OS: Lecture 7

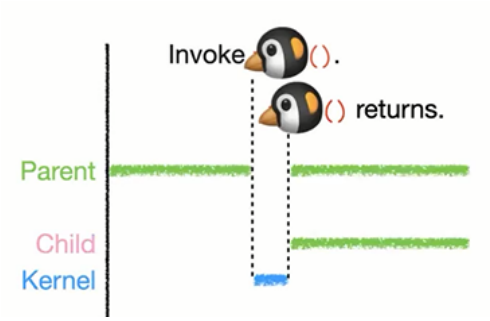
OS: Lecture 7

- Review
  - System calls for process management
    `Fork()`
- Process Organization
  - Booting the computer
    - BIOS: Basic input/output system
    - Boot device
    - Boot loader
  - The first process - `init` (Linux)
  - Orphans
    - Reparenting
    - What happens in the kernel?
    - Will `init` clear the zombie?
    - How does `init` clear the zombie?
  - Some observations
- Process scheduling
  - What and Why
  - Process states
    - Ready
    - Running
    - Blocked
    - Zombie
  - Context switching
- Homework
  - Question 1
  - Undefined behavior

Review

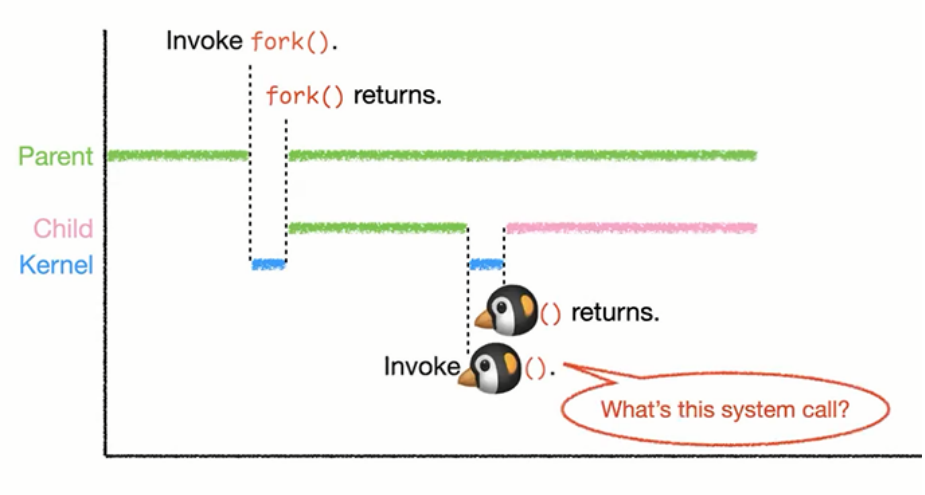
- System calls for process management

1. What's 🐧?



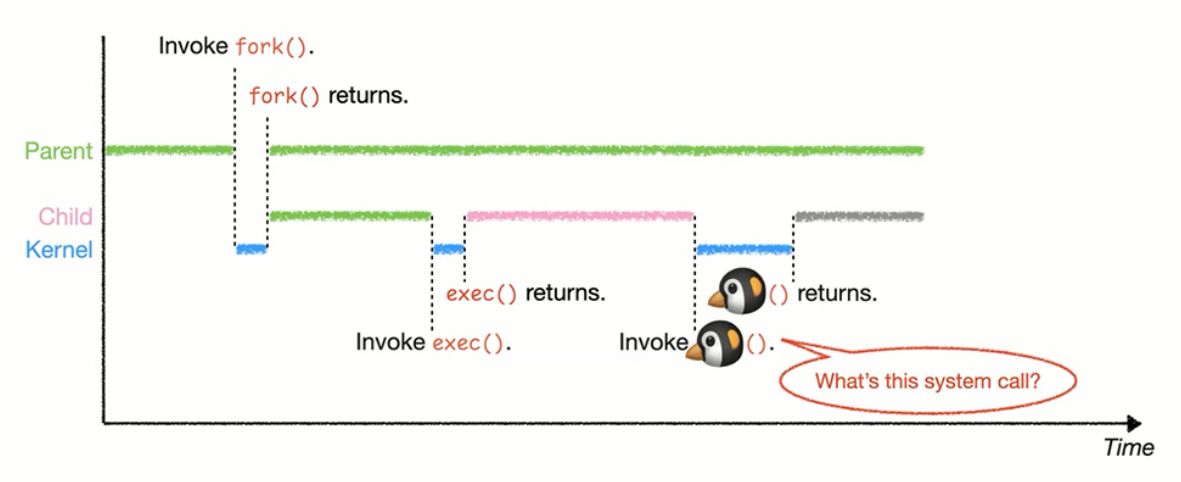
   - It's `fork()`!

2. What's 🐧?



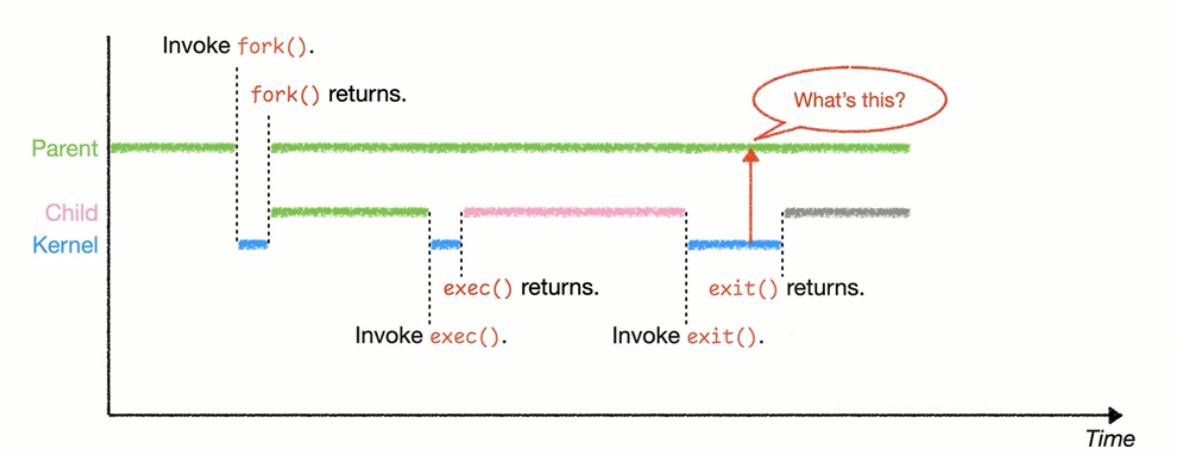
   - It's `exec()`!

3. What's 🐧?



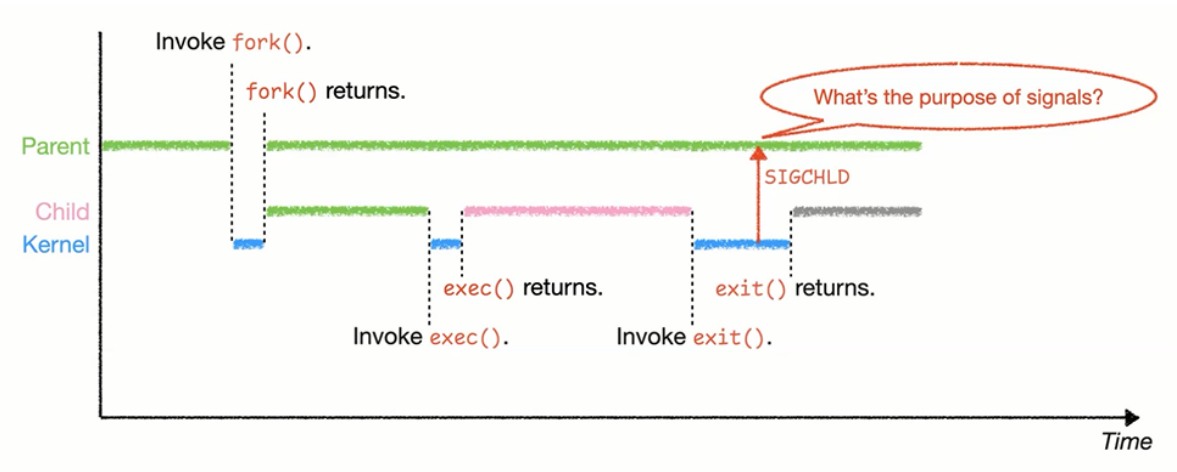
   - It's `exit()`!

4. What's this?



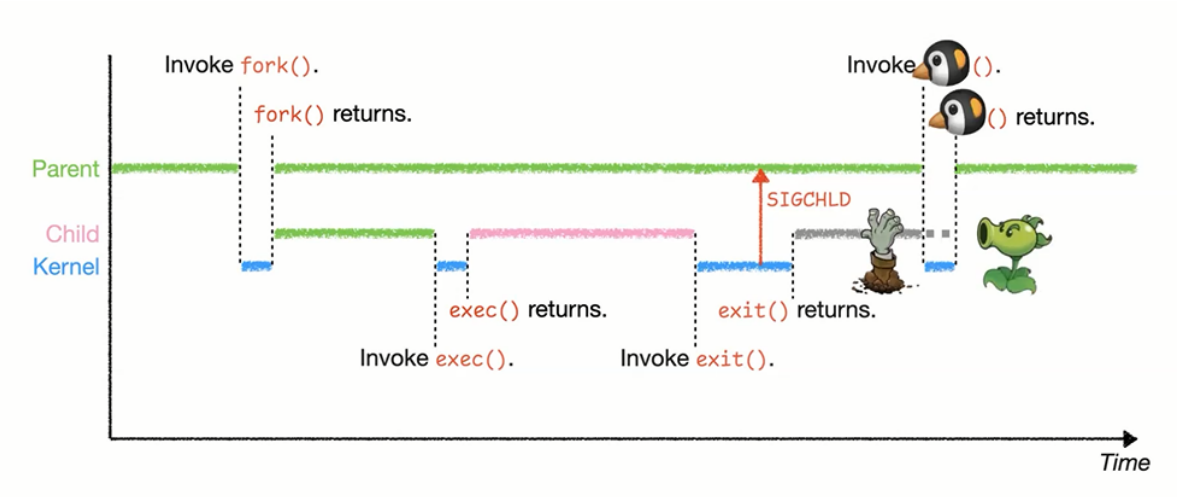
- It's the signal `SIGCHLD` !

5. What's the purpose of it?



- The purpose of the signal is the **interprocess communication**

6. What's 🐧?

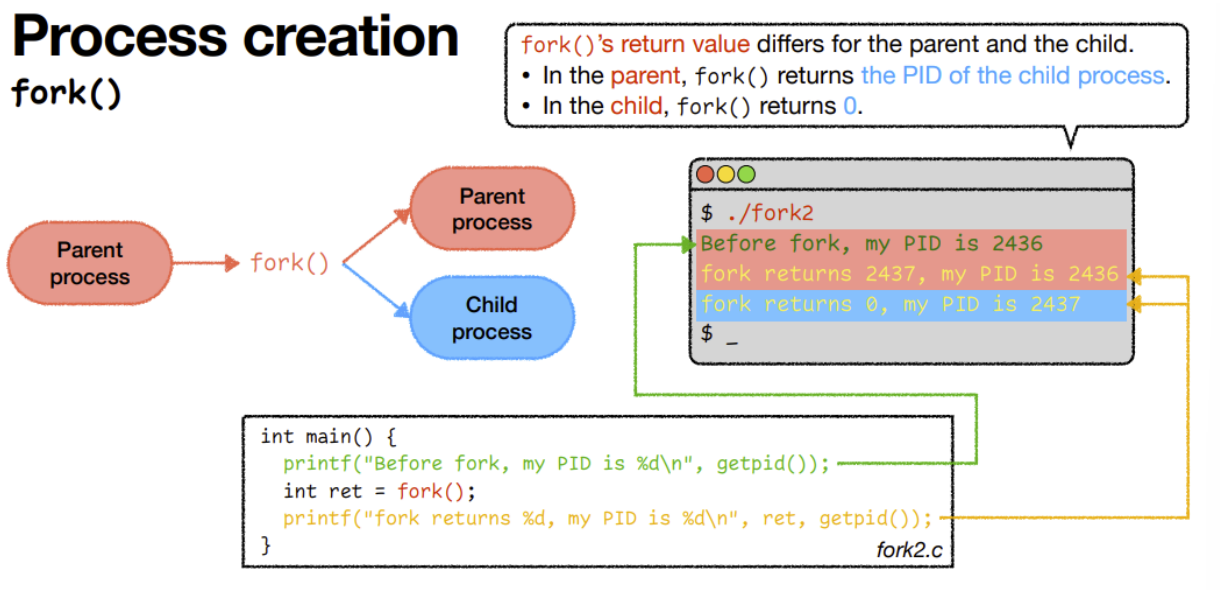


- It's `wait()` !

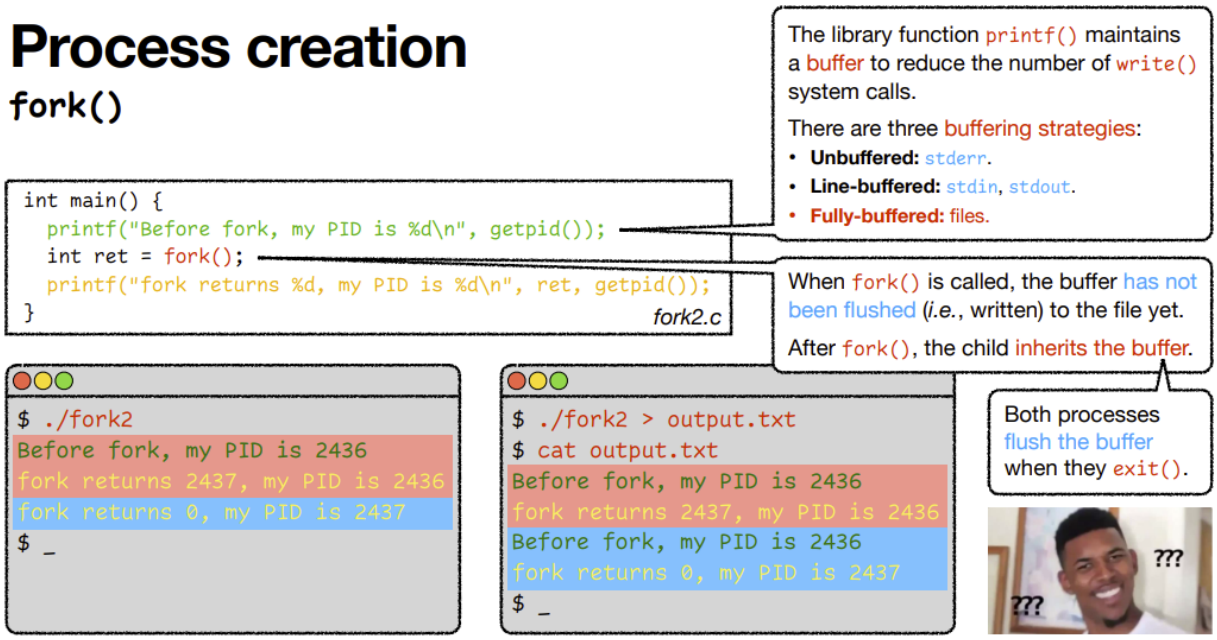
System call	Purpose	Return value on success	Return value on failure
`fork()`	Create a child process (by cloning the calling process).	In the parent: PID of the child. In the child: 0.	In the parent: -1 (check `errno`). No child process is created.
`exec*()`	Execute a program.	It never returns.	-1 (check `errno`).
`exit()`	Terminate the calling process.	It never returns.	It never fails.
`wait()` `waitpid()`	Wait for any child to terminate. Wait for a child to change state.	PID of the child. (status is stored in the argument.)	-1 (check `errno`).
`signal()`	Change the signal handler.	The previous signal handler.	`SIG_ERR` (check `errno`).
`kill()`	Send a signal to a process.	0.	-1 (check `errno`).

System call	Purpose	Return value on success	Return value on failure
`fork()`	Create a child process (by cloning the calling process).	In the parent: PID of the child. In the child: 0.	In the parent: -1 (check errno). No child process is created
`exec*()`	Execute a program.	It never returns.	-1 (check errno).
`exit()`	Terminate the calling process.	It never returns.	It never fails.
`wait()` `waitpid()`	Wait for any child to terminate. Wait for a child to change state.	PID of the child. (status is stored in the argument.)	-1 (check errno).
`signal()`	Change the signal handler.	The previous signal handler.	SIG_ERR (check errno).
`kill()`	Send a signal to a process.	0	-1 (check errno).

- **`Fork()`**

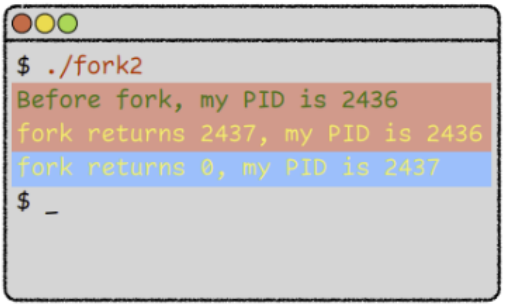


- `fork()` returns the PID of the child to the parent process, and returns 0 to the chold process

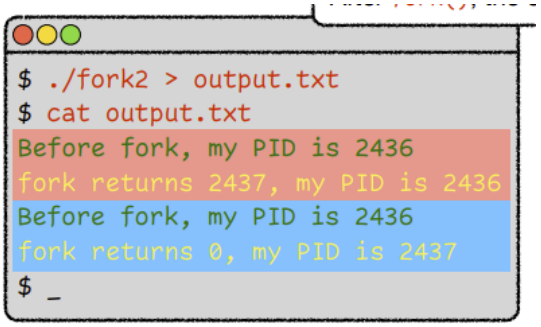


NOTE:

- For this execution, the order of the last two printed lines can be switched



- For this program, there're 4 lines printed.




```
yt2475@linserv1 proc]$ cat fork2.c
include <stdio.h>
include <unistd.h>

nt 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
efore fork, my PID is 8957
ork returns 8958, my PID is 8957
ork returns 0, my PID is 8958
```

```
[yt2475@linserv1 proc]$ ./fork2 > output.txt
[yt2475@linserv1 proc]$ cat output.txt
Before fork, my PID is 8965
fork returns 8966, my PID is 8965
Before fork, my PID is 8965
fork returns 0, my PID is 8966
```

    - This is because it's output to a file, and a file outpt stream is fully buffered.
    - When `fork()` is called, the buffer has not been flushed, so it's inherited by the child.
    - Both process flushes the buffer when they terminates.
- You can also change the default buffering strategy
    - Using the library function `setvbuf ()`

Process Organization

- **Booting the computer**

BIOS locates the boot device.

The boot code on the boot device is executed.

The boot code starts the OS.

- BIOS: Basic input/output system

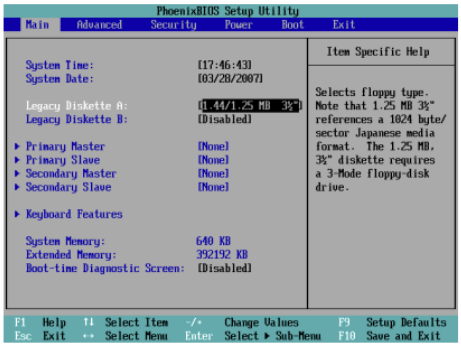
BIOS is firmware (*i.e.*, software providing low-level control for hardware).

It is stored in a ROM chip or flash memory on the motherboard.

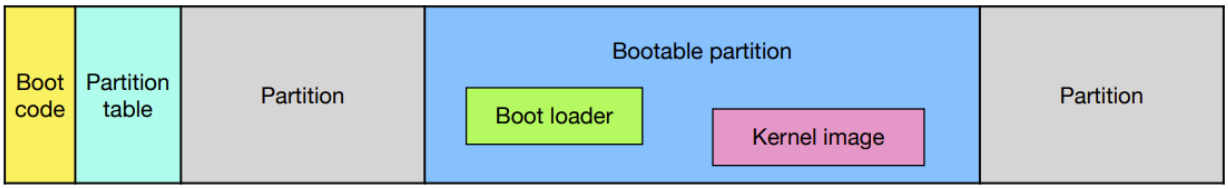
The BIOS starts when the computer boots up.

It checks all devices attached to the computer and locates the boot device.

Note: *recent systems have replaced BIOS with UEFI (Unified Extensible Firmware Interface), which is a complete boot manager.*



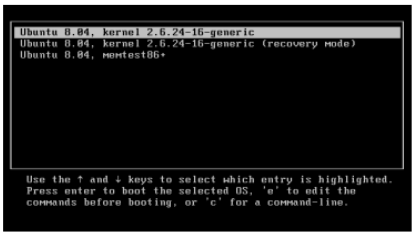
- Boot device



1. The boot code in the boot device's first sector is executed.
2. The boot code reads the partition table and locates the bootable partition.
3. The boot loader from the bootable partition is executed.
4. The boot loader starts the OS kernel.

- Boot loader

The boot loader locates and boots the kernel image.



The kernel image is just a file…

- Linux: `/boot/vmlinuz-*`
- macOS: `/System/Library/Kernels/kernel`
- Windows: `C:\windows\system32\ntoskrnl.exe`

The kernel initializes the memory layout, device drivers, *etc.*, and creates the first process.

The first process - `init` (Linux)

During booting, the kernel creates the first process (PID = 1): `/sbin/init`.

- Recent Linux systems have replaced `init` with `systemd`.
- However, for the purpose of this course, you can ignore their differences.

`init` creates more processes using `fork()` and `execve()`.

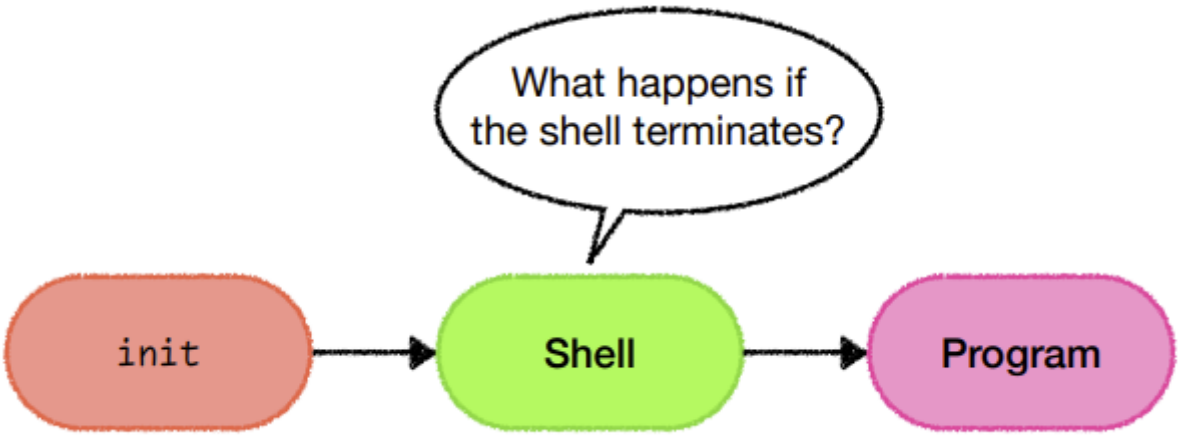
- It reads configurations from `/etc/rc.d` about which programs to run.
- You can view the entire process tree using the `pstree` command.

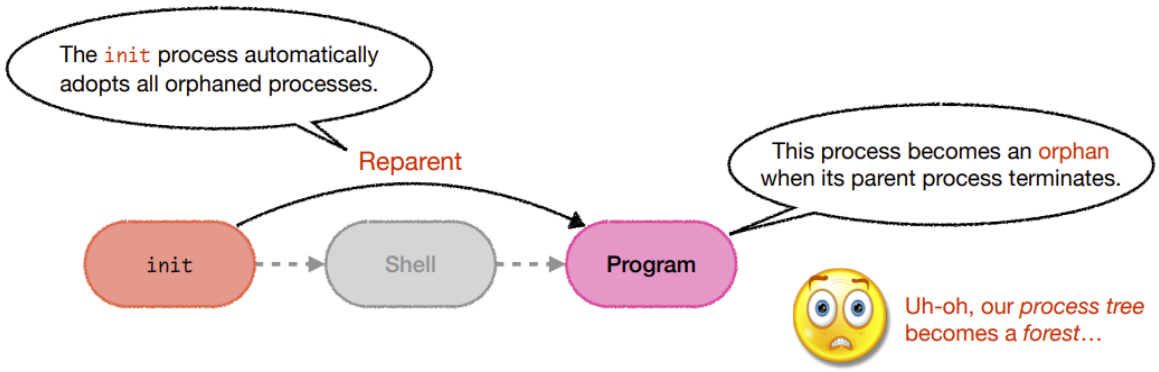
`init` continues running in the background until the system is shut down.

Why?

- Why do we want `init`, the parent of all other processes?

Orphans





Reparenting

- the `init` process will adopt all the orphans

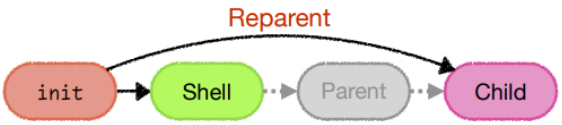
Example of reparenting

```c
int main() {
  if (fork() == 0) {
    // child process
    printf("My parent's PID is %d\n", getppid());
    sleep(2);
    printf("My parent's PID is %d\n", getppid());
  } else {
    // parent process
    sleep(1);
  }
  // both processes
  printf("Terminated\n");
}
```

orphan.c

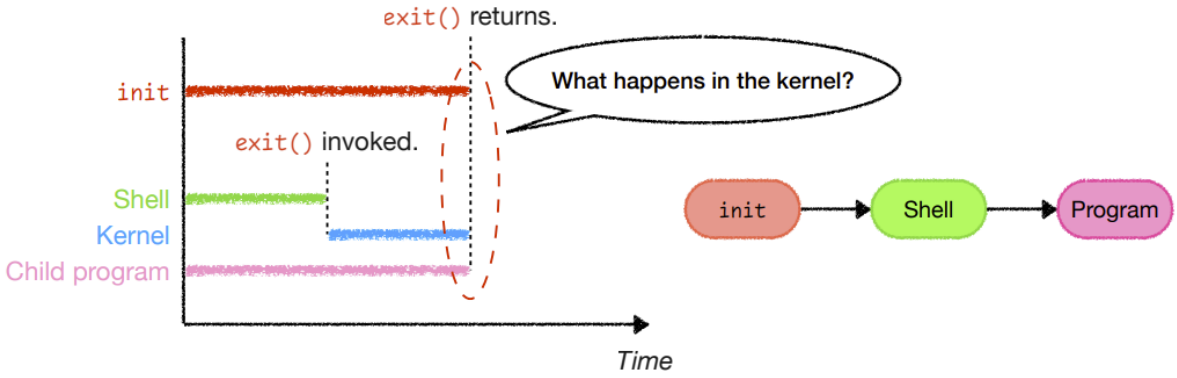
What does it output?

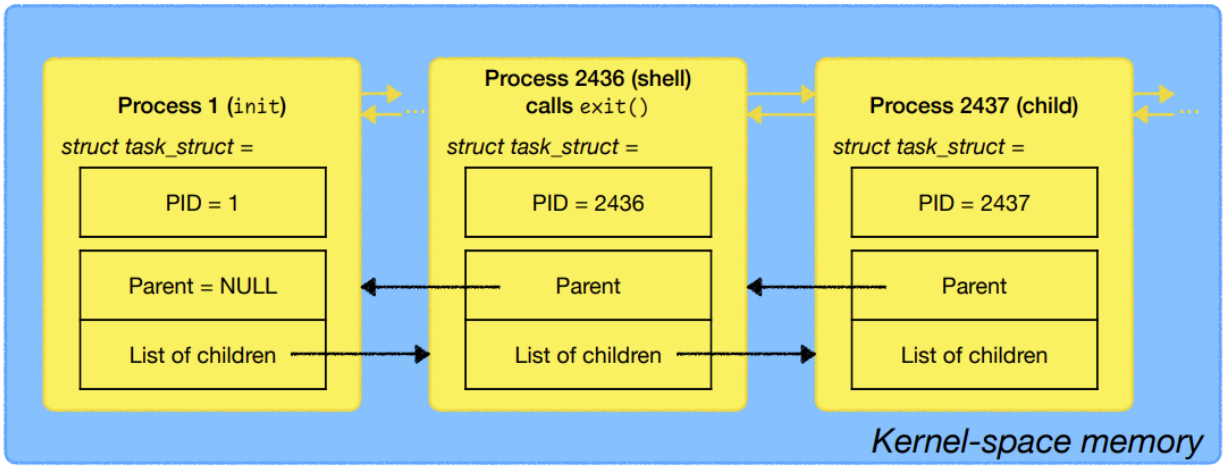
```
$ ./orphan
My parent's PID is 2436
Terminated
$ My parent's PID is 1
Terminated
_
```

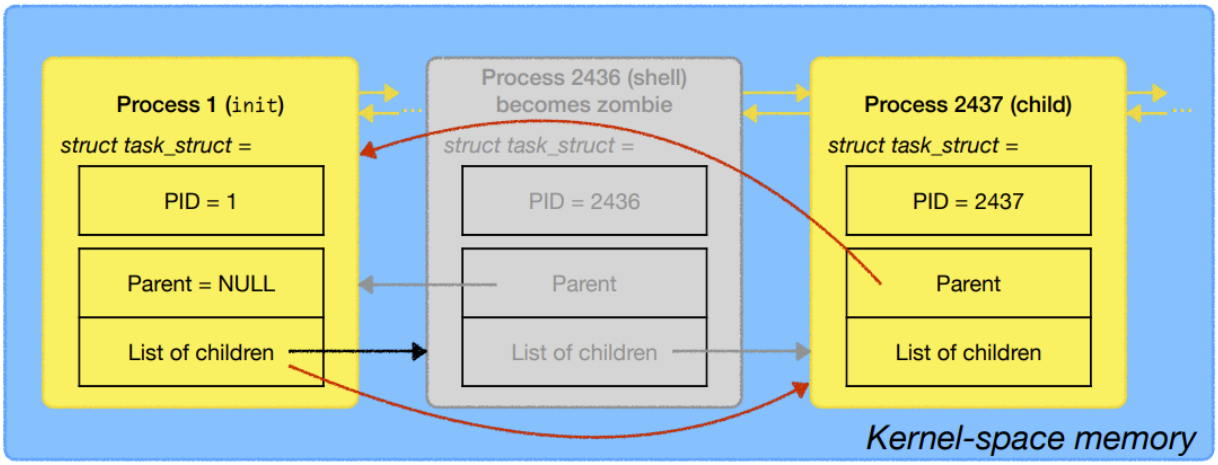


> In the program above, the child sleep for 2 secs between the line printed, and the parent sleep for 1 second and then terminates.
>
> This means that before the parent terminated, the child has already print out a line, and it prints the second line after the parent terminates.
>
> From the output, we can see that it's not the grandparents that has adopted the orphan. It's the `init` process that adopted the orphan.

What happens in the kernel?

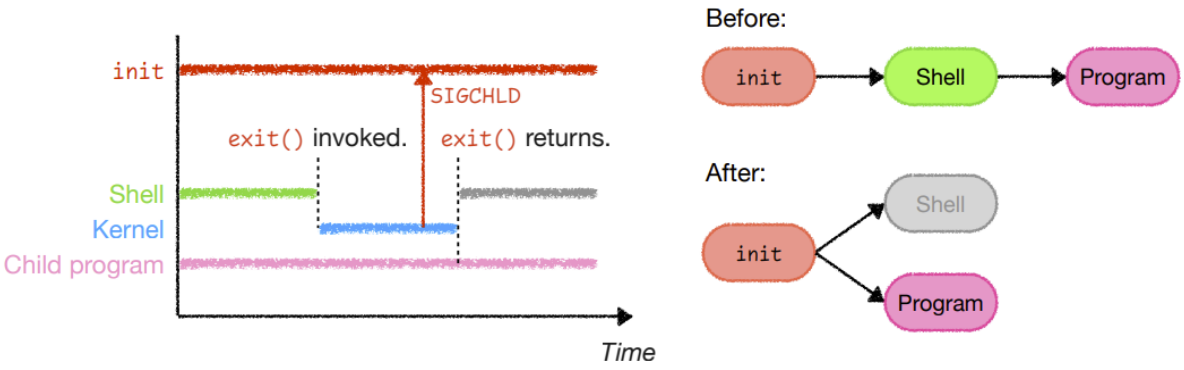






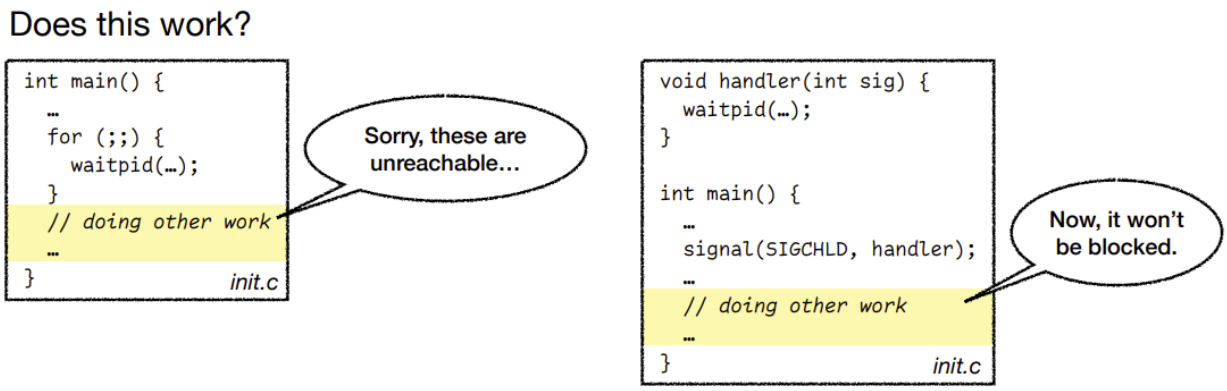
> The grey pointers remains as the zombie process is not yet removed.

- Will `init` clear the zombie?



> The `init` process adopted the orphans. Now it has two children.

How does `init` clear the zombie?



Left: We can make an infinite loop, and continously call `waitpid()` to clear all the zombies

- Nice try, but it won't work.
- `init` would be blocked if it calls `waitpid()`, but we can't let it to be blocked.
- The `init` process is not just for clearing the zombies, it has many other useful works to do.

Right: We can overwrite the signal handler, let the signal handler of `SIGCHLD` to call `waitpid()`

- In this way, the `init` process will not be blocked.
- Whenever there's a `SIGCHLD`, the signal handler will call the `waitpid()` and clear the zombie.
- Now the `init` won't be blocked and can do other work

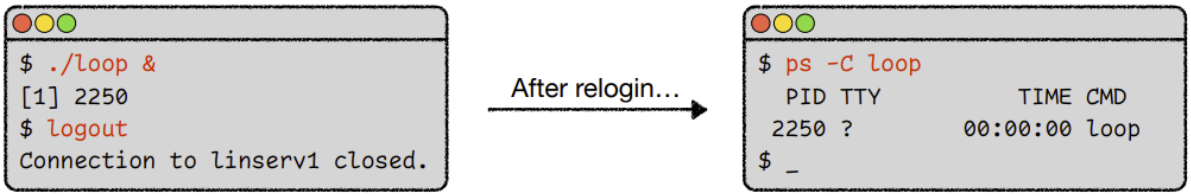
Some observations

Processes in Linux are organized as a single tree.

- Why not a forest?
- Windows does maintain a forest-like process hierarchy, but we'll focus on Unix/Linux in this course.

Reparenting allows processes to run without a parent shell.

- Therefore, background jobs can survive even after the shell exits.



Process scheduling

What and Why

Computers often do several things concurrently, even if it has only one CPU. •

- It's called multiprogramming.

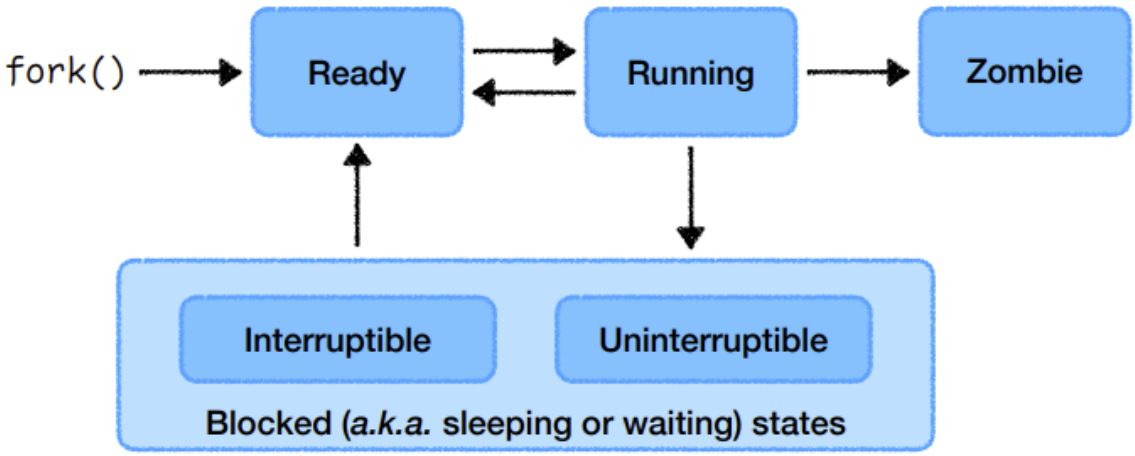
The CPU switches from process to process quickly, running each for a few *ms* .

- It's called multitasking.

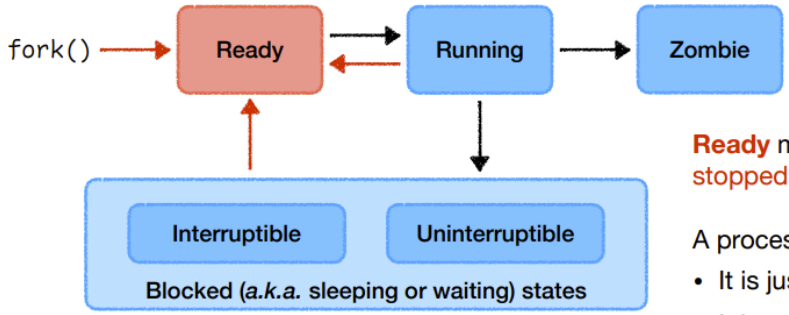
The OS needs to choose which process to run next.

- It's called scheduling.
- The part of the OS that makes the choice is called the scheduler.

Process states



- Ready

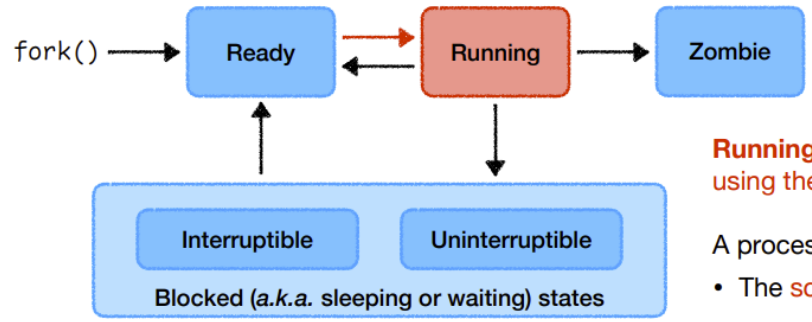


Ready means runnable but temporarily stopped to let another process run.

A process may become ready after…

- It is just created by `fork()`;
- It has run on the CPU for long enough, and the scheduler picks another process to run;
- It returns from a blocked state.

- Running

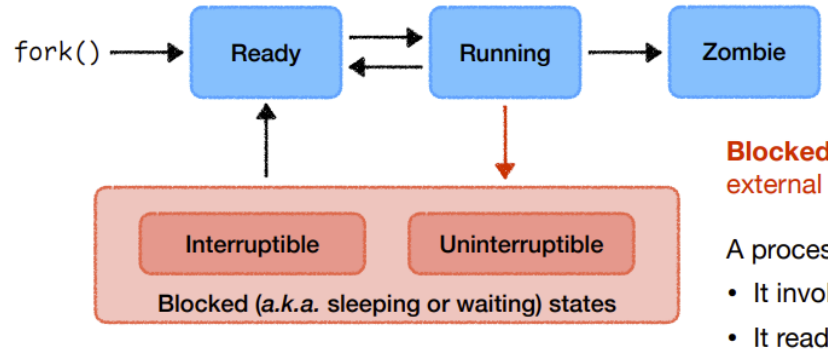


Running means the process is actually using the CPU at that instant.

A process may become running after…

- The scheduler picks this process to run.

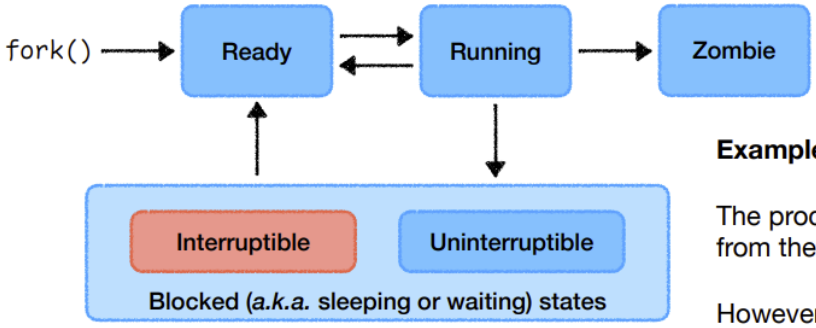
- Blocked



Blocked means unable to run until some external event happens.

A process may become blocked after…

- It invokes the `pause()` system call;
- It reads from an I/O device;
- …

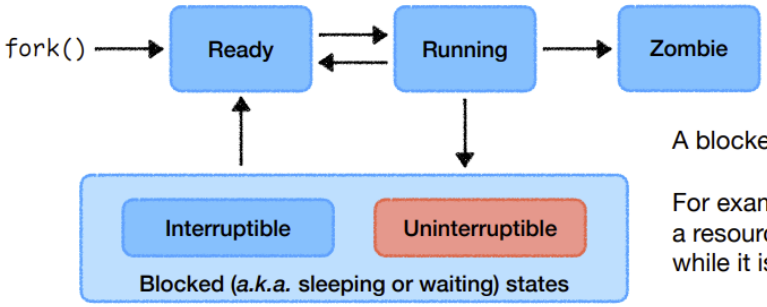


Example: reading a file on the disk.

The process has to wait for the response from the disk drive. So, it is blocked.

However, this blocked state is interruptible.

For example, pressing Ctrl-C can get the process out of the blocked state.

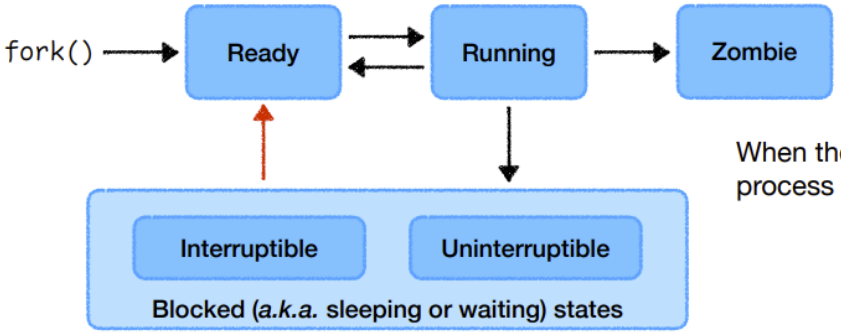


A blocked state may also be uninterruptible.

For example, a process may need to wait for a resource, but it doesn't want to be disturbed while it is waiting.

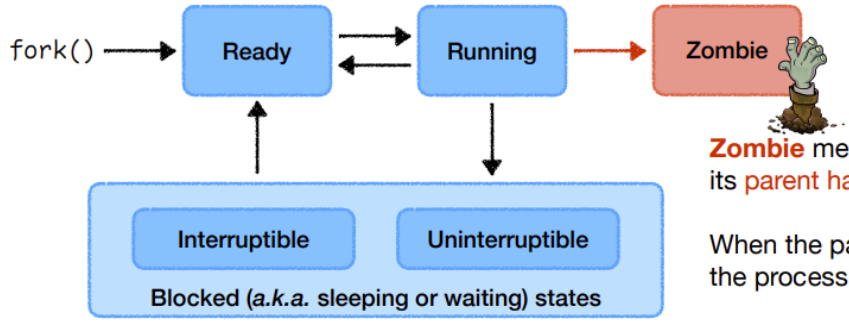
This scenario usually happens within the kernel.

> "
>
> This is very rare



When the external event happens, the process becomes ready again.

- Zombie



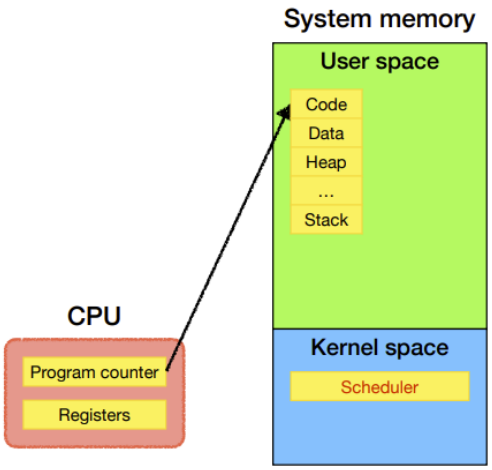
Zombie means the process exits and its parent has not yet waited for it.

When the parent finally waits for it, the process terminates.

- Context switching

When it's time for a process to give up running on the CPU…

The scheduler in the kernel will choose the next process to run.

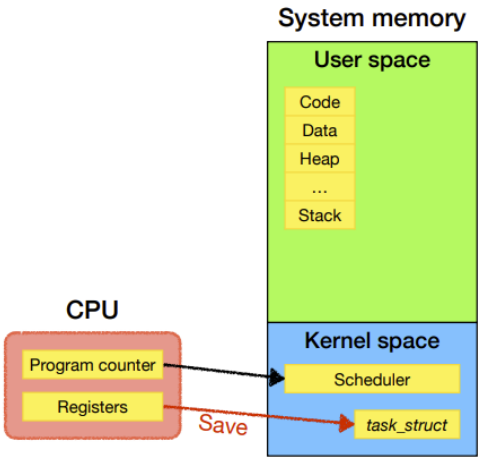


Before the scheduler can take up the CPU, it has to back up register values.

- Where should the backup be stored?

The **context** of a process consists of…

- Its user-space memory;
- Register values.



> "
>
> - The best place to store these register data is our task_struct
> - During context-switching, those register values are all saved to the task_struct
> - Context:
>   - User-space memory
>   - Register values
>   - because you need to switch the user-space memory and the register values between different contexts

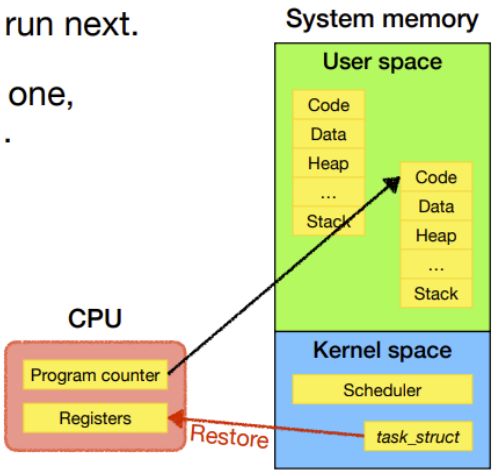
Then, the scheduler decides which process to run next.

If the next process is different from the current one, the OS performs a **context switch**, including…

- Saving and restoring registers;
- Switching memory maps;
- Flushing and reloading the cache;
- …

Context switching may be expensive…

…and the system isn't doing any useful work.



Homework

- Question 1

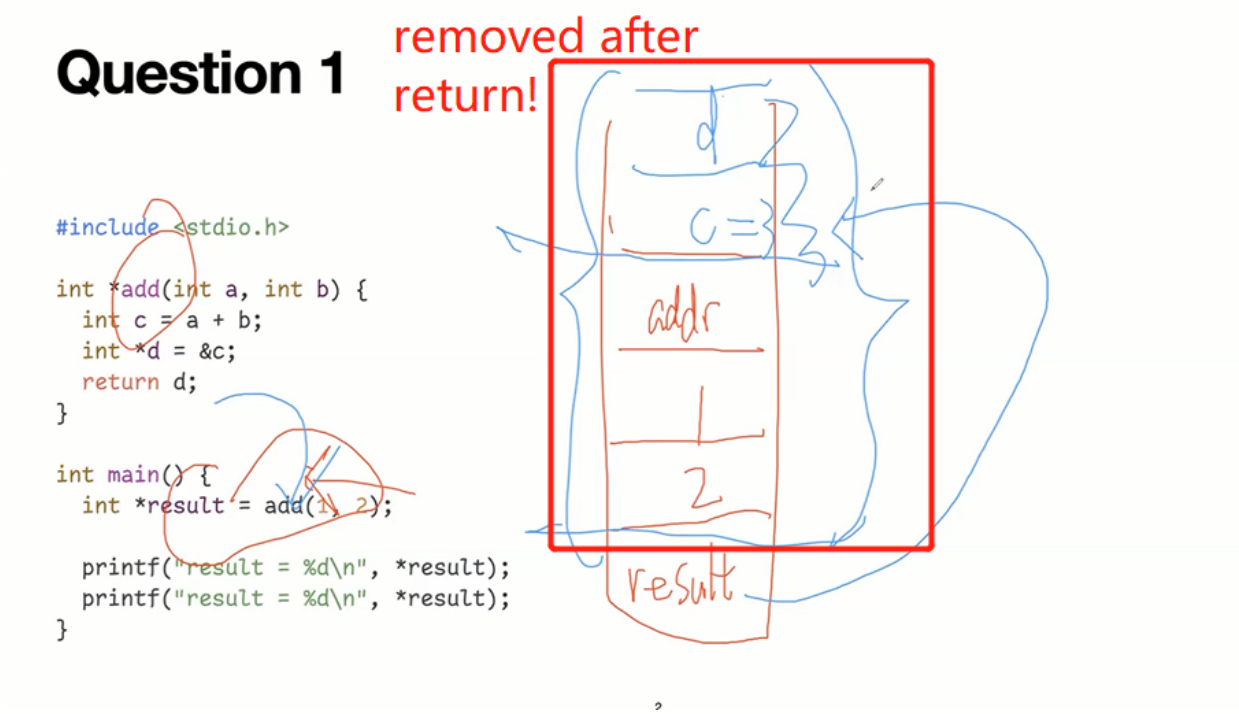
```c
#include <stdio.h>

int *add(int a, int b) {
  int c = a + b;
  int *d = &c;
  return d;
}

int main() {
  int *result = add(1, 2);

  printf("result = %d\n", *result);
  printf("result = %d\n", *result);
}
```

Question 1



- Note that in this stack, everything above the result is removed after the function `add()` is returned.
- However, the result points to somewwhere inside the removed stack.
  - will cause an undefined behavior!
  - The compiler would be able to generat any code they want

- Undefined behavior

Returning the address of a local variable is **undefined behavior**.

In the C community, undefined behavior may be humorously referred to as **"nasal demons"** since the compiler is allowed to do anything it chooses, **even "to make demons fly out of your nose."**

If you're interested in learning more about undefined behavior, I recommend that you read Schrödinger's Code.