# CSCI 2410 Systems Programming

## Fall 2026


### Instructor: Jie Zhou

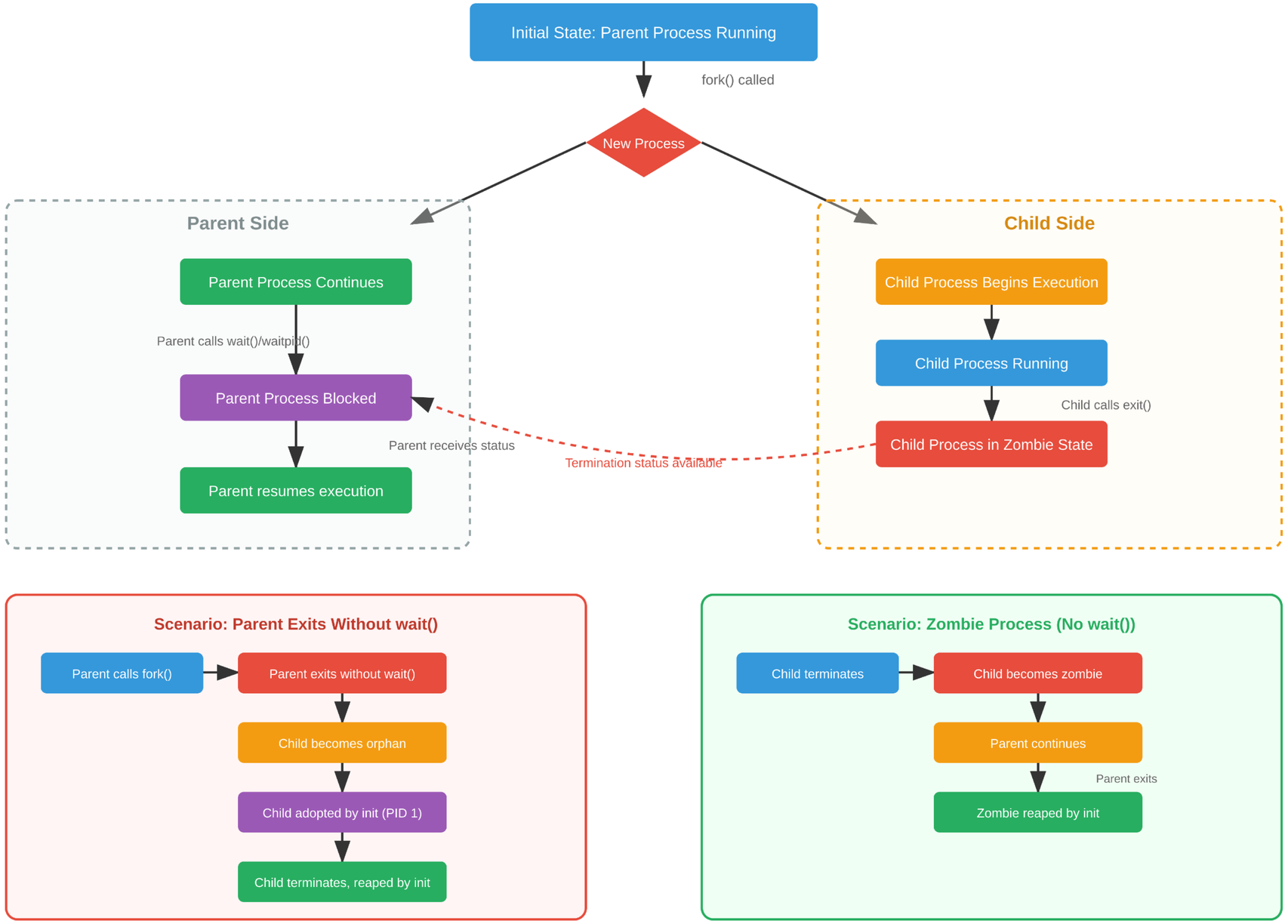
Department of Computer Science

George Washington University




Slides materials are mostly credited to Gabriel Parmer and Sibin Mohan at GWU.

# Fork(), Wait() and Process Lifecycle in Unix/Linux



# Executing A Program

- Program: a set of instructions that specifies a computation
  - can be a compiled executable (machine code) or a script (interpreted)
- `fork()` lets us make a new process from an existing one.



How do we launch a new program?

# execve()

- Replace current program without creating a new process
- Replace the process's virtual address space with a new one
- Load target new program and start executing target program

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

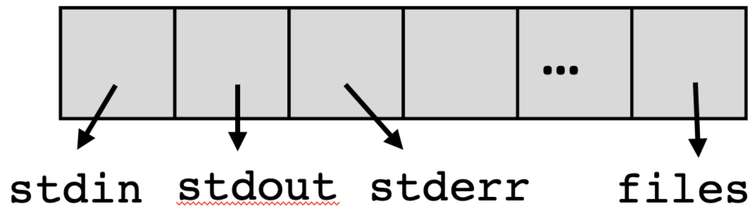
- From `<unistd.h>`
- `path`: executable's file location (e.g., `/bin/ls`)
- `argv`: an array of command line arguments
  - `argv[0]` is conventionally the program name.
  - The last element must be NULL.
- `envp`: An array of environment variables
  - Each string has the form "`NAME=value`", e.g., "`USER=alice`"
  - The last element must be NULL.

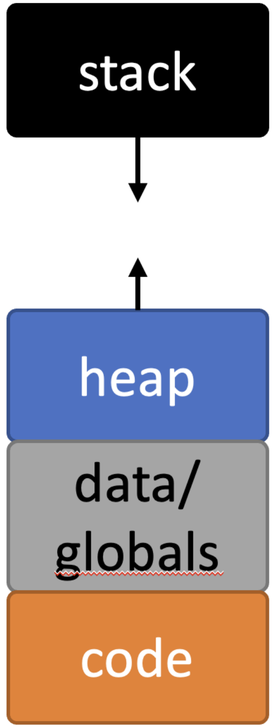
# Process

pid
ppid
user: sibin
cwd: /home/sibin

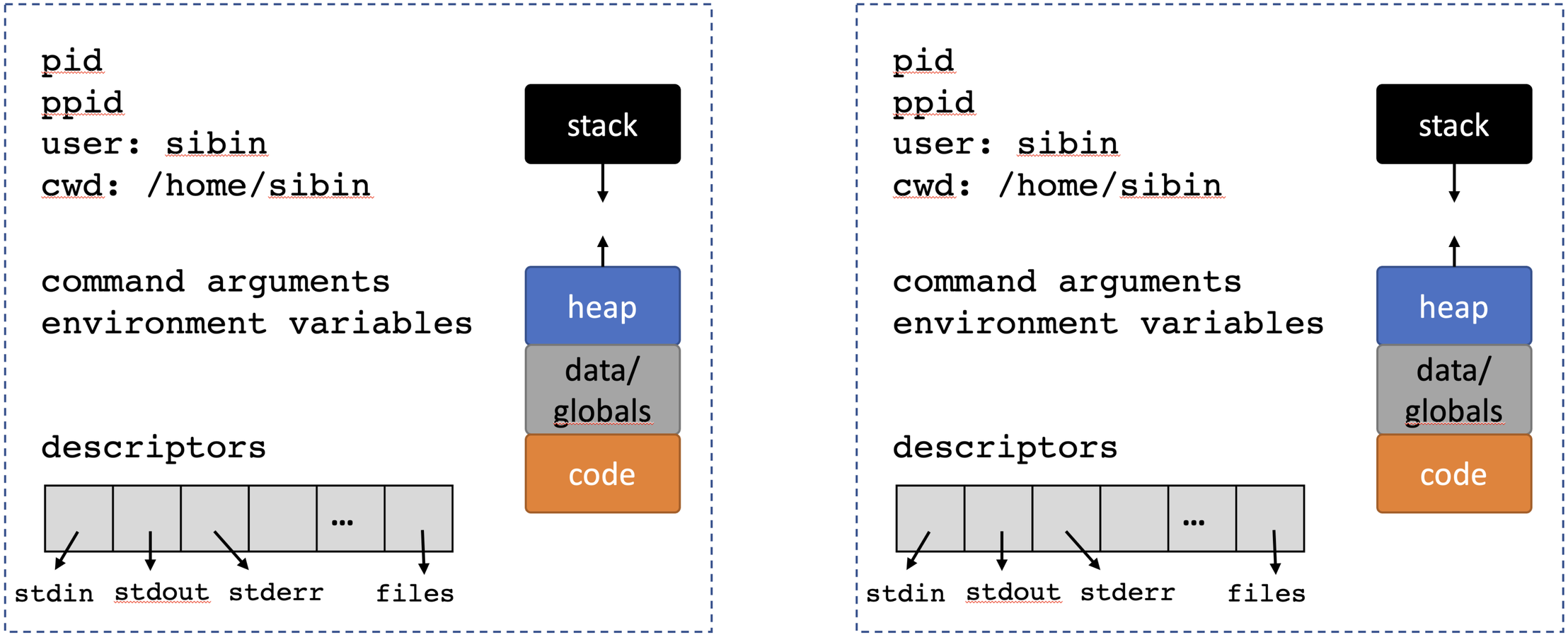
command arguments
environment variables

descriptors

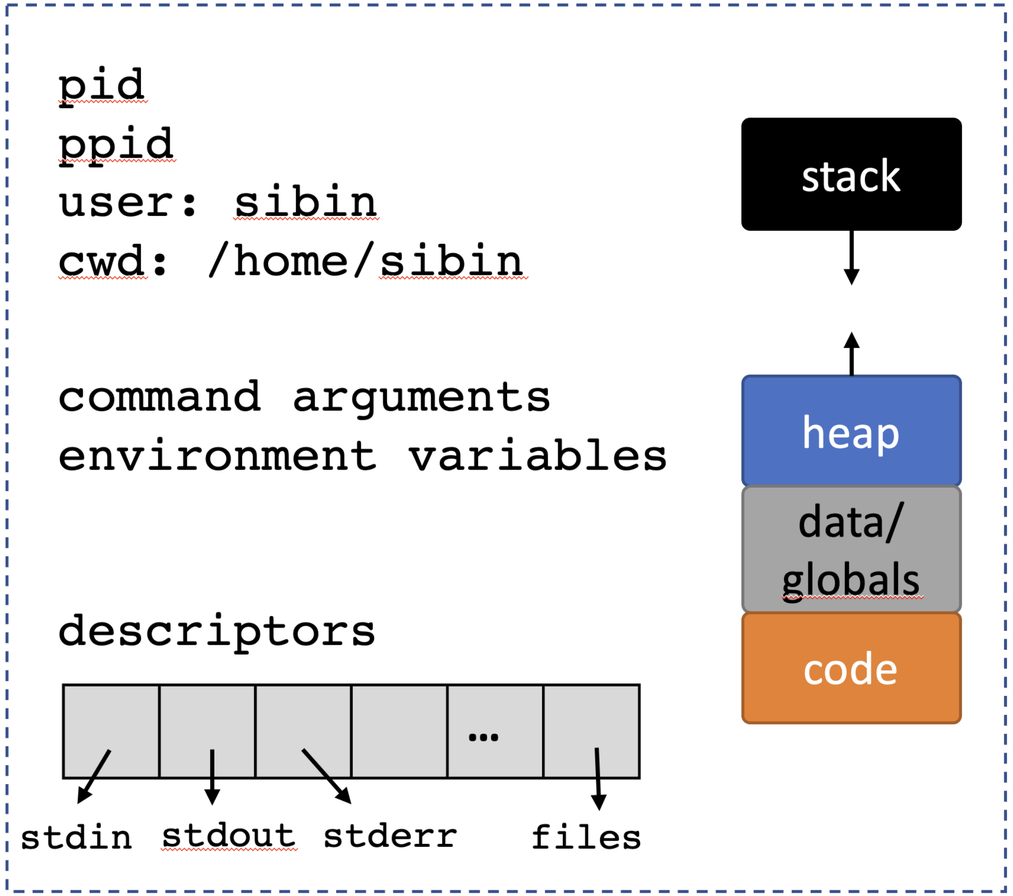


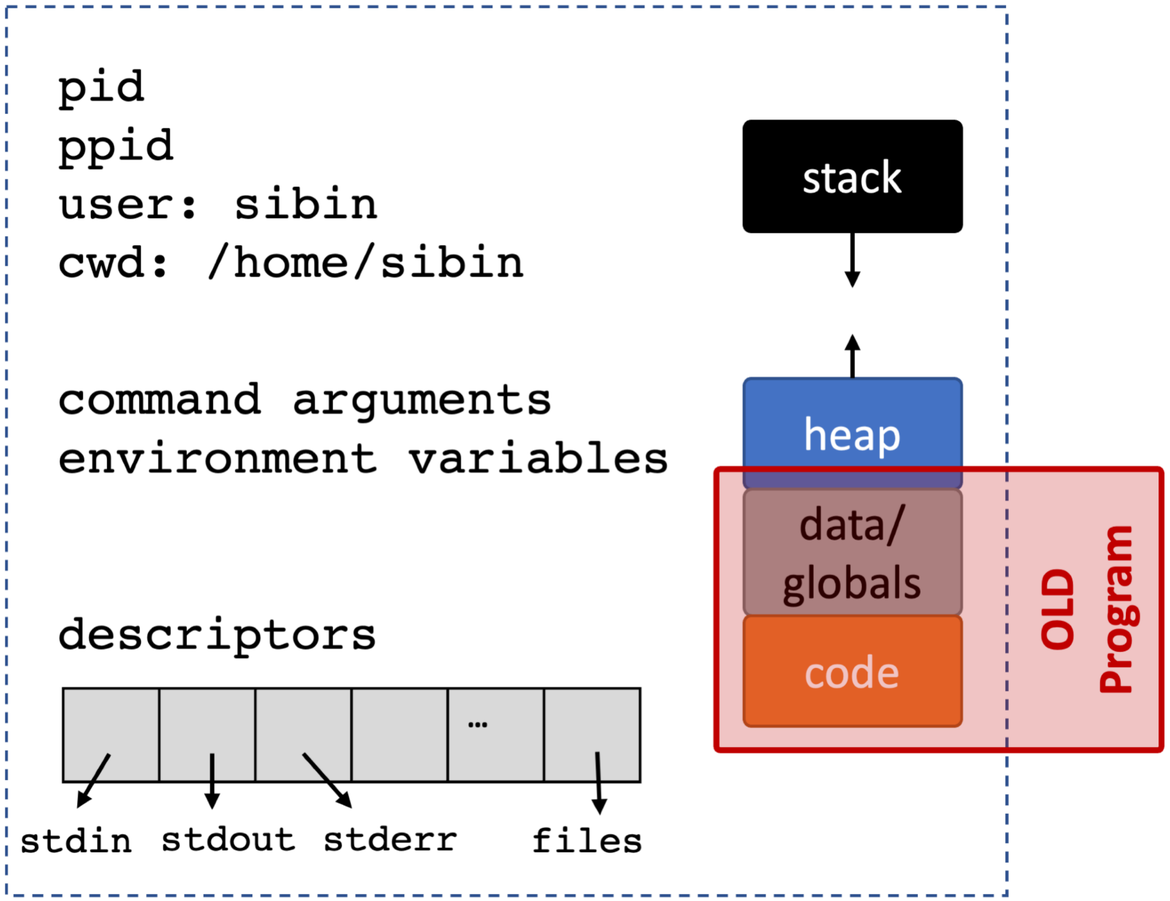



# After fork()

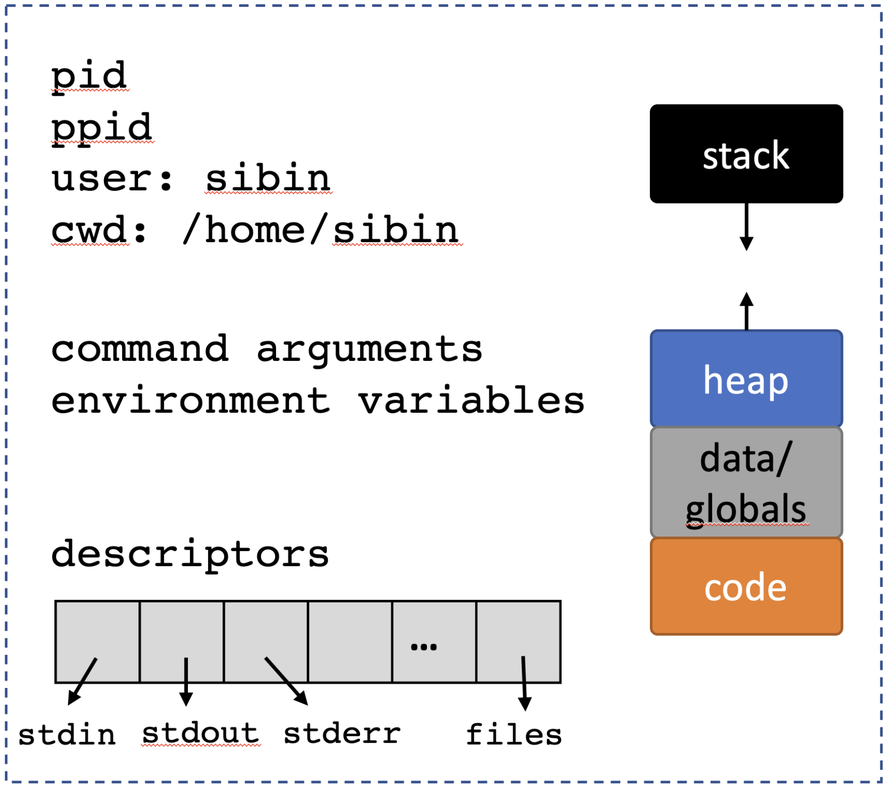

pid
ppid
user: sibin
cwd: /home/sibin
command arguments
environment variables
descriptors
stack
heap
data/
globals
code
...
stdin
stdout
stderr
files
pid
ppid
user: sibin
cwd: /home/sibin
command arguments
environment variables
descriptors
stack
heap
data/
globals
code
...
stdin
stdout
stderr
files

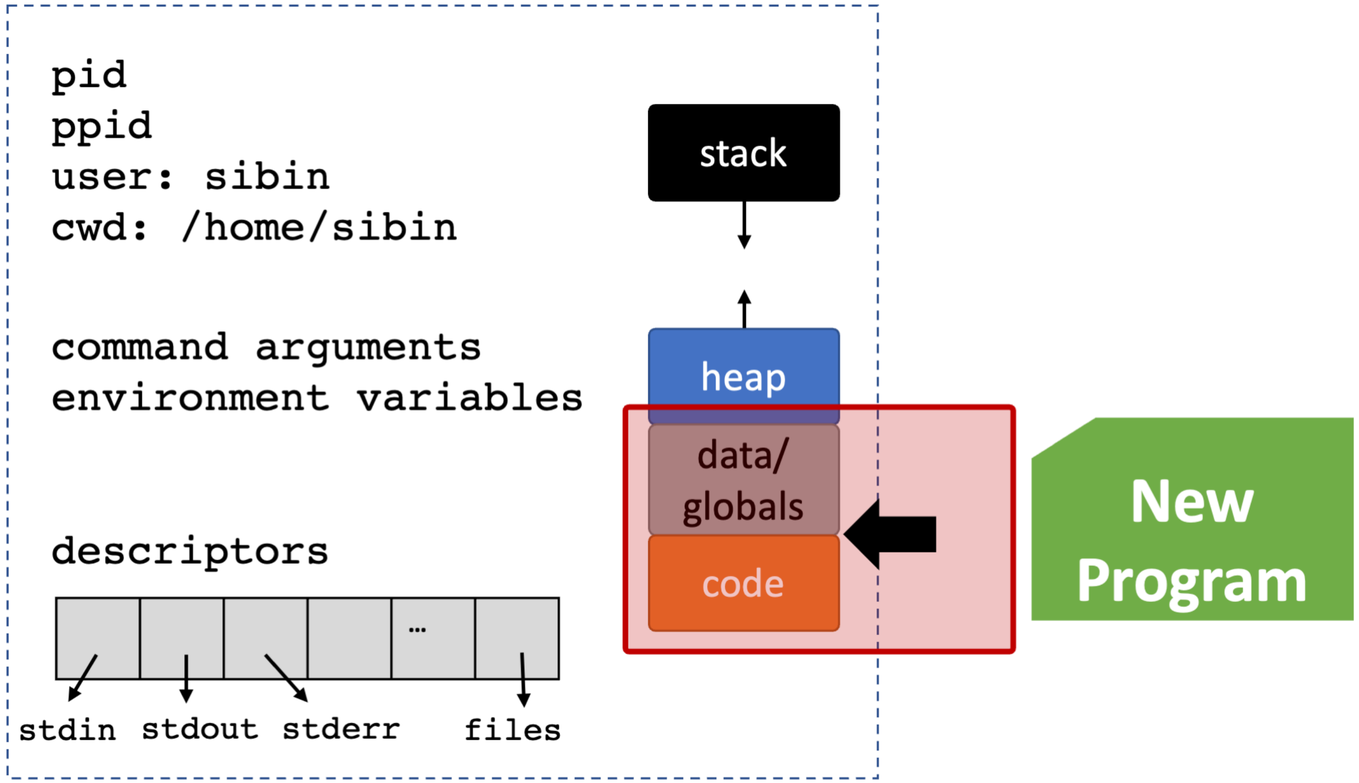
# Replace Old Code



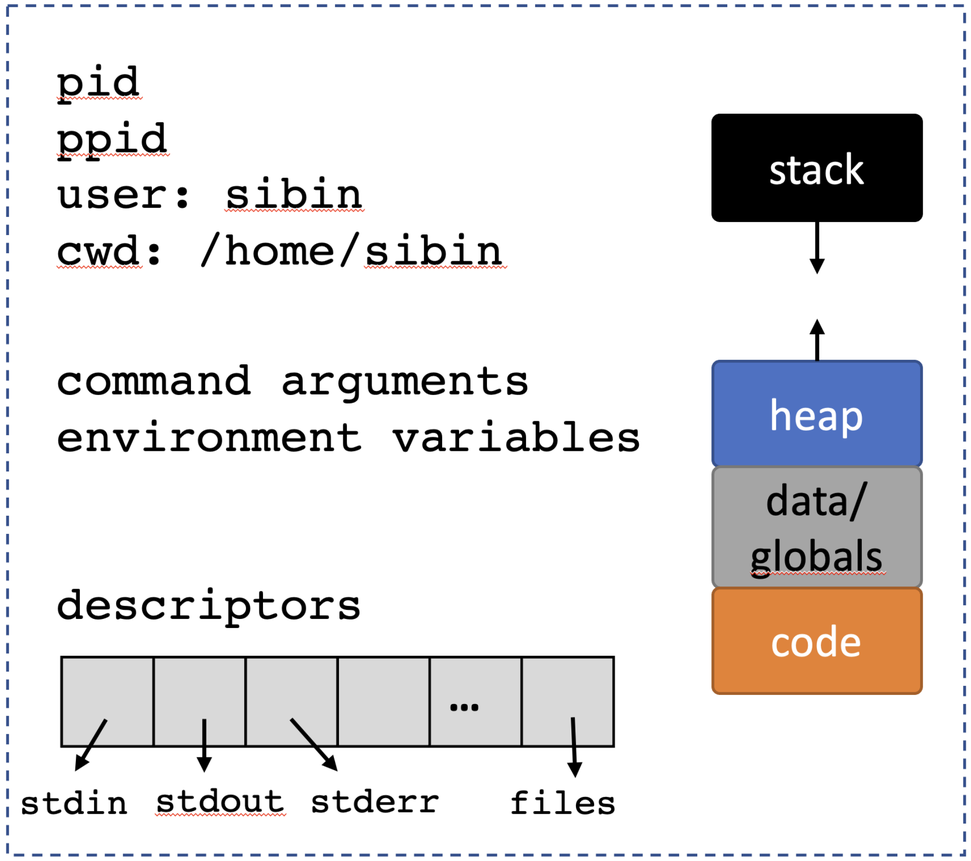



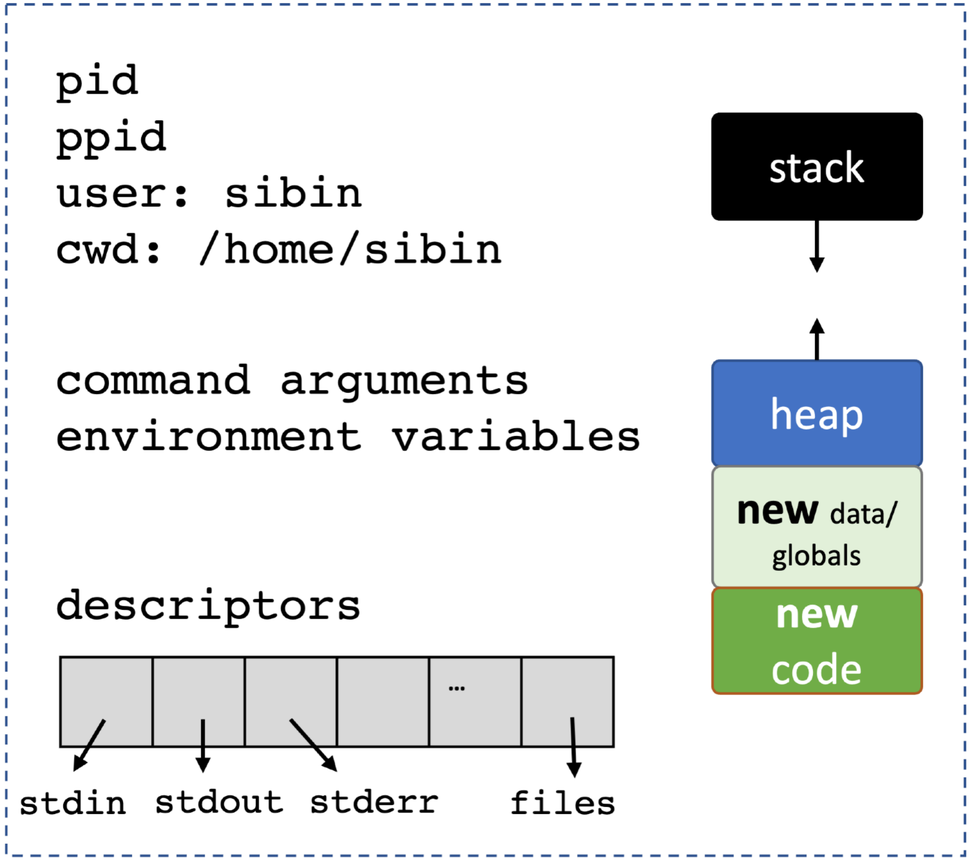
# With New Code



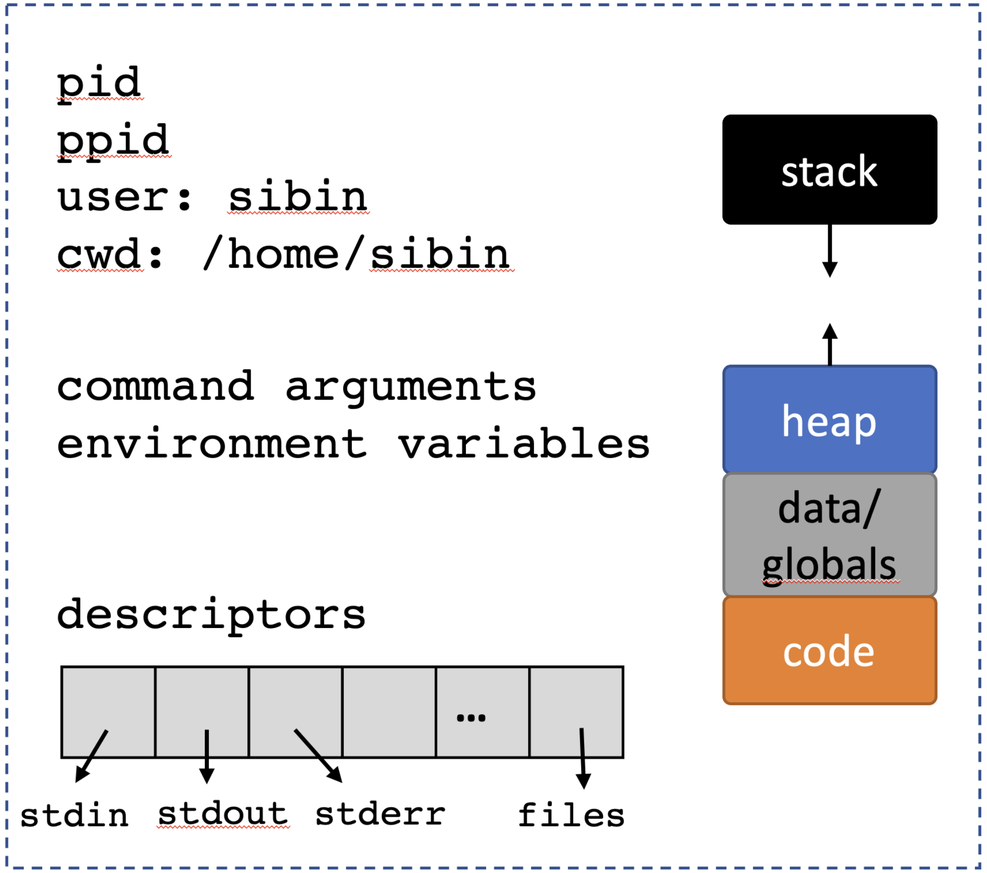



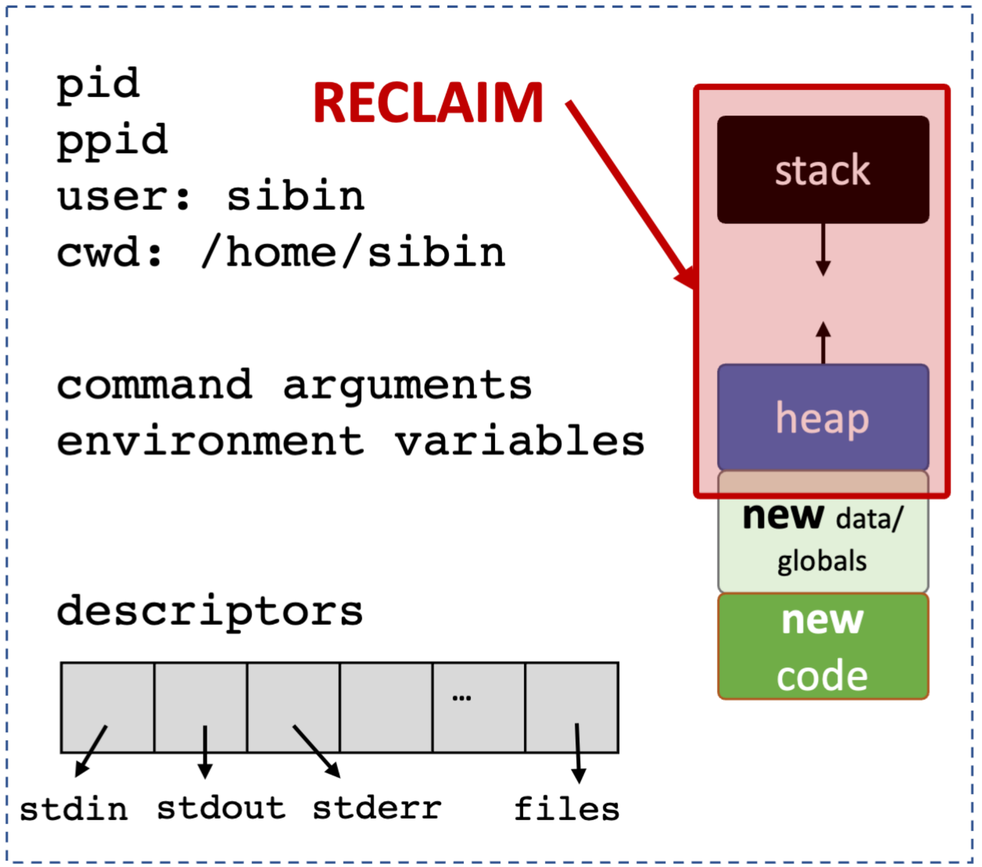
# With New Code



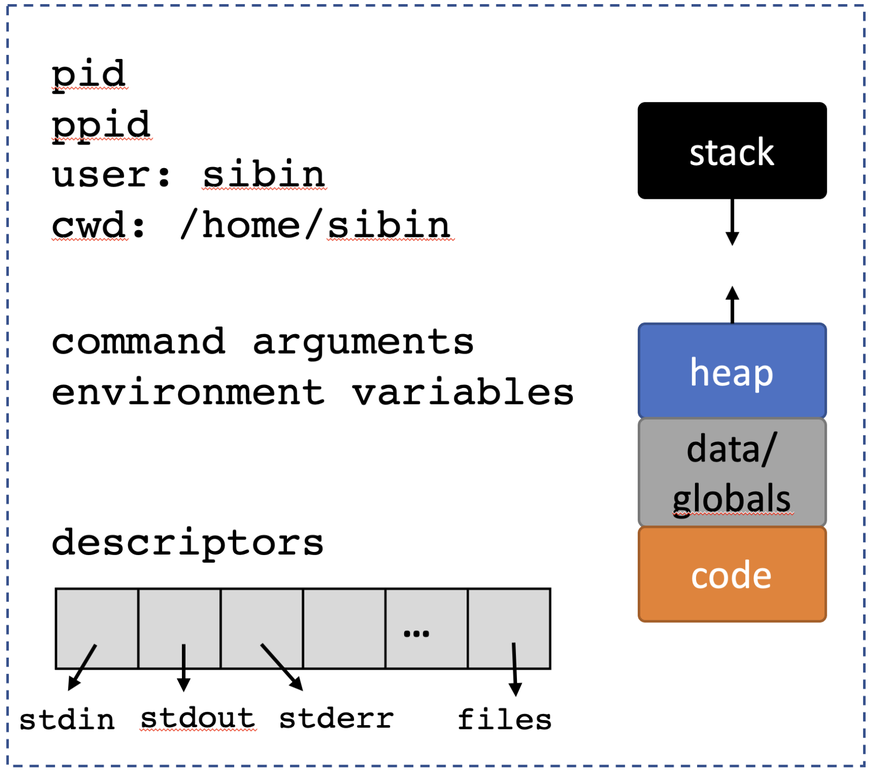



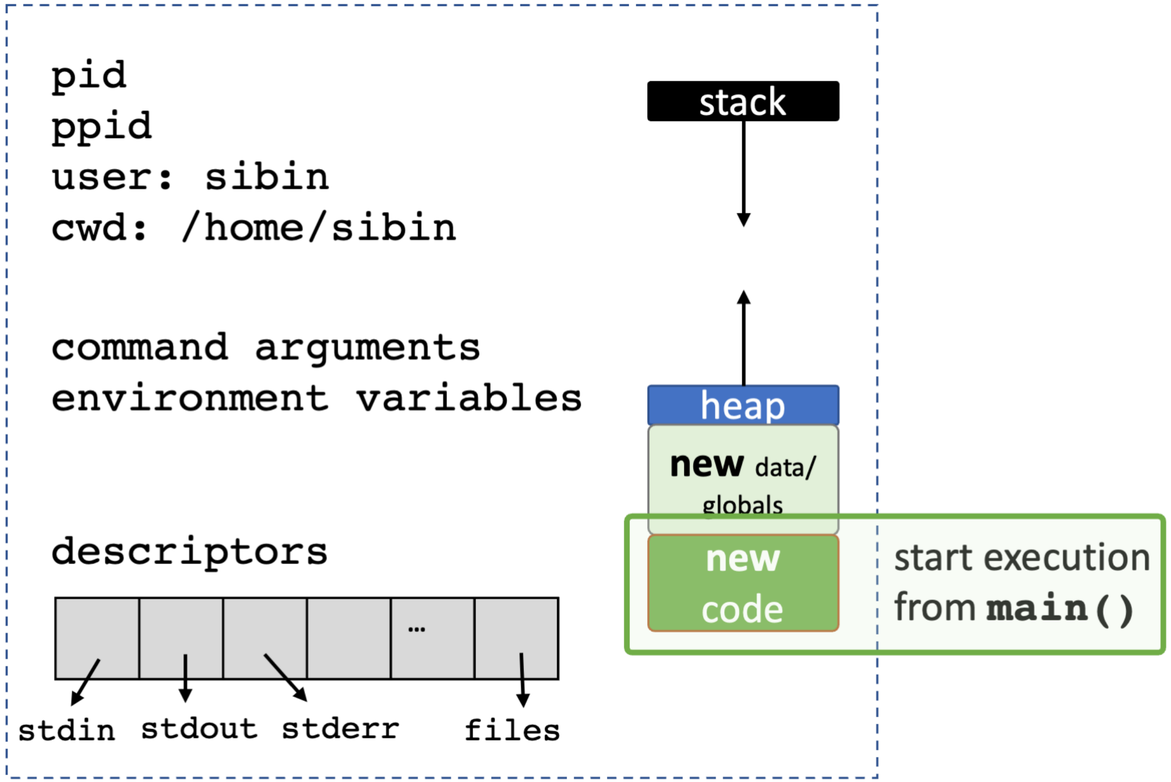
# Reclaim Resources



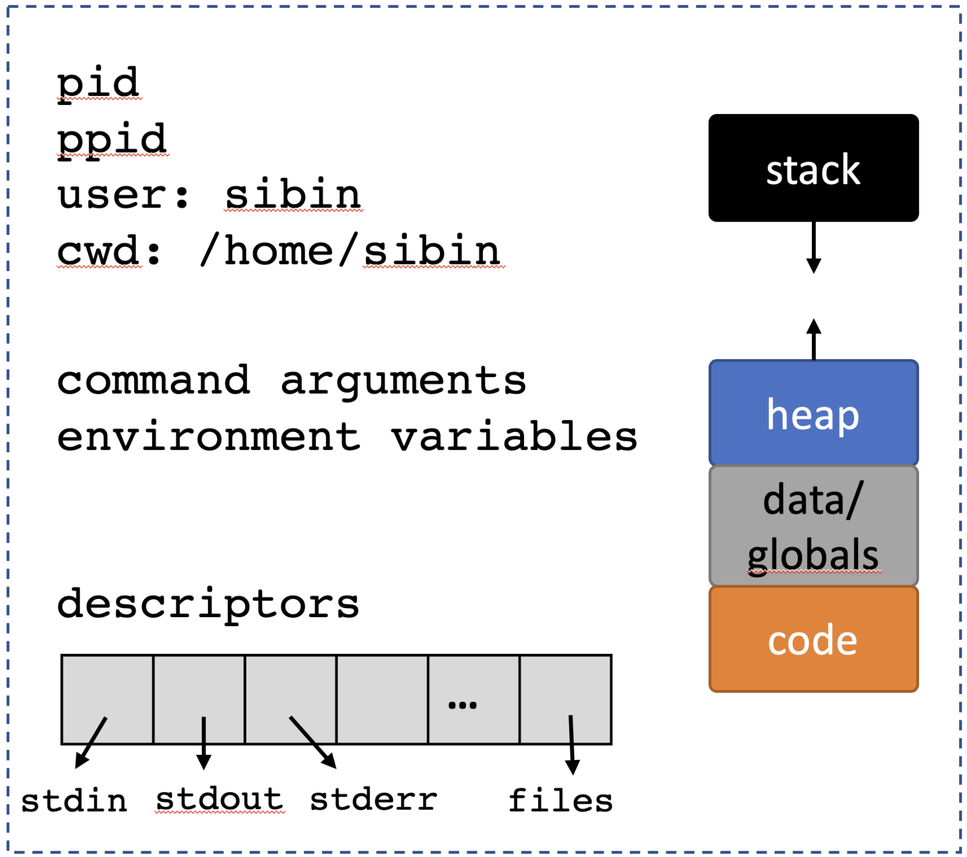
# Execute New Program

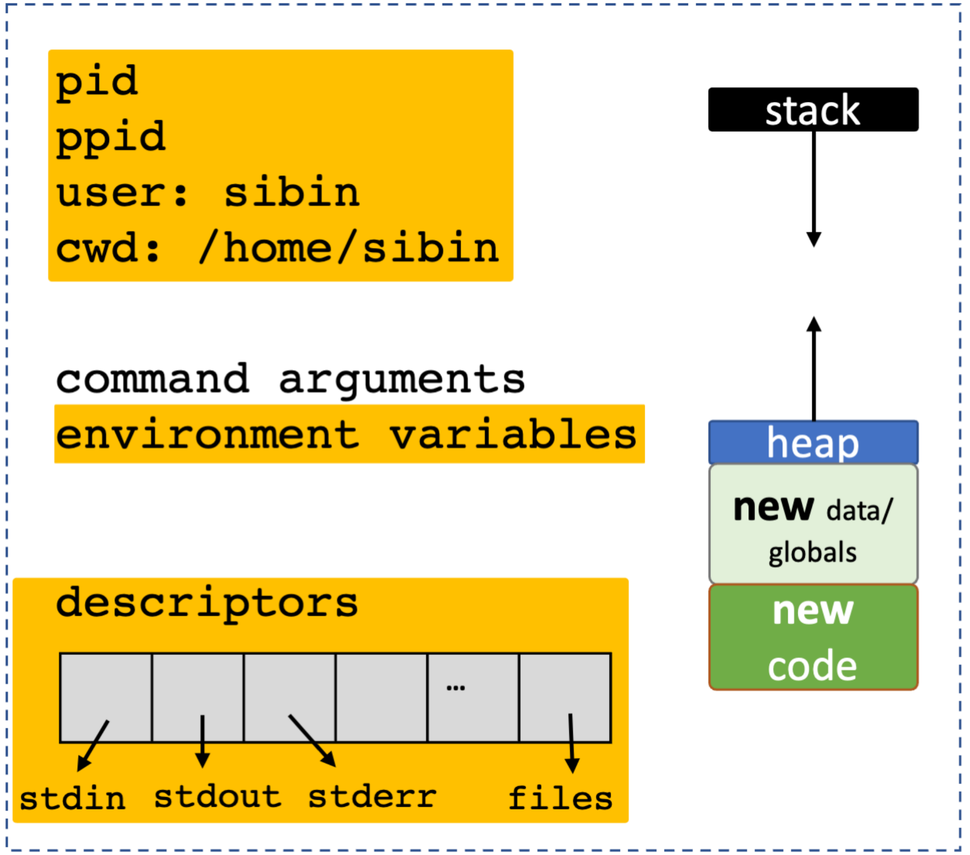


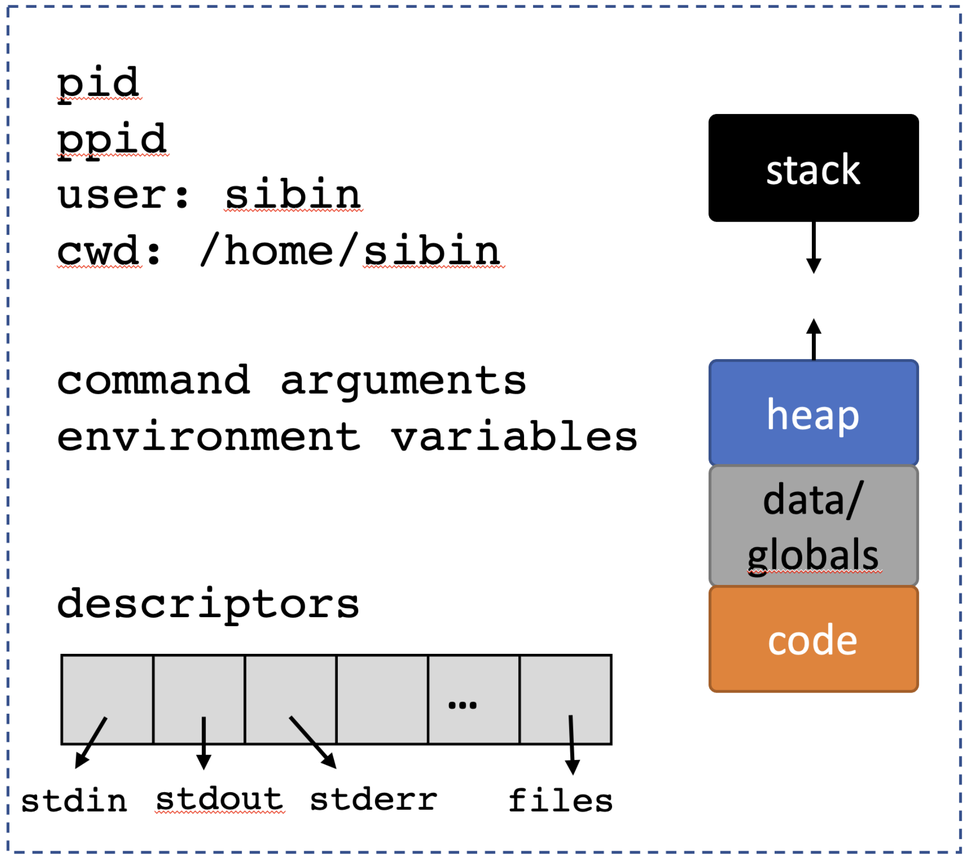


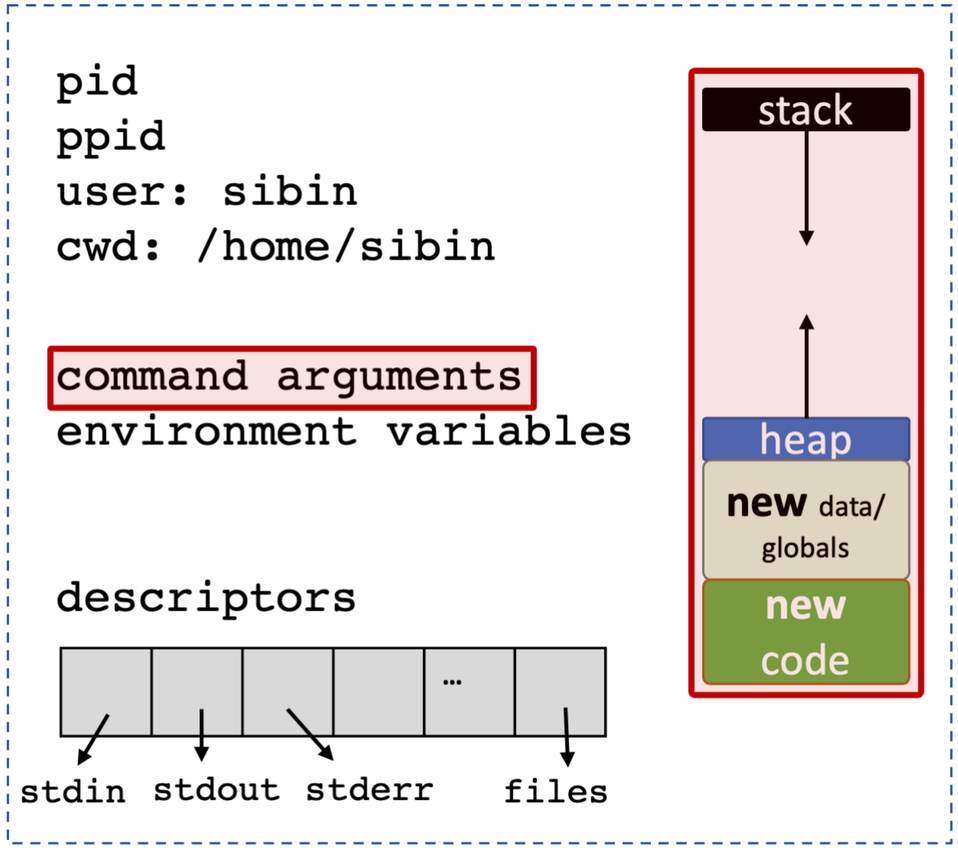
# Good Side Effects



# Only Memory Not Shared



# `execve()` Variants

- `execl()`
- `execlp()`
- `execv()`
- `execvp()`
- `execle()`

Check the `man` page.

# fork() and execve() Example

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        // Child: execute ls
        char *argv[] = {"ls", "-l", NULL};
        char *envp[] = {NULL};

        execve("/bin/ls", argv, envp);
        perror("execve"); // Only reached if execve fails
    } else {
        // Parent: wait for child
        wait(NULL);
        printf("Child finished!\n");
    }

    return 0;
}
```

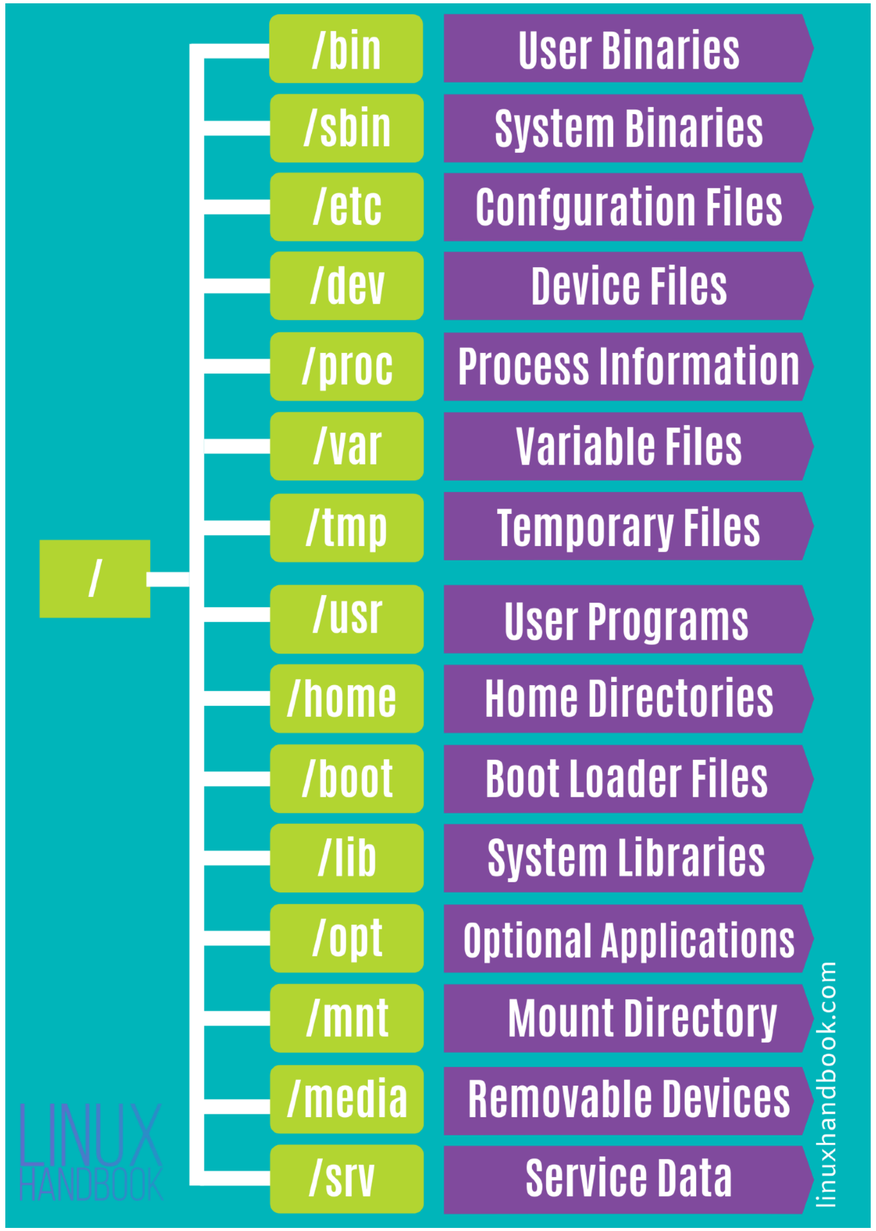
# More on Processes

- Several key mechanism for managing a mechanism and interactions
  - current working directory, `cwd`
  - file descriptors
  - controlling “exceptional” control flow via signals

# cwd

- Each process has a “working directory”
- Base for any relative paths
- All file and directory paths are one of
  - absolute paths beginning with “`/`”, e.g., “`/bin/ls`”
  - relative paths, e.g., “`docs/test.txt`”
- Simple and useful APIs:
  - `getcwd()` gets the current process’ working directory
  - `chdir()` changes the calling process’ working directory
  - both defined in `unistd.h`

# Directory Structure of Linux



# File Descriptors

- Each process maintains a set of file descriptors (FDs).
- Refer to regular files and other I/O resources.
  - regular files
  - sockets
  - devices
- Each FD is a nonnegative integer.
- By convention, the first three FDs are:
  - `0`: `stdin`, standard input, usually from keyboard/terminal
  - `1`: `stdout`, standard output, usually to terminal, but can be redirected to files
  - `2`: `stderr`, standard error, usually to terminal, but can be redirected to files

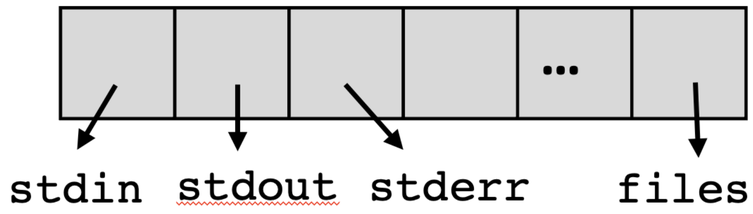
| number | descriptor |
|---|---|
| `0` | `STDIN_FILENO` |
| `1` | `STDOUT_FILENO` |
| `2` | `STDERR_FILENO` |
| | |

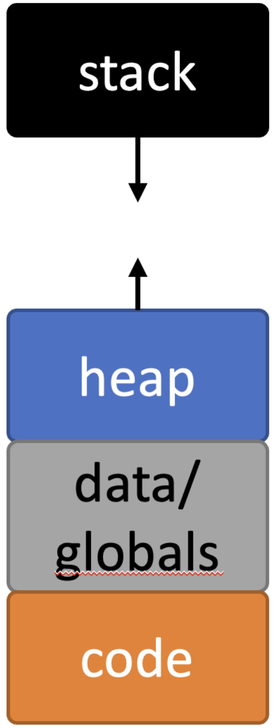
# Process

pid
ppid
user: sibin
cwd: /home/sibin

command arguments
environment variables

descriptors



# File Descriptor Operations

| Operations | Interface |
|---|---|
| pull bytes from channel into buffer | `read()` |
| send bytes from buffer to channel | `write()` |
| duplicate file descriptors | `dup()/dup2()` |
| close a file descriptor | `close()` |

- `printf()` writes to standard out.

Channel: I/O resources, e.g., a file, socket

# dup()/dup2()

```
int dup(int fildes);
```

- duplicates an existing FD (`fildes`) and returns the smallest unused `fd`

```
int dup2(int fildes, int fildes2);
```

- Makes `fildes2` refer to the same I/O resources as `fildes`
- If `fildes2` is already open, it is closed first.
- Returns `fildes2` on success.
- Both return −1 on failure.

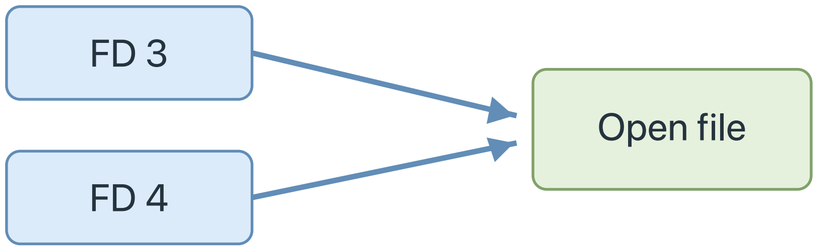
```
1 int fd = open("output.txt", O_WRONLY);
2 dup2(fd, STDOUT_FILENO);  // FD 1 now refers to output.txt
3 close(fd);
4
5 printf("Hello\n");        // Goes to output.txt
```

# Closing a File Descriptor

- `close()` does not necessarily remove the channel.

```
1 int fd1 = open("data.txt", O_RDONLY);  // Assume FD 3
2 int fd2 = dup(fd1);                    // FD 4
3
4 close(fd1);                            // Close FD 3
5 read(fd2, buf, 100);                   // Still works!
6 close(fd2);
```

- Before `close(fd1);`




- After `close(fd1);`

