CSC 112: Computer Operating Systems
Lecture 2

Processes

Department of Computer Science,

Hofstra University

Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming

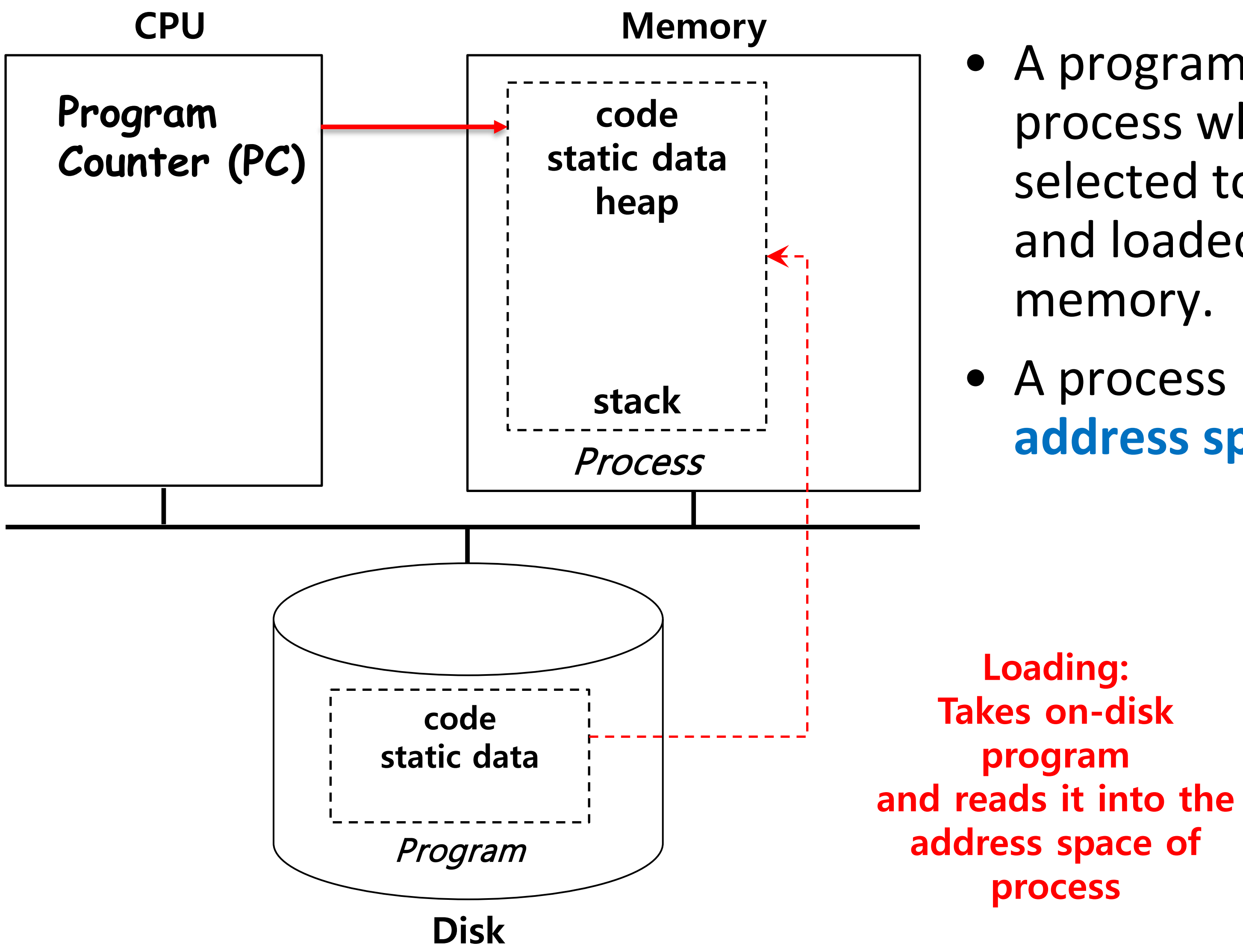
Overview

- Process concept
- Process state
- Process API (creation, wait)
- Process tree

Process

- Program is a ***static*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
  - Process is an abstraction of CPU
- Execution of program started via Graphic User Interface (GUI) mouse clicks, command line entry of its name, etc
- A physical CPU is shared by many processes
  - Time sharing: run one process for a little while, then run another one, and so forth.
  - Processes believe they are using CPU alone

Process

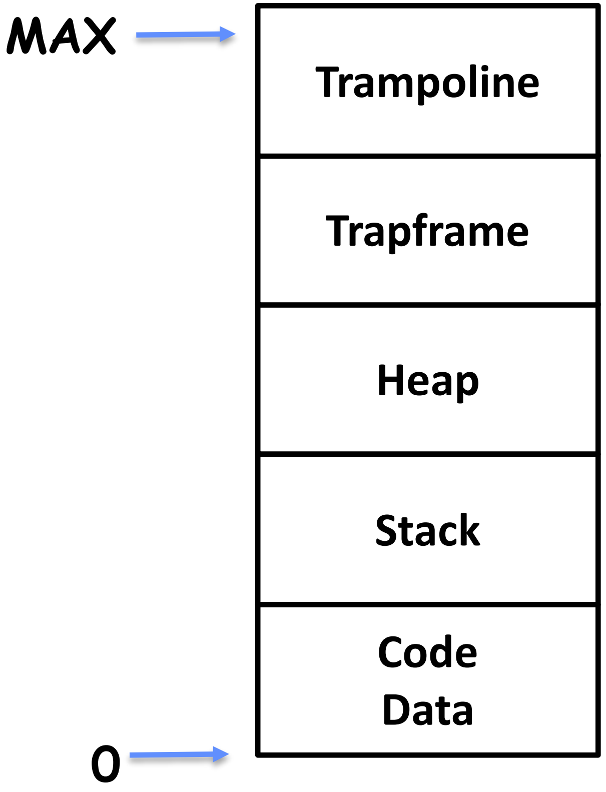


- A program becomes a process when it is selected to execute and loaded into memory.
- A process has an **address space**

Process

Process: a running program

- **Consists of:**
  - Code: Instructions
  - Stack: Temporary data, e.g., function parameters, returned addresses, local variables
  - Registers: Program counter (PC), general purpose, stack pointer
  - Data: Global variables
  - Heap: Dynamically allocated



XV6 Address Space

Process

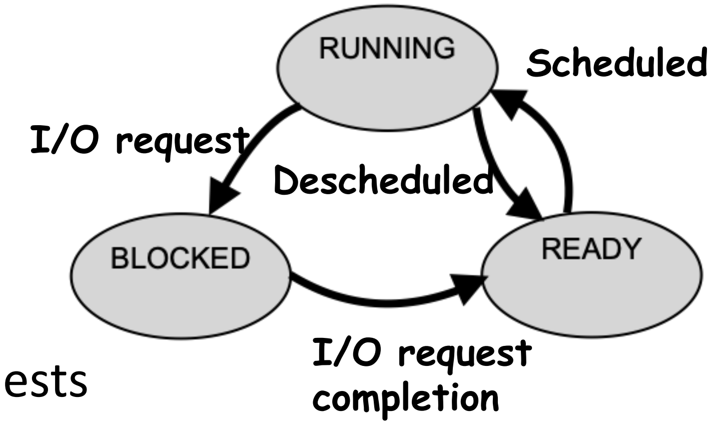
```c
struct proc {
	struct spinlock lock; // p->lock must be held when using these:
	enum procstate state; // Process state
	void *chan; // If non-zero, sleeping on chan
	int killed; // If non-zero, have been killed
	int xstate; // Exit status to be returned to parent's wait
	int pid; // Process ID
	// wait_lock must be held when using this:
	struct proc *parent; // Parent process
	// these are private to the process, so p->lock need not be
held.
	uint64 kstack; // Virtual address of kernel stack
	uint64 sz; // Size of process memory (bytes)
	pagetable_t pagetable; // User page table
	struct trapframe *trapframe; // data page for trampoline.S
	struct context context; // swtch() here to run process
	struct file *ofile[NOFILE]; // Open files
	struct inode *cwd; // Current directory
	char name[16]; // Process name (debugging)
};
```

XV6 (proc.h)

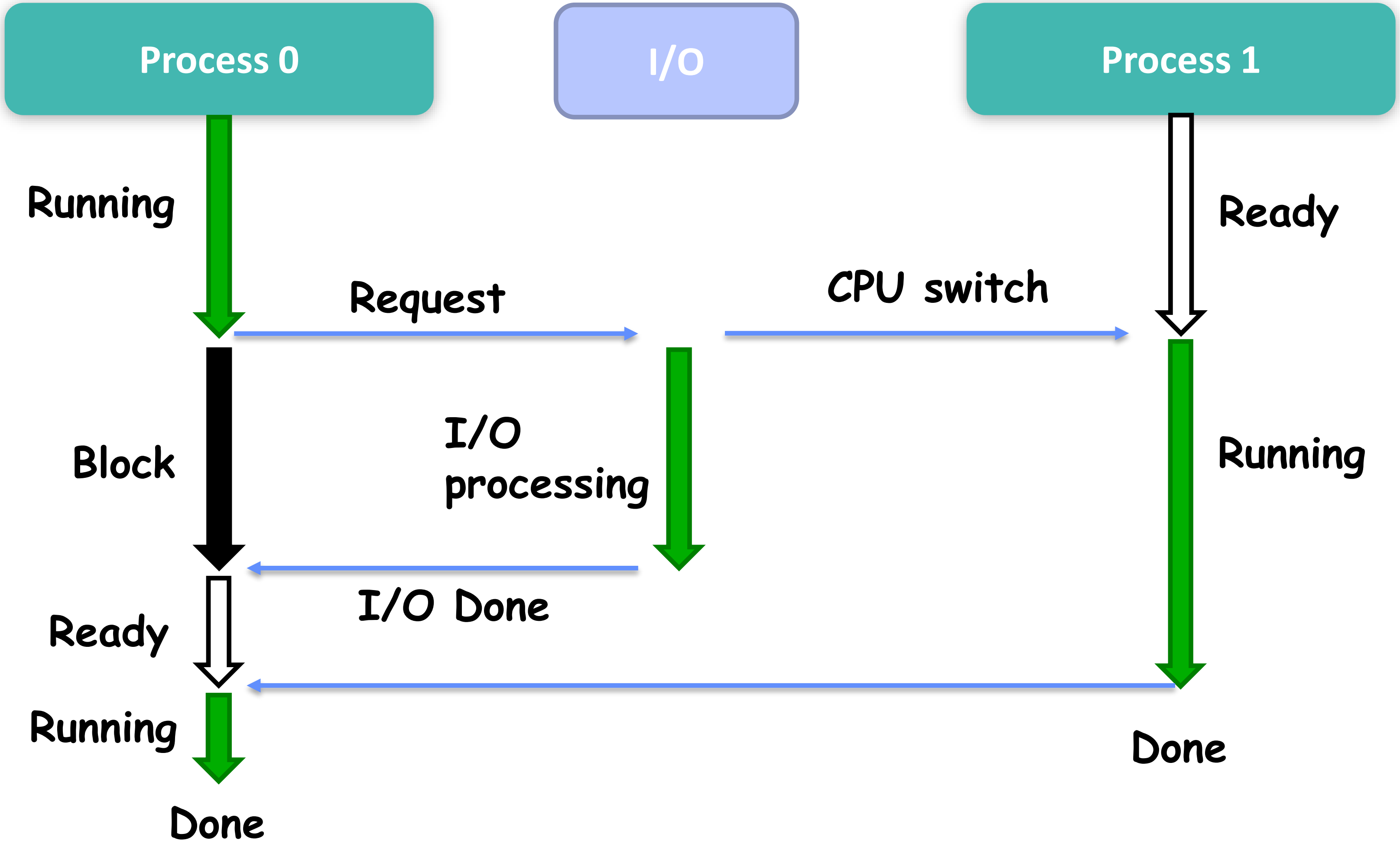
- A process is represented by a **process control block (PCB)**
  - Process ID (PID, unique)
  - State
  - Parent process
  - Opened files
  - etc.

Process State

- Process has different states
  - **READY**
    - » Ready to run and pending for running
  - **RUNNING**
    - » Being executed by OS
  - **BLOCKED**
    - » Suspended due to some other events, e.g., I/O requests



Process State



Process API

- Process API to manipulate processes
  - **CREATE**
    - » Create a new process, e.g., double click, a command in terminal
  - **WAIT**
    - » Wait for a process to stop
    - » Like I/O request
  - **DESTROY**
    - » Kill the processes
  - **STATUS**
    - » Obtain the information of a process
  - **OTHERS**
    - » Suspend or resume a process

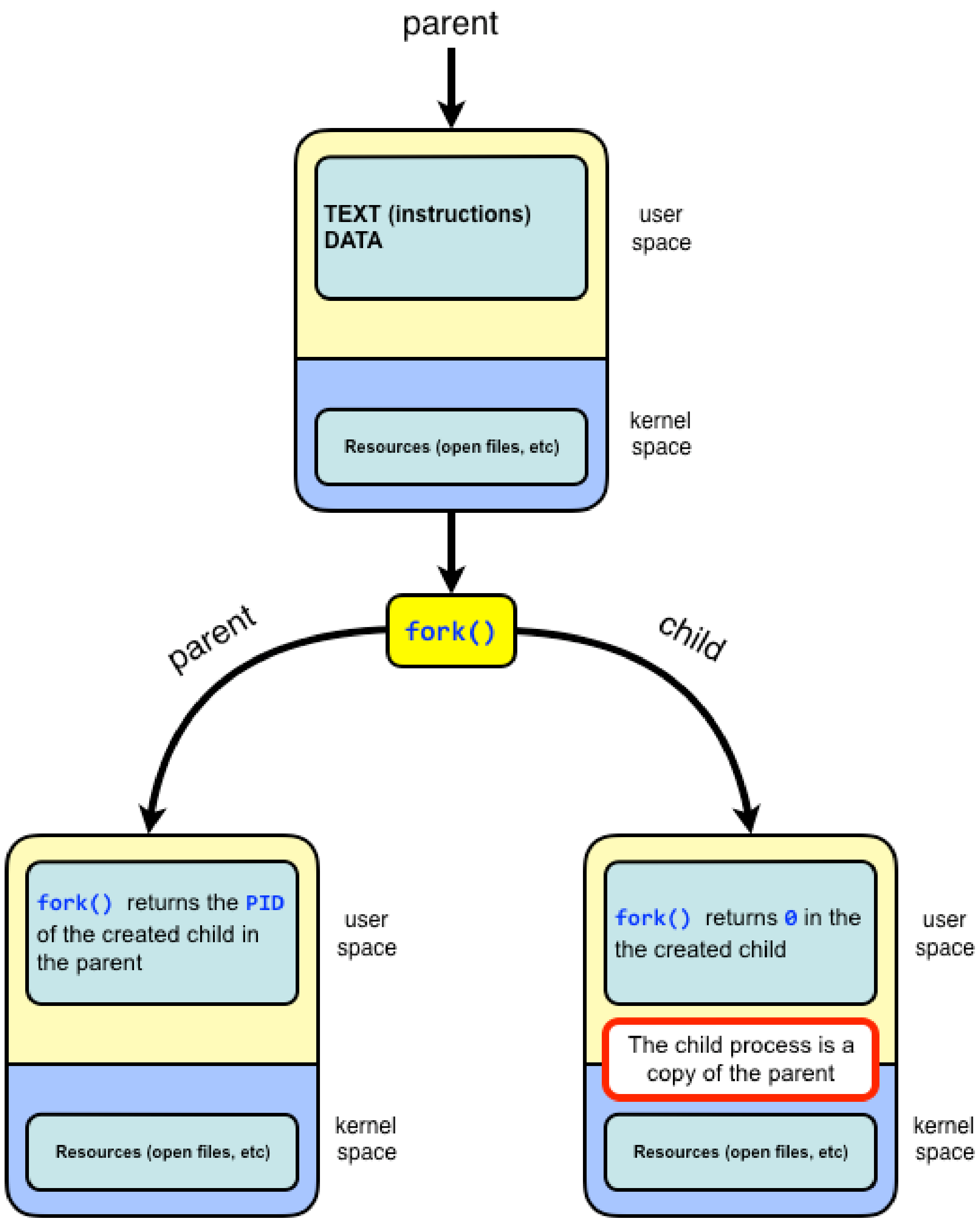
Process Creation

- A process is created by another process, **parent process** or **calling process**

- Process creation relies on two system calls
  - **fork()**
    - » Create a new process and clone its parent process
  - **exec()**
    - » Overwrite the created process with a new program

fork()

- A function without any arguments
  - **pid = fork()**
- Both **parent process** and **child process** continue to execute **the instruction following the fork()**
- The return value indicates which process it is (**parent** or **child**)
  - **Non-0 pid (**pid of child process**)** : return value of the parent process,
  - **0** : return value of the new child process
  - **-1** : an error or failure occurs when creating new process
- Child process is a duplicate of its parent process and has same
  - **instructions, data, stack**
- Child and parents have different
  - **PIDs, memory spaces**



fork()

```
int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
            // fork failed; exit
            fprintf(stderr, "fork failed\n"); exit(1);
    } else if (rc == 0) {
            // child (new process)
            printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
            // parent goes down this path (original process)
            printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

Output

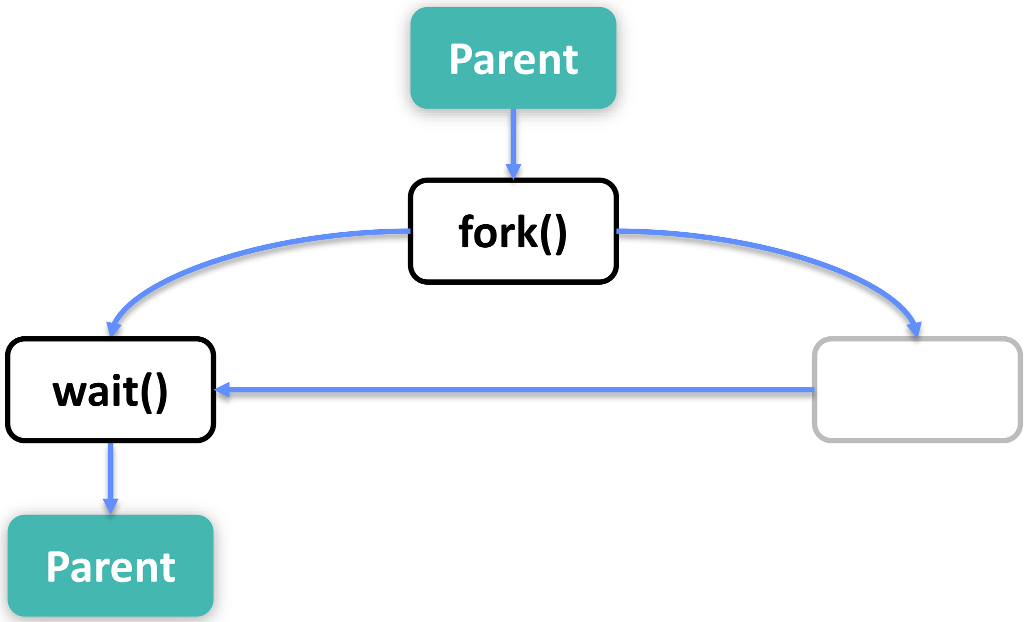
```
hello world (pid:96744)
hello, I am parent of 96745 (pid:96744)
hello, I am child (pid:96745)
```

Child Process

Parent Process

wait()

- Let the parent process wait for the completion of the child process
  - **pid = wait()**
- **waitpid()** is an alternative of wait()



wait()

```c
int main(int argc, char *argv[])
{
        printf("hello world (pid:%d)\n", (int) getpid());
        int rc = fork();
        if (rc < 0) {
                // fork failed; exit
                fprintf(stderr, "fork failed\n");
                exit(1);
        } else if (rc == 0) {
                // child (new process)
                printf("hello, I am child (pid:%d)\n", (int) getpid());
                sleep(1);
        } else {
                // parent goes down this path (original process)
                int wc = wait(NULL);
                printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc, (int) getpid());
        }
        return 0;
}
```

Child process sleeps for 1 second
Parent process waits for the child process to finish sleeping

Child Process

Parent Process

wait()

- **Without wait(): it is nondeterministic which process (parent or child) runs first**

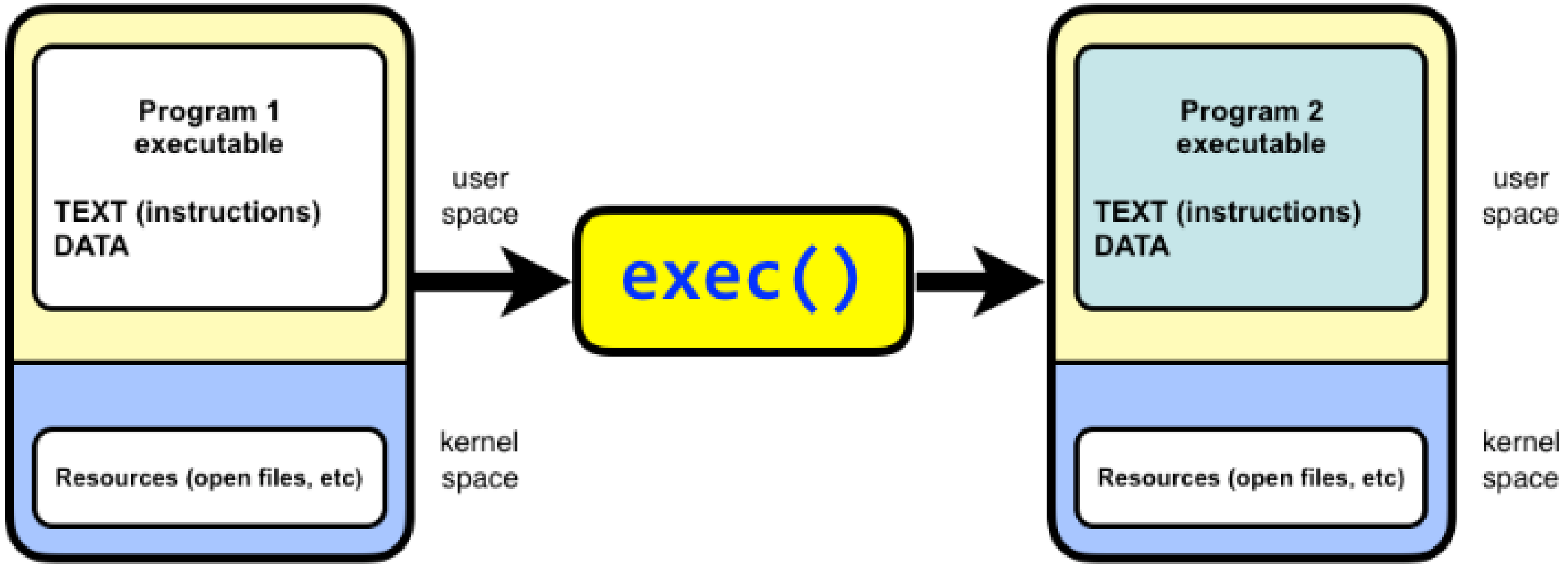
```
hello world (pid:96744)
hello, I am parent of 96745 (pid:96744)
hello, I am child (pid:96745)
```

- **With wait(): child runs first, and parents waits for child to finish**

```
hello world (pid:96848)
hello, I am child (pid:96849)
hello, I am parent of 96849 (wc:96849) (pid:96848)
```

exec()

- **exec(cmd, argv)** replaces the current process image with a new process image specified by the path to an executable file.
  - **exec()** does not return. It starts to execute the new program.
- There is a family of **exec(),** e.g., **execl(), execvp()**
  - **execl()** takes a variable number of arguments that represent the program name and its arguments.
  - **execvp()** takes an array of arguments instead of a variable-length argument list



exec() Example

```c
int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
            // fork failed; exit
            fprintf(stderr, "fork failed\n"); exit(1);
    } else if (rc == 0) {  // child (new process)
            printf("hello, I am child (pid:%d)\n", (int) getpid());
            char *myargs[3];
            myargs[0] = strdup("wc");  // program: "wc" (word count)
            myargs[1] = strdup("p3.c");  // argument: file to count
            myargs[2] = NULL; // marks end of array
            execvp(myargs[0], myargs); // run word count
            printf("this will be replaced, so not printed out");
    } else {  // parent
            int rc_wait = wait(NULL);
            printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n", rc, rc_wait, (int) getpid());}
    return 0;
}
```

Output

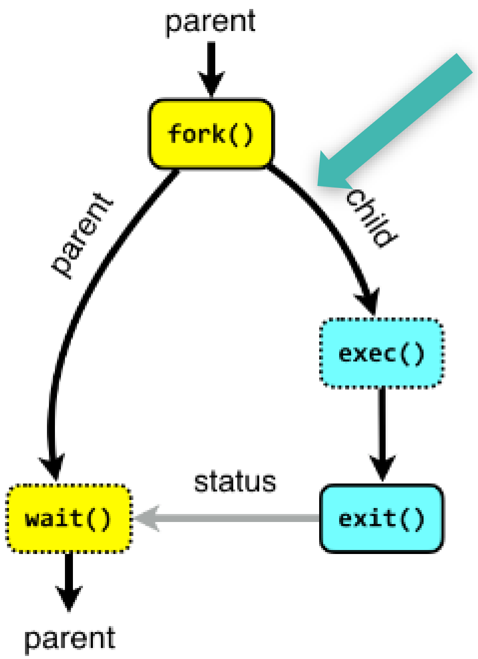
```
hello world (pid:97511)
hello, I am child (pid:97512)
      32      123      966 p3.c
hello, I am parent of 97512 (wc:97512) (pid:97511)
```

Why fork() + exec()

- Why don’t we directly create a new process instead of “**fork**+**exec**”?
  - **Simple** and **powerful**
  - **Important** to build Unix Shell (an interface to the Unix system)
- By separating **fork()** and **exec()**, we can manipulate various settings just before executing a new program and **make the IO redirection and pipe possible**.
  - IO redirection
  - pipe

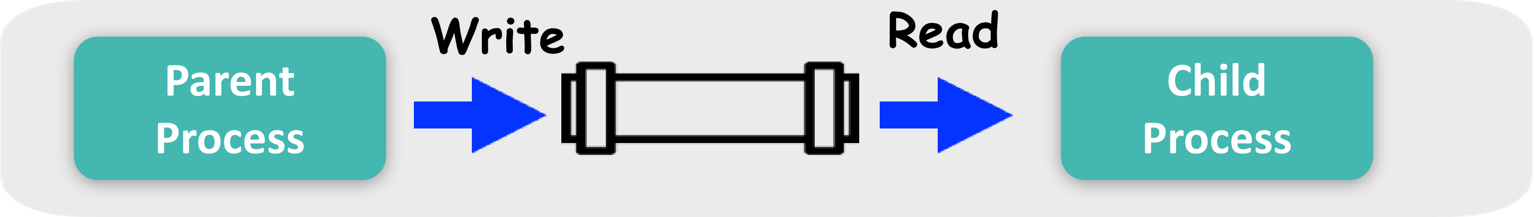
```
% cat w3.c > newfile.txt
```

```
% echo hello world | wc
```



pipe

- A communication method between two processes

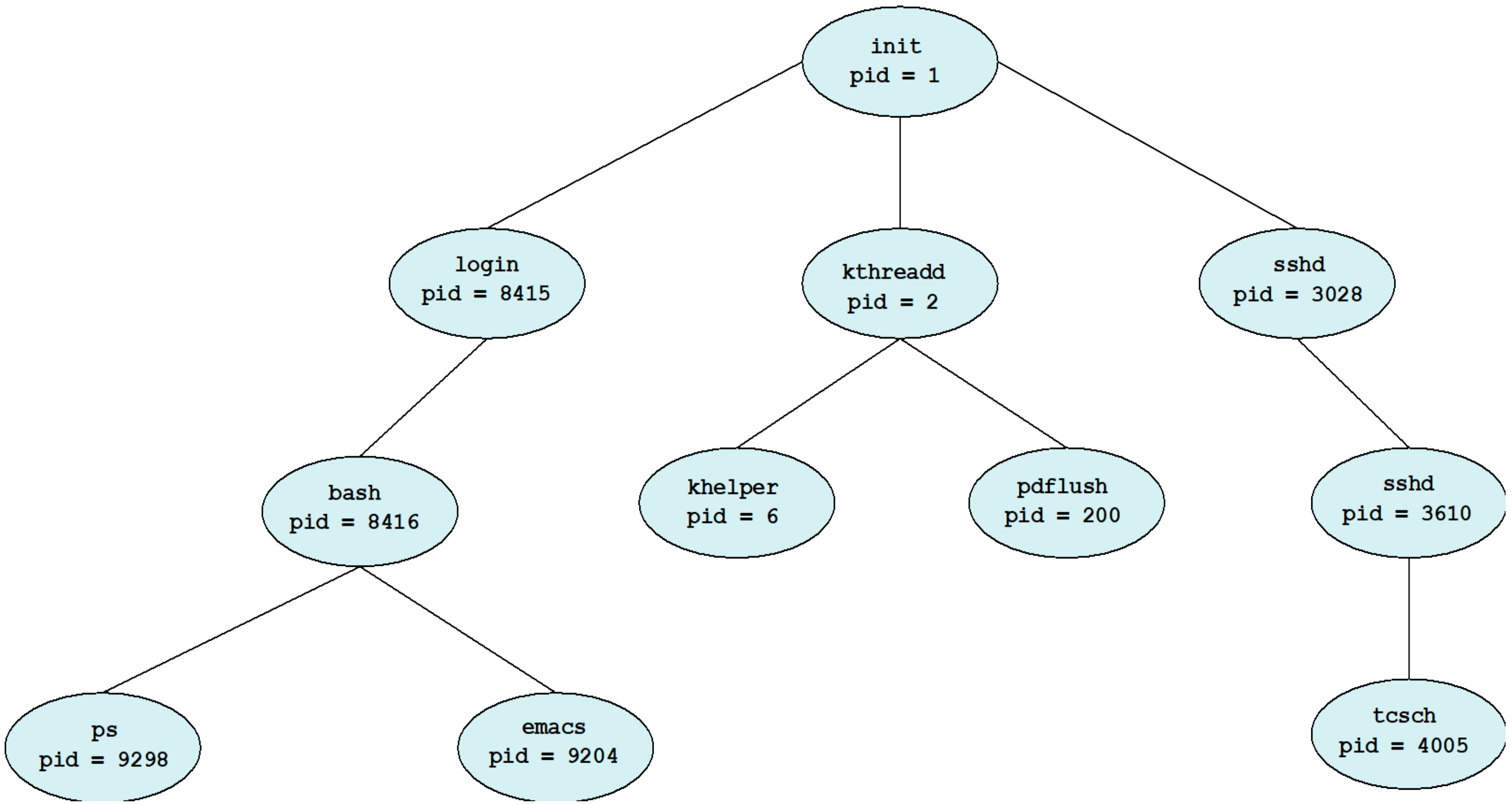



```
(base) dliu@dhcp-10-24-18-121 my_code % cat hello.c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

```
(base) dliu@dhcp-10-24-18-121 my_code % cat hello.c |grep printf
    printf("Hello World!\n");
(base) dliu@dhcp-10-24-18-121 my_code %
```

Process Tree



Process Tree

- % pstree (to show the process tree)

```
(base) dliu@dhcp-10-24-17-236 ~ % pstree
-+= 00001 root /sbin/launchd
 |--= 00322 root /usr/libexec/logd
 |--= 00323 root /usr/libexec/smd
 |--= 00324 root /usr/libexec/UserEventAgent (System)
```

- % ps (to show all processes)

```
PID   TT  STAT      TIME COMMAND
  1   ??  Ss    17:57.36 /sbin/launchd
322   ??  Rs     6:29.86 /usr/libexec/logd
323   ??  Ss     0:00.19 /usr/libexec/smd
324   ??  Ss     0:19.58 /usr/libexec/UserEventAgent (System)
```

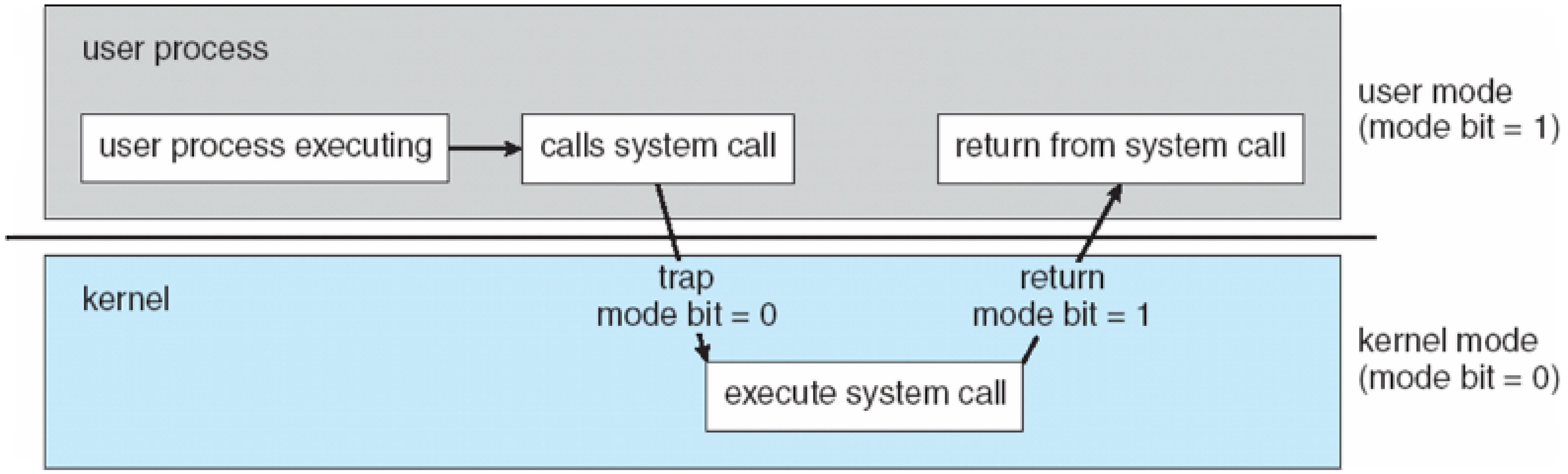
User/Kernel Mode Separation

- **User mode**: restricted, limited operations
  - Processes start in user mode
- **Kernel mode**: privileged, not restricted
  - OS starts in kernel mode

- What if a process wants to perform some restricted operations?
  - **System calls:** Allow the kernel services to provide some functionalities to user programs

User/Kernel Mode Separation

- A process starts in **user mode**
- If it needs to perform a restricted operation, it calls a system call by executing a **trap instruction**.
- The state and registers of the calling process are stored, the system enters **kernel mode**, OS completes the syscall work.
- **Return from syscall,** restore the states and registers of the process, and resume the execution of the process



Process Scheduling

- **Switching Between Processes**
  - **Cooperative approach**
  - **Non-cooperative approach**

- **Cooperative approach**
  - **Trust process to relinquish CPU to OS through traps**
    - » **System calls**
    - » **Illegal operations, e.g., divided by zero**
  - **Issue: if no system call**

- **Non-cooperative approach**
  - **The OS takes control**
  - **OS obtains control periodically, e.g., timer interrupter**

Summary

- In OS, process is a running program and has an address space
- We use process API to create and manage processes
- Fork() to duplicate a process, exec() to replace the command
- Process scheduling