

# Programming Tools II (System Calls)

---

## **Creating and Executing Processes**

Motilal Nehru National Institute of Technology Allahabad,  
Prayagraj

# Unix system calls

---

fork( )

wait( )

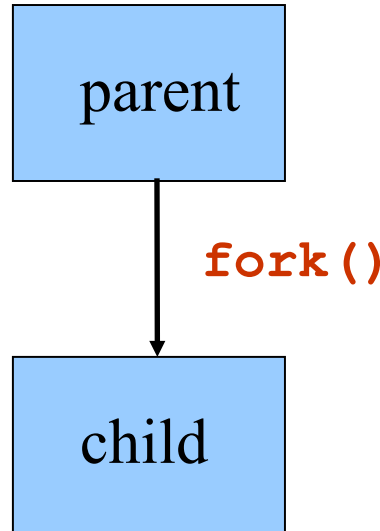
exit( )

# How To Create New Processes?

---

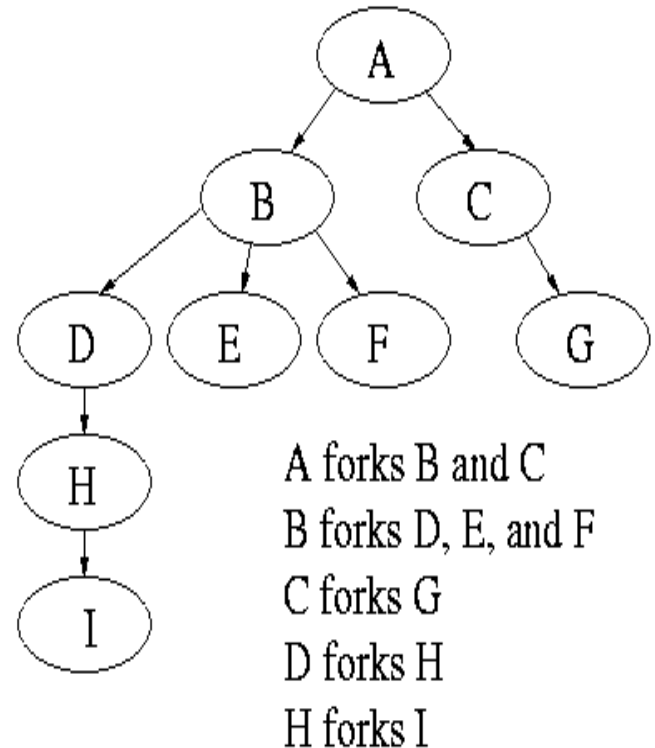
## ■ Underlying mechanism

- A process runs **fork** to create a child process
- Parent and children execute concurrently
- Child process is a duplicate of the parent process



# Process Creation

- After a **fork**, both parent and child keep running, and each can fork off other processes.
- A **process tree** results. The root of the tree is a special process created by the OS during startup.
- A process can *choose* to wait for children to terminate. For example, if C issued a **wait()** system call, it would block until G finished.



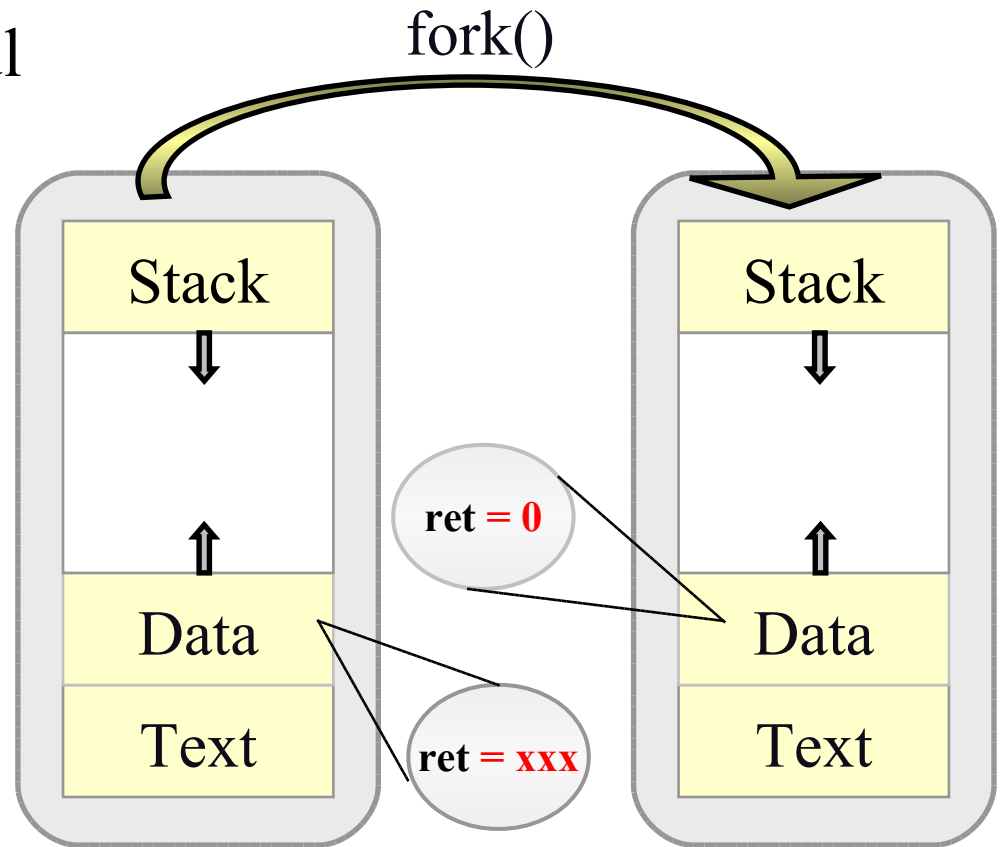
# Bootstrapping

---

- When a computer is switched on or reset, there must be an initial program that gets the system running
- This is the bootstrap program
  - Initialize CPU registers, device controllers, memory
  - Load the OS into memory
  - Start the OS running
- OS starts the first process (such as “init”)
- OS waits for some event to occur
  - Hardware interrupts or software interrupts (traps)

# Fork System Call

- Current process split into 2 processes: parent, child
- Returns -1 if unsuccessful
- Returns 0 in the child
- Returns the child's identifier in the parent

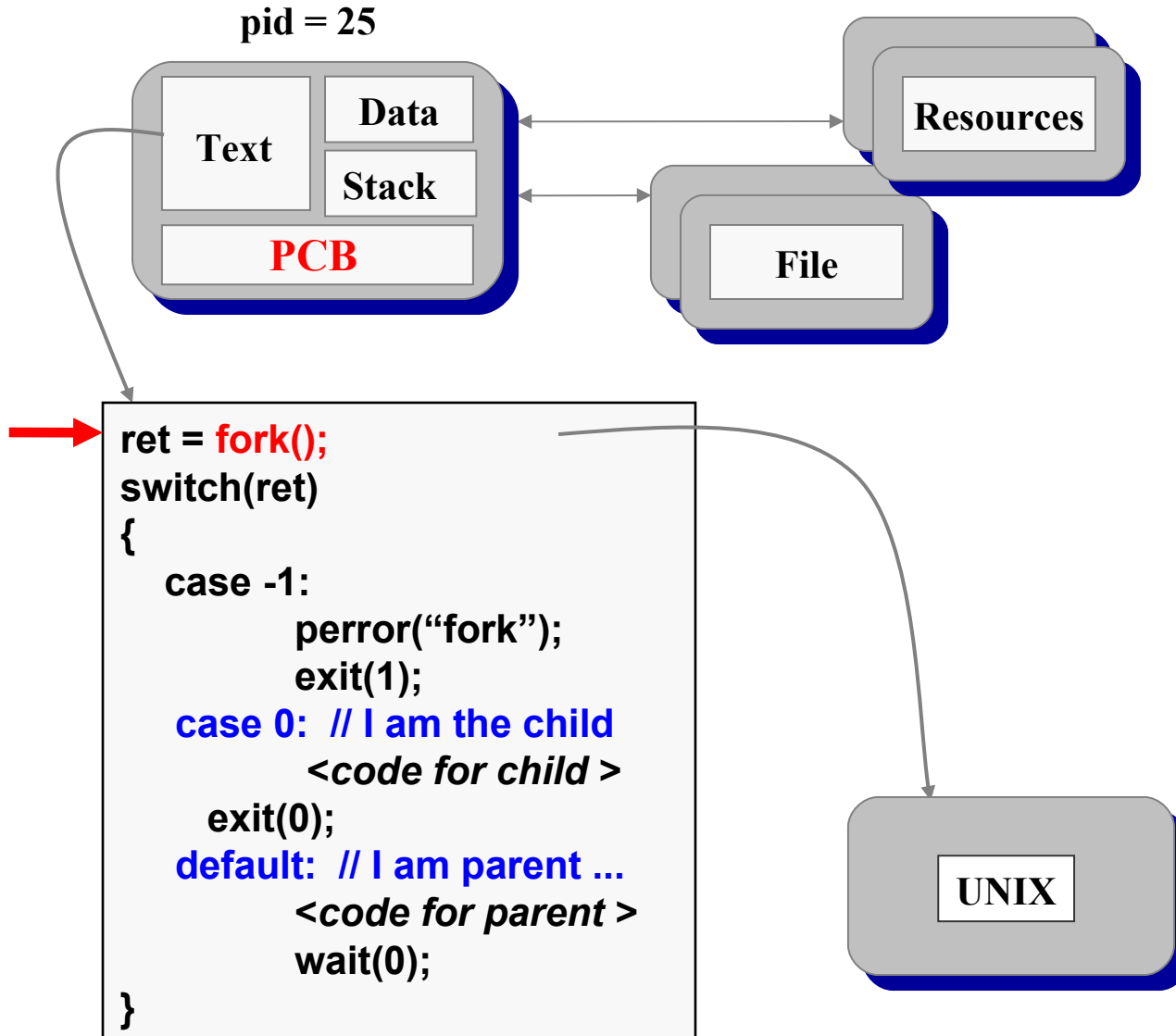


# Fork System Call

---

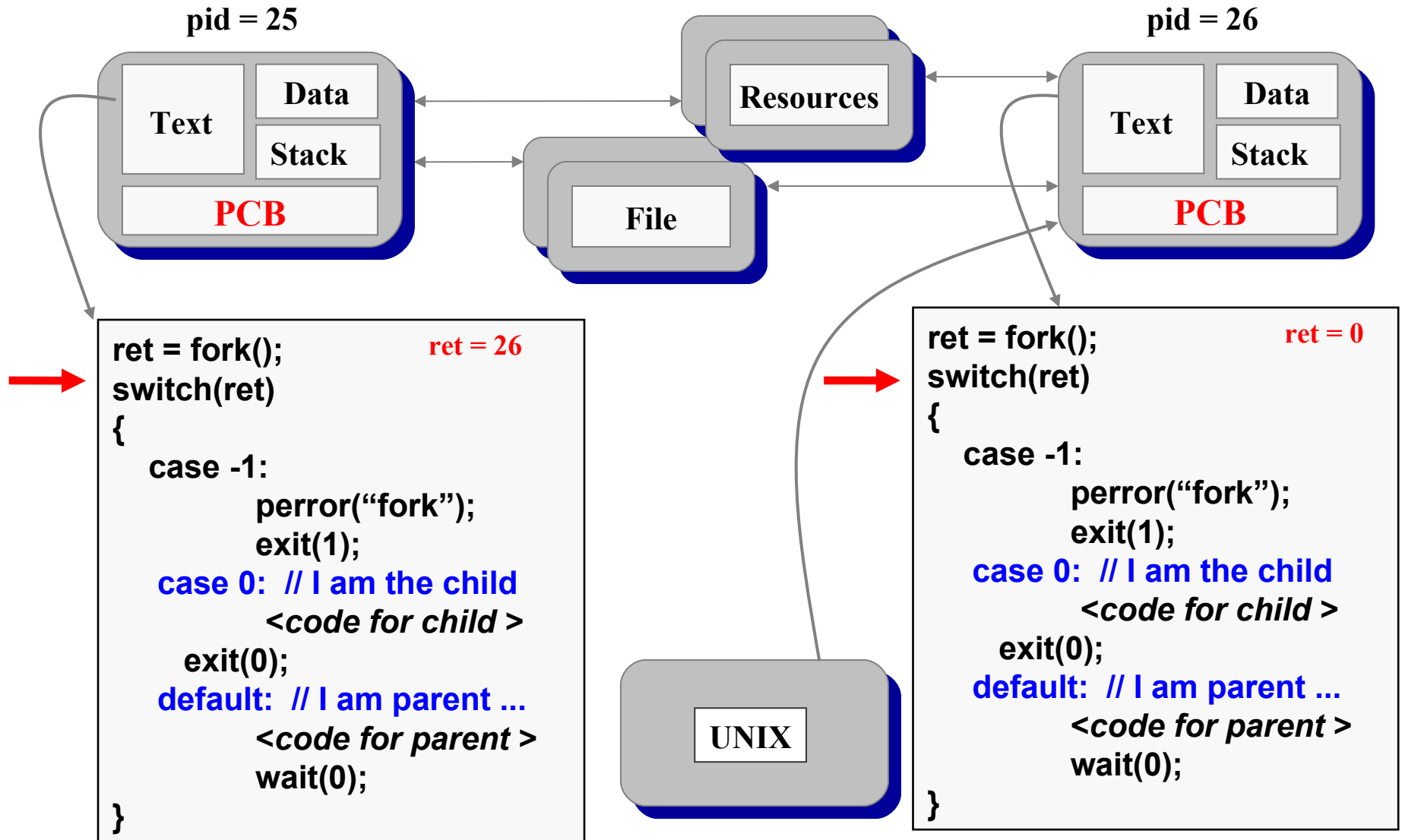
- The child process inherits from parent
  - identical copy of memory
  - CPU registers
  - all files that have been opened by the parent
- Execution proceeds **concurrently** with the instruction following the fork system call
- The execution context (PCB) for the child process is a copy of the parent's context at the time of the call

# How fork Works (1)

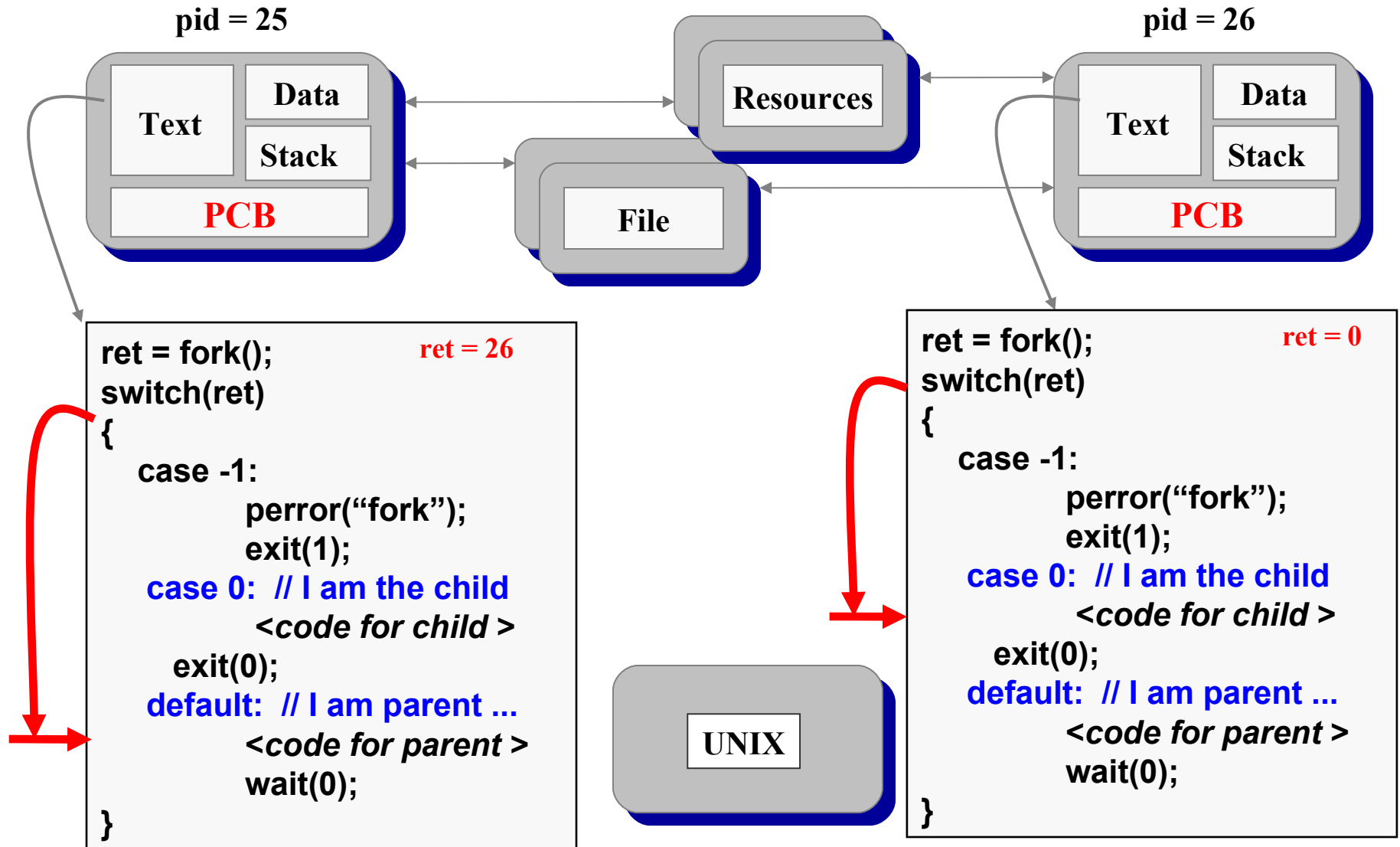




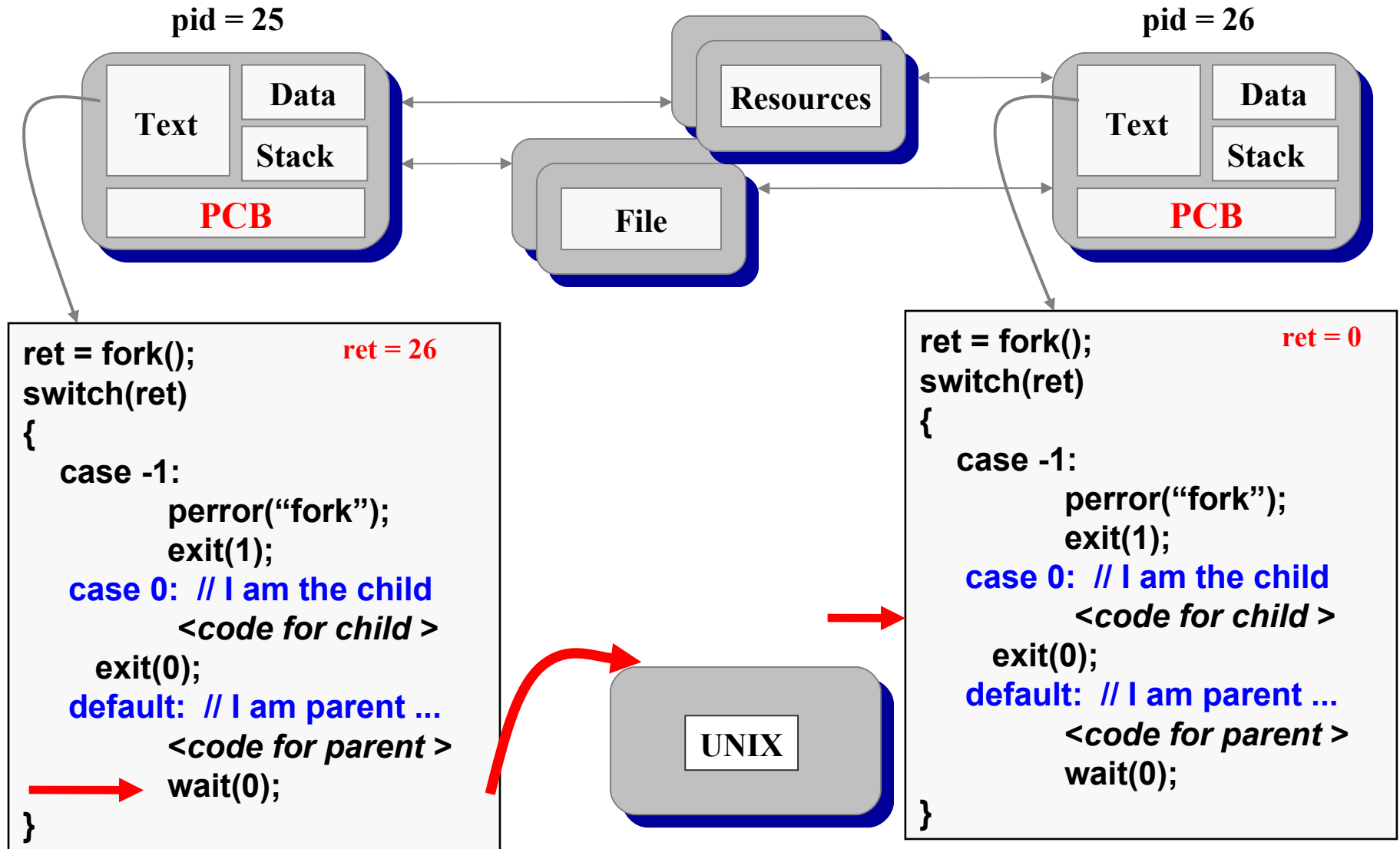
# How fork Works (2)



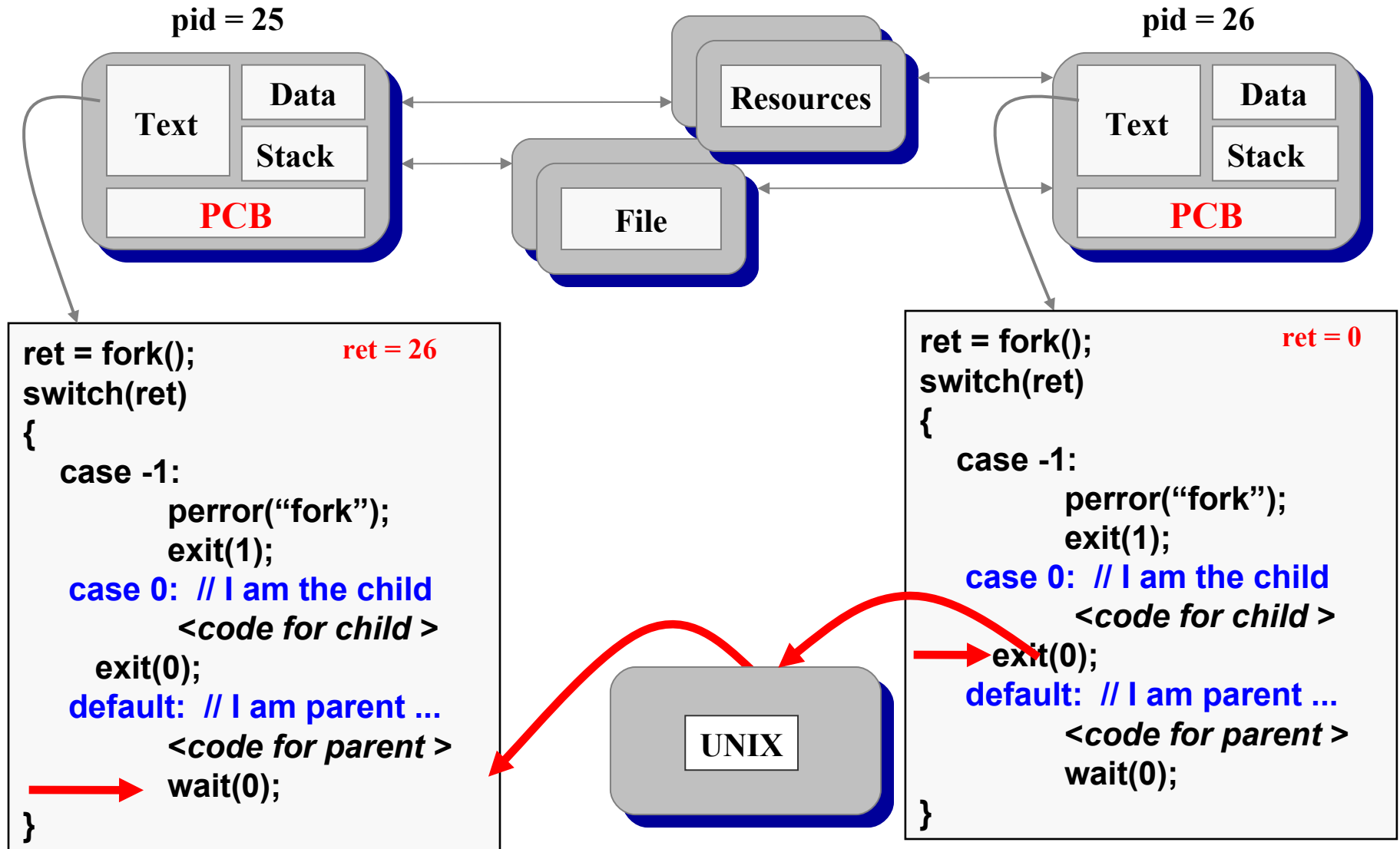
# How fork Works (3)



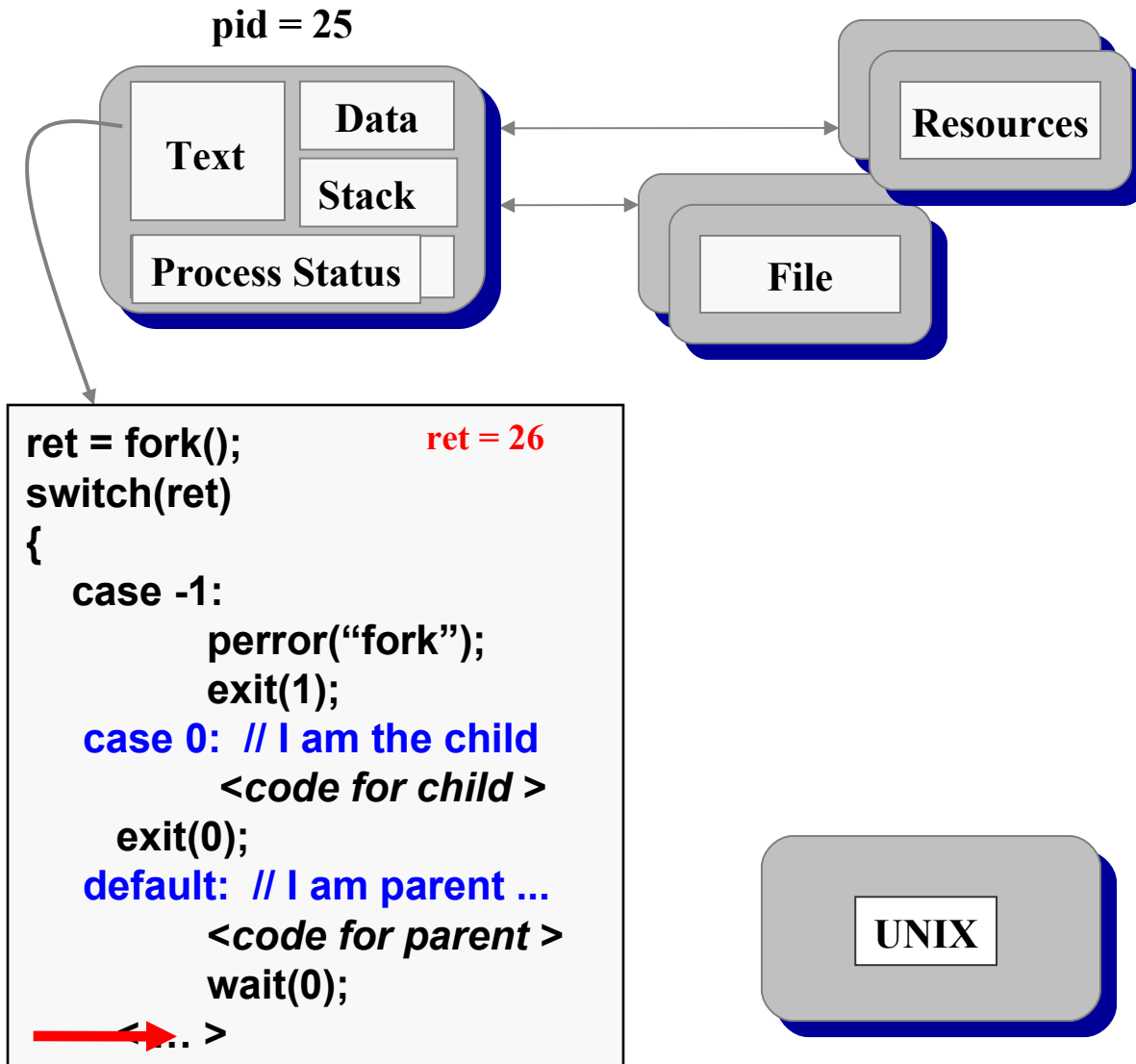
# How fork Works (4)



# How fork Works (5)



# How fork Works (6)



# Orderly Termination: `exit()`

---

- To finish execution, a child may call `exit(number)`
- This system call:
  - Saves result = argument of `exit`
  - Closes all open files, connections
  - Deallocates memory
  - Checks if parent is alive
  - If parent is alive, holds the result value until the parent requests it (with `wait`); in this case, the child process does not really die, but it enters a zombie/defunct state
  - If parent is not alive, the child terminates (dies)

# Waiting for the Child to Finish

---

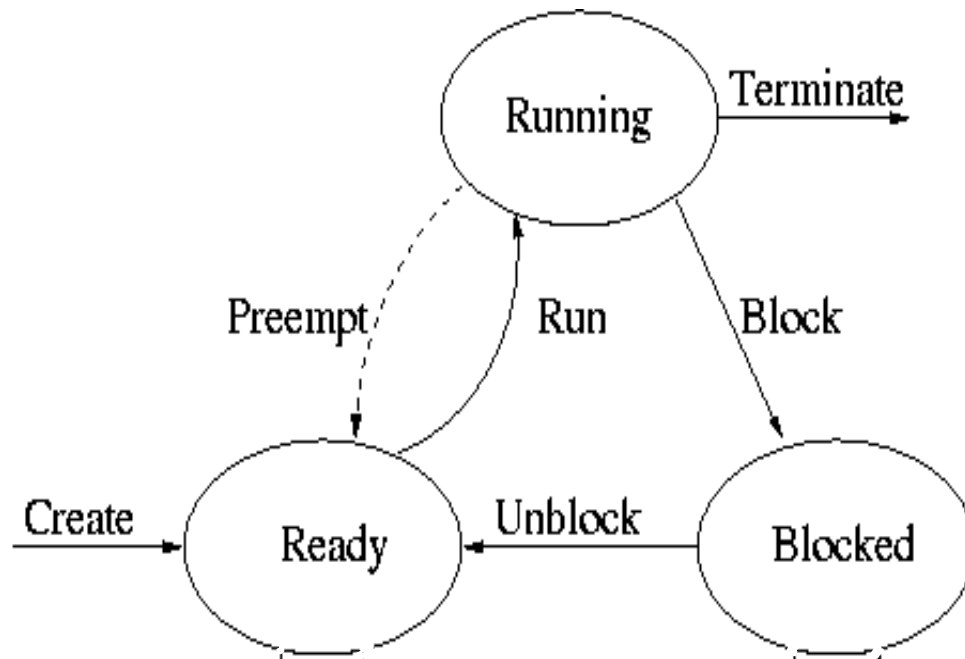
- Parent may want to wait for children to finish
  - Example: a shell waiting for operations to complete
- Waiting for any some child to terminate: `wait()`
  - Blocks until some child terminates
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- Waiting for a specific child to terminate: `waitpid()`
  - Blocks till a child with particular process ID terminates

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

# State Transition on wait and exit Calls

---





# Other useful system calls: `getpid`, `getppid`

---

- `getpid` returns the identifier of the calling process. Example call (`pid` is an integer):

```
pid = getpid();
```

- `getppid` returns the identifier of the parent.
- Note:
  - Zombies can be noticed by running the '`ps`' command (shows the process list); you will see the string "<`defunct`>" as their command name:

```
ps -ef
```

```
ps -ef | grep mdamian
```

# Fork Example 1: What Output?

---

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid != 0) {
        printf("parent: x = %d\n", --x);
        exit(0);
    } else {
        printf("child: x = %d\n", ++x);
        exit(0);
    }
}
```

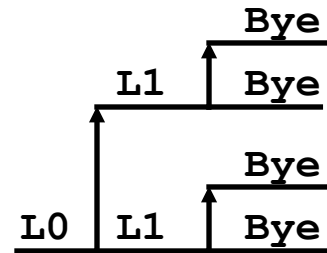
# Fork Example 2

---

## ■ Key Points

- Both parent and child can continue forking

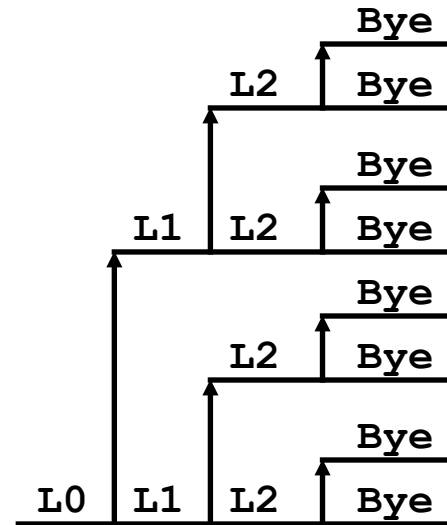
```
void fork2()  
{  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("Bye\n");  
}
```



# Fork Example 3

---

```
void fork3()  
{  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("L2\n");  
    fork();  
    printf("Bye\n");  
}
```



# Fork Example 4

---

```
void fork4()  
{  
    printf("L0\n");  
    if (fork() != 0) {  
        printf("L1\n");  
        if (fork() != 0) {  
            printf("L2\n");  
            fork();  
        }  
    }  
    printf("Bye\n");  
}
```

# Fork Example 5

---

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```

# Summary

---

## ■ Fork

- Creates a duplicate of the calling process
- The result is two processes: parent and child
- Both continue executing from the same point on

## ■ Exit

- Orderly program termination
- Unblocks waiting parent

## ■ Wait

- Used by parent
- Waits for child to finish execution

# Unix system calls

---

execv

execl



# Unix's `execv`

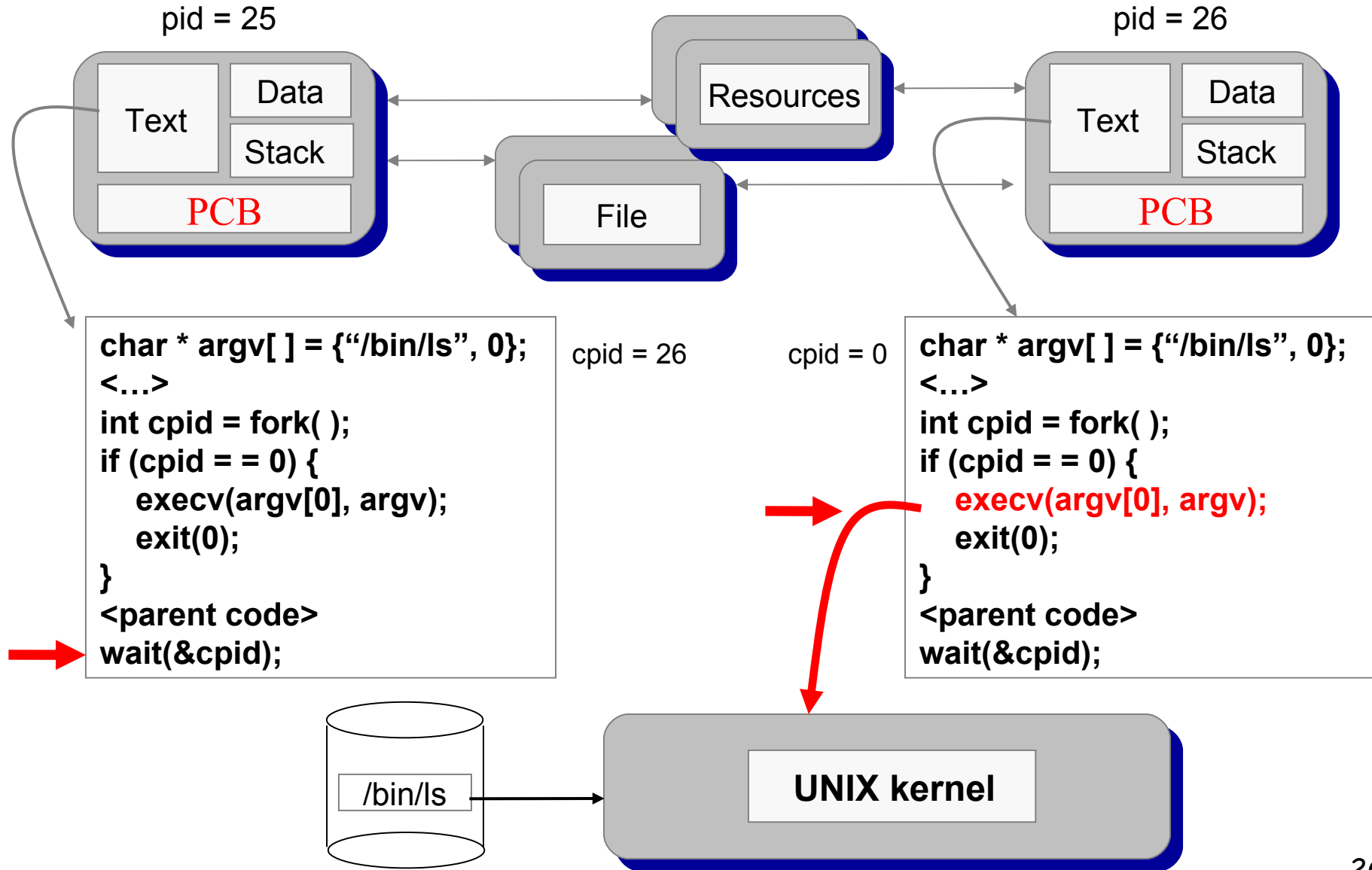
---

- The system call `execv` executes a file, transforming the calling process into a new process. After a successful `execv`, there is no return to the calling process.

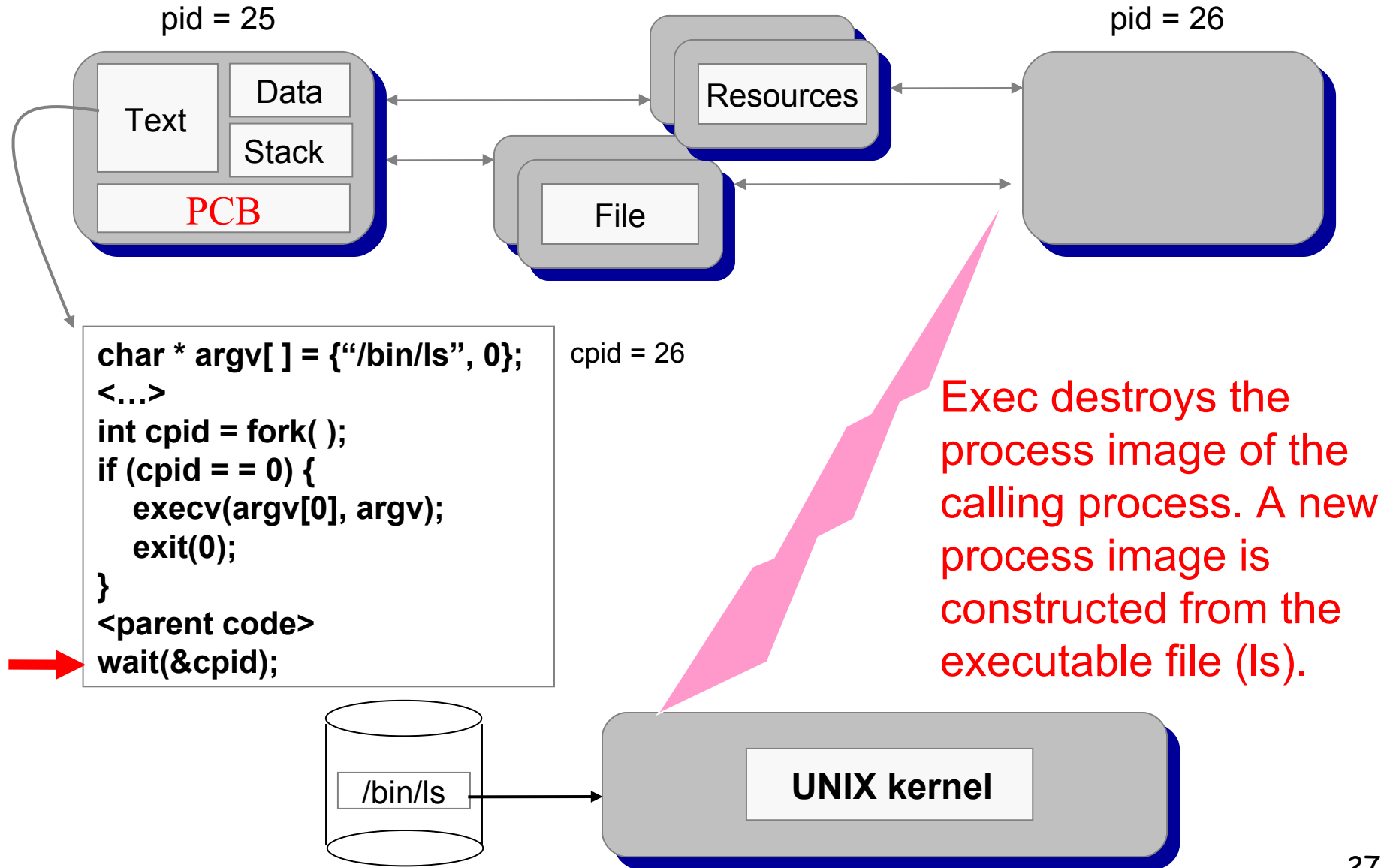
```
execv(const char * path, char * const argv[])
```

- `path` is the full path for the file to be executed
- `argv` is the array of arguments for the program to execute
  - each argument is a null-terminated string
  - the first argument is the name of the program
  - the last entry in `argv` is `NULL`

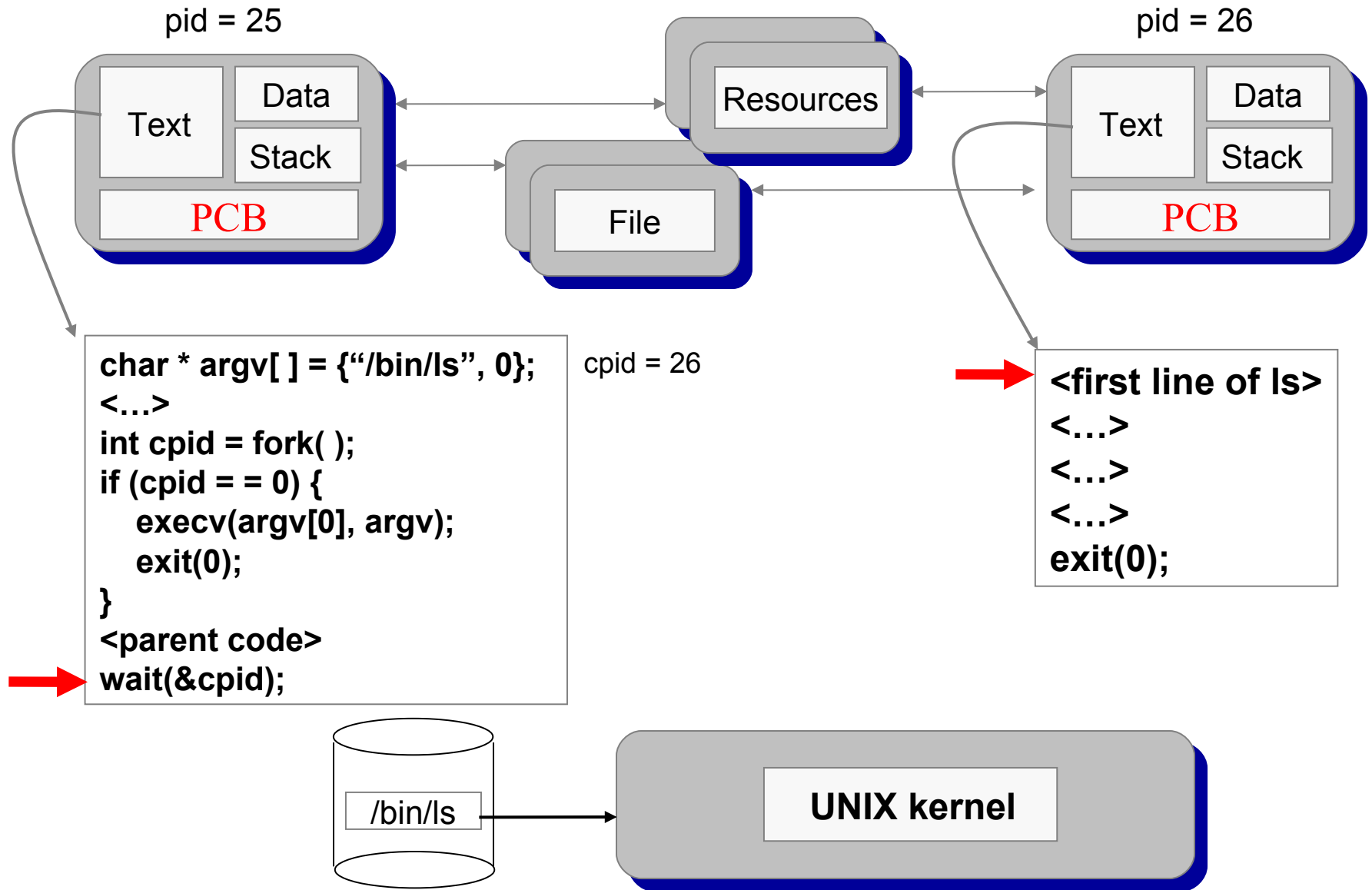
# How execv Works (1)



# How execv Works (2)

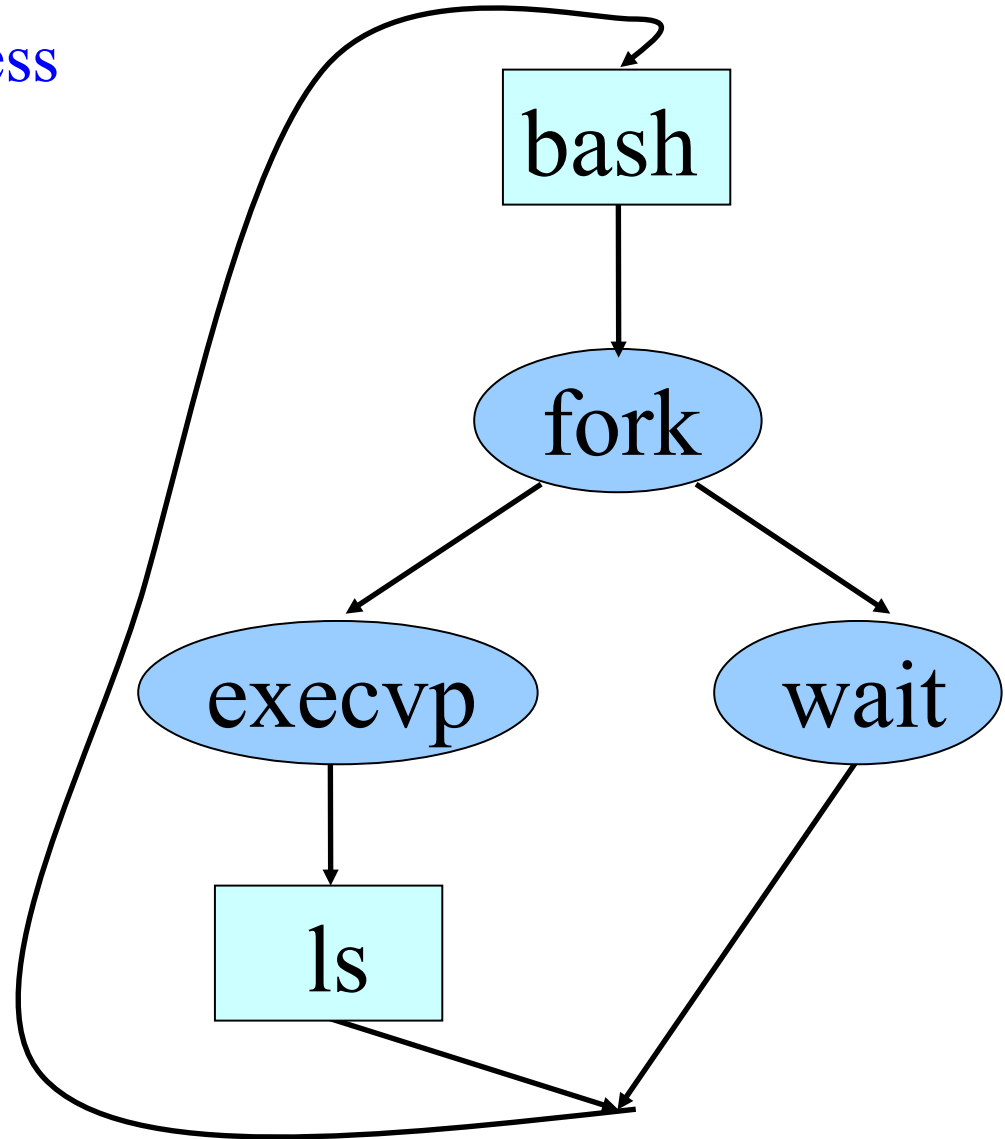


# How execv Works (3)



# Example: A Simple Shell

- Shell is the parent process
  - E.g., bash
- Parses command line
  - E.g., “ls -l”
- Invokes child process
  - Fork, execvp
- Waits for child
  - Wait



# execv Example

---

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

char * argv[] = {"/bin/ls", "-l", 0};
int main()
{
    int pid, status;

    if ( (pid = fork() ) < 0 )
    {
        printf("Fork error \n");
        exit(1);
    }
    if(pid == 0)    { /* Child executes here */
        execv(argv[0], argv);
        printf("Exec error \n");
        exit(1);
    } else /* Parent executes here */
        wait(&status);
    printf("Hello there! \n");
    return 0;
}
```

Note the NULL string  
at the end

# execv Example – Sample Output

---

- Sample output:

**total 282**

**drwxr-xr-x 2 mdamian faculty 512 Jan 29 14:02 assignments**

**-rw-r--r-- 1 mdamian faculty 3404 Jan 29 14:05 index.html**

**drwxr-xr-x 2 mdamian faculty 512 Jan 28 15:02 notes**

**Hello there!**

# execl

---

- Same as execv, but takes the arguments of the new program as a list, not a vector:

- Example:

```
execl("/bin/ls", "/bin/ls", "-l", 0);
```

- Is equivalent to

Note the NULL string at the end



```
char * argv[] = {"/bin/ls", "-l", 0};  
execv(argv[0], argv);
```

- execl is mainly used when the number of arguments is known in advance



# General purpose process creation

---

- In the parent process:

```
int childPid;
char * const argv[ ] = {...};

main {
    childPid = fork();
    if(childPid == 0)
    {
        // I am child ...
        // Do some cleaning, close files
        execv(argv[0], argv);
    }
    else
    {
        // I am parent ...
        <code for parent process>
        wait(0);
    }
}
```

# Combined fork/exec/wait

---

- Common combination of operations
  - Fork to create a new child process
  - Exec to invoke new program in child process
  - Wait in the parent process for the child to complete
- Single call that combines all three
  - `int system(const char *cmd);`
- Example

```
int main()
{
    system("echo Hello world");
}
```

# Properties of fork / exec sequence

---

- In 99% of the time, we call `execv(...)` after `fork()`
  - the memory copying during `fork()` is useless
  - the child process will likely close open files and connections
  - overhead is therefore high
  - might as well combine both in one call (`OS/2`)
- `vfork()`
  - a system call that creates a process without creating an identical memory image
  - sometimes called “lightweight” fork
  - child process is understood to call `execv()` almost immediately

# Variations of execv

---

- `execv`

- Program arguments passed as an array of strings

- `execvp`

- Extension of `execv`
- Searches for the program name in the PATH environment

- `execl`

- Program arguments passed directly as a list

- `execlp`

- Extension of `execl`
- Searches for the program name in the PATH environment

# Summary

---

- `exec(v,vp,l,lp)`

- Does NOT create a new process
- Loads a new program in the image of the calling process
- The first argument is the program name (or full path)
- Program arguments are passed as a vector (`v,vp`) or list (`l,lp`)
- Commonly called by a forked child

- `system:`

- combines fork, wait, and exec all in one