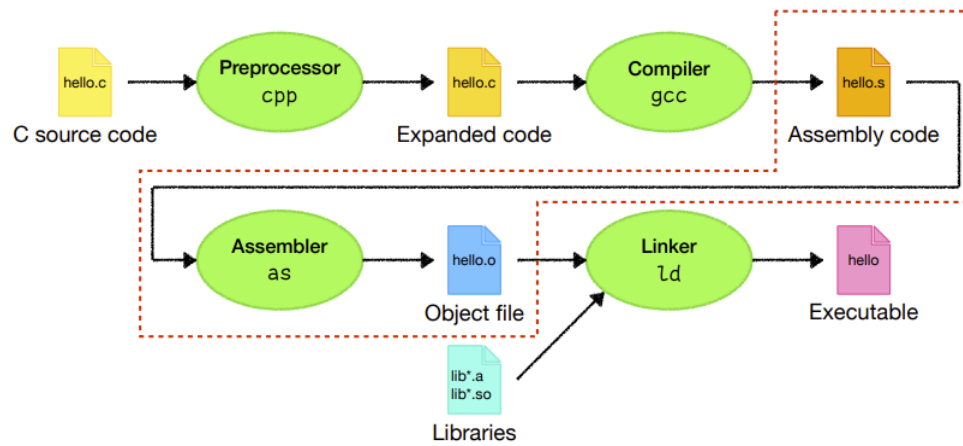
“

-O2 is the code for optimization.

There're different levels of optimization

Why not higher level? O3 might cause bugs.

- Assembler

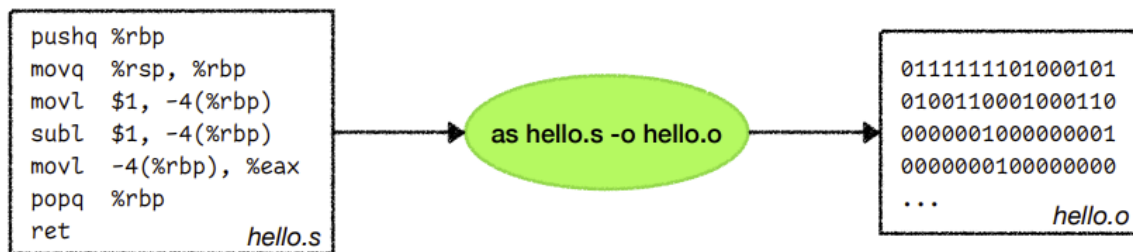


The **assembler** converts the generated assembly code to an **object file**.

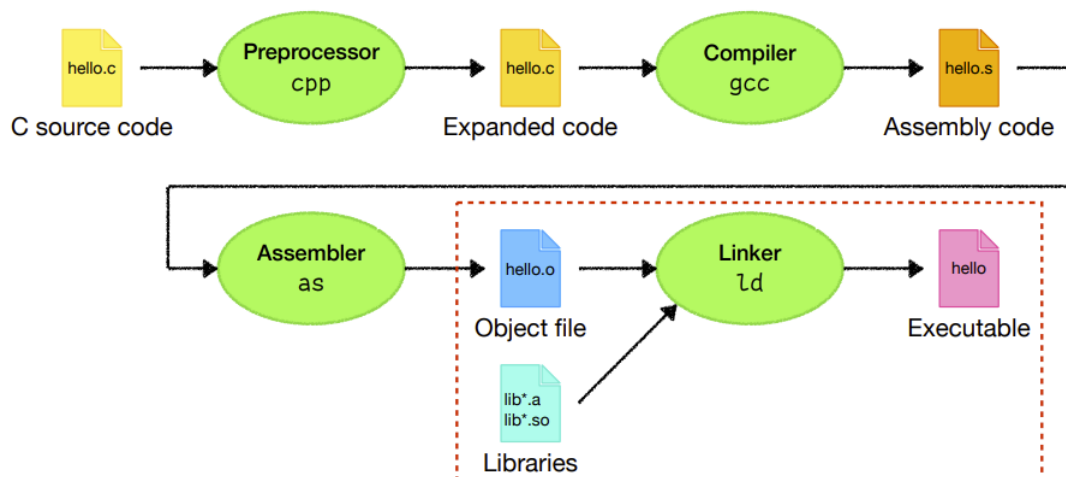
The object file contains **machine code**, but **isn't yet executable**.

“

It may contains functions from other libraries, so you must link them together before execution.

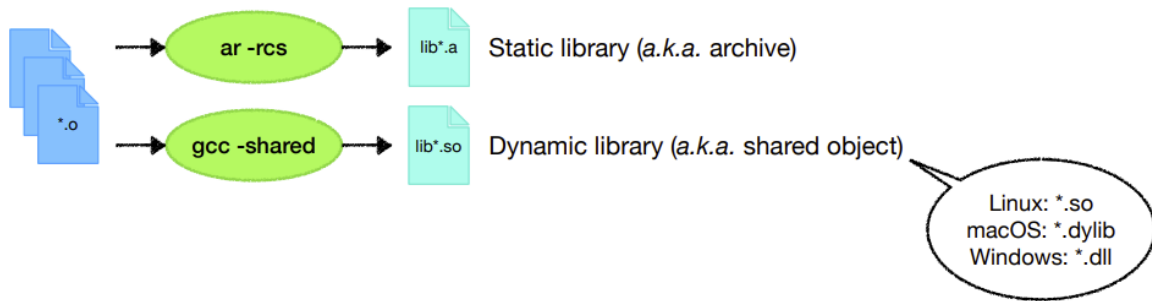


• Linker

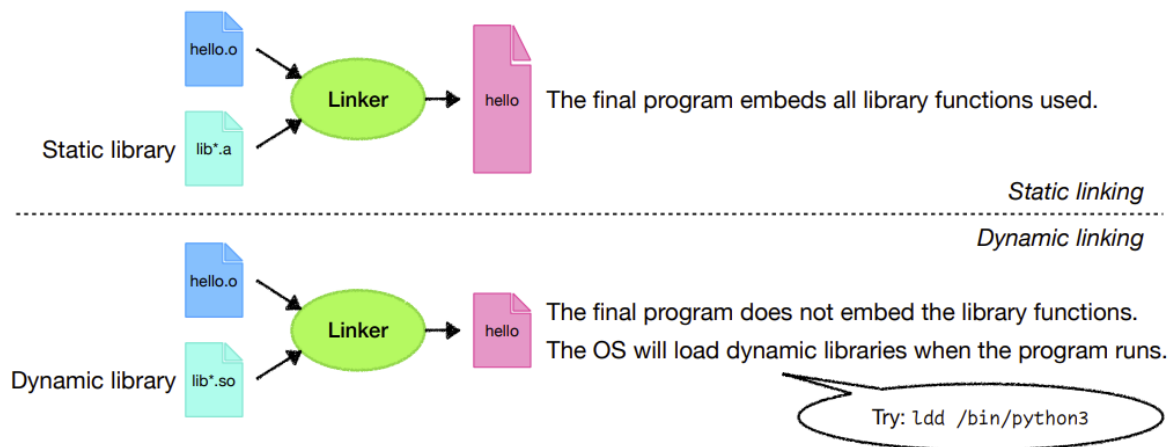


The **linker** combines **object files** and **libraries** to produce the **executable file**.

A **library** is just a collection of functions and variables.



- What's the difference between Static Library and Dynamic Library?
- Static & Dynamic linking



- The dynamic linking is the default one, because it saves space, reuses codes, and is easy to update if there's any bugs in the library
- The static linked file is useful for embedded systems
- The Dynamic Library and the Static Library has different formats (`.a` and `.so`), so you can't replace one with another

Processes

The **process** is the most central concept in an operating system.

- It's an abstraction of a **running program**.
- It attaches to all the **memory** that is allocated for the process.
- It associates with all the **files** opened by the process.
- It contains **accounting information** such as its owner, running time, memory usage...

Let's start with some system calls.

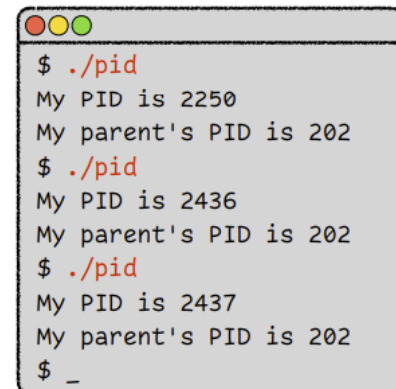
- Process identification: `getpid()` and `getppid()`

The OS gives each process a unique identification number, the **Process ID (PID)**.

- `getpid()` returns the PID of the calling process.
- `getppid()` returns the PID of the parent of the calling process.

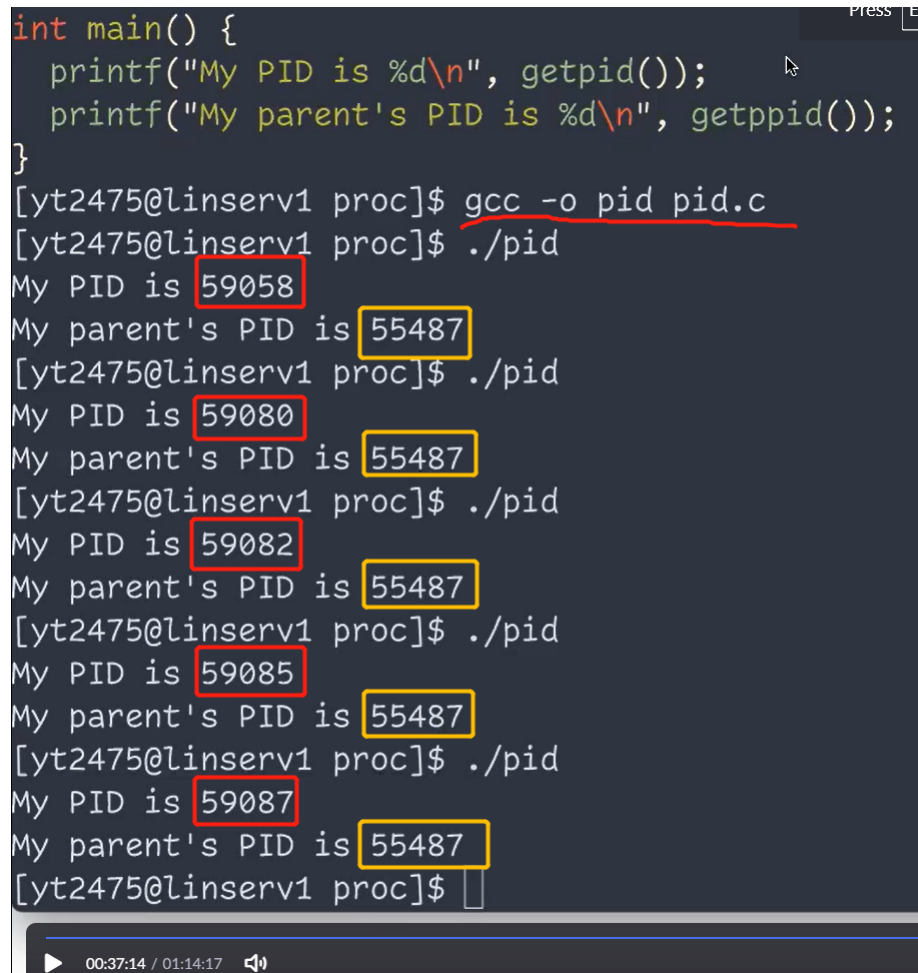
```
int main() {  
    printf("My PID is %d\n", getpid());  
    printf("My parent's PID is %d\n", getppid());  
}
```

pid.c



```
$ ./pid  
My PID is 2250  
My parent's PID is 202  
$ ./pid  
My PID is 2436  
My parent's PID is 202  
$ ./pid  
My PID is 2437  
My parent's PID is 202  
$ _
```

- Example:



```
int main() {  
    printf("My PID is %d\n", getpid());  
    printf("My parent's PID is %d\n", getppid());  
}  
[yt2475@linserv1 proc]$ gcc -o pid pid.c  
[yt2475@linserv1 proc]$ ./pid  
My PID is 59058  
My parent's PID is 55487  
[yt2475@linserv1 proc]$ ./pid  
My PID is 59080  
My parent's PID is 55487  
[yt2475@linserv1 proc]$ ./pid  
My PID is 59082  
My parent's PID is 55487  
[yt2475@linserv1 proc]$ ./pid  
My PID is 59085  
My parent's PID is 55487  
[yt2475@linserv1 proc]$ ./pid  
My PID is 59087  
My parent's PID is 55487  
[yt2475@linserv1 proc]$
```

“

Note that the PID each time is different, but the PPID remains the same.

This is because we launch a new process every time we run the program.

But the parent process is always the shell, and we didn't start a new session for the shell.

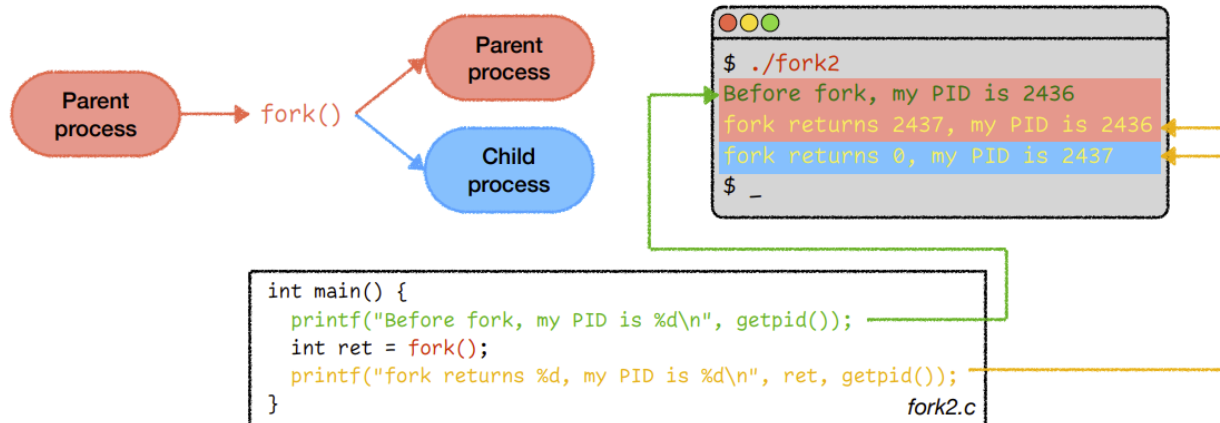
- Process creation: **fork()**

fork() 's return value differs for the parent and the child.

- In the **parent**, **fork()** returns the PID of the child process.
- In the **child**, **fork()** returns 0.

Process creation

fork()



How do we know the process is the parent or child?

- return value of **fork()** is different
- Example:

```
[yt2475@linserv1 proc]$ cat fork2.c  
#include <stdio.h>  
#include <unistd.h>  
  
int main() {  
    printf("Before fork, my PID is %d\n", getpid());  
    int ret = fork();  
    printf("fork returns %d, my PID is %d\n", ret, getpid());  
}
```

```
[yt2475@linserv1 proc]$ gcc -o fork2 fork2.c  
[yt2475@linserv1 proc]$ ./fork2  
Before fork, my PID is 59685  
fork returns 59686, my PID is 59685  
fork returns 0, my PID is 59686
```

- `fork()` Behaves like "cloning."

The child **inherits** (but is independent from) the parent's...

- Program code
 - Both the parent and child share the same code.
- Program counter
 - Therefore, both the parent and the child execute from the same location after `fork()`.
- Memory
 - This includes global variables, local variables, and dynamically allocated memory.
- Opened files
 - If the parent has opened a file, then the child also has the same file opened.

However, the child **differs** from the parent in a few things...

- Return value of `fork()`
 - The parent returns the PID of the child, or -1 if `fork()` fails. The child returns 0.
- Process ID
 - The child gets a new PID, which is **not necessarily** the parent's PID + 1.
- Parent
 - The child process's parent is the parent process, not the grandparent.
- Running time
 - The child's running time is reset to 0.
- File locks
 - The child does not inherit file locks from its parent.

Challenge: what will be the output?

```
int main() {  
    printf("Hello ");  
    fork();  
    printf("CS202\n");  
}
```

fork_buffer.c

Process creation

`fork()` behaves like “cloning.”

Challenge: what will be the output?

```
int main() {  
    printf("Hello ");  
    fork();  
    printf("CS202\n");  
}
```

fork_buffer.c

The library function `printf()` invokes the `write()` system call.

There is a **buffer** in the **FILE** structure to reduce the number of system calls.

At `fork()`, the child **inherits the buffer**.

Unbuffered: invoke `write()` **immediately**.

- `stderr` is **unbuffered** by default.

Line-buffered: write data to the **buffer**.

Invoke `write()` when a **newline character** is encountered.

- `stdin` and `stdout` are **line-buffered** by default.

Fully-buffered: write data to the **buffer**.

Invoke `write()` when the **buffer becomes full** or before the **process terminates**.

You can call `setvbuf()` to **change the buffering strategy** or call `fflush()` to **force a write**.

```
$ ./fork_buffer  
Hello CS202  
Hello CS202  
$ _
```



```
int main() {  
    printf("Hello ");  
    fflush(stdout);  
    fork();  
    printf("CS202\n");  
}
```

fork_buffer2.c

```
$ ./fork_buffer2  
Hello CS202  
CS202  
$ _
```

“

At `printf("Hello")`, the `Hello` is not immediately printed. It's saved to a buffer, which was cloned by the forked process

- Use `setvbuf()` or `fflush()` to set the buffer
- `fflush()` force the output to standard output

```
[yt2475@linserv1 proc]$ cat fork_buffer2.c  
#include <stdio.h>  
#include <unistd.h>  
  
int main() {  
    printf("Hello ");  
    fflush(stdout);  
    fork();  
    printf("CS202\n");  
}  
[yt2475@linserv1 proc]$ gcc -o fork_buffer2 fork_buffer2.c  
[yt2475@linserv1 proc]$ ./fork_buffer2  
Hello CS202  
CS202
```