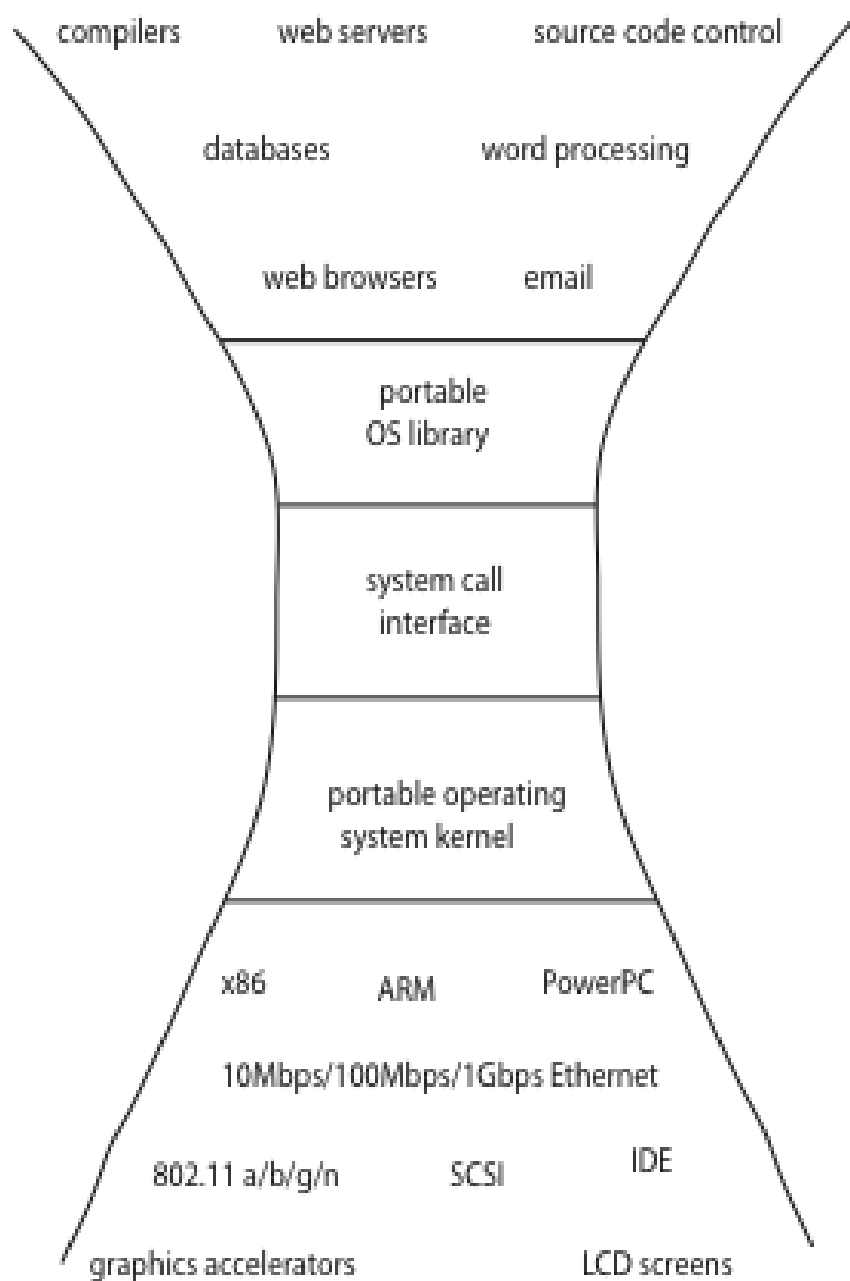
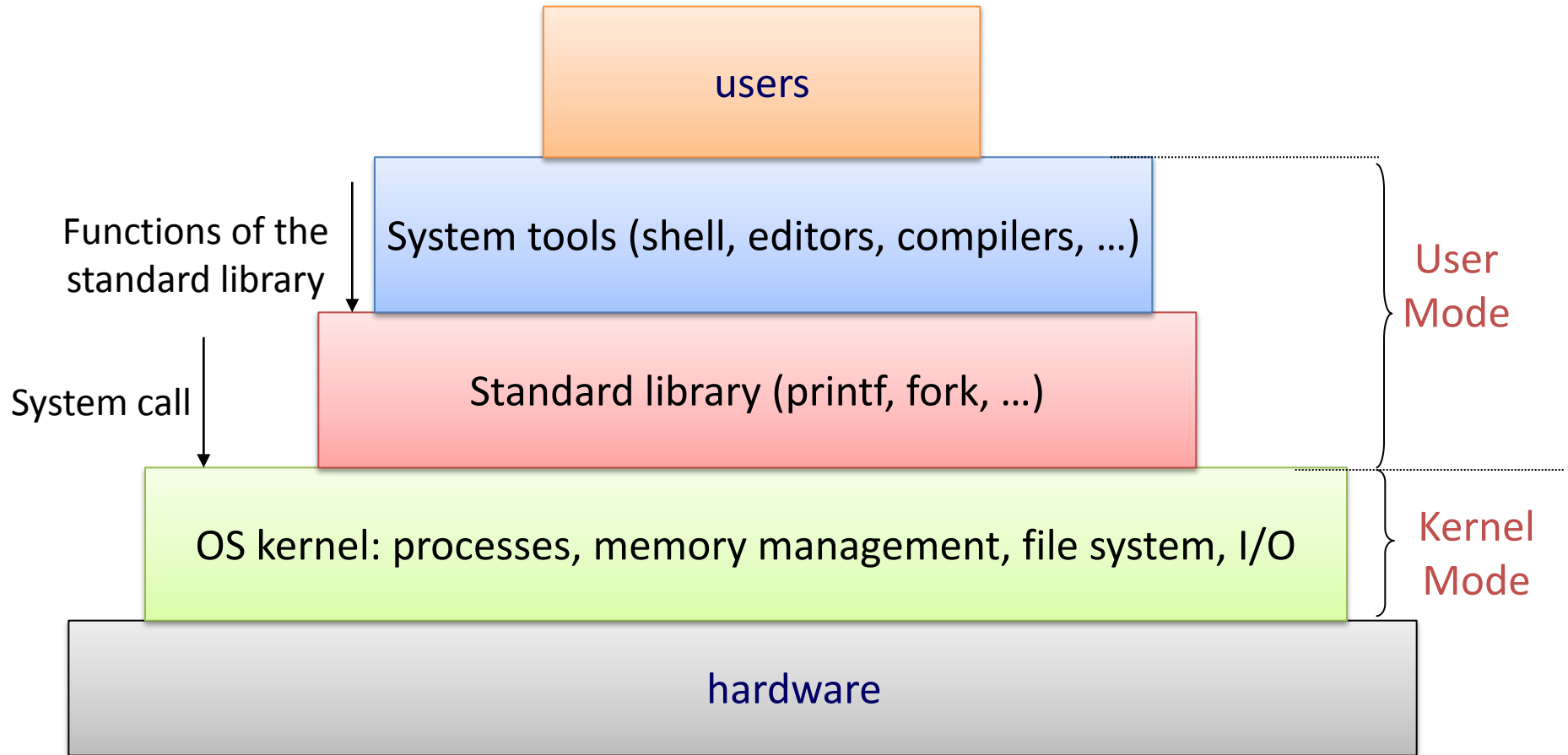


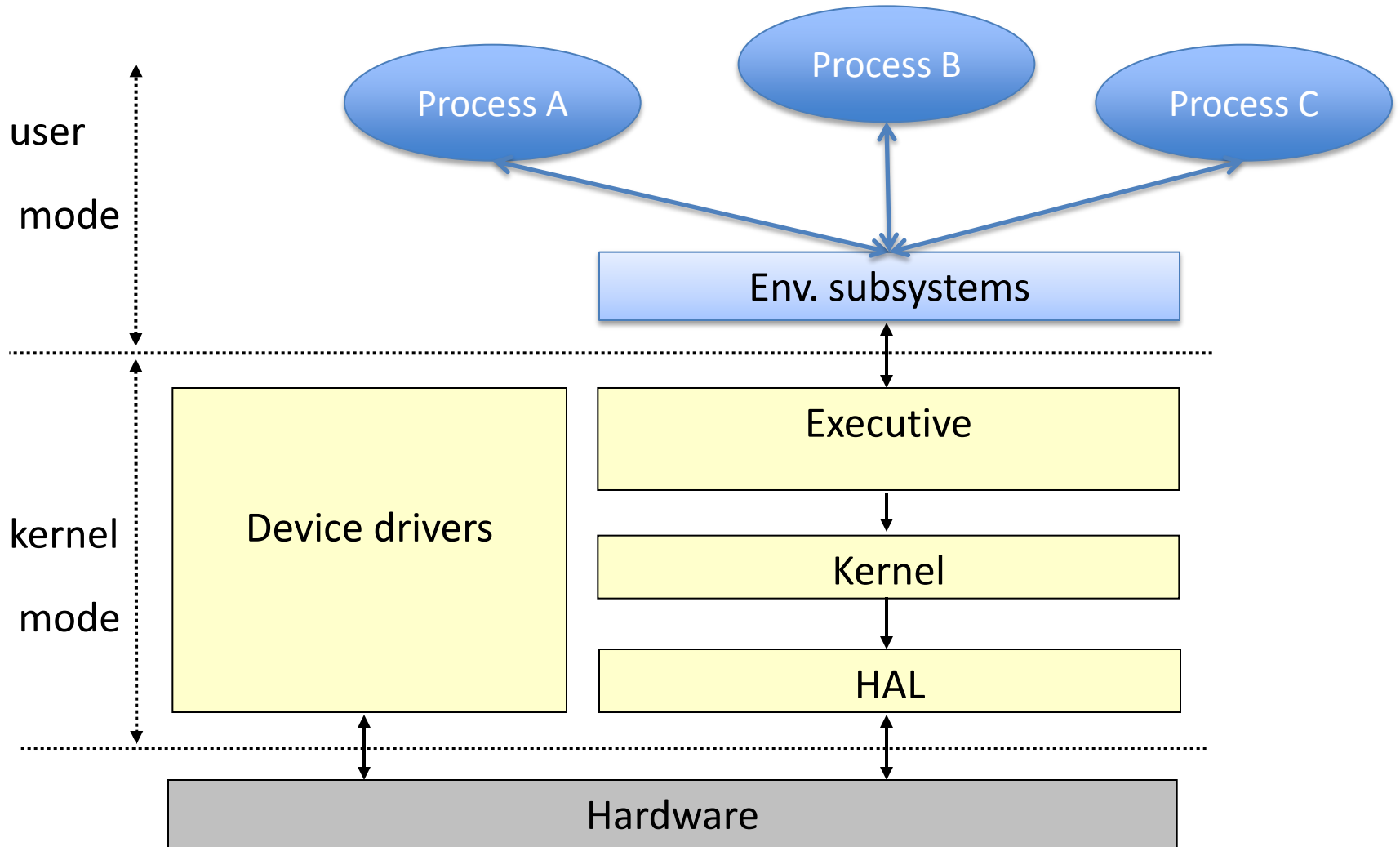
Structure



Unix architecture



Windows architecture



Programming Interface

Main Points

- Creating and managing processes
 - fork, exec, wait
- Performing I/O
 - open, read, write, close
- Communicating between processes
 - pipe, dup, select, connect
- Example: implementing a shell

Shell

- A shell is a job control system
 - Allows programmer to create and manage a set of programs to do some task
 - Windows, MacOS, Linux all have shells
- Example: to compile a C program

```
cc -c sourcefile1.c
cc -c sourcefile2.c
ln -o program sourcefile1.o sourcefile2.o
```

Question

- If the shell runs at user-level, what system calls does it make to run each of the programs?
 - Ex: `cc`, `ln`

Windows CreateProcess

- System call to create a new process to run a program
 - Create and initialize the process control block (PCB) in the kernel
 - Create and initialize a new address space
 - Load the program into the address space
 - Copy arguments into memory in the address space
 - Initialize the hardware context to start execution at ``start''
 - Inform the scheduler that the new process is ready to run

Windows CreateProcess API (simplified)

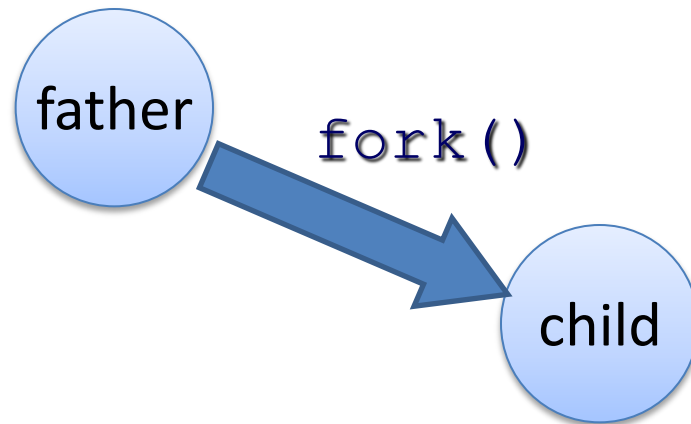
```
if (!CreateProcess(  
    NULL,          // No module name (use command line)  
    argv[1],       // Command line  
    NULL,          // Process handle not inheritable  
    NULL,          // Thread handle not inheritable  
    FALSE,         // Set handle inheritance to FALSE  
    0,             // No creation flags  
    NULL,          // Use parent's environment block  
    NULL,          // Use parent's starting directory  
    &si,            // Pointer to STARTUPINFO structure  
    &pi )           // Pointer to PROCESS_INFORMATION structure  
)
```

UNIX Process Management

- UNIX fork – system call to create a copy of the current process, and start it running
 - No arguments!
- UNIX exec – system call to change the program being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send a notification to another process

UNIX fork()

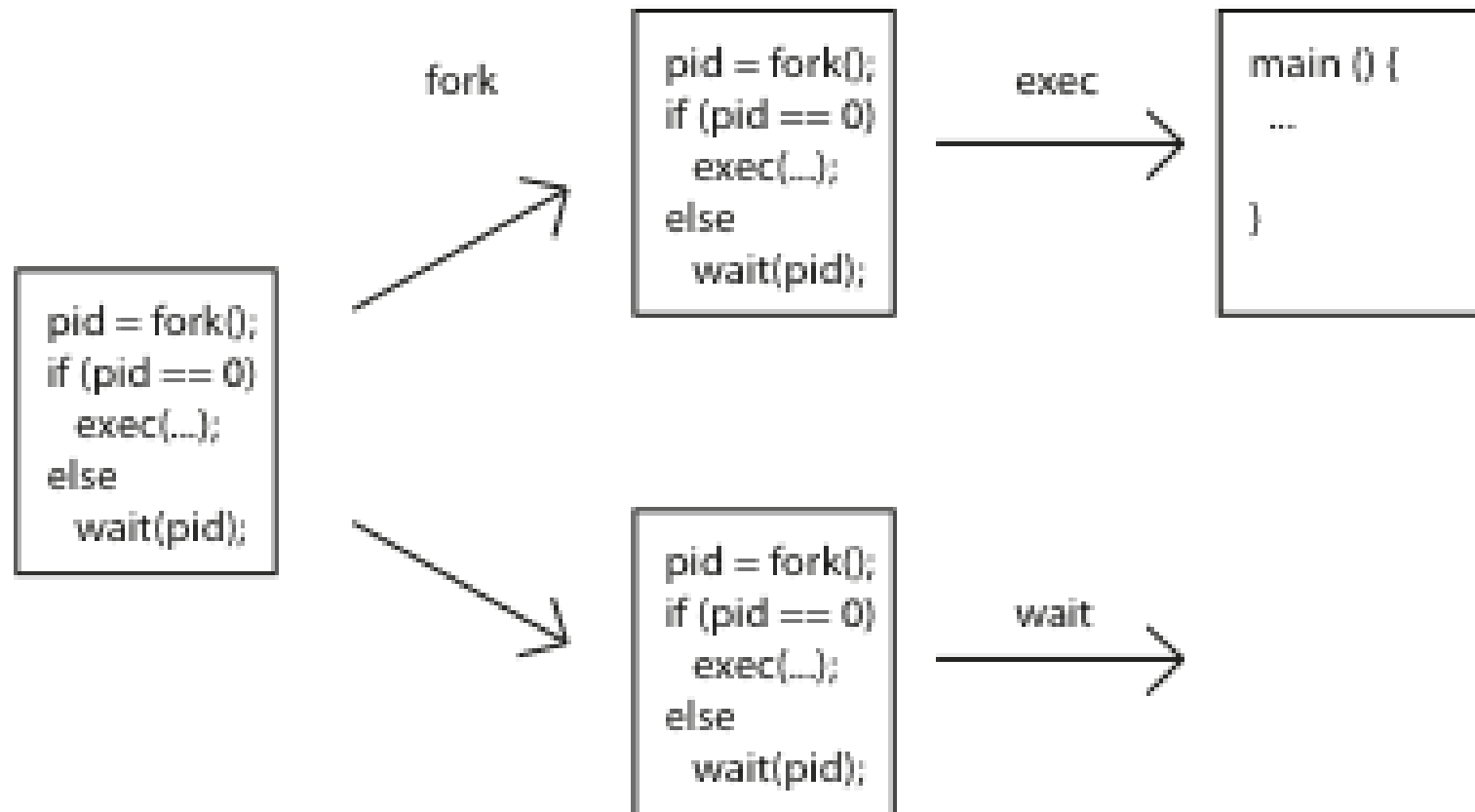
- `fork()` is used to generate a child process:
 - The father and its child share the same code
 - The child process inherits a copy of the kernel and user data of the father



UNIX fork()

- `fork()` does not take input params
- Returns an integer:
 - For the child it is 0
 - For the father is:
 - A positive value that represents the PID of the child
 - A negative value that represents an error code

UNIX Process Management



Question: What does this code print?

```
int child_pid = fork();  
if (child_pid == 0) {           // I'm the child process  
    printf("I am process # %d\n", getpid());  
    return 0;  
} else {                       // I'm the parent process  
    printf("I am parent of process # %d\n", child_pid);  
    return 0;  
}
```

Questions

- Can UNIX `fork()` return an error? Why?
- Can UNIX `exec()` return an error? Why?
- Can UNIX `wait()` ever return immediately? Why?

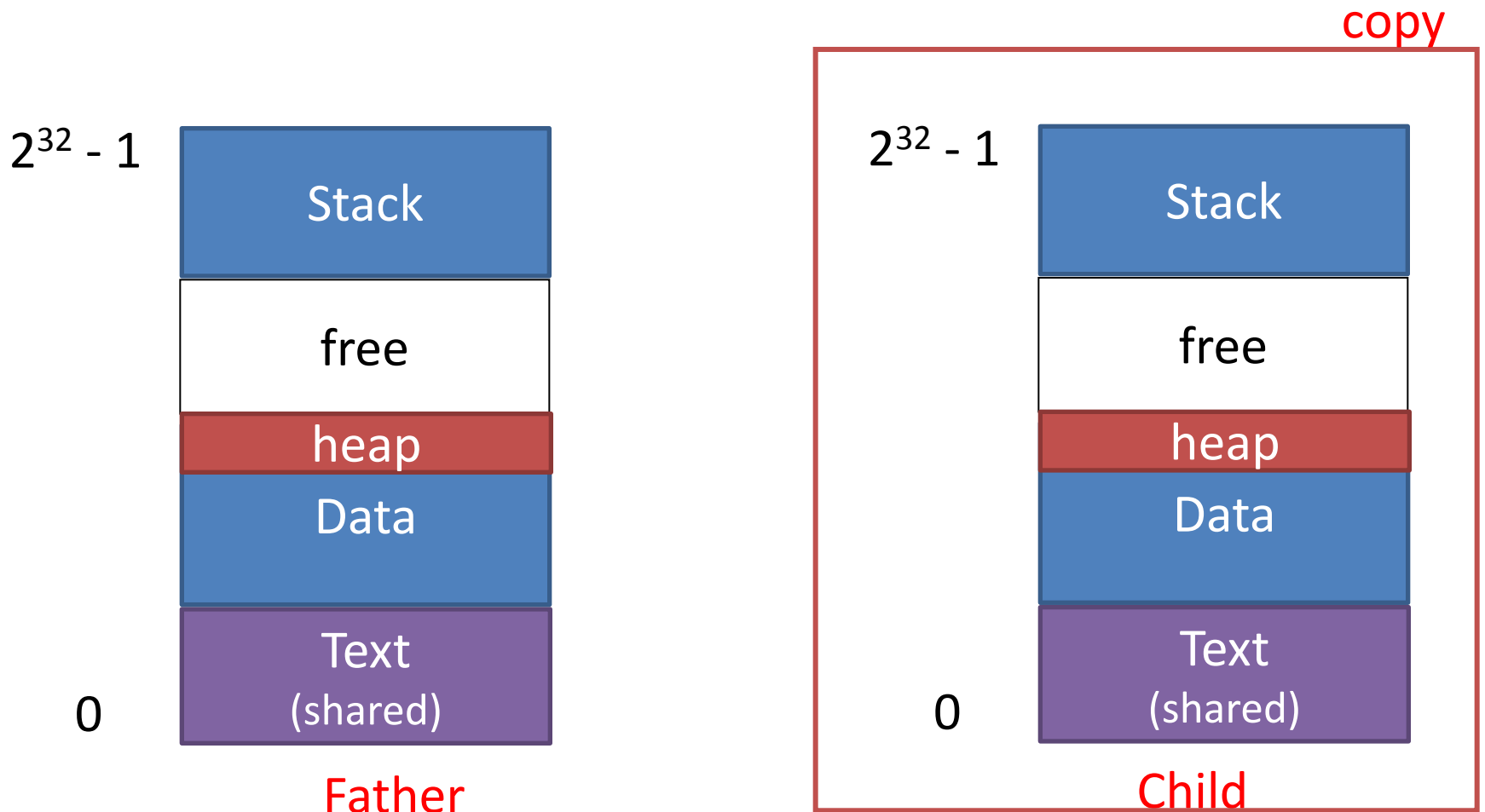
Implementing UNIX fork

Steps to implement UNIX fork

- Create and initialize the process control block (PCB) in the kernel
 - Namely, a **process structure** and a **user structure**
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

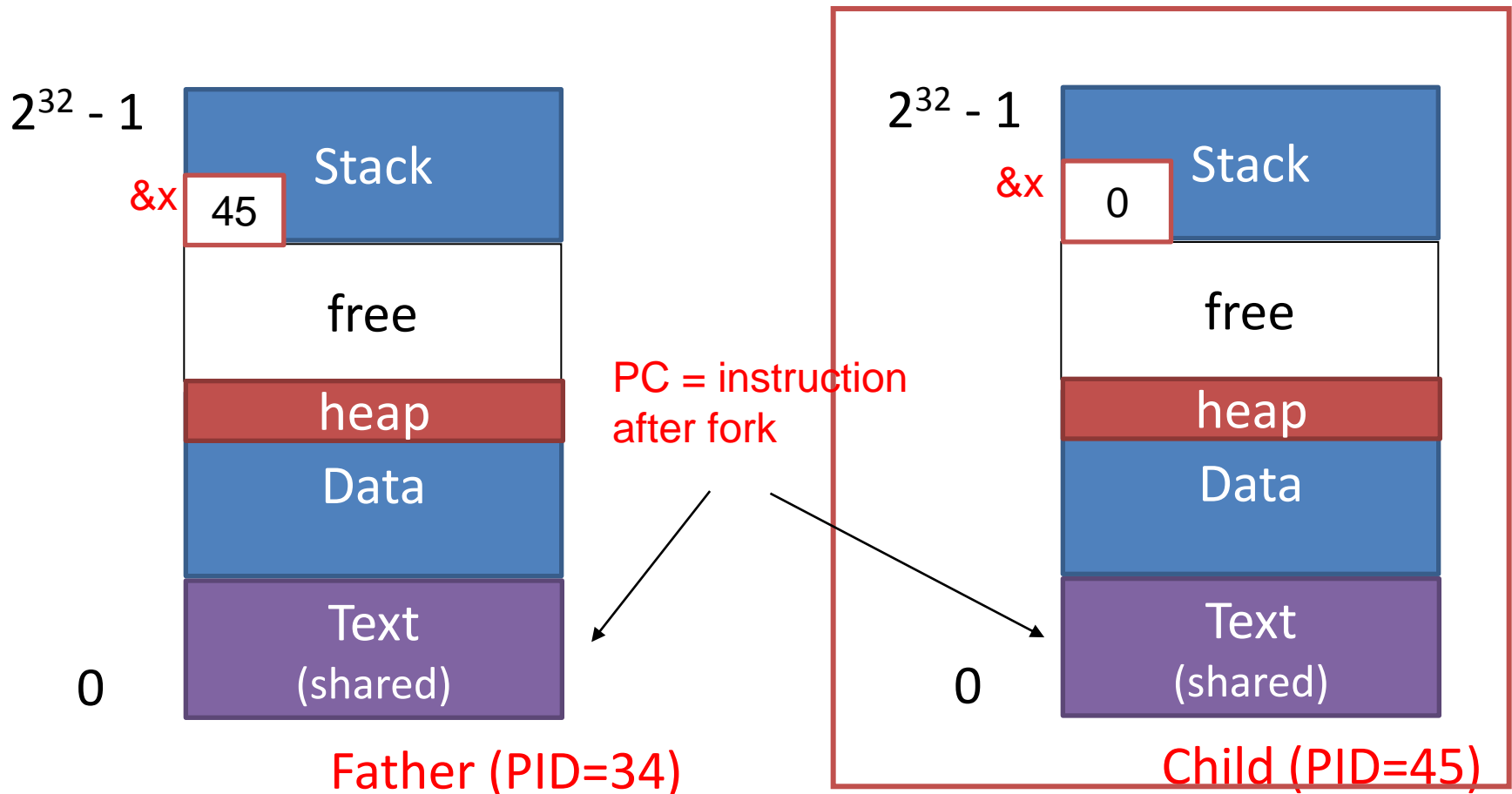
Implementing UNIX fork

Addressing spaces of the father and the child after a successful fork



Implementing UNIX fork

Addressing spaces of the father and the child after a successful fork



Implementing UNIX fork

- Steps to implement UNIX fork
 - Load the program into the current address space
 - Copy arguments into memory in the address space
 - Initialize the hardware context to start execution at ``start''

UNIX exec

- Replaces the code executed by a process
 - Does not create a new process, inherits PBC and changes address space
- Replaces the data
- Example:
 - `int execl(char *pathname, char *arg0, .. char *argN, (char*) 0)`
 - Pathname is the name of an executable file
 - `argN[1],..argN[...]` are the arguments passed to the program
 - list terminated with `(char *) 0`

UNIX exec

- If it's successful it does not return
 - The process executes another program
- If it fails it returns an error code
- After exec the process:
 - Keeps the PID
 - Keeps the PCB (process and user structures)
 - But it changes references to code and data memory
 - Resets the pending signals
 - Keeps the kernel stack
 - Keeps the assigned resources (open files)

Process termination in UNIX

- A process can terminate:
 - Because of an exception due to illegal actions
 - By invoking the system call `exit`
- The terminated process returns an exit value to its father
 - The father receives the value by the system call `wait`
 - If the father didn't already call the `wait`, the terminated process switches to zombie state
 - If the father is already terminated, the `init` process adopts its children

`exit()` and `wait()`

- `void exit(int status);`
 - Status is the termination code
 - `exit` never returns
 - Frees memory, releases resources
 - If it switches to zombie, keeps the PBC until the father invokes `wait`
- `int wait(int *status);`
 - Status is the PID of the terminated process or an error code

UNIX I/O

- Uniformity
 - All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use
 - Open returns a handle (file descriptor) for use in later calls on the file
- Byte-oriented
- Kernel-buffered read/write
- Explicit close
 - To garbage collect the open file descriptor

UNIX File System Interface

- UNIX file open is a Swiss Army knife:
 - Open the file, return file descriptor
 - Options:
 - if file doesn't exist, return an error
 - If file doesn't exist, create file and open it
 - If file does exist, return an error
 - If file does exist, open file
 - If file exists but isn't empty, nix it then open
 - If file exists but isn't empty, return an error
 - ...

Interface Design Question

- Why not separate syscalls for open/create/exists?

```
if (!exists(name))
```

```
    create(name); // can create fail?
```

```
fd = open(name); // does the file exist?
```

Implementing a Shell

```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
    child_pid = fork();    // create a child process
    if (child_pid == 0) {
        exec(prog, args);    // I'm the child process. Run program
        // NOT REACHED
    } else {
        wait(child_pid);    // I'm the parent, wait for child
        return 0;
    }
}
```