Operating Systems

(Sistemas Operativos)

The Process Abstraction

University of Minho

2024-2025

What will we learn?

CPU Management

- Although a computer has a limited number of CPUs, users get the impression that several programs (a lot more than the available CPUs) are running simultaneously
  - How does the OS provide this illusion?
  - How can users use the OS APIs to start and stop programs?
  - How does the OS choose what programs will be running, be switched, …?

- Let us start with the **process** abstraction!

Process
Definition

- Process: a running program[1]
- The process abstraction is fundamental for virtualizing the CPU
  - In a given CPU, the OS can start a process, stop it, run another one
- With the right mechanisms (**context switch**) and policies (**scheduling algorithms**)
  - Time sharing and multiprogramming become possible!
  - But, first, one must understand the Process abstraction
- Definitions to take in account:
  - Program: passive entity (e.g., an executable file sitting on disk with several instructions)
  - Process: active entity, executing instructions and accessing system resources

[1] Informal definition, as we will see, one program can create several processes

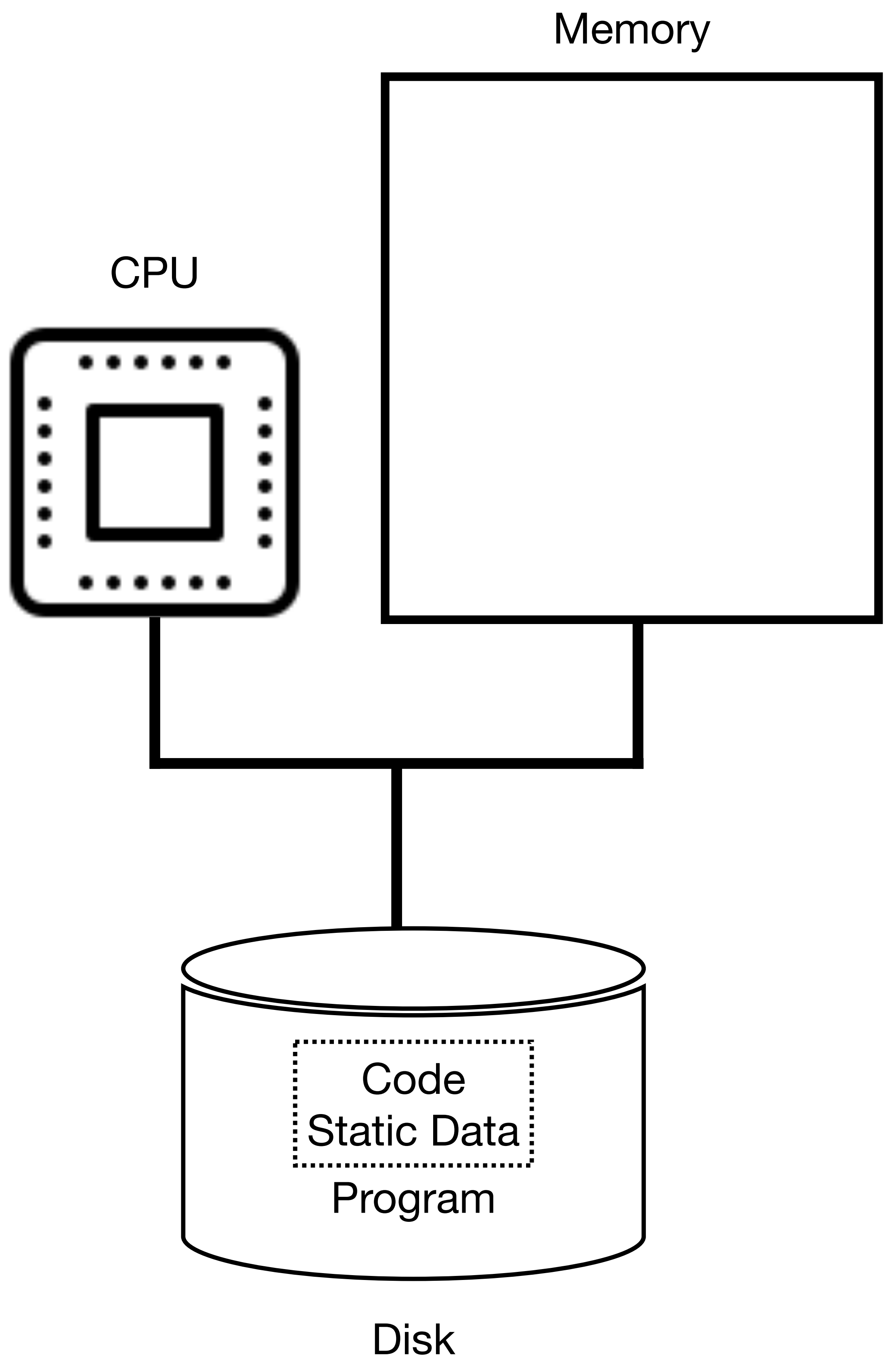
Process
The abstraction

- Being an active entity, the process will access/use system resources
  - CPU - to execute instructions of a program
  - Memory - to keep the instructions, variables, data structures … of a program
  - Disk - to store the program's information persistently
- The current activity of a process can be represented by
  - The value of the **Program Counter** (PC) and other **CPU registers**
    - The PC is a special register identifying the instruction to execute next
    - Many program instructions read/update CPU registers
  - **Address space**, i.e., the memory layout of a process
    - Where the instructions ("code"), data (variables, structures, …) of a process reside
  - List of **open files**, as processes may do I/O

Process Creation

How are programs transformed into processes

- Programs (*i.e.*, code[1] and static data[2]) initially reside on disk as executable files
  - The source code files of programs are compiled into object files (with the aid of a **compiler**)
  - Object files, and required libraries (*e.g.*, *standard C library*), are combined into one executable binary file (with the **linker**)

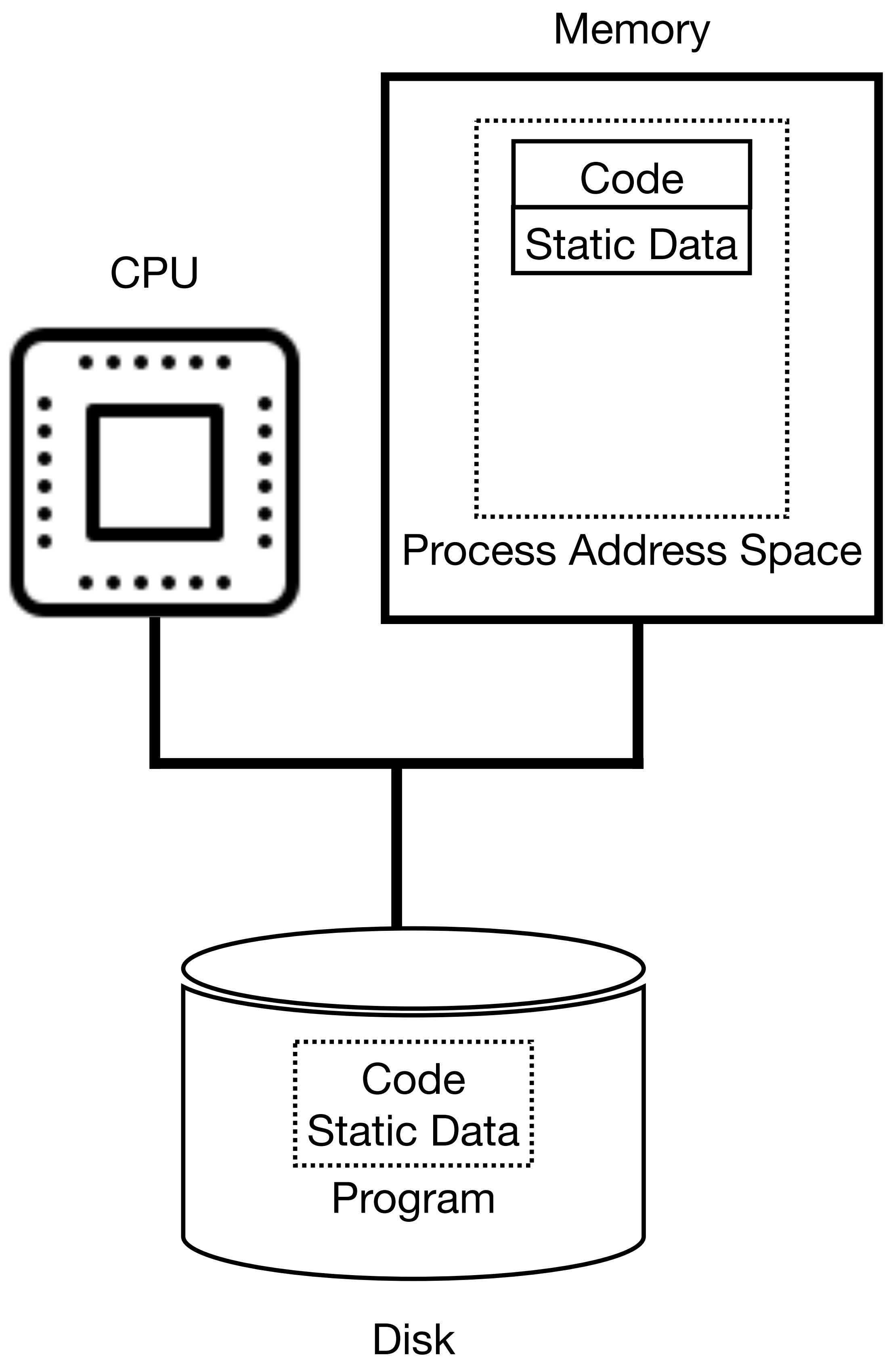


[1] In the literature, the Code segment is also known as Text
[2] Static data (also known as the Data segment in the literature) has, for instance, initialized (global) variables

Process Creation

How are programs transformed into processes

- Programs (*i.e.*, code[1] and static data[2]) initially reside on disk as executable files
  - The source code files of programs are compiled into object files (with the aid of a **compiler**)
  - Object files, and required libraries (*e.g.*, *standard C library*), are combined into one executable binary file (with the **linker**)
- Bytes from the binary file (stored on disk) are loaded into memory (with the **loader**)
  - Loading can be done **eagerly** (all bytes are loaded before running the program) or **lazily** (pieces of the program are loaded as these are needed and while the program is executing)[3]

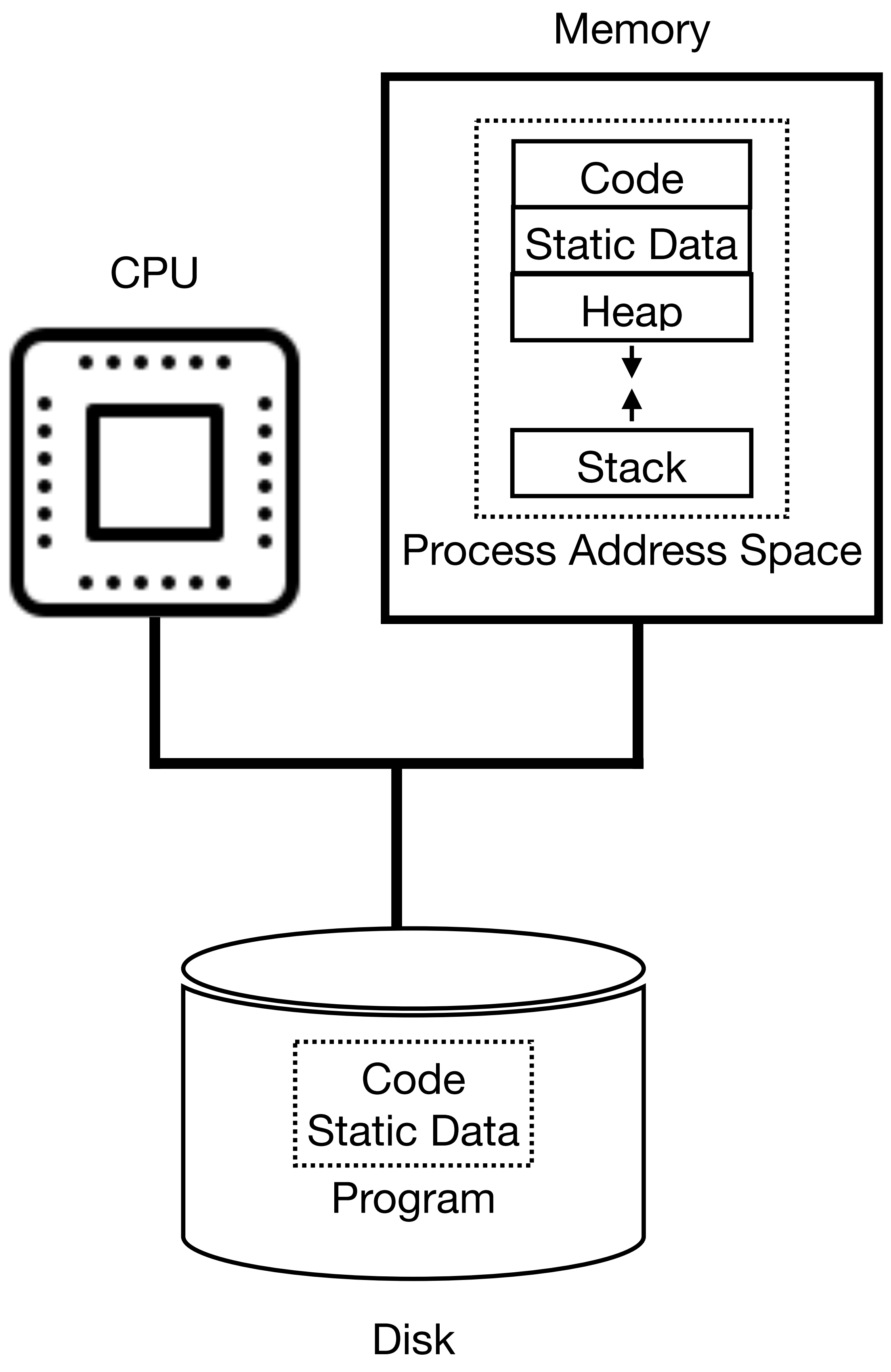


[1] In the literature, the Code segment is also known as Text
[2] Static data (also known as the Data segment in the literature) has, for instance, initialized (global) variables
[3] Modern OSs use lazy loading

Process Creation

How are programs transformed into processes

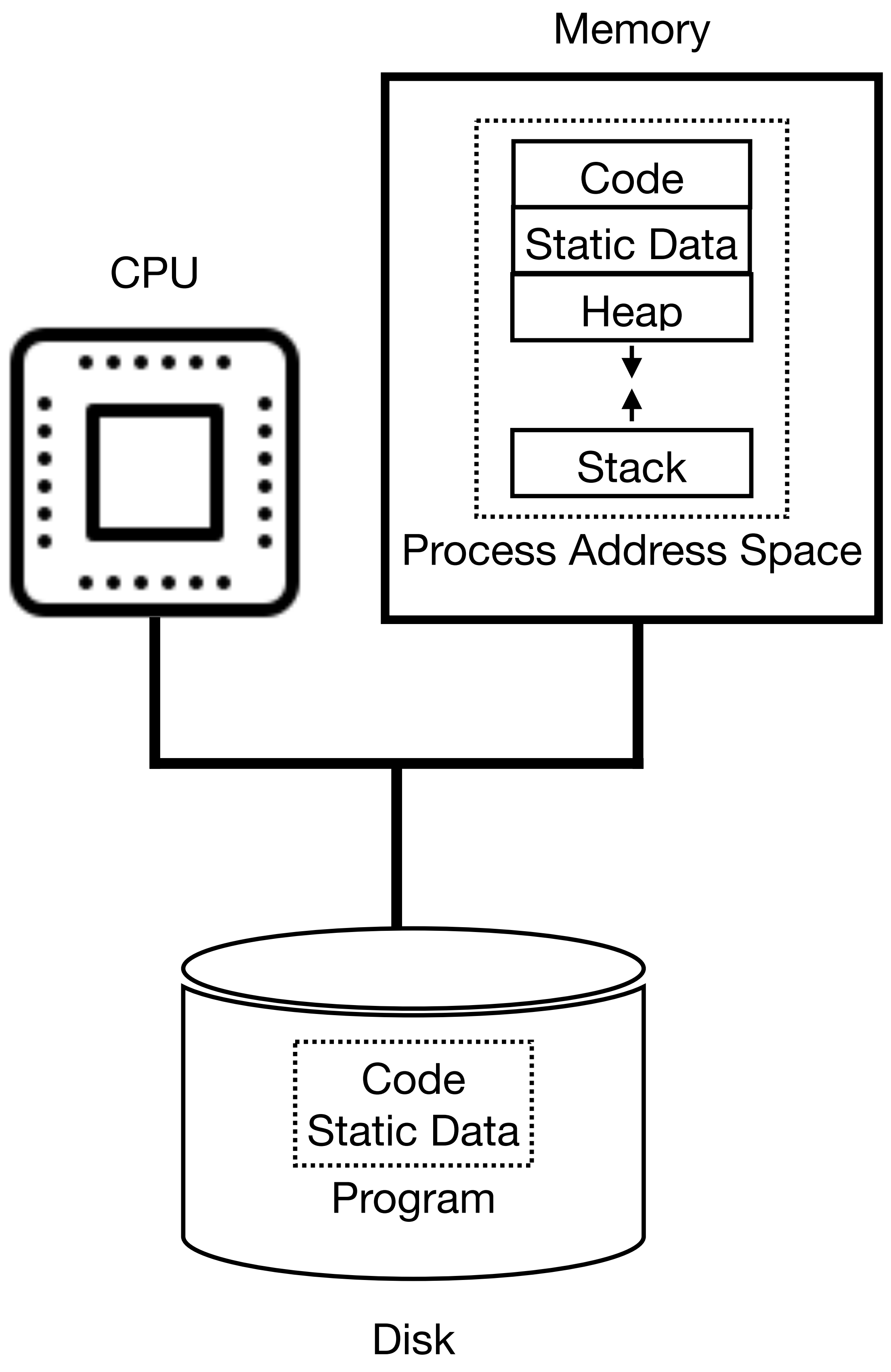
- Once code and static data are loaded into memory, the OS
  - Allocates memory for the **stack** (e.g., arguments, local variables, function parameters, return addresses)
  - Allocates memory for the **heap** (dynamically allocated memory)
  - Does other tasks related to I/O (e.g., initialize default file descriptors for *stdin*, *stdout* and *stderr*)



Process Creation

How are programs transformed into processes

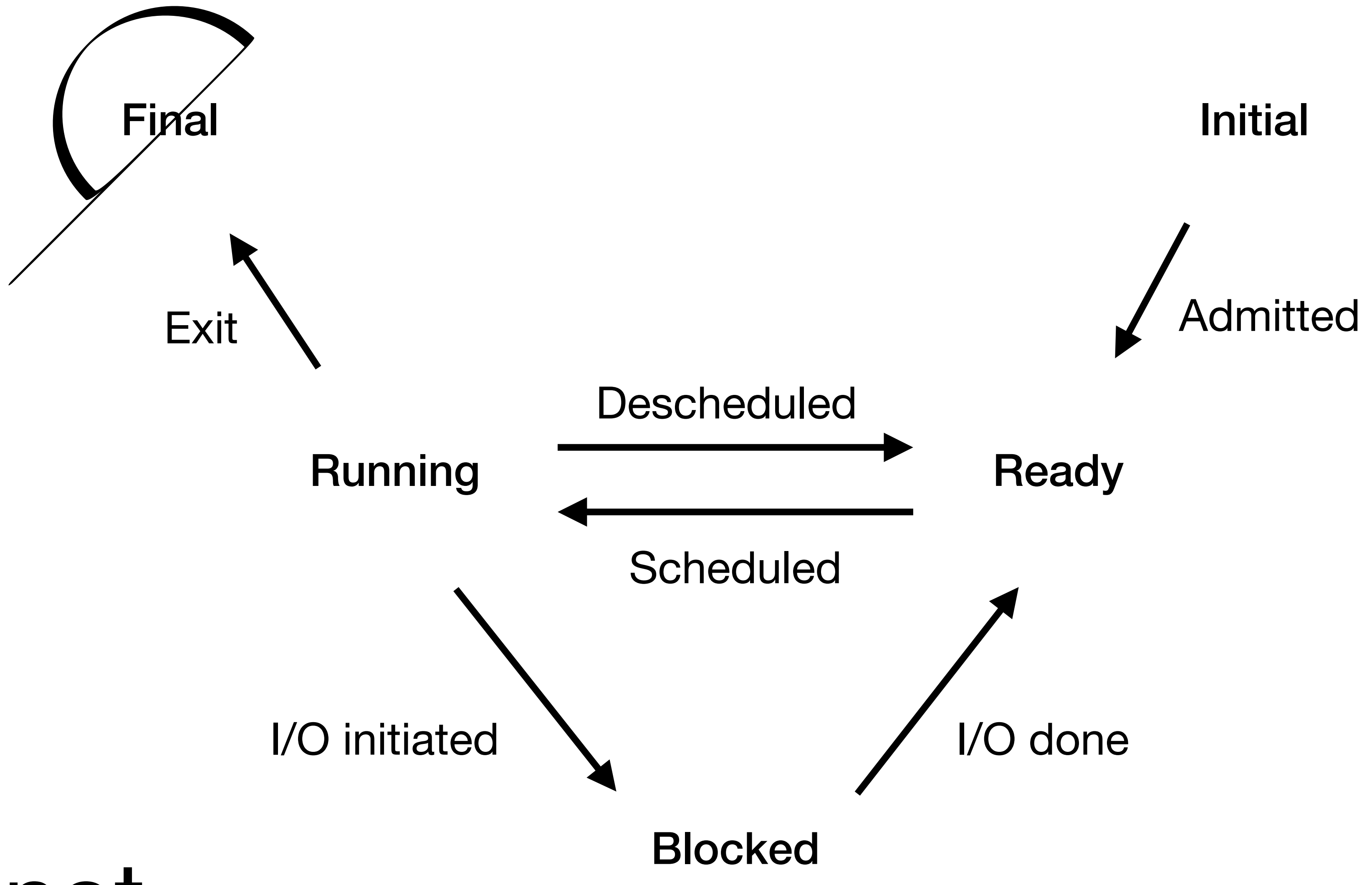
- Once code and static data are loaded into memory, the OS
  - Allocates memory for the **stack** (e.g., arguments, local variables, function parameters, return addresses)
  - Allocates memory for the **heap** (dynamically allocated memory)
  - Does other tasks related to I/O (e.g., initialize default file descriptors for *stdin*, *stdout* and *stderr*)
- Run the program at its entry point, namely the *main()* routine
  - The OS transfers control of the CPU to the newly created process to begin executing



Process Lifecycle

States

- In a simplified view a process can be
  - **Running** on the processor (executing instructions)
  - **Ready** to run (waiting for processor time)
  - **Blocked** until some other event takes place (e.g., when a process makes a disk I/O request it becomes blocked until the request is served)
- We can extend the above states with
  - **Initial -** the process is being created
  - **Final -** the process has finished execution but it was not cleaned up yet (**zombie** state in Unix)
  - Some OSs may further delineate their states

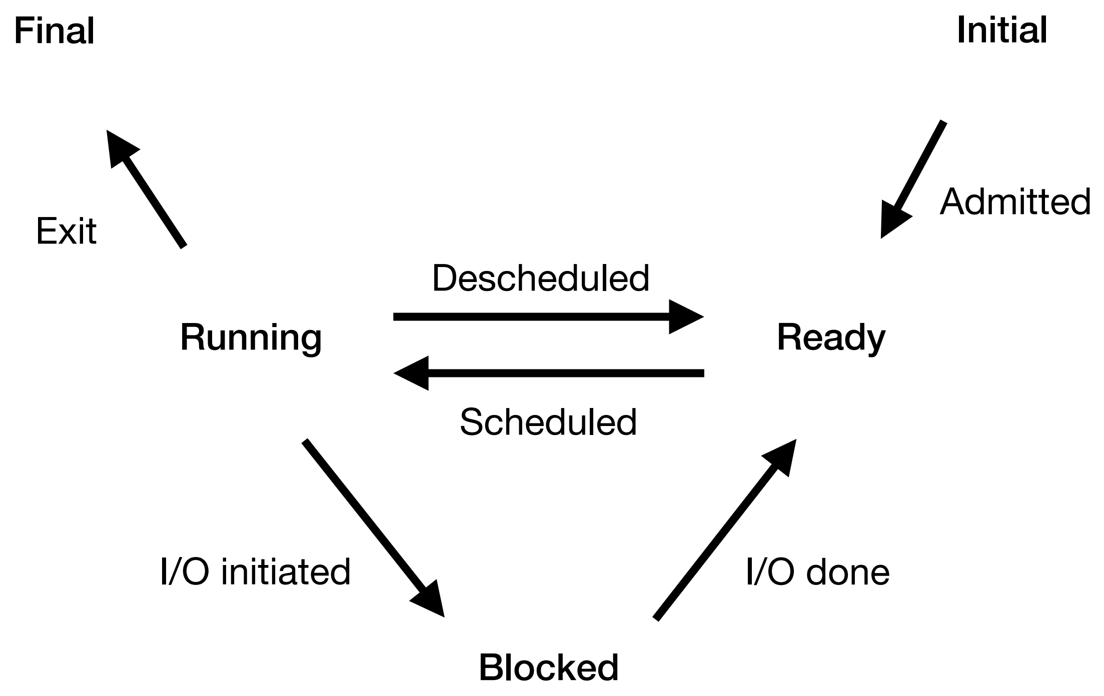


Process Lifecycle

States

- During execution, a process may transition several times through the **Running**, **Ready** and **Blocked** states
- At the example on the right
  - $Process_0$ blocks for I/O and $Process_1$ is scheduled (time 4)
  - $Process_0$ is ready but the OS decides to keep $Process_1$ running (time 7)
- Questions
  - Why is it a good idea to schedule another process when $Process_1$ blocks for I/O (time 4)?
  - Should one schedule immediately $Process_0$ when I/O is done (time 7)
- These decisions are made by the OS **scheduler**

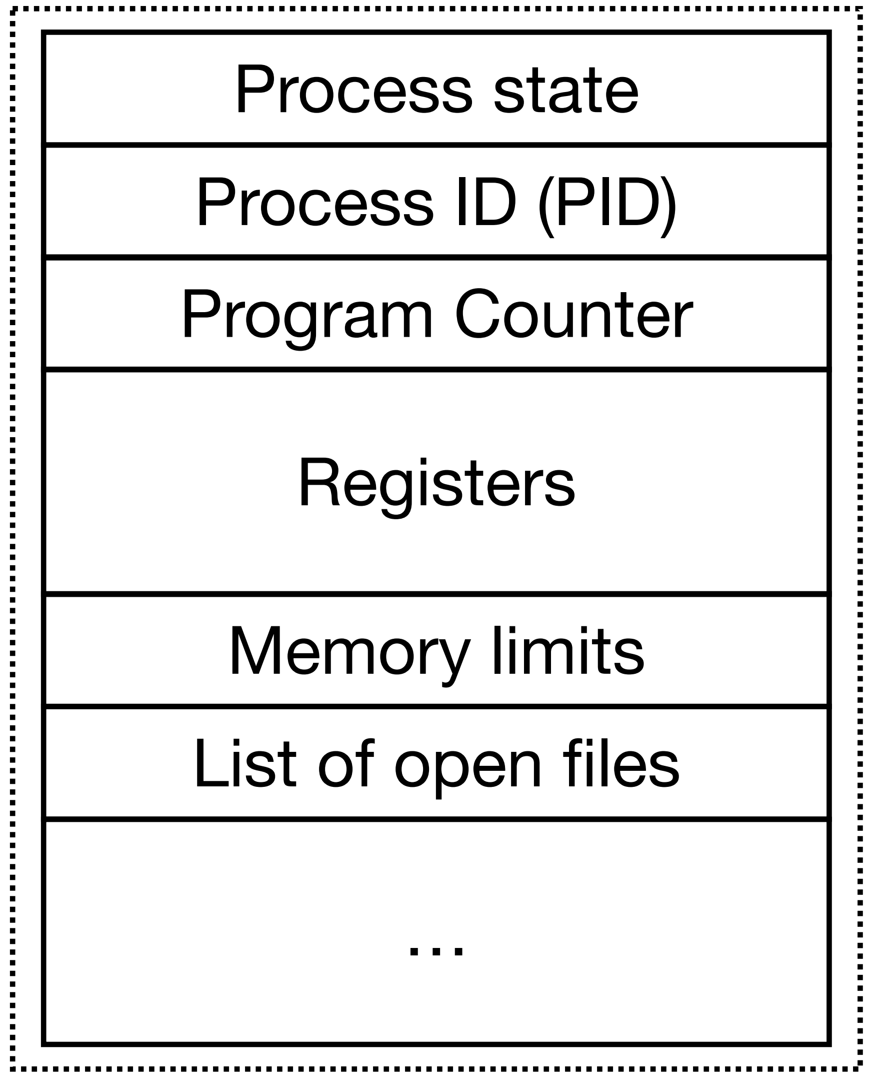
Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked, so $Process_1$ runs
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O is done
8	Ready	Running	$Process_1$ finishes
9	Running	Finished	
10	Running	Finished	$Process_0$ finishes



Operating System

Data Structures

- The OS is a program so it uses data structures!
  - A **process list** is used to track the state of processes (*i.e.,* **Running**, **Ready**, **Blocked**)
  - Each entry of this list is called a **Process Control Block** (**PCB**)



PCB

- PCB
  - **Process State:** Running, Ready, Blocked, …
  - **Process ID (PID):** Unique identifier for the process
  - **Program Counter:** memory address of the next instruction to execute
  - **Registers:** CPU registers that must be saved in memory so the process may be resumed later (e.g., stack pointers)
  - **Memory limits:** The memory limits (address space) the process may access[1]
  - **List of open files:** Files opened by the process[1]

[1] To be further discussed at the memory management and persistence classes!

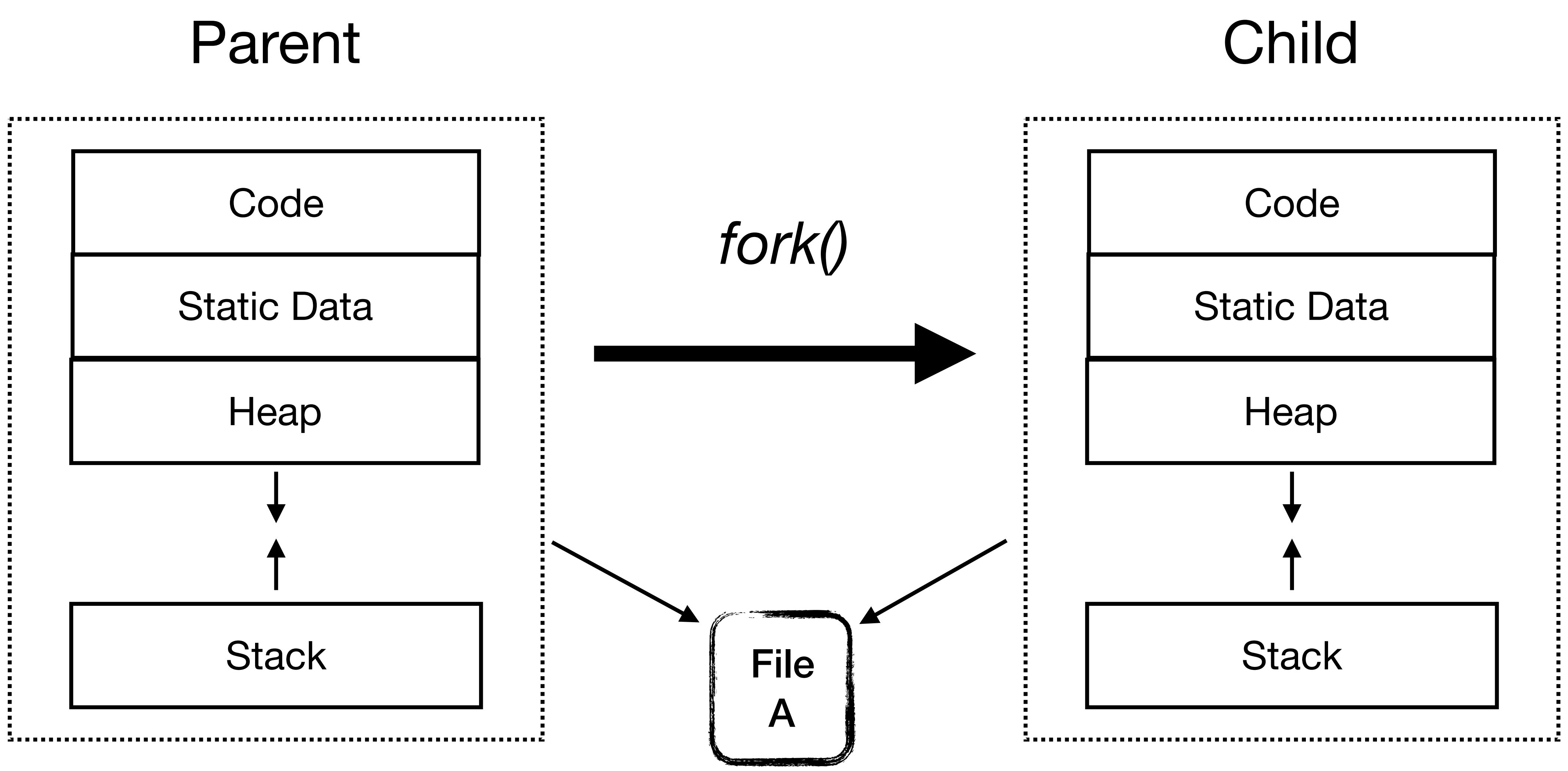
Process API

- The API must allow to
  - **Create** new processes, otherwise, how can one launch a new program?
  - **Destroy** processes, when one wishes to terminate processes non-gracefully
  - **Wait** for a process until it finishes its execution
  - Check the **status** of a process (e.g., its current state, execution time)
  - Other controls such as **suspending** and **resuming** a process on demand

UNIX Process API

Fork

- The *fork()* system call is used to create a new process - *pid_t fork(void)*
- When fork is called by the **parent** process, the **child** process, an almost identical copy of the calling process is created
  - The **child process has its own copy** of the address space, registers, PC, list of open files

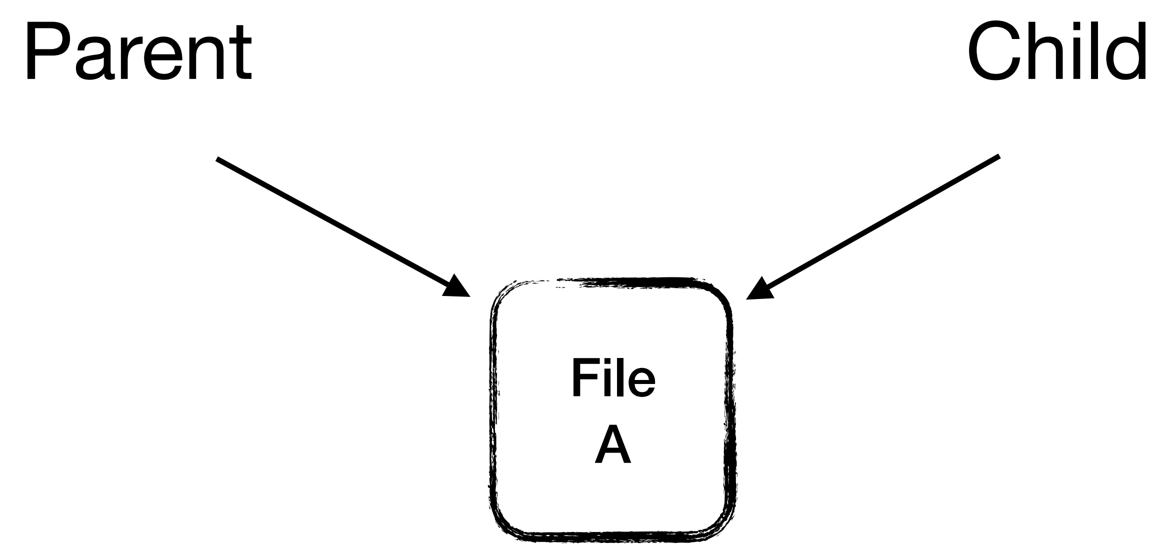


Copying != Sharing

UNIX Process API

Fork

- The **value returned** by *fork()* to each process is different
  - The parent receives the **PID of the newly-created child**
  - The child receives a return code of **zero**
- Two processes are now running **concurrently**
  - Accessing an independent address space (memory)
  - **Question:** What if the two processes write to *file A*?



- Two processes are now executing the instructions of the same program
  - **Question:** How can one specify code to be executed by each process?

Fork

Example (Try it!)

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char* argv[]){

    printf("Hello (pid:%d)\n", getpid());
    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        // Do stuff (child)
    }else{
        printf("Parent of %d (pid: %d)\n", rc_fork, getpid());
        //Do stuff (parent)
    }

    return 0;
}
```

Prompt

```
prompt> ./p1
```

- Try compiling and running this code!

Fork

Example (Try it!)

Process 44562

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char* argv[]){

    printf("Hello (pid:%d)\n", getpid());
    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        // Do stuff (child)
    }else{
        printf("Parent of %d (pid: %d)\n", rc_fork, getpid());
        //Do stuff (parent)
    }

    return 0;
}
```

Prompt

```
prompt> ./p1
Hello (pid:44562)
```

- When the program starts, the process prints the message “Hello” and its PID - 44562

Fork

Example (Try it!)

Process 44562
(parent)

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char* argv[]){

    printf("Hello (pid:%d)\n", getpid());
    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        // Do stuff (child)
    }else{
        printf("Parent of %d (pid: %d)\n", rc_fork, getpid());
        //Do stuff (parent)
    }

    return 0;
}
```

Process 44563
(child)

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char* argv[]){

    printf("Hello (pid:%d)\n", getpid());
    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        // Do stuff (child)
    }else{
        printf("Parent of %d (pid: %d)\n", rc_fork, getpid());
        //Do stuff (parent)
    }

    return 0;
}
```

Prompt

prompt> ./p1
Hello (pid:44562)

- When *fork()* is called a new process (child) is created, which is an almost identical copy of the calling process (parent)

Fork

Example (Try it!)

Process 44562
(parent)

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char* argv[]){

    printf("Hello (pid:%d)\n", getpid());
    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        // Do stuff (child)
    }else{
        printf("Parent of %d (pid: %d)\n", rc_fork, getpid());
        //Do stuff (parent)
    }

    return 0;
}
```

Process 44563
(child)

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char* argv[]){

    printf("Hello (pid:%d)\n", getpid());
    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        // Do stuff (child)
    }else{
        printf("Parent of %d (pid: %d)\n", rc_fork, getpid());
        //Do stuff (parent)
    }

    return 0;
}
```

Prompt

prompt> ./p1
Hello (pid:44562)
Parent of 44563 (pid: 44562)
Child (pid: 44563)
prompt>

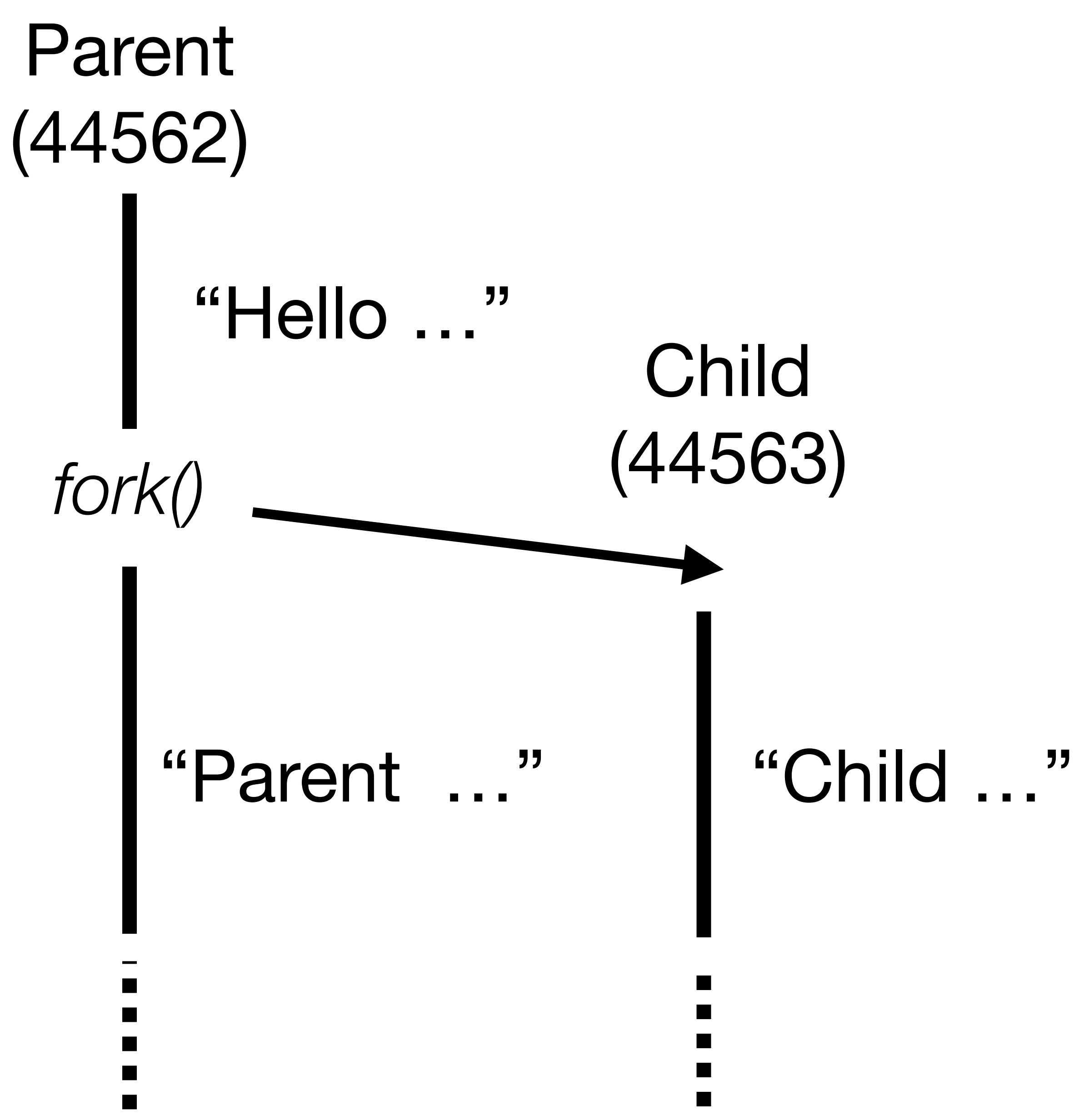
- The child does not start running at *main(),* it starts as it if had called *fork()* itself (“Hello” is only printed once)
- The child is not an identical copy since it receives a different return code from *fork().*
  Useful for specifying different code (doing different actions) for each process

Fork

Concurrency

- **Question:** If the previous program runs multiple times, the messages *“Parent …”* and *“Child …”* may be switched. Why?
- **Question:** even assuming a single CPU, the output is non deterministic, why?
- Non-determinism leads to interesting **concurrency** problems
  - Although the memory of each process is isolated (private), processes may share access to other resources (e.g., files)
  - Memory is not isolated for multi-threaded programs
    - **Threads** of a process have their own program counter but share the process’s address space

```
prompt> ./p1
Hello (pid:44562)
Parent of 44563 (pid: 44562)
Child (pid: 44563)
prompt>
```



Fork

Another example

- **Question:** And with this code…
  - How many times is the message *“I’m Here”* printed?
  - What process(es) print that message?
  - What messages can be switched?

```c
int main(int argc, char* argv[]){

    printf("Hello (pid:%d)\n", getpid());
    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        // Do stuff (child)
    }else{
        printf("Parent of %d (pid: %d)\n", rc_fork, getpid());
        //Do stuff (parent)

    }
    printf("I'm here (pid: %d)\n", getpid());
    return 0;
}
```

UNIX Process API

Wait

- What if one wants the parent to wait for its child to finish executing?
- The *wait()* system call blocks the execution of the calling process until one of the child processes ends - *pid_t wait(int *status);*
  - **Disclaimer:** Some variants of wait() may not block until the child process exits

Wait

example

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

int main(int argc, char* argv[]){

    printf("Hello (pid:%d)\n", getpid());
    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        // Do stuff (child)
        _exit(1);
    }else{
        //Do stuff (parent)
        pid_t rc_wait = wait(NULL);
        printf("Parent of %d (rc_wait: %d) (pid: %d)\n", rc_fork, rc_wait, getpid());
        //Do stuff (parent)
    }
    return 0;
}
```

```
prompt> ./p1
Hello (pid:44866)
Child (pid: 44867)
Parent of 44867 (rc_wait: 44867) (pid: 44866)
prompt>
```

Wait

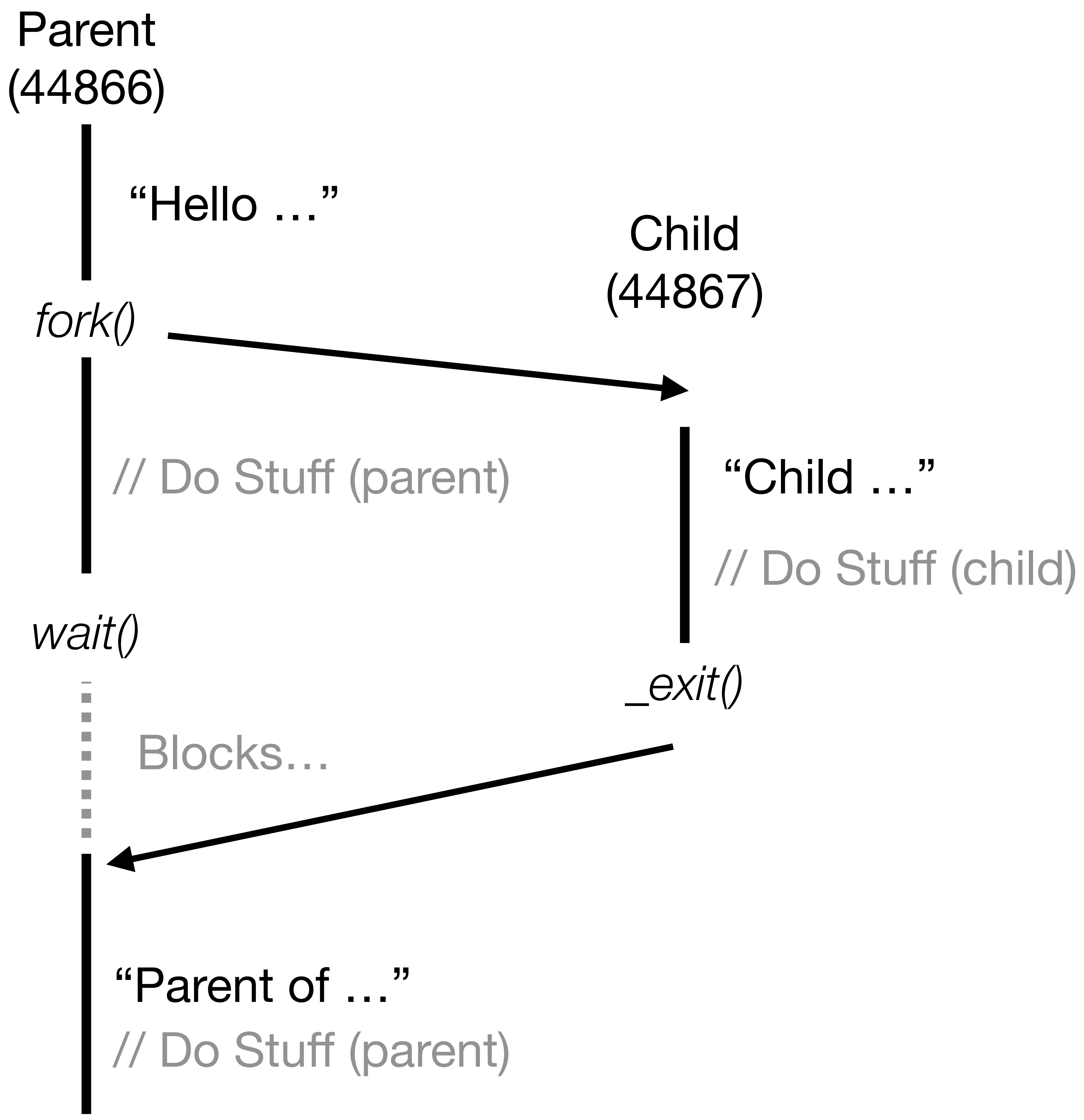
example

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

int main(int argc, char* argv[]){

    printf("Hello (pid:%d)\n", getpid());
    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        // Do stuff (child)
        _exit(1);
    }else{
        //Do stuff (parent)
        pid_t rc_wait = wait(NULL);
        printf("Parent of %d (rc_wait: %d) (pid: %d)\n", rc_fork, rc_wait, getpid());
        //Do stuff (parent)
    }
    return 0;
}
```

```
prompt> ./p1
Hello (pid:44866)
Child (pid: 44867)
Parent of 44867 (rc_wait: 44867) (pid: 44866)
prompt>
```

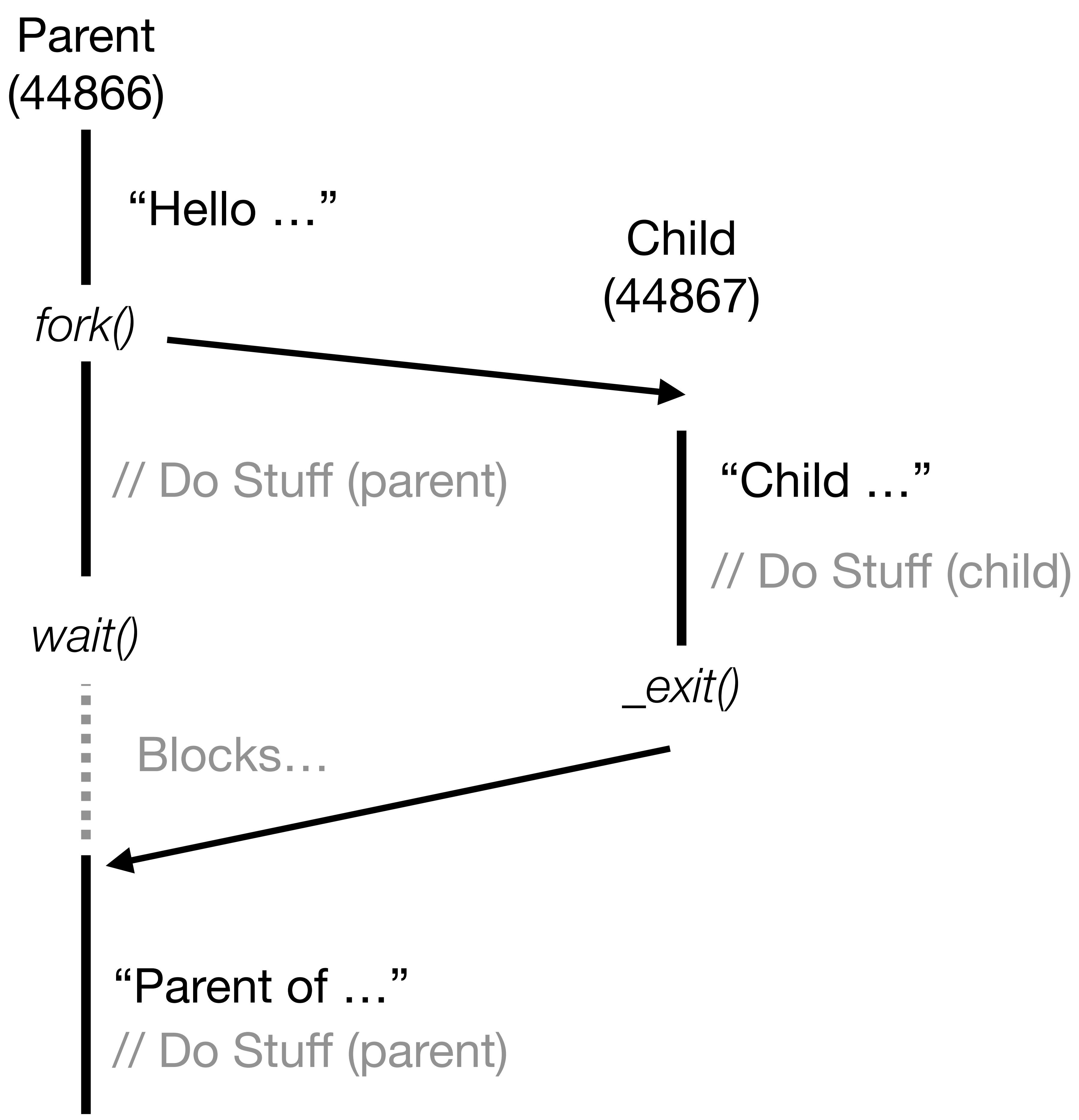


Fork and Wait

Concurrency

- **Question:** If the previous program runs multiple times, the messages *“Child …”* and *“Parent …”* are printed always in the same order. Why?

```
prompt> ./p1
Hello (pid:44866)
Child (pid: 44867)
Parent of 44867 (rc_wait: 44867) (pid: 44866)
prompt>
```



Other details

Interesting things to see at the practical labs (Pt1)

- Do not forget to handle errors (when system calls return -1…)!
- Look at the *wait()* man page
  - Explore the *status* argument and the *WIFEXITED* and *WEXITSTATUS* macros
  - Explore the *waitpid()* call
- What happens if the child exits before the parent calls *wait()*?
  - The child goes into a ***zombie*** state (waiting for the parent to collect its status)
- What happens if the parent exits while the child is still executing?
  - The child becomes **orphaned**, being adopted by another process.
    For example, the *init process* (*i.e.,* the initial process ran by UNIX systems)

Other details

Interesting things to see at the practical labs (Pt 2)

- How many processes does the code below creates?
- Can you draw the tree of processes?

PID
1945

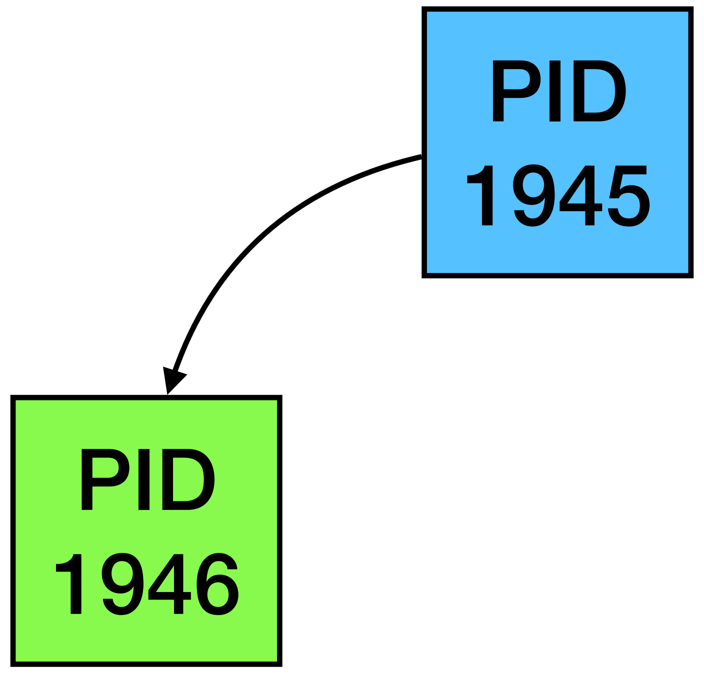
```
int main(int argc, char* argv[]){
    fork()
    fork()
    fork()
}
```

Other details

Interesting things to see at the practical labs (Pt 2)

- How many processes does the code below creates?
- Can you draw the tree of processes?

```
int main(int argc, char* argv[]){
    fork()
    fork()
    fork()
}
```

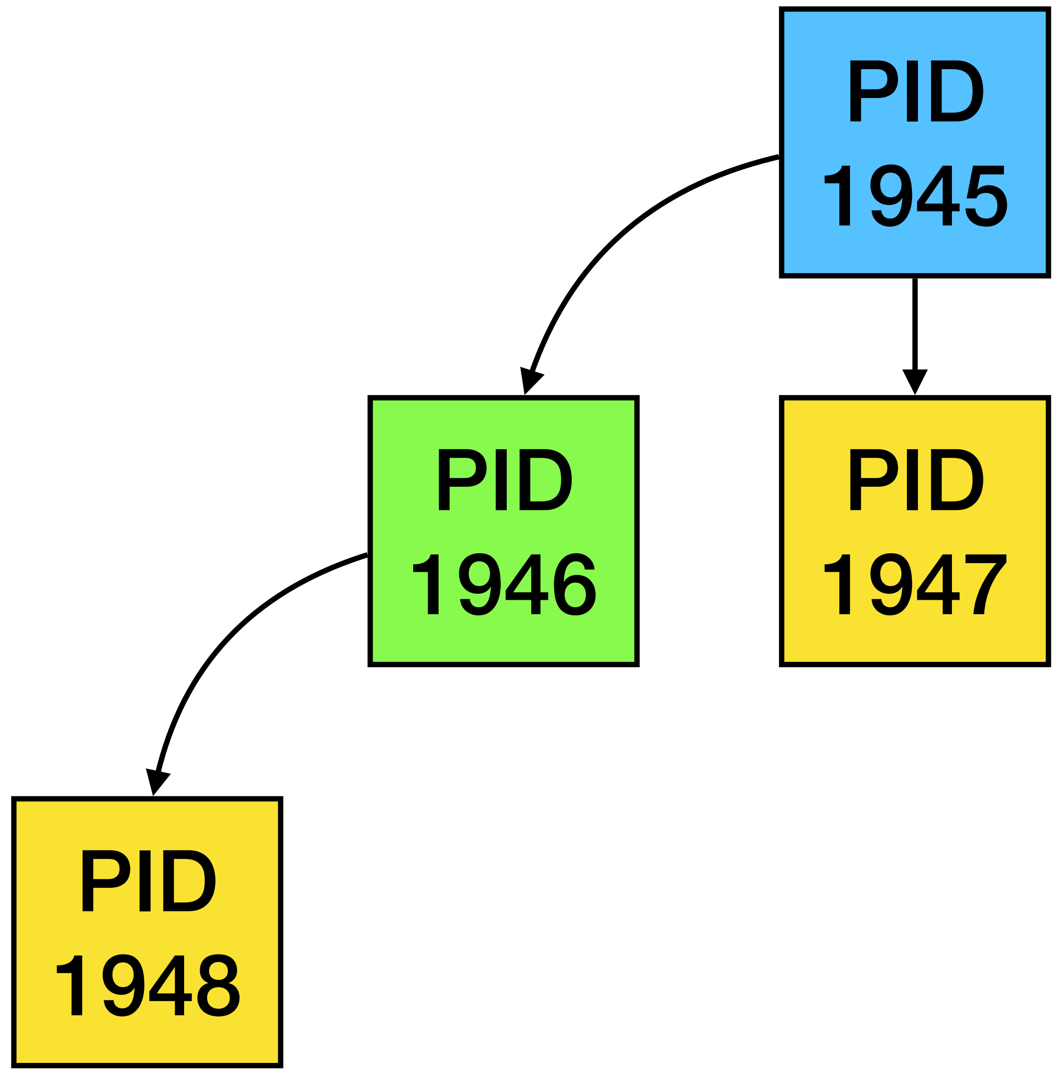


Other details

Interesting things to see at the practical labs (Pt 2)

- How many processes does the code below creates?
- Can you draw the tree of processes?

```
int main(int argc, char* argv[]){
    fork()
    fork()
    fork()
}
```

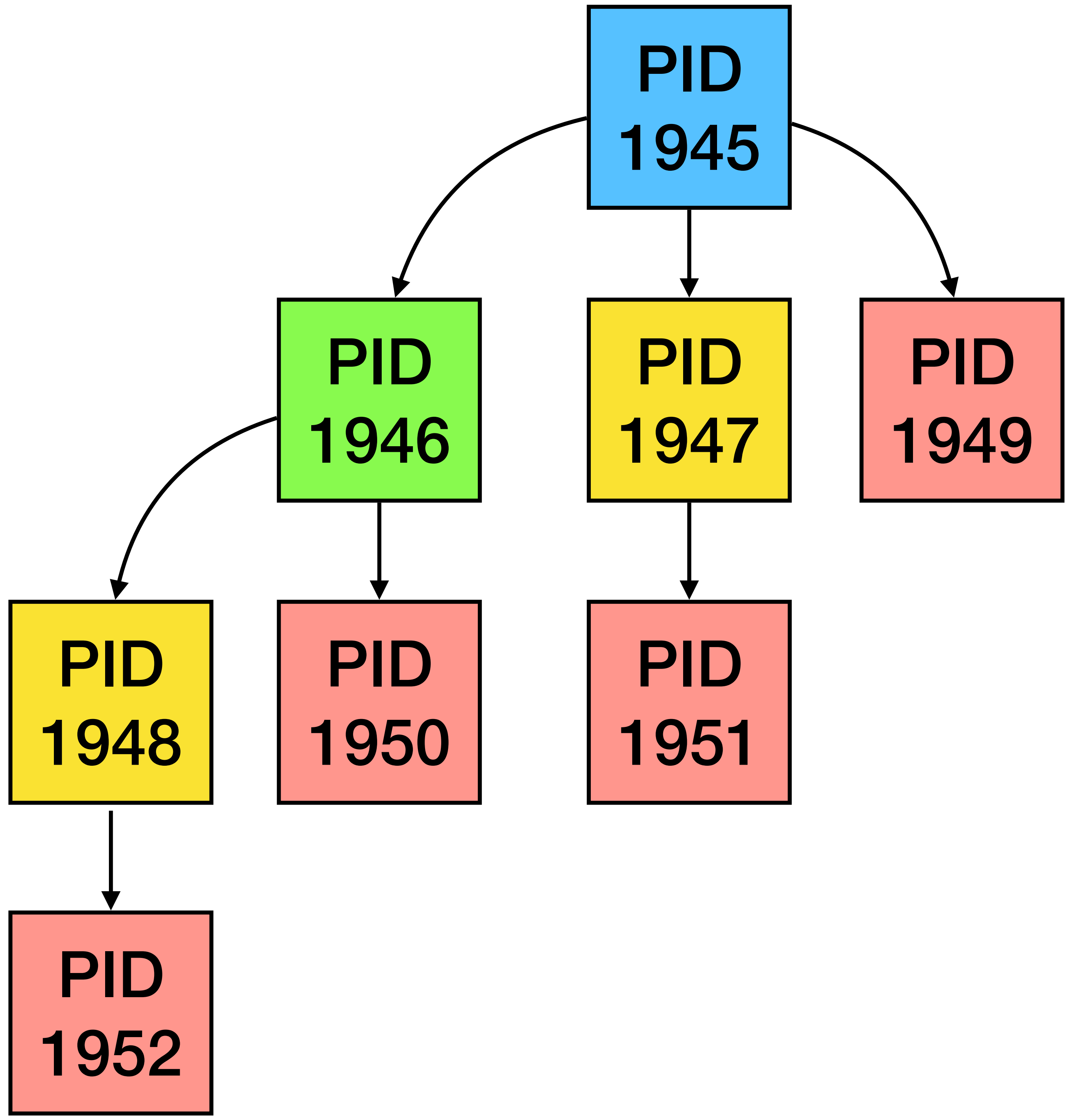


Other details

Interesting things to see at the practical labs (Pt 2)

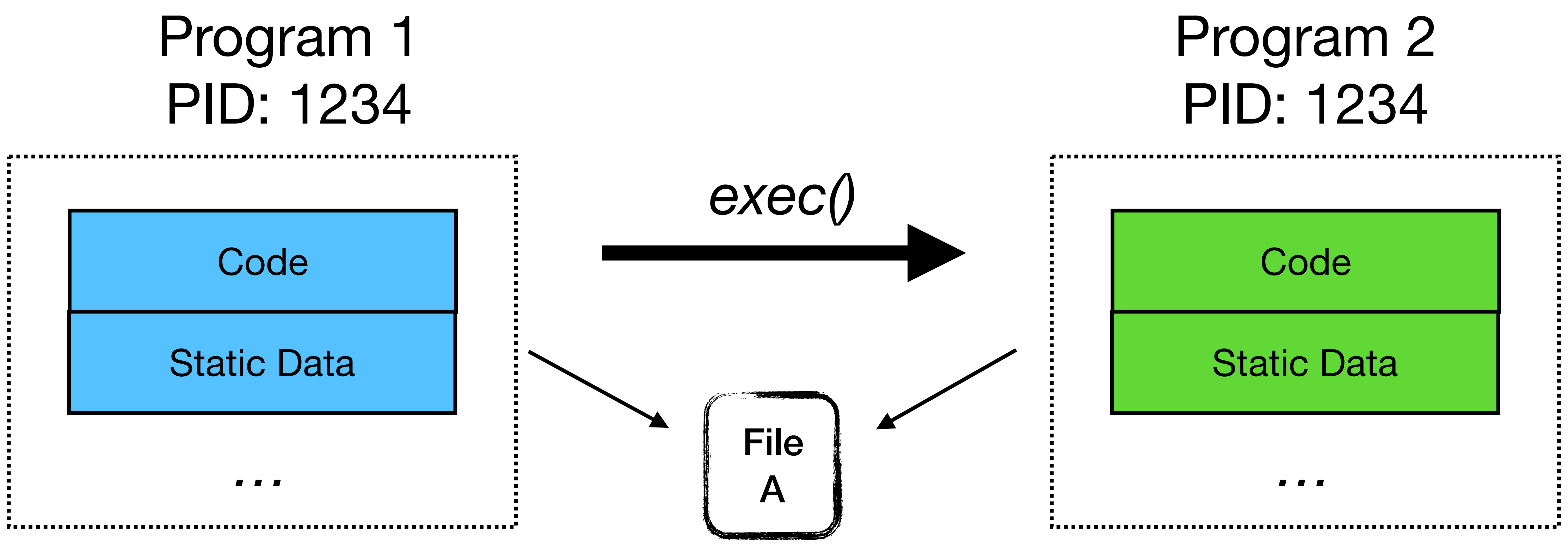
- How many processes does the code below creates?
- Can you draw the tree of processes?

```
int main(int argc, char* argv[]){
    fork()
    fork()
    fork()
}
```



UNIX Process API
Exec

- What if one wants the child process to run a different program?
- The *exec()* system call replaces the code and static data of the current process with the ones from a new program
  - Heap and stack are reinitialized
  - The new program starts at the *main()* entry point
  - PID is maintained (same process)
  - Initialized resources (e.g., opened files) are still usable by the process



- There are several variants of exec (e.g., *execv()*, *execp()*, *execlp()*, …)

Exec
example

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

int main(int argc, char* argv[]){

    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");      // program: wc
        myargs[1] = strdup("exec.c"); // arg: file to count
        myargs[2] = NULL;              // mark end of array
        execvp(myargs[0], myargs);     // runs word count
        printf("I'm Here Child (pid: %d)\n", getpid());
        _exit(1);
    }else{
        pid_t rc_wait = wait(NULL);
        printf("Parent of %d (pid: %d)\n", rc_fork, getpid());
    }
    return 0;
}
```

- *int execvp(const char *file, char *const argv[]);*
  - **file:** path/name of the program
  - **argv:** list of arguments of the program (terminated with NULL)
- The *wc* program executes on the child process
- **Question:** Is *“I’m Here Child …”* printed in this program ?

Output of *wc*

prompt> ./p1
Child (pid: 77262)
23 77 648 exec.c
Parent of 77262 (pid: 77261)
prompt>

Try running the following command in your shell: *wc exec.c*

Exec

example

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

int main(int argc, char* argv[]){

    pid_t rc_fork = fork();
    if(rc_fork == 0){
        printf("Child (pid: %d)\n", getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");      // program: wc
        myargs[1] = strdup("exec.c"); // arg: file to count
        myargs[2] = NULL;              // mark end of array
        execvp(myargs[0], myargs);    // runs word count
        printf("I'm Here Child (pid: %d)\n", getpid());
        _exit(1);
    }else{
        pid_t rc_wait = wait(NULL);
        printf("Parent of %d (pid: %d)\n", rc_fork, getpid());
    }
    return 0;
}
```

- *int execvp(const char *file, char *const argv[]);*
  - **file:** path/name of the program
  - **argv:** list of arguments of the program (terminated with NULL)
- The *wc* program executes on the child process
- **Question:** Is *"I'm Here Child …"* printed in this program ?
  - exec() only returns to the original program if unsuccessful

Output of *wc*

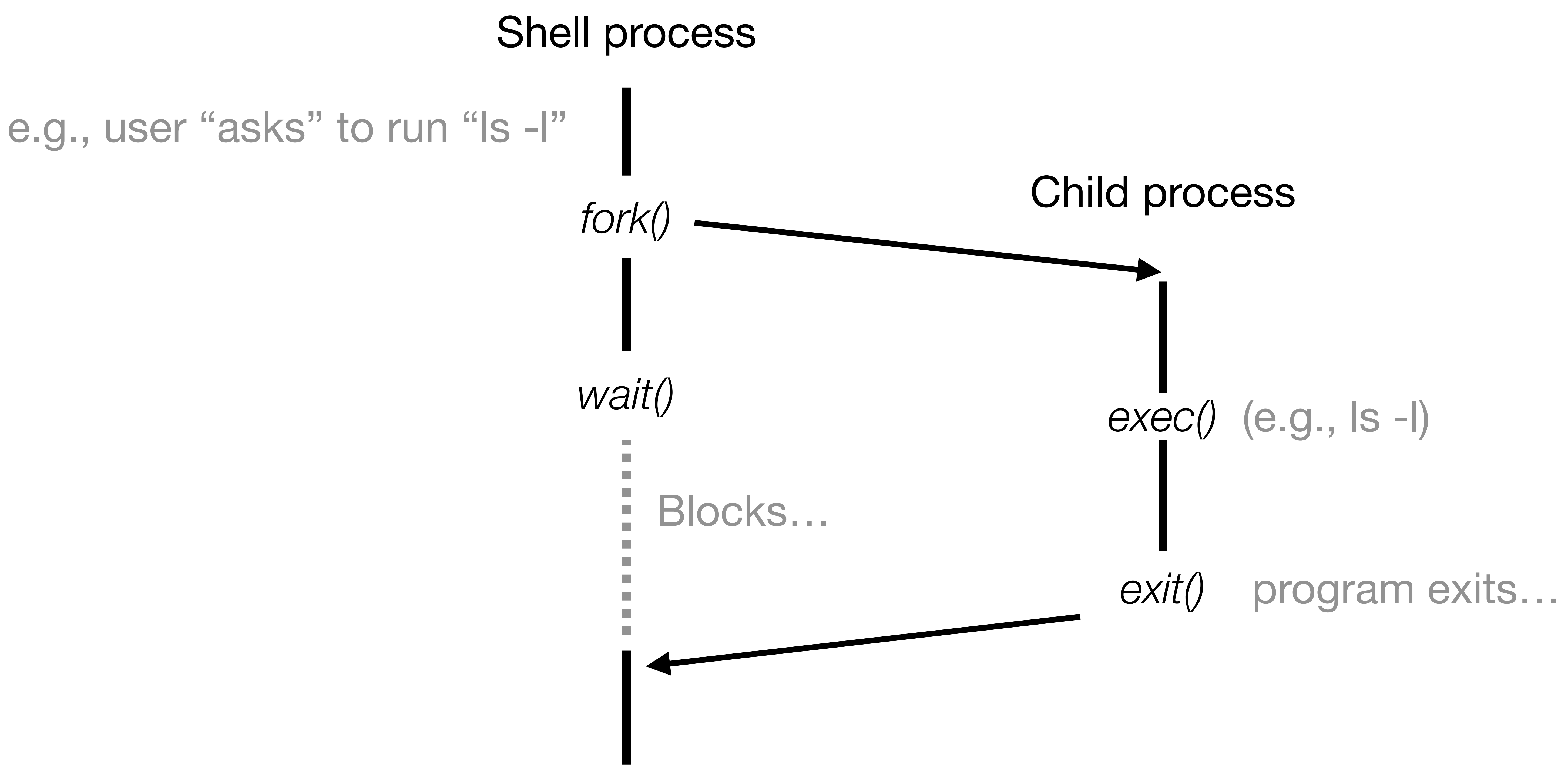
```
prompt> ./p1
Child (pid: 77262)
   23      77     648 exec.c
Parent of 77262 (pid: 77261)
prompt>
```

Try running the following command in your shell: *wc exec.c*

Fork, Exec, Wait

A powerful combination

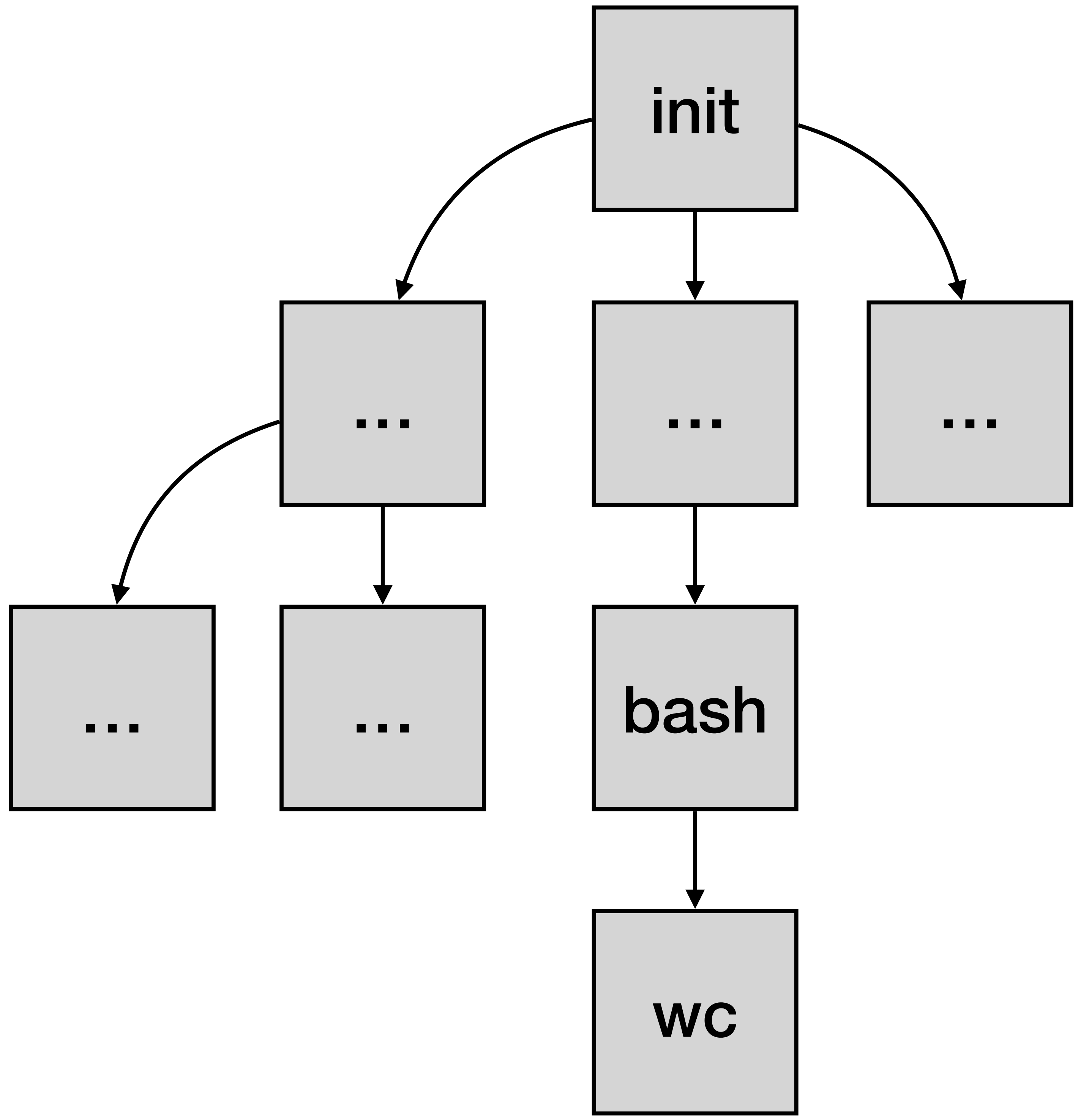
- **Question**: How does shell (bash) executes your programs?



Process Tree

In Unix and Linux

- In UNIX and Linux systems, processes are organized into a tree hierarchy
  - The first process to run after booting the OS is called **init**
  - All other processes descend from **init**
- Check the **pstree** tool



Motivating the API

Why using fork() + exec() to run programs?

- Calling fork() and then exec() allows setting up the environment for new programs and to do interesting commands…
- **Example 1:** *“wc /etc/passwd > newfile.txt”*
  - Check the practical labs for the *dup()* system call
- **Example 2:** *“grep -v ^# /etc/passwd | wc”*
  - Check the practical labs for the *pipe()* and *dup()* system calls

Other system calls and tools

In Unix

- *kill()* system call used to
  - Pause, resume, force termination of processes
  - Configured as keystroke combinations in most UNIX shells
- Useful tools
  - **ps**, **pstree**, and **top** (check information about processes and system resources)
  - **kill** (same name as function but provided as a shell tool)

More Information

- **Chapters 4 and 5** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces.** Arpaci-Dusseau Books, 2018.
- Avi Silberschatz, Peter Baer Galvin, Greg Gagne. **Operating System Concepts (10. ed)**. John Wiley & Sons, 2018.

Questions?