Processes

Operating Systems

Assoc. Prof. Milos Jovanovik, PhD

Processes

- **Processes are programs in execution**
- Unit of work in a modern time-sharing system
- An instance of an executing program, including the current values of the program counter, registers, and variables
- They are the basic OS abstraction!
    - Users think that they are working on systems with several (virtual) CPU, while working on a single-CPU-system

Simultaneous Processes

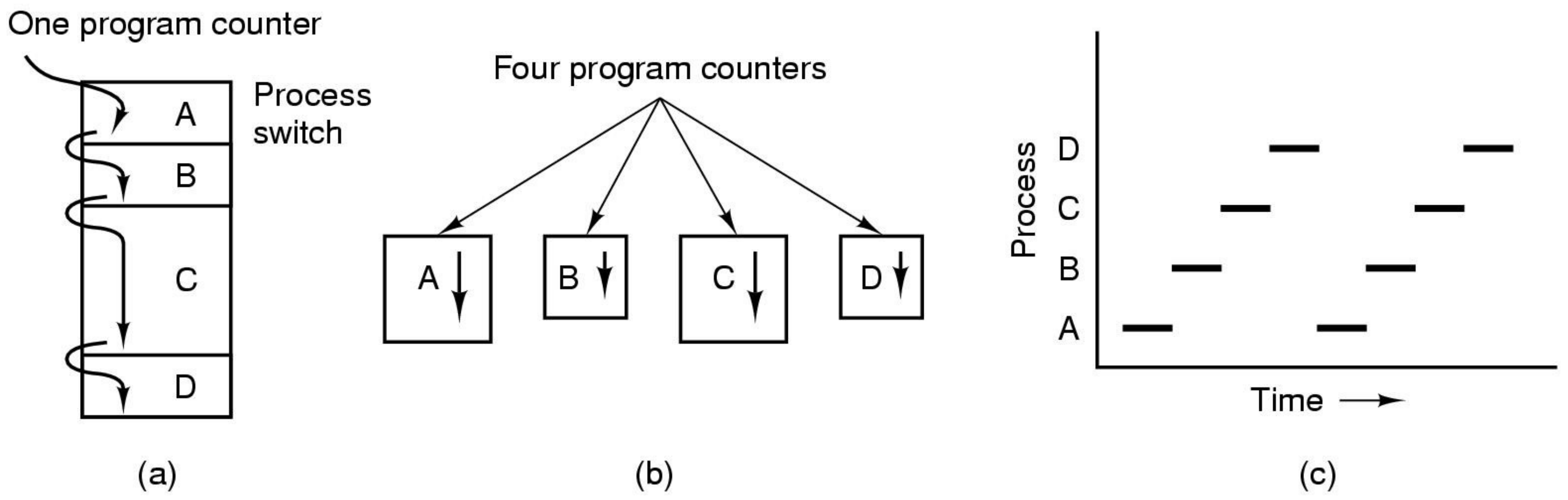
- At every moment, there are several processes working in parallel
  - System processes
  - User processes
- The users have an illusion that the processes are executing in parallel
- Example: the user is surfing the Web and at the same time in the background there is:
  - A process that waits for incoming email;
  - A process of the antivirus program to check periodically for new virus definition;
  - Processes for printing files and backing up the user’s data on a USB stick;

Pseudo Parallelism

- Parallel execution illusion
  - Several processes run on one CPU
  - Each process obtains a time quantum

- OS creators have made a conceptual **model of sequential processes**

The Process Model



a) Multiprogramming four processes;
b) Conceptual model of four independent, sequential processes;
c) Only one process is active at once;

Process Creation

- Four principal events that cause processes to be created:
  1. System initialization
  2. One process creates another processes
  3. Process creation by user demand
  4. Initiation of batch jobs

- A process is created with a system call:
  - UNIX: fork();
  - Windows: CreateProcess(10 parameters);

Process Termination

- Conditions for Process Termination:
  1. Normal exit (voluntarily)
  2. Error exit (voluntary)
  3. Fatal error (by force, involuntary)
  4. Killed by another process (by force)

Process Termination

- A process is voluntarily terminated with a system call:
  - UNIX: exit
  - Windows: ExitProcess
- A process is involuntarily terminated with a system call:
  - UNIX: kill
  - Windows: TerminateProcess

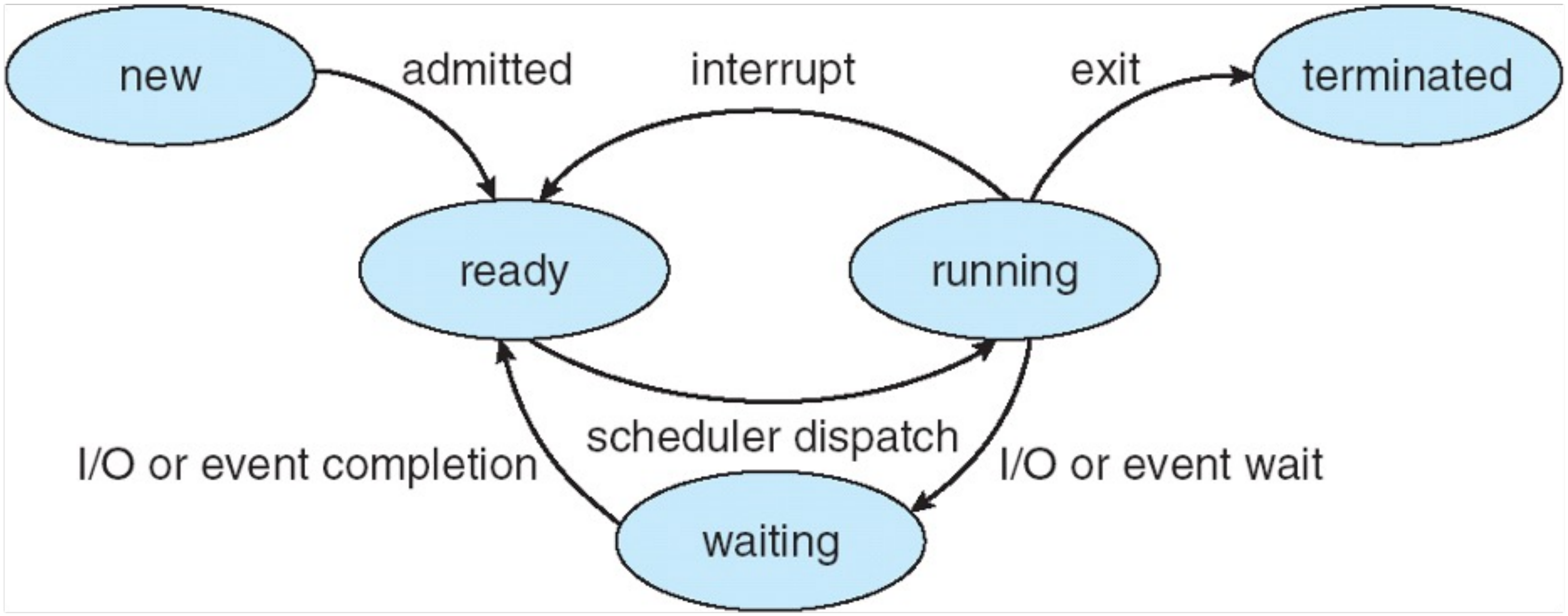
Process Hierarchy

- Parent processes create child processes
- Child processes can create their own child processes
- In UNIX, there is hierarchy of processes (process group)
  - The init process is started after boot;
  - Init forks a new process per terminal, waiting for login;
  - If a login is successful, the login process executes a shell to accept commands which may start up more processes, and so forth;
  - All the processes in the entire system belong to a single tree, with init at the root;
- Windows does not have a concepts of hierarchy. All processes are equal.

Process States

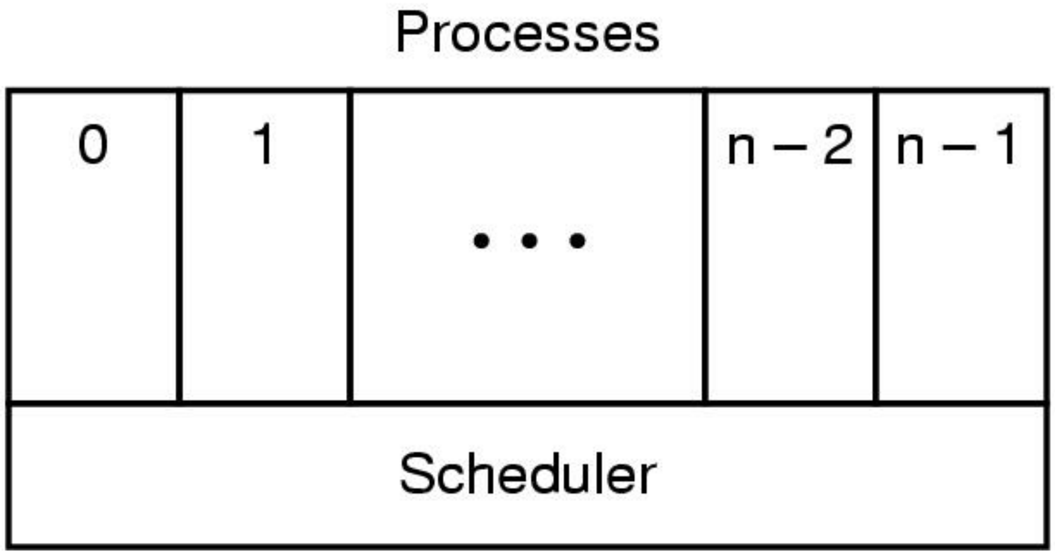
- A state defines the present activity of a process:
  - **New**: A process is created;
  - **Active** (**Running**): Execution of its instructions;
  - **Blocked** (**Waiting**): The process is waiting for something to happen (I/O completion or reception of a signal);
  - **Ready**: Waiting for the CPU – the other resources are already assigned;
  - **Terminated**: The process has ended;

Process Life-Cycle



- Transition:
  - Program actions (system calls)
  - OS actions (scheduling)
  - External events (interrupts)

Process Model in the OS



- Scheduler: the lowest level of the OS responsible for the interrupts and process scheduling.
- The higher level are processes that can be executed sequentially.
- The scheduler is a process that schedules execution of higher level processes.

Process Scheduler

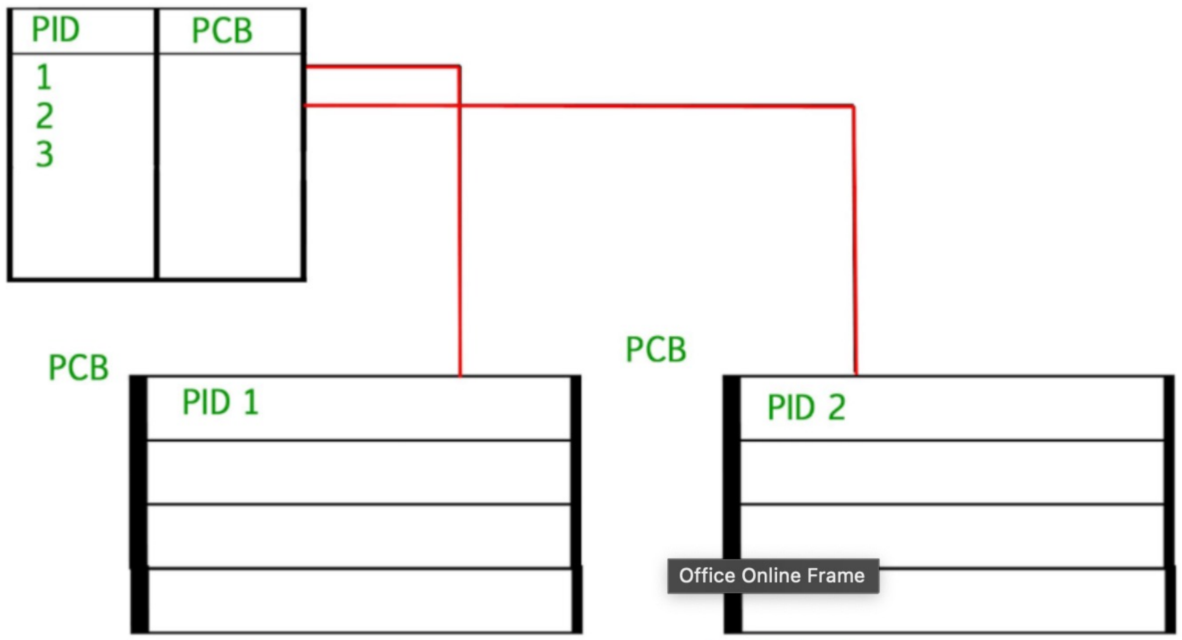
- The scheduler is the part of the OS that decides which process runs at a certain point in time
- Tasks of the process scheduler:
  - Pause a running process, moving it to the “ready” queue;
  - Start a process from the “ready” queue;
    - The decision on which process will be run, is based on a scheduling strategy;
- Preemptive scheduling: The executing process is stopped and moved to the “ready” queue, so that other processes can get CPU time.
- Non-preemptive scheduling: The process decides itself when to move to the “ready” queue.

Implementation of Processes

- The OS maintains a Process Table (array of structures)
  - One process = One entry
  - Each entry is called Process Control Block (PCB)
  - PCB contains important data related to the process
  - It allows efficient and centralized access to all the process information
  - Ready queues usually use pointers to PCB

Process Model Implementation

- The OS has table of all running processes
  - UNIX / Linux: ps –a
  - Windows: tasklist



Process table and process control block

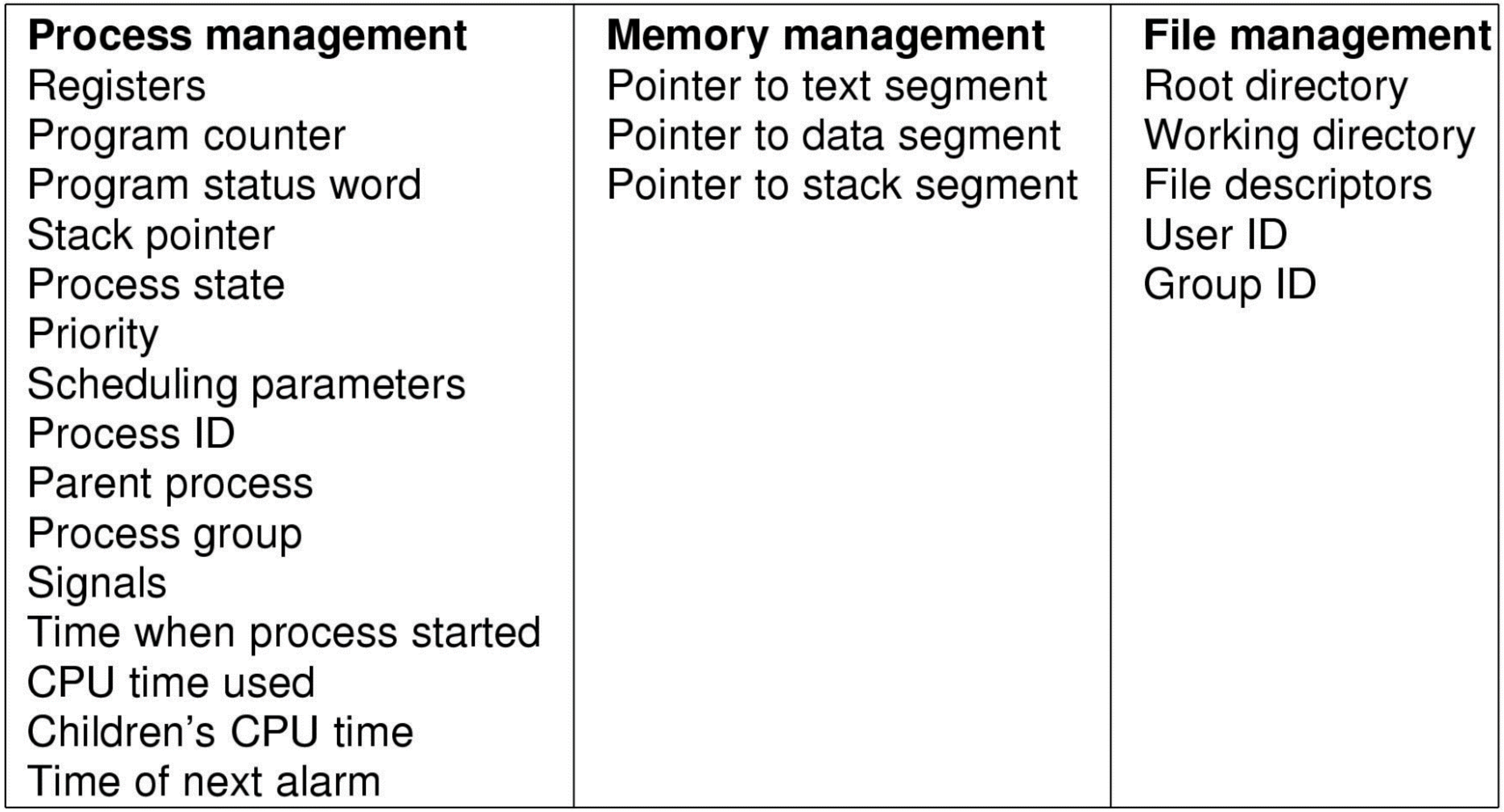
PCB Information …

- Process ID (PID)
  - Unique;
- Process state
- Program counter
  - Address of next instruction;
- CPU registers
  - To keep the state info when a process is interrupted, so the process can continue after the interrupt;
- CPU-scheduling information
  - Pointers to the scheduling queues;
  - Process priority;

… PCB Information

- Memory management info
  - Memory description (base and limit registers, page tables or the segment tables);
- User identification numbers
  - uid, gid, euid, egid;
- I/O status
  - It includes list of I/O devices assigned to the process, list of open files, etc.;
- Information about:
  - CPU usage time;
  - Execution time (real);
  - Limits and usage quotas;
- Pointers to parent and child processes

PCB Entries



Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Some fields in a typical PCB

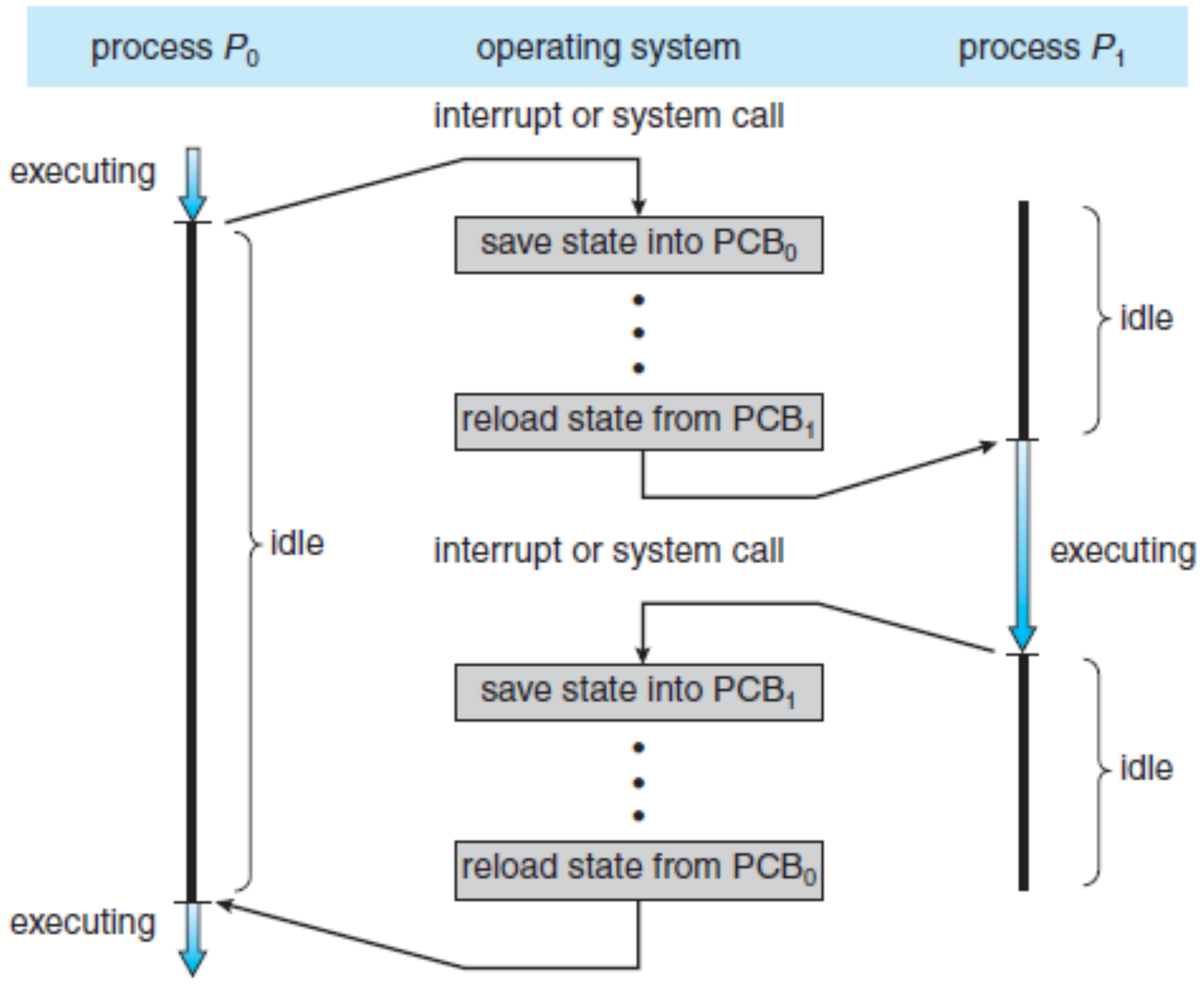
Process Context

- The process context is represented in PCB and has:
  - CPU register values;
  - Process status;
  - Memory management info;
- The process has a set of “private” CPU registers in main memory
  - The data is saved to main memory when the process switches from ready to active (running);
  - They must be saved back into the main memory when the process switches from active to ready or waits I/O;

Context Switch

- Context switch is CPU assignment to another process
- Context switch is done when:
  - There is an interrupt;
  - A process terminates;
  - A process goes to sleep;
  - There is a system call;
- Context switching is expensive
  - The OS kernel saves the context of the old process in its PCB;
  - The kernel loads the context of the new process;
  - It is a crucial factor for efficiency and its price rises with the CPU speedup;
  - The OS is more complex – there is a lot of work when context switching;

CPU switch from Process to Process



When to Switch?

- Time is up (clock interrupt)
  - The process has spent the dedicated time;
- I/O interrupt
- Memory Faults
  - The requested memory address from the virtual memory has to be stored in the working memory;
- Trap
  - There was a certain error
  - The processes could go to exit state
- System Call
  - Open File
  - Exit

UNIX: Creating, Executing and Terminating Processes

- A new process is created using fork()
- Program execution: exec() family
- Terminating: exit()

UNIX fork() (1)

- It creates a child process
  - Two different process have the same copy of one program;
- The child inherits from the parent:
  - Identical copies of the variables and the memory (address space);
  - Identical copies of all CPU registers;

UNIX fork() (2)

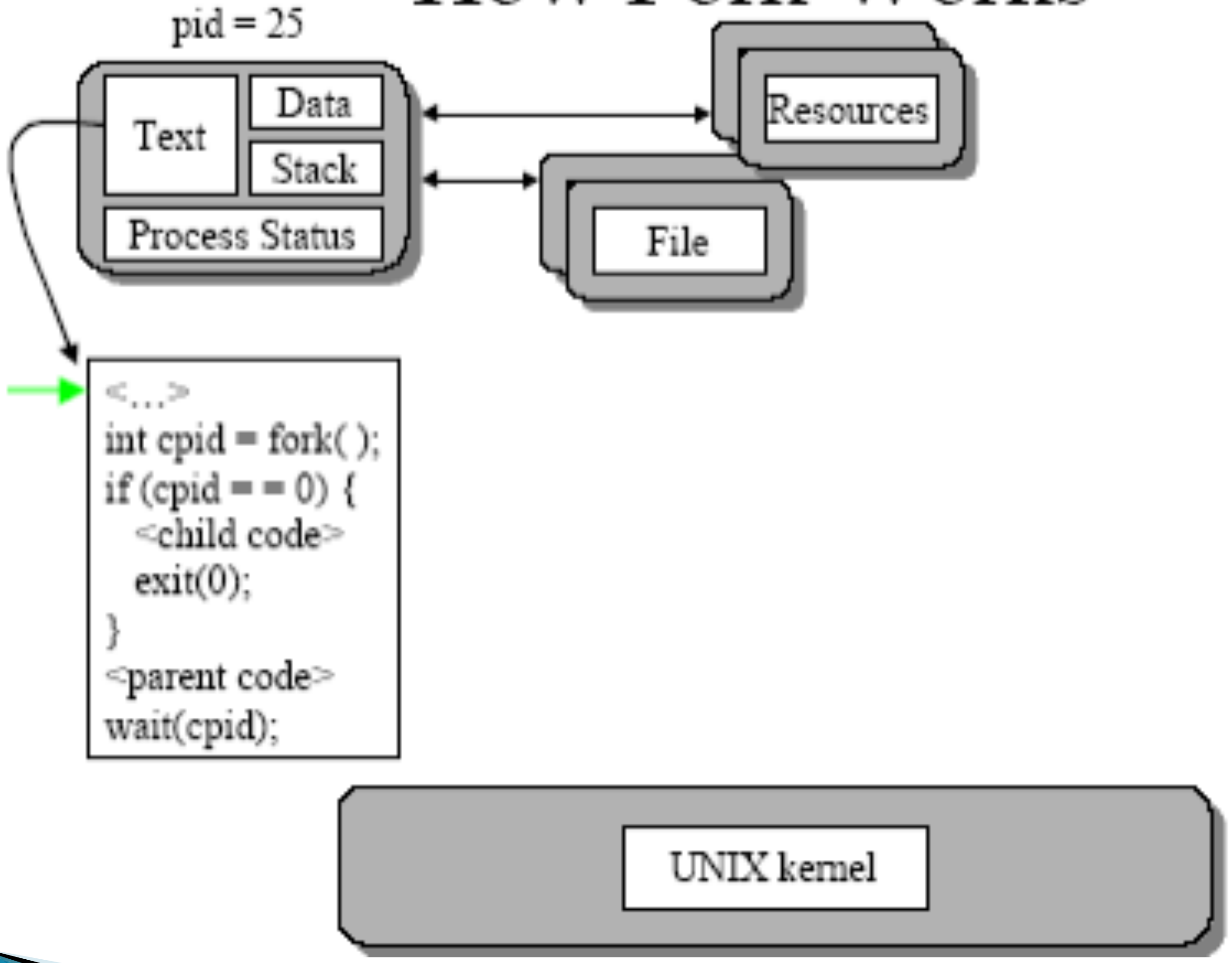
- The two processes after fork (the parent and the child), return to the same point of execution:
  - For the child process, fork() returns 0 (PID for the child);
  - For the parent process, fork() returns the ID of the child process;
- Simple implementation of fork():
  - Allocate memory for the child process;
  - Copy the memory and the CPU registers from the parent to the child process;
- Expensive operation!

Using fork()

```
main()
  …
int pid = fork();    // create a child
if (pid == 0) {      // child continues here
  …
}
else {               // parent continues here
  …
}
```

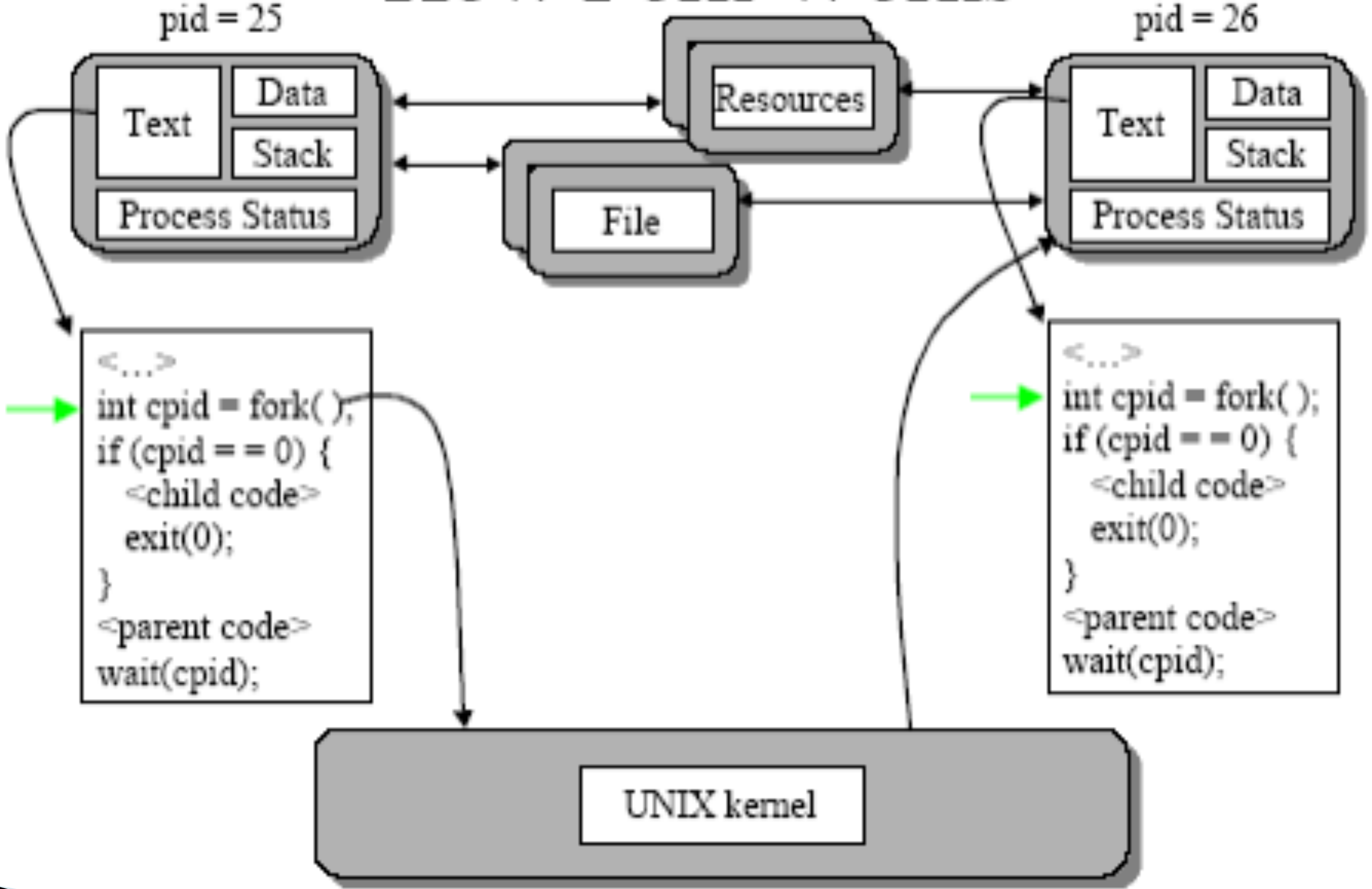
- The code for the child and the parent is in the same program
- The child inherits all open files and network connections

How Fork Works



```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

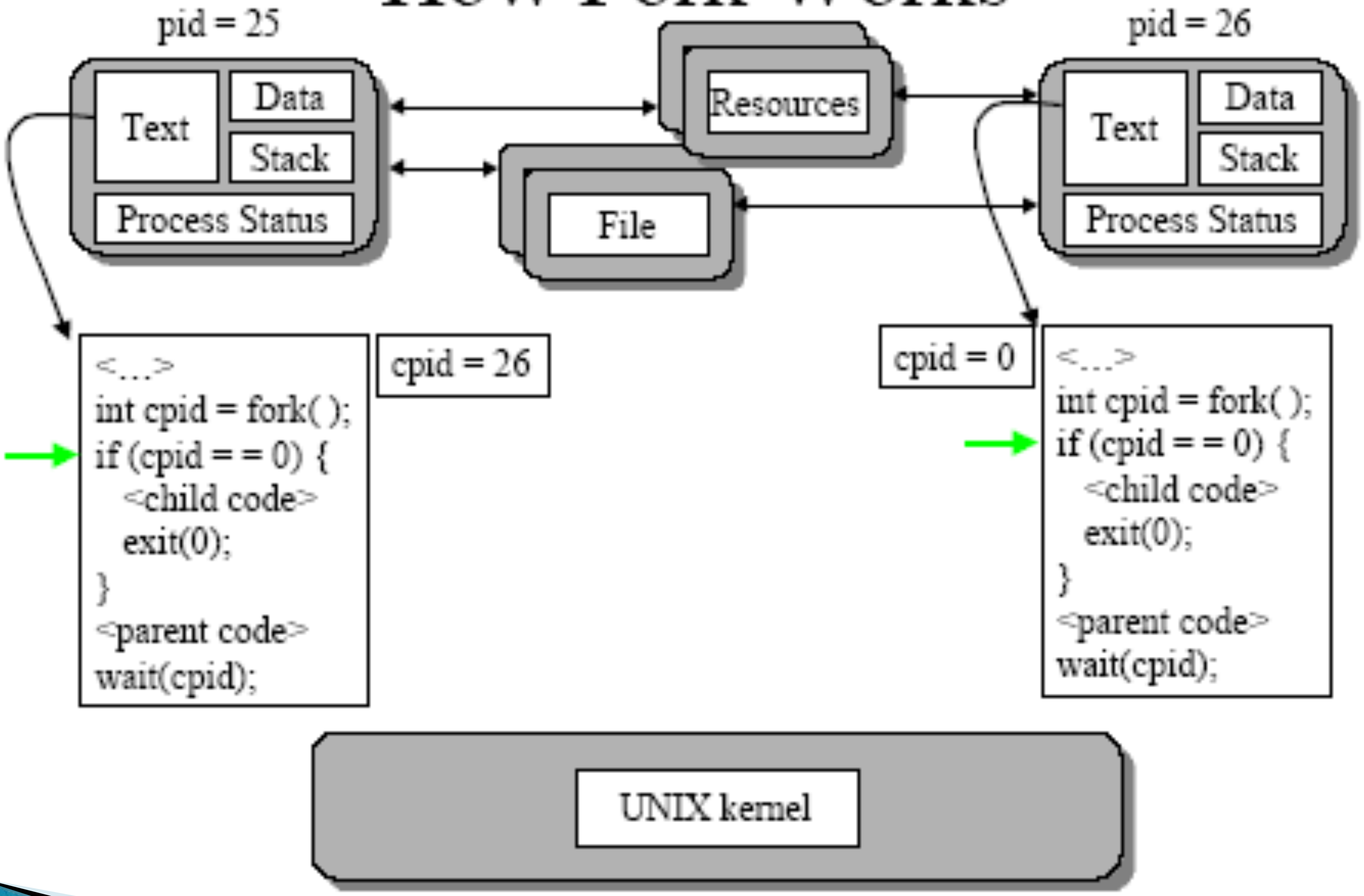
How Fork Works



```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

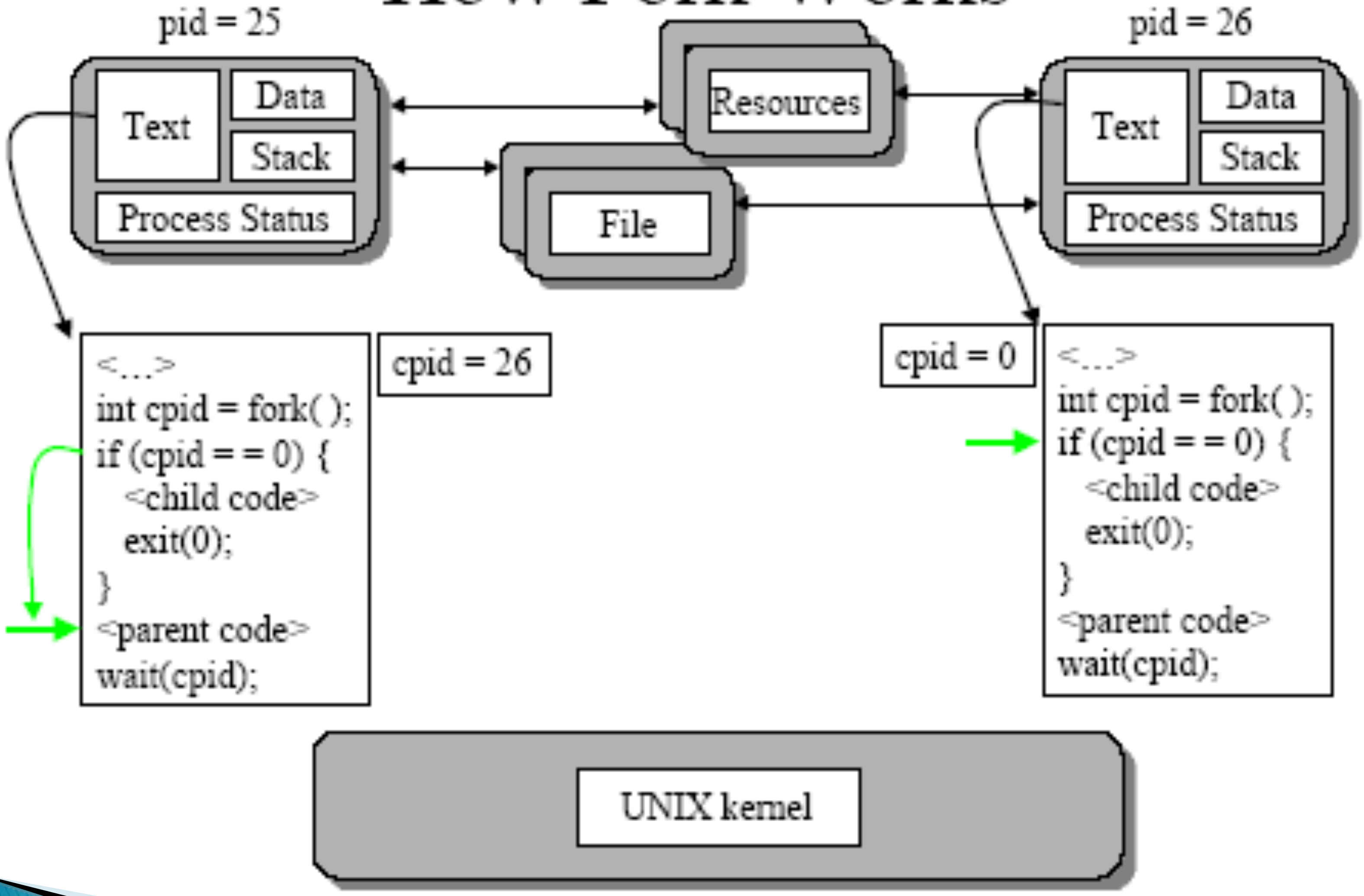
How Fork Works



```
<…>
int cpid = fork( );
if (cpid = = 0) {
  <child code>
  exit(0);
}
<parent code>
wait(cpid);
```

```
<…>
int cpid = fork( );
if (cpid = = 0) {
  <child code>
  exit(0);
}
<parent code>
wait(cpid);
```

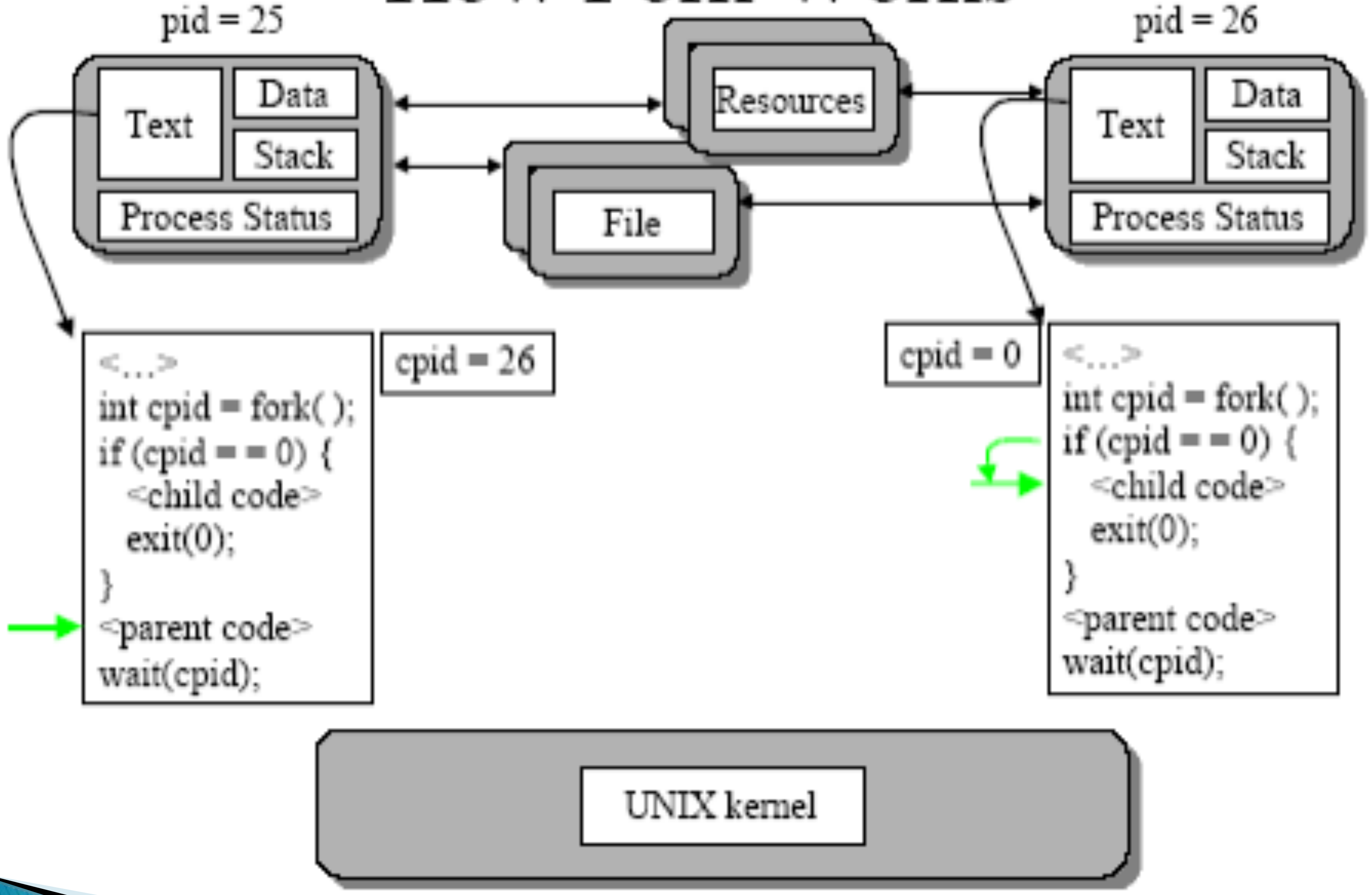
How Fork Works



```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

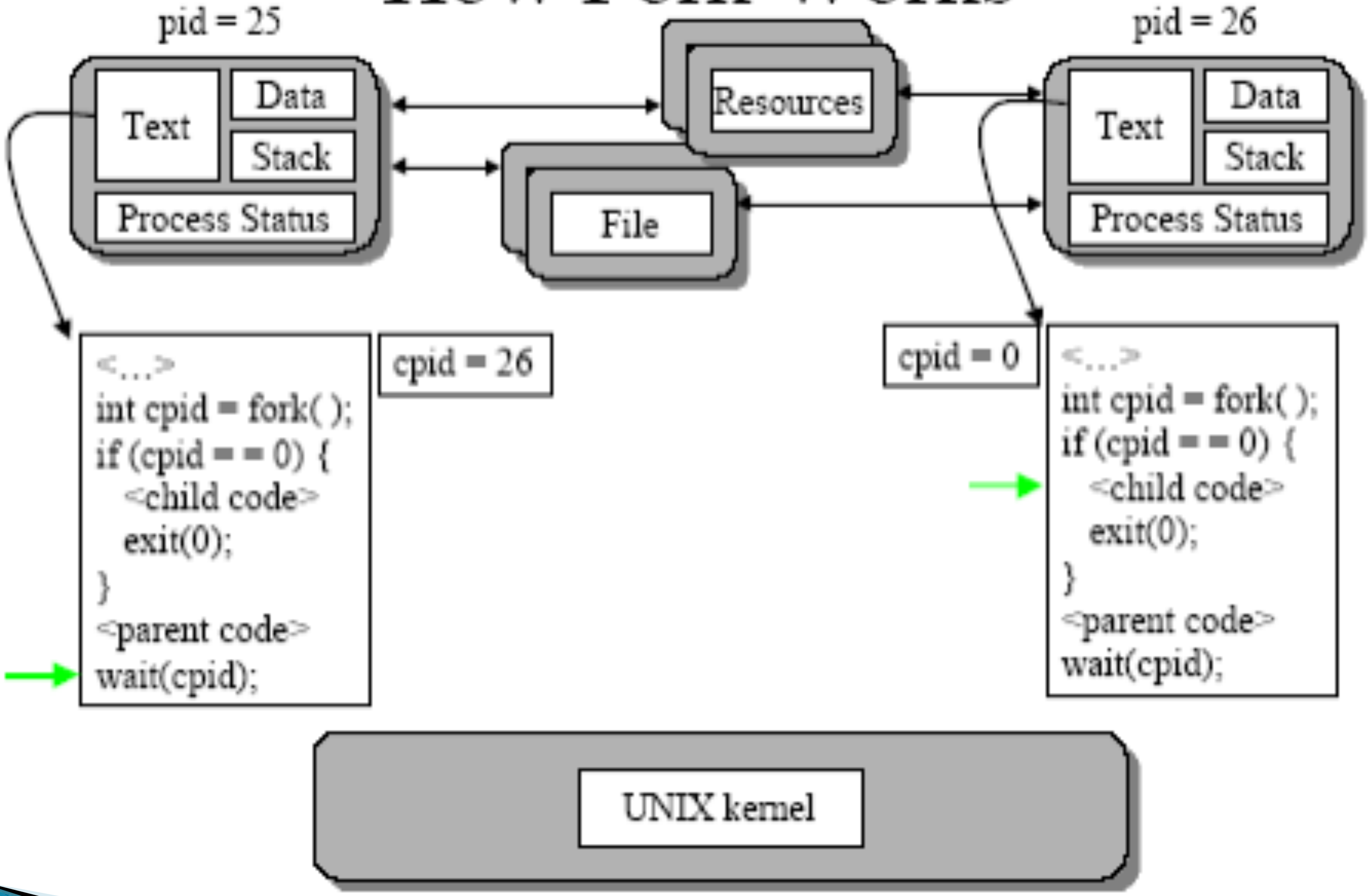
How Fork Works



```
<...>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

```
<...>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

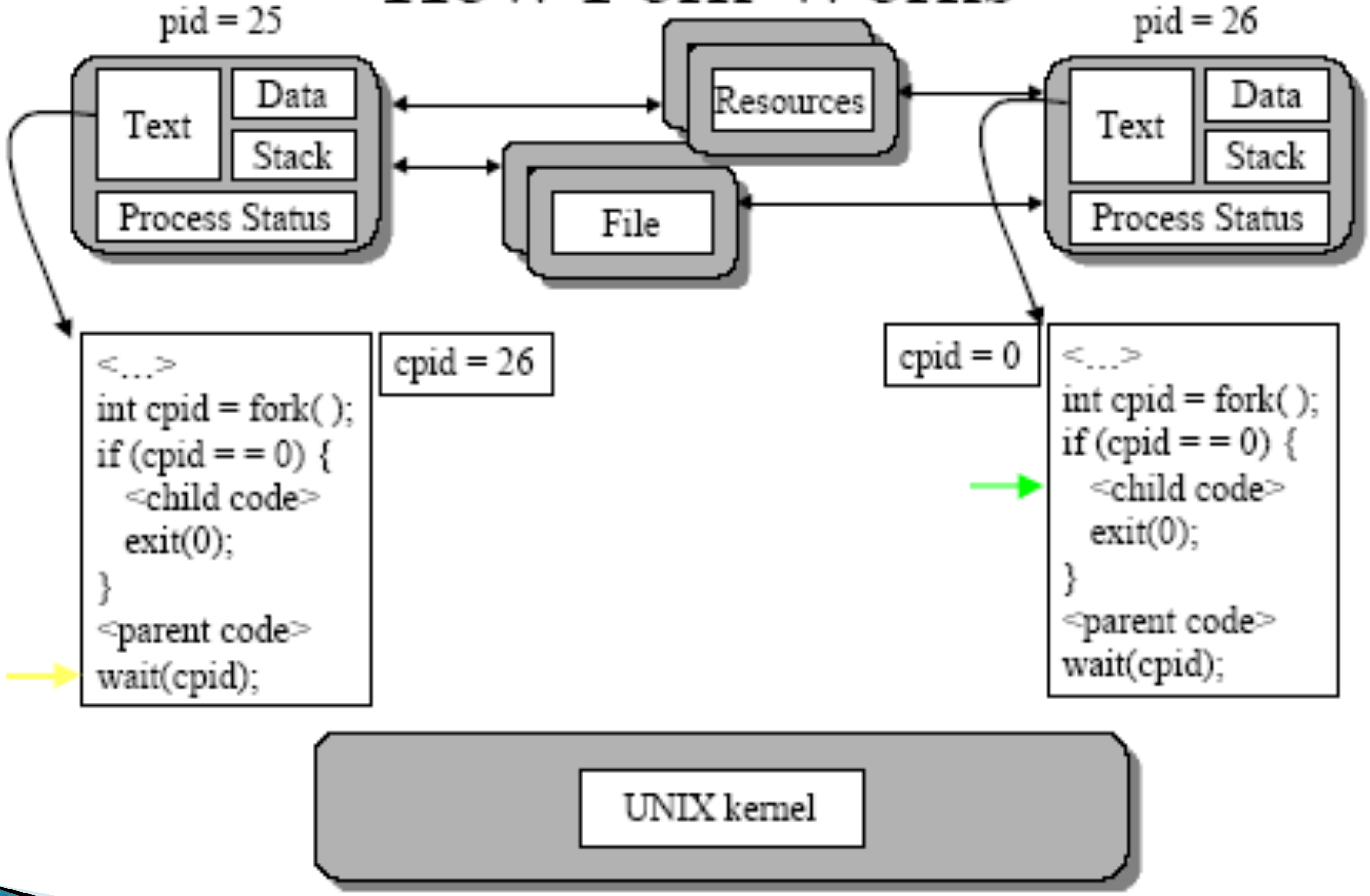
How Fork Works



```
<...>
int cpid = fork( );
if (cpid = = 0) {
  <child code>
  exit(0);
}
<parent code>
wait(cpid);
```

```
<...>
int cpid = fork( );
if (cpid = = 0) {
  <child code>
  exit(0);
}
<parent code>
wait(cpid);
```

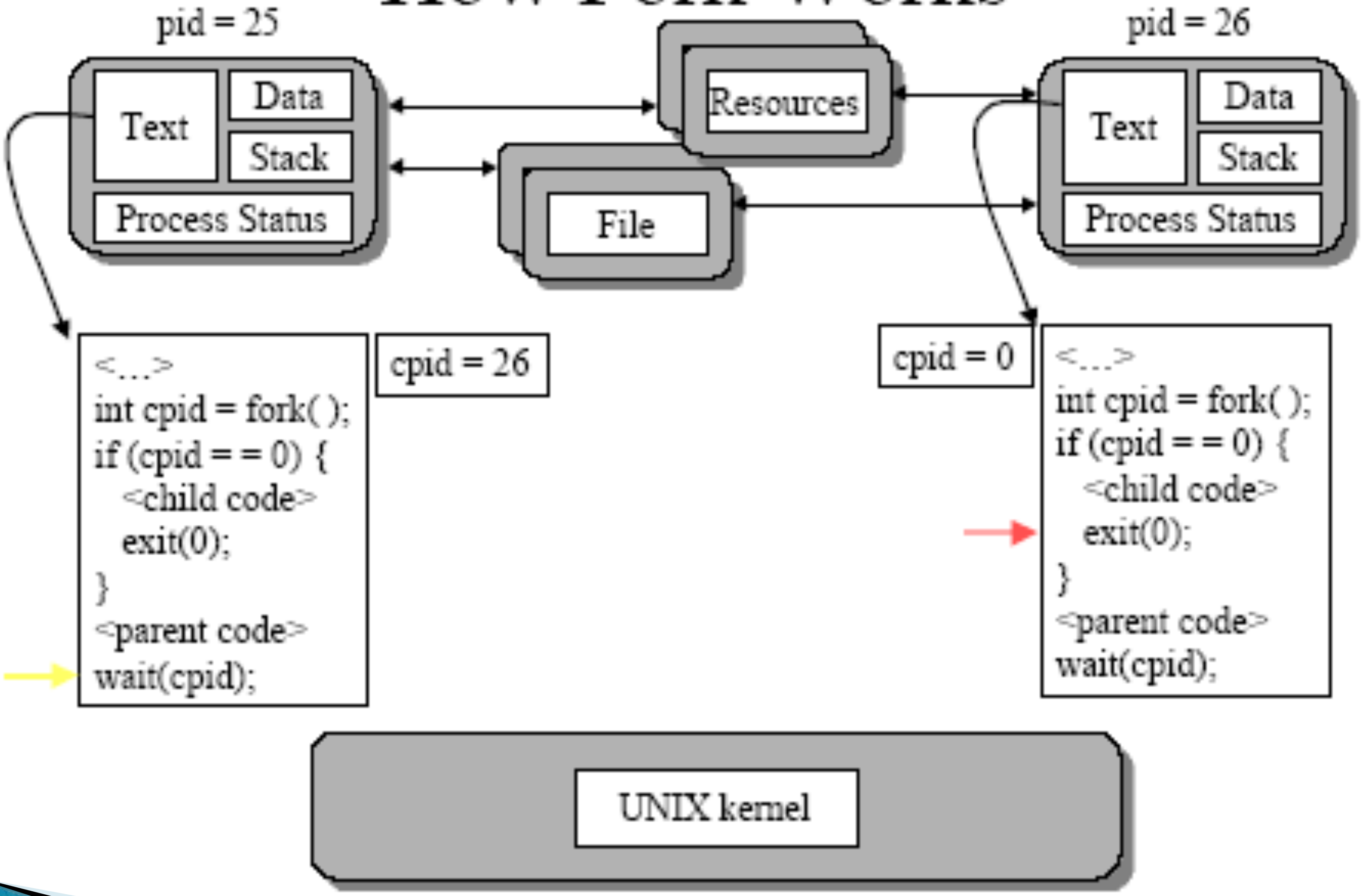
How Fork Works



```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

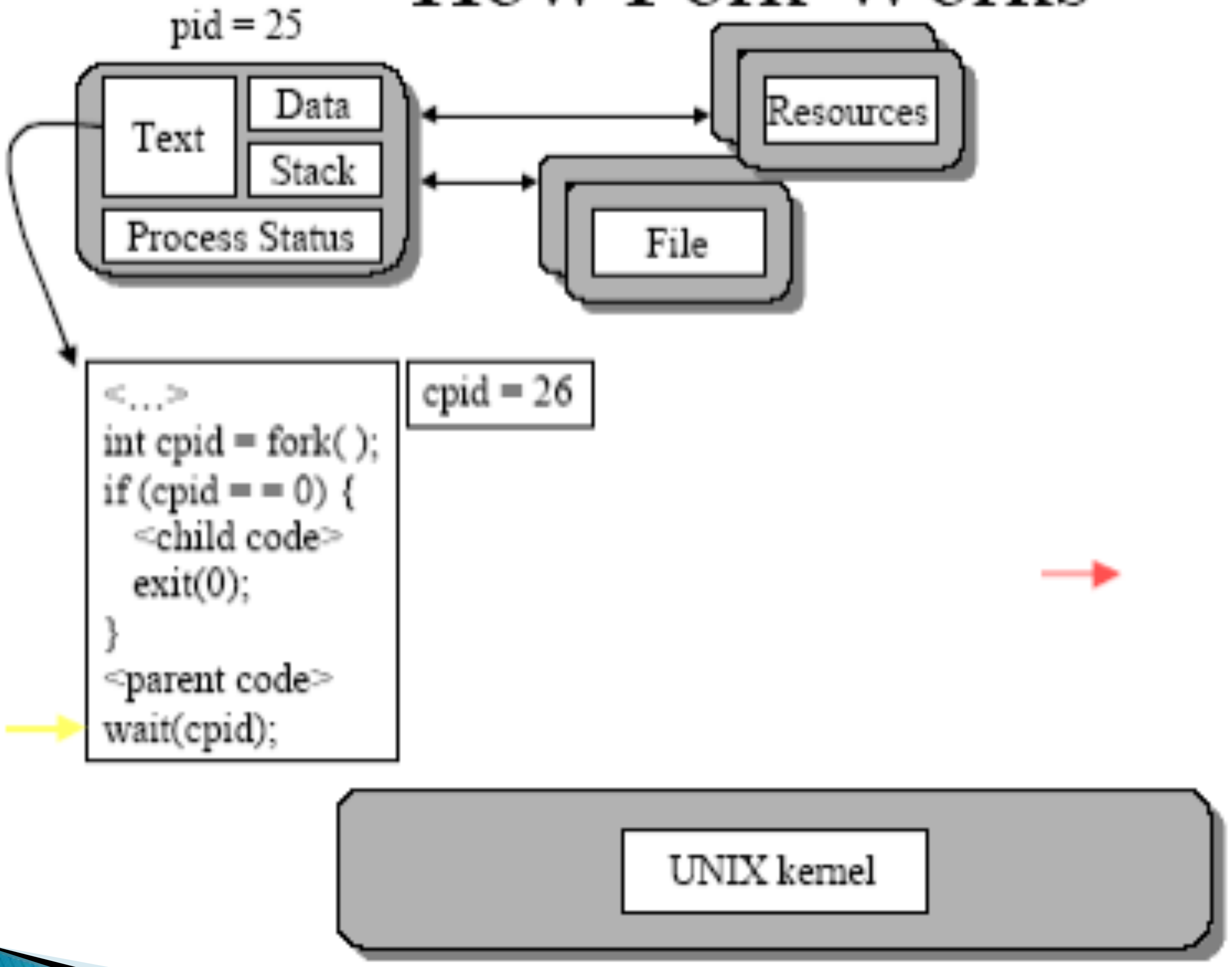
How Fork Works



```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

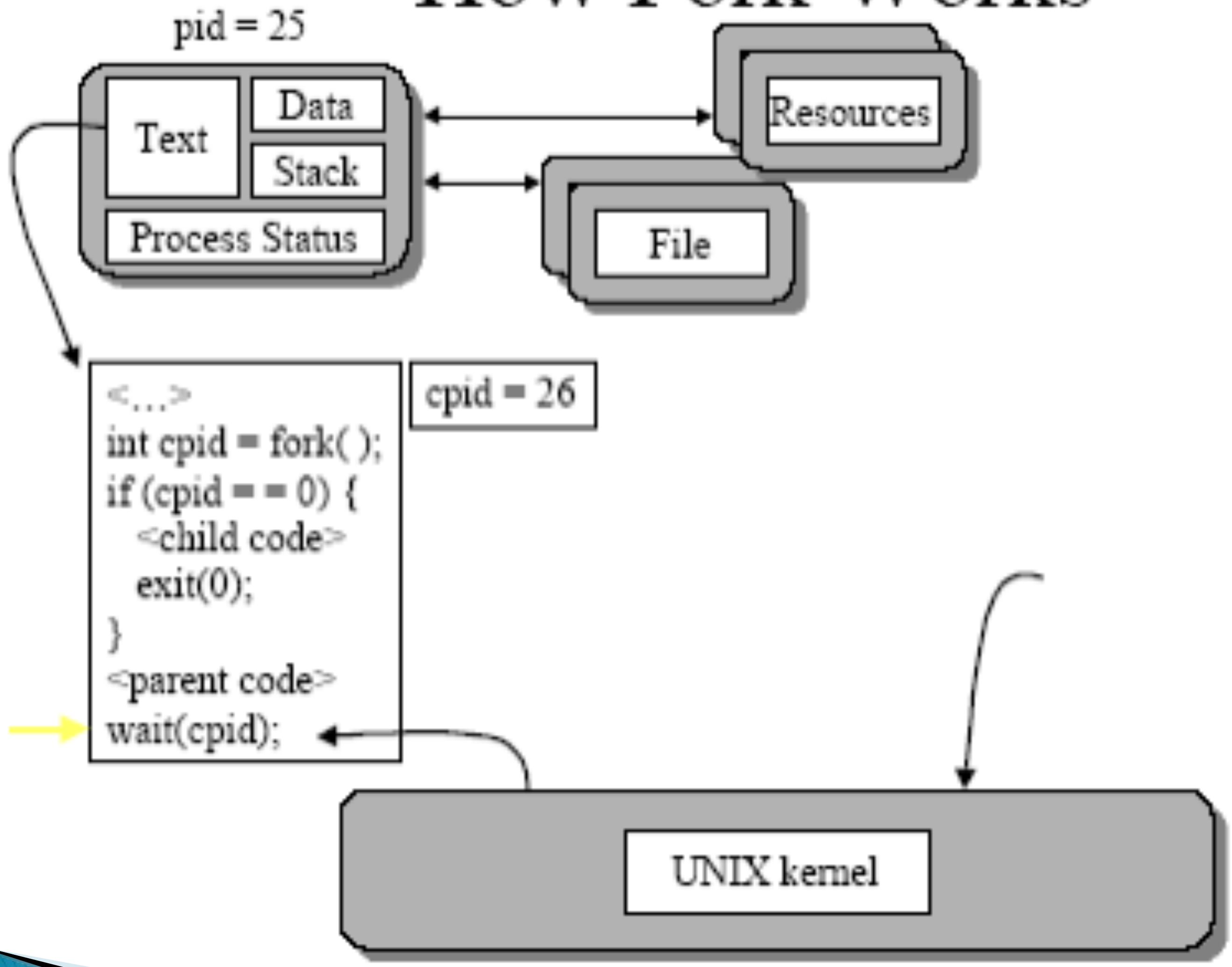
```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

How Fork Works



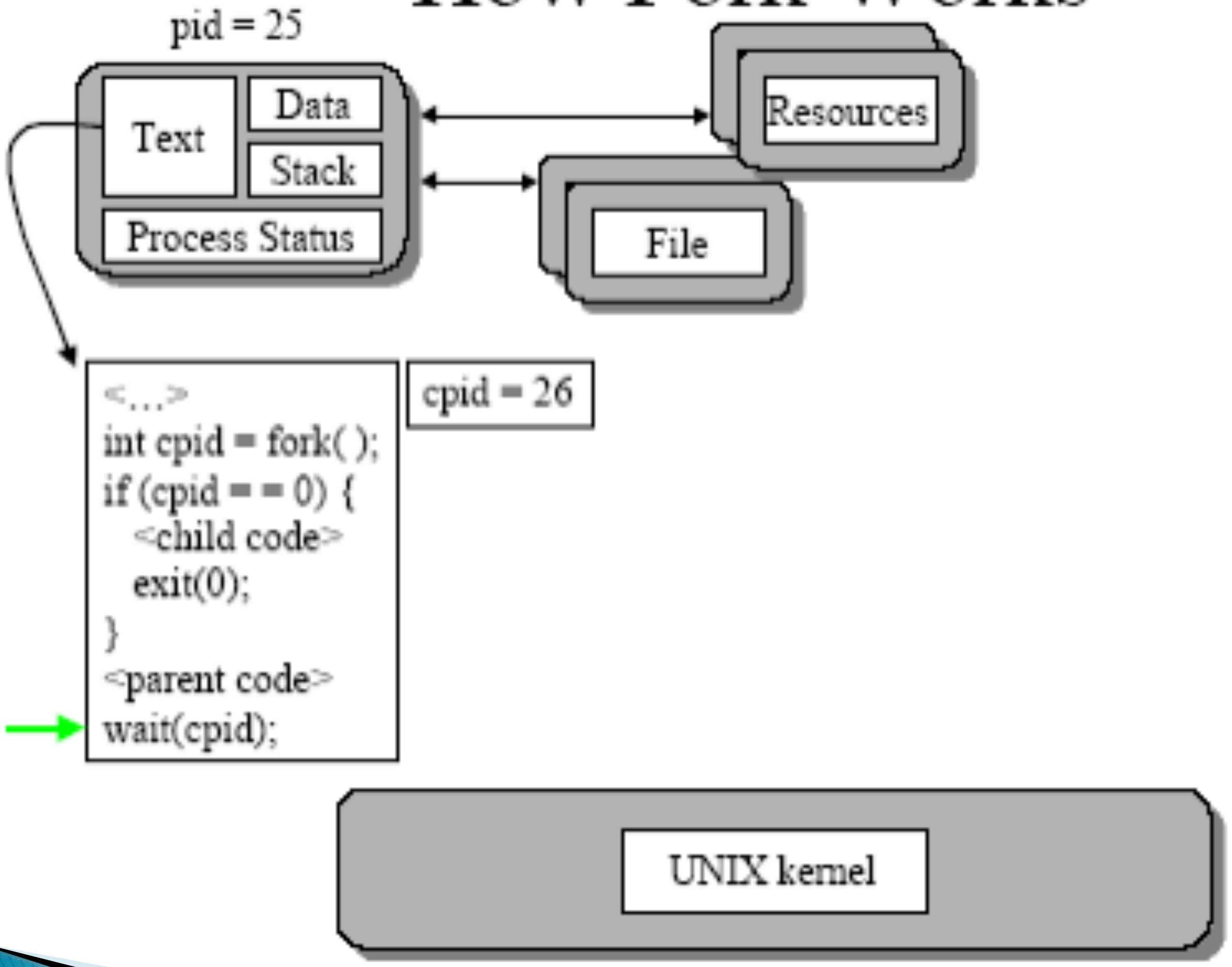
```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

How Fork Works



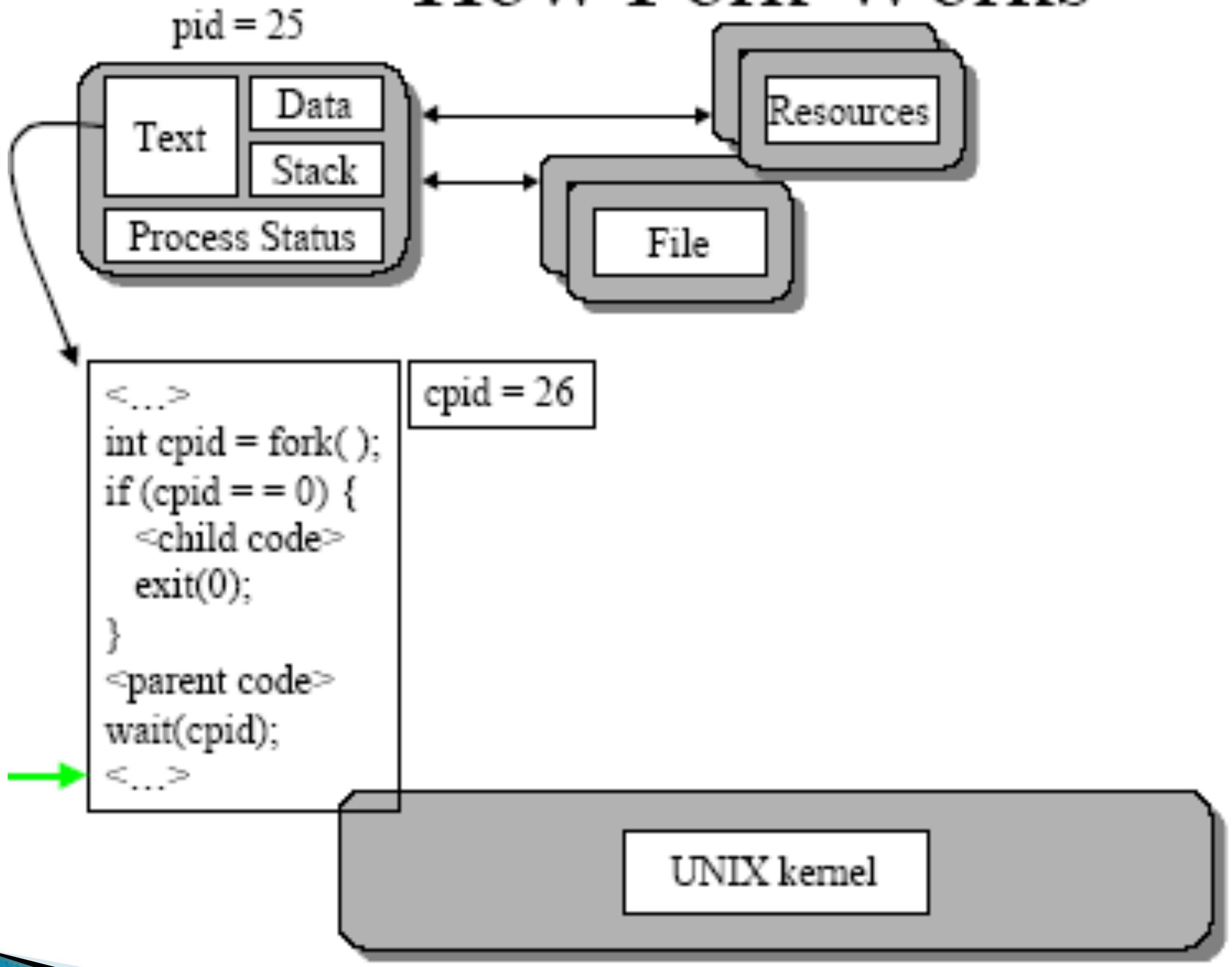
```
<...>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

How Fork Works



```
<...>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

How Fork Works



```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
<…>
```

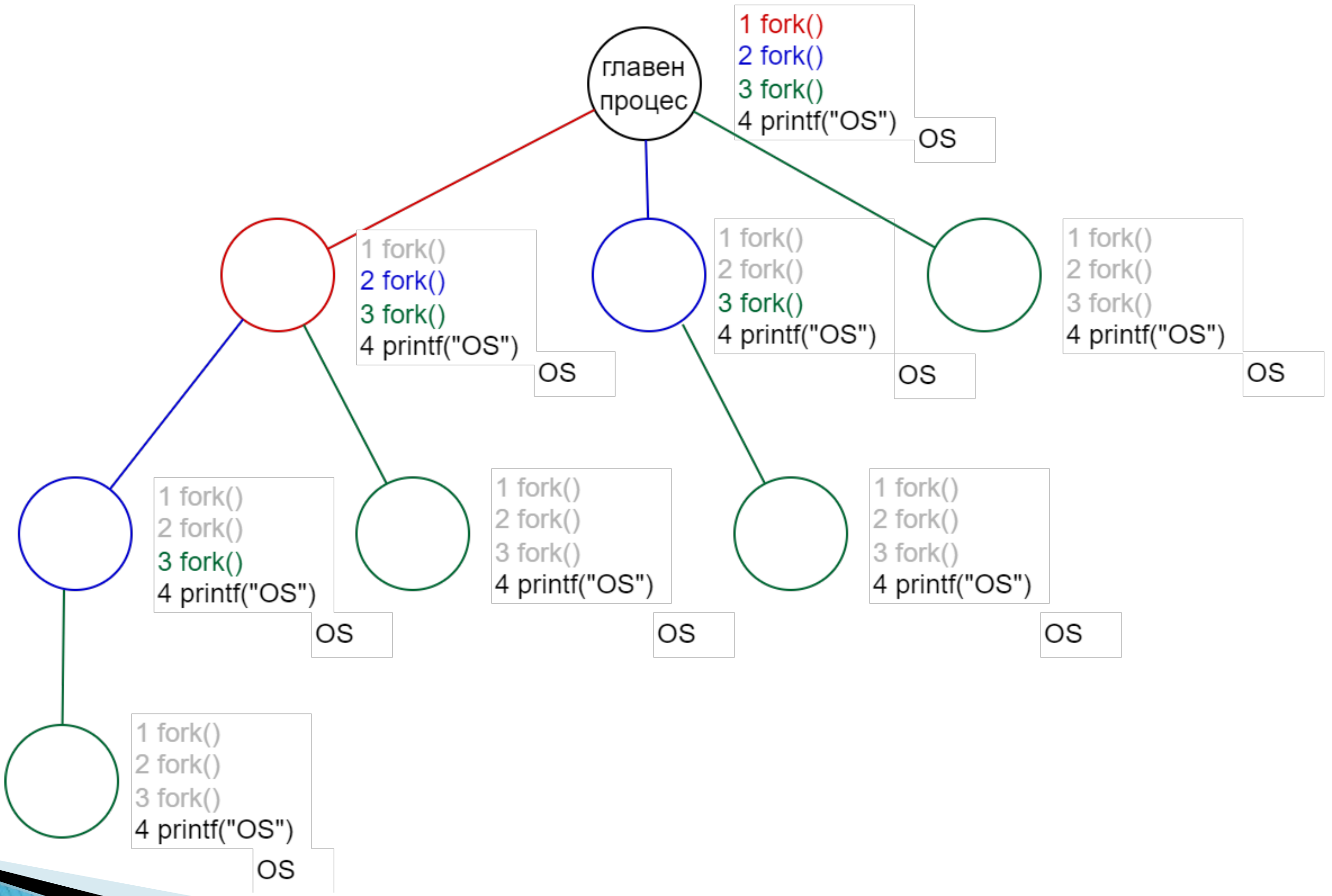
fork() example

- How many times will "OS" be printed after the entire code executes?

```
fork()
fork()
fork()
printf(“OS”)
```

fork() example

- ”OS“ will be printed 8 times.



Change of the Executable: exec()

- exec: execl, execle, execlp, execv, execve, execvp, or exect.
- The exec subprogram, in all its forms, executes a new program in the calling process.
- So, exec does not create a new process, but it overrides the existing program with the new one (new process image).
- It allows the process to get a new argument count (argc) and new arguments (argv).

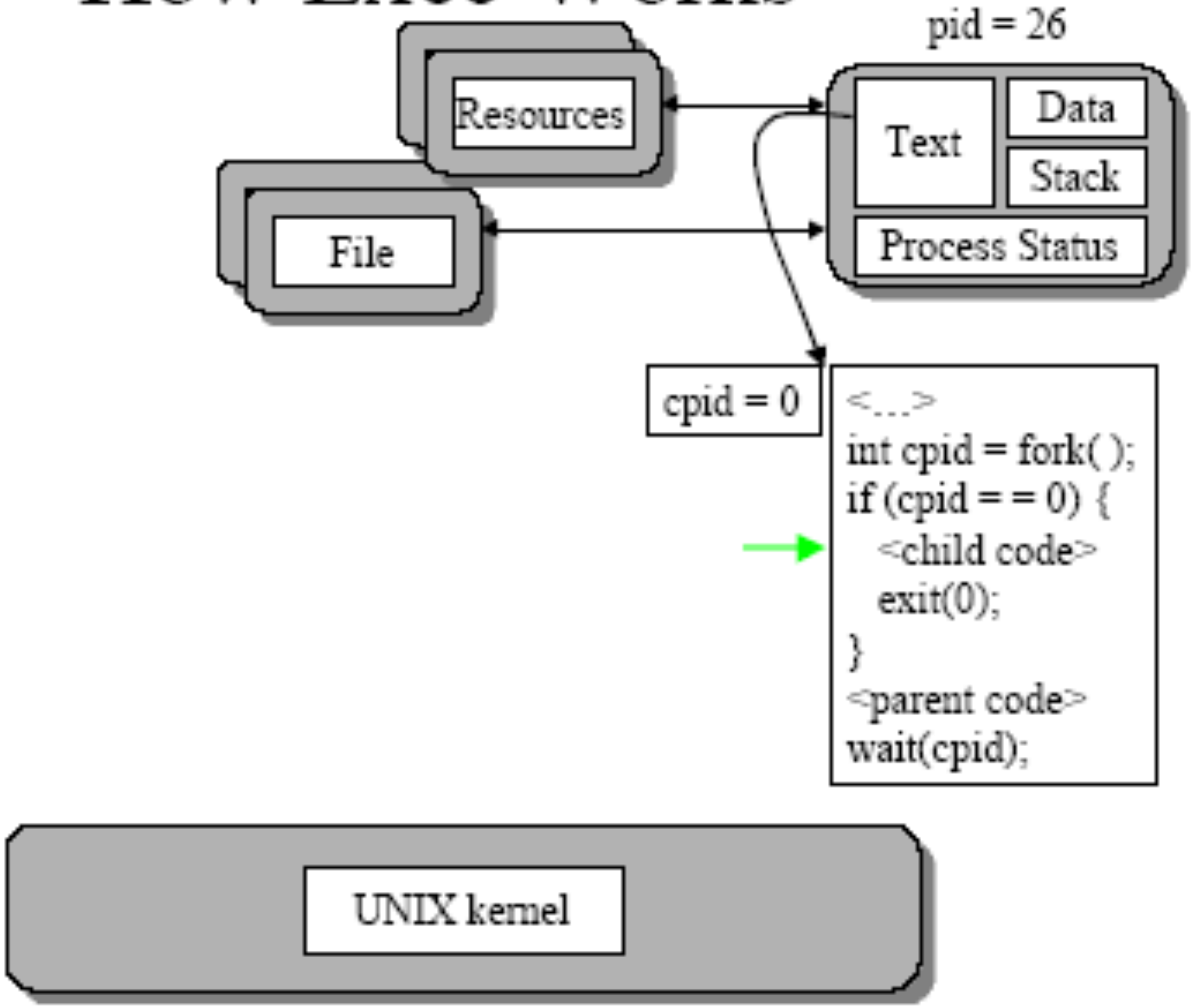
Using exec()

- In the parent process:

```
main()
  …
int pid = fork();         // create a child
if (pid == 0) {           // child continues here
  exec(“program”, argc, argv0, argv1, …);
}
else {
  …                       // parent continues here
}
```

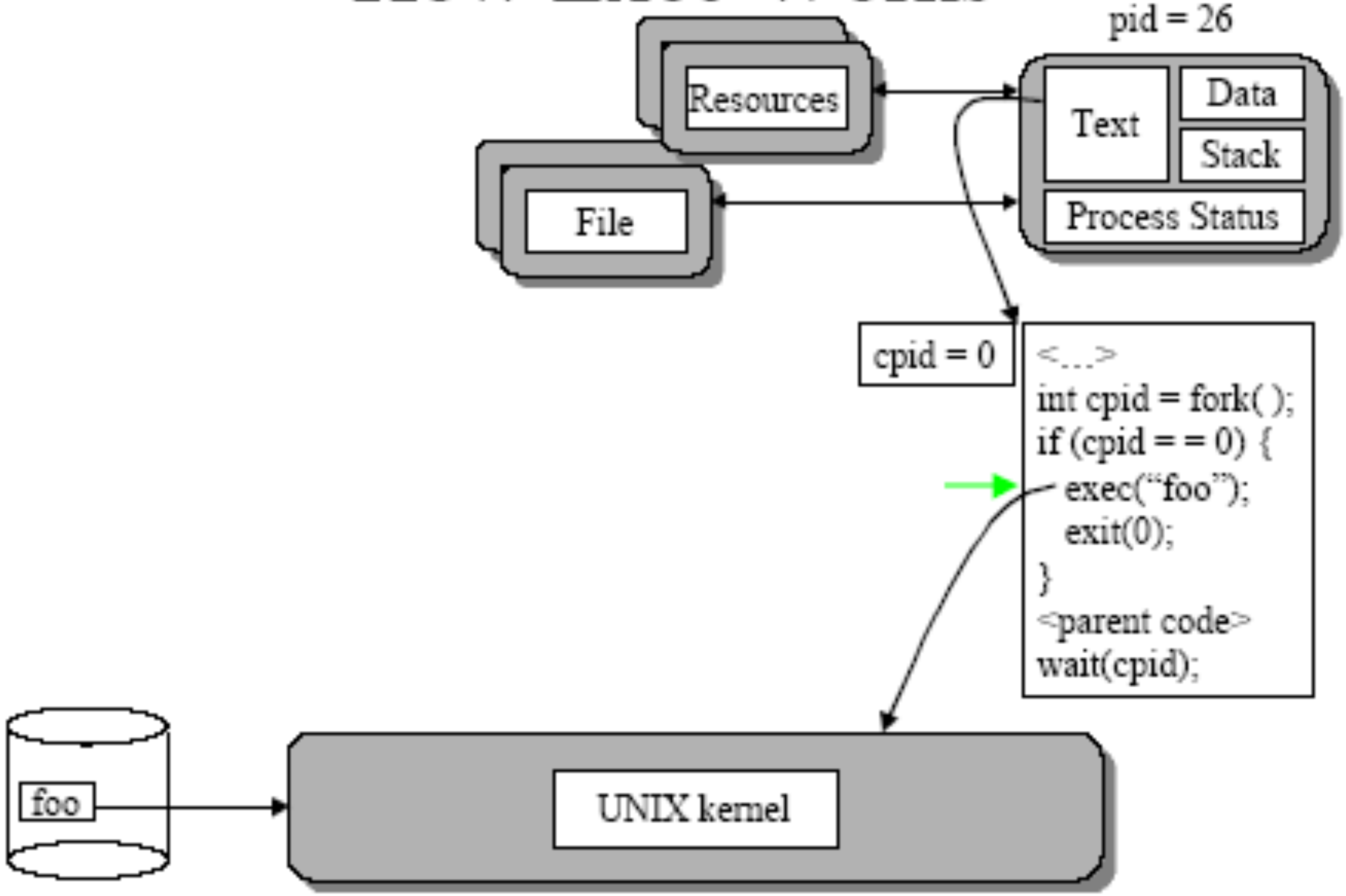
- In 99% of cases, we use exec() after fork().

How Exec Works



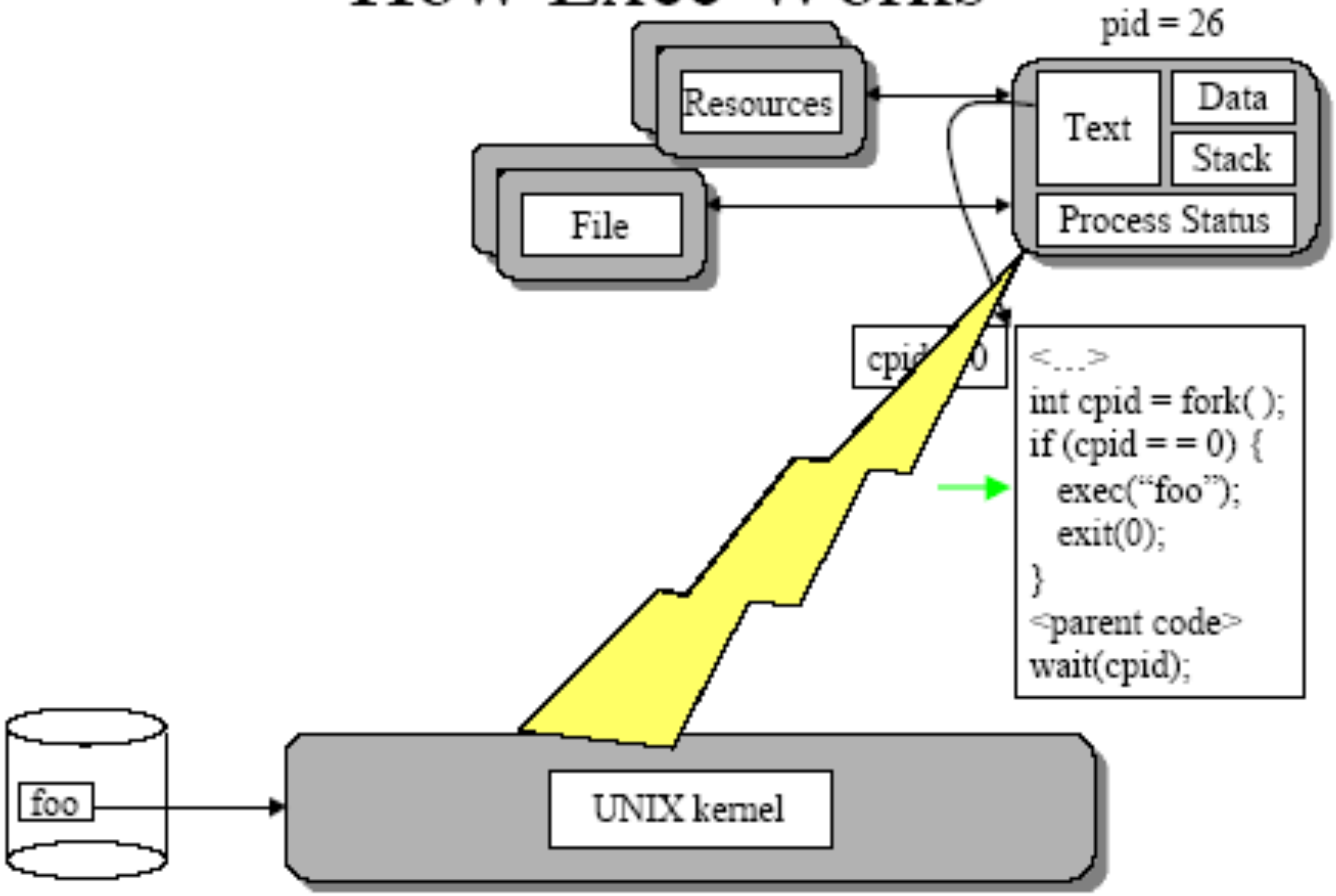
```
<…>
int cpid = fork( );
if (cpid = = 0) {
  <child code>
  exit(0);
}
<parent code>
wait(cpid);
```

How Exec Works



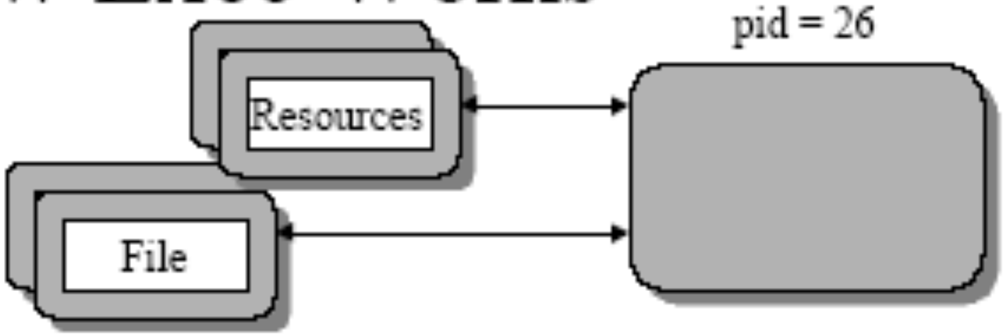
```
<...>
int cpid = fork( );
if (cpid = = 0) {
  exec("foo");
  exit(0);
}
<parent code>
wait(cpid);
```

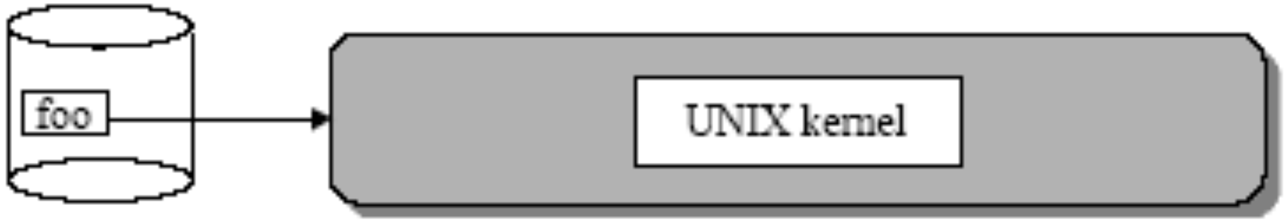
How Exec Works



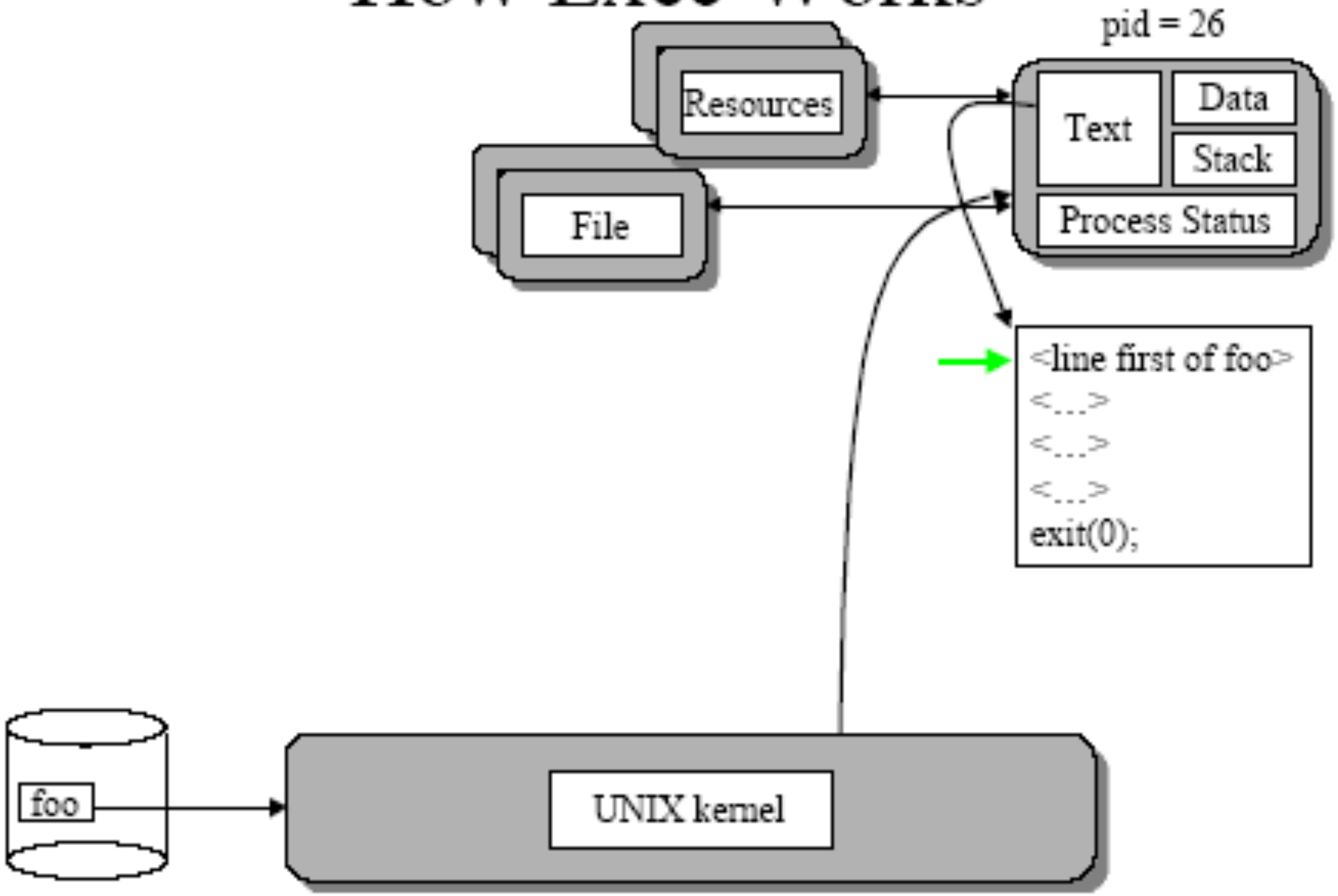
```
<…>
int cpid = fork( );
if (cpid = = 0) {
   exec("foo");
   exit(0);
}
<parent code>
wait(cpid);
```

How Exec Works

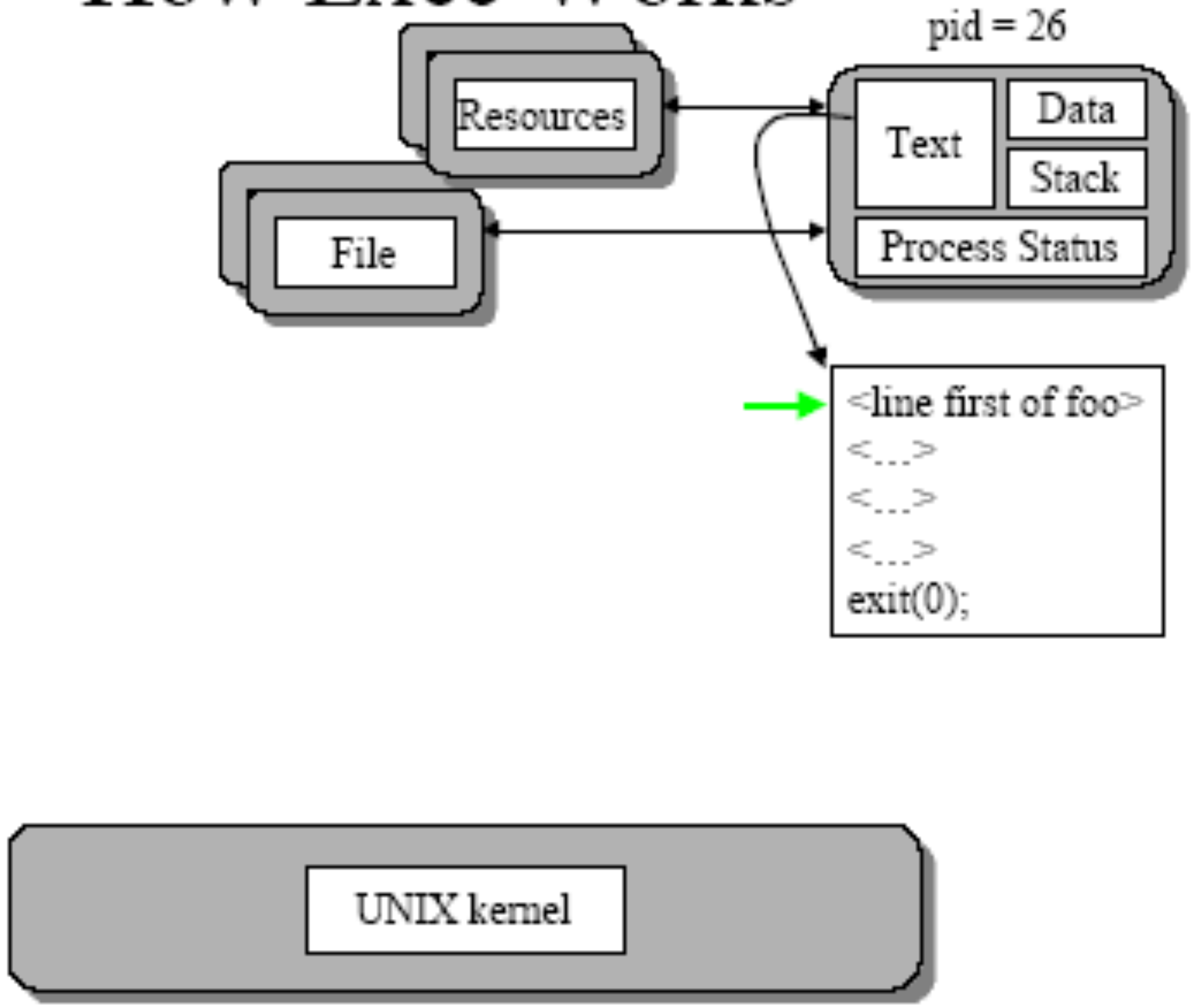




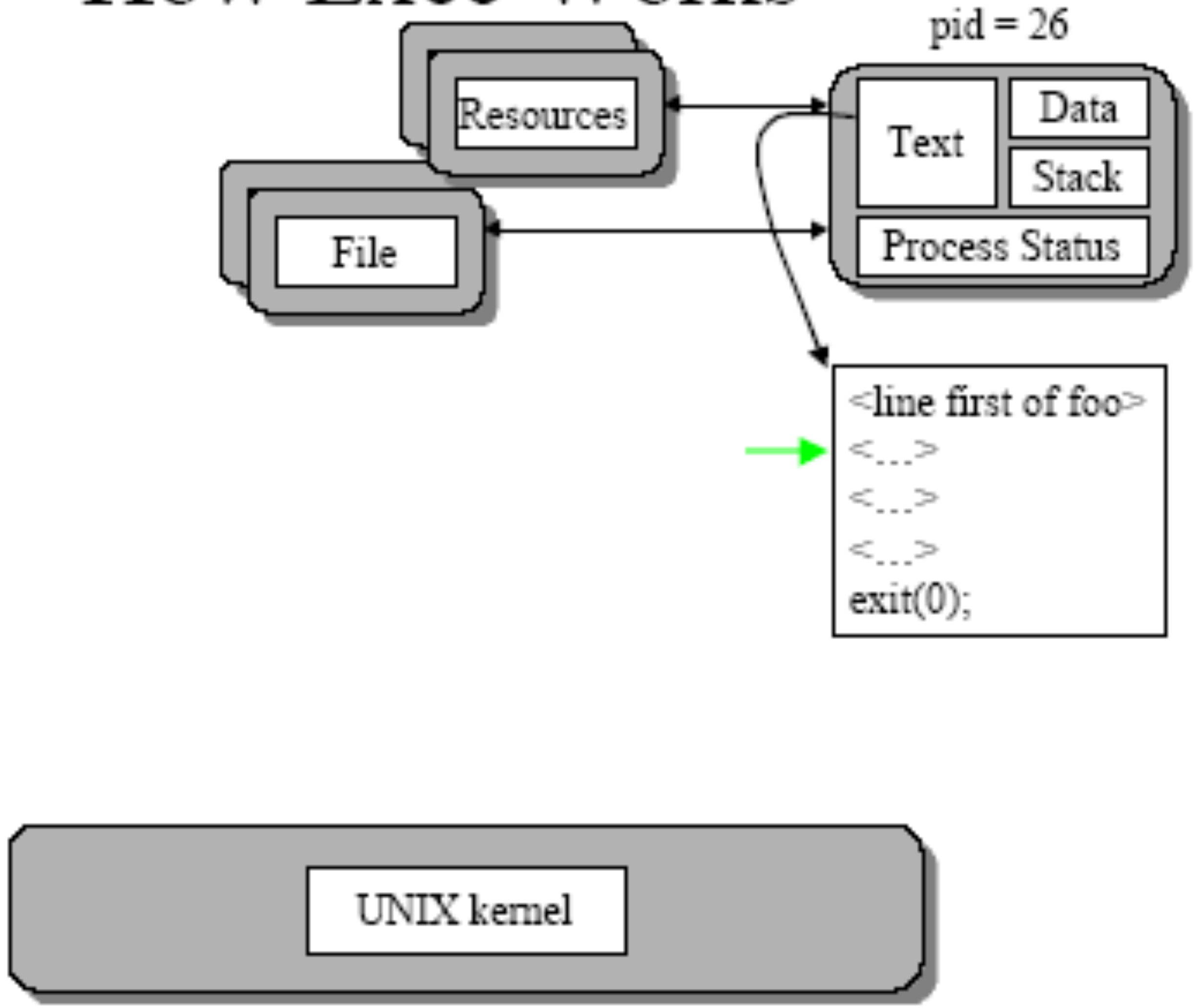
How Exec Works



How Exec Works



How Exec Works



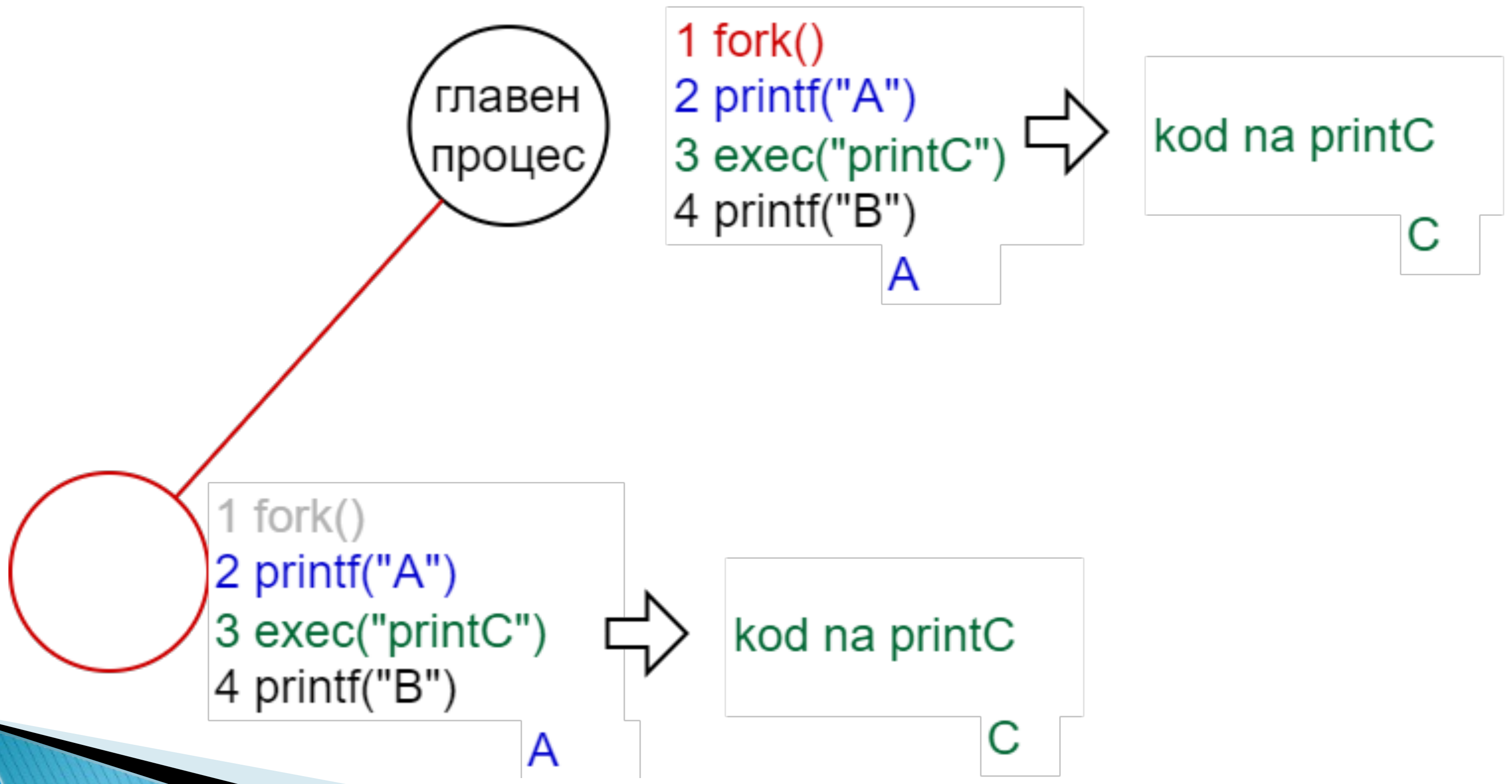
exec() example

- How many times will “B” be printed after the entire code finishes?
- Possible answers: 0, 1, 2 or 3

```
fork()
printf(“A”)
exec(“printC”)
printf(“B”)
```

exec() example

- "B" will not be printed at all.
- "A" and "C" will be printed twice, each.



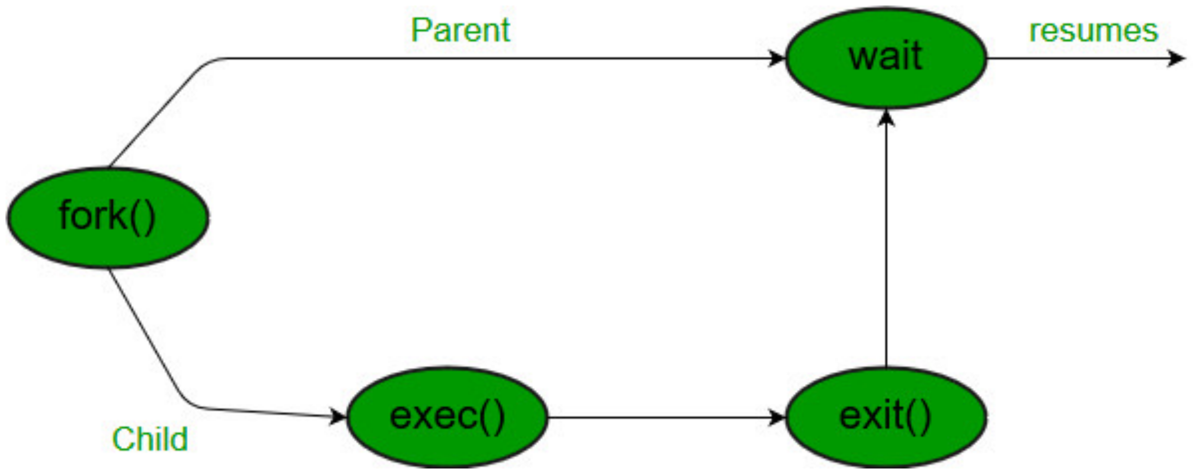
Normal Execution of exit()

- After the execution, the program makes a system call using exit().
  - Usually, exit(0) means a successful end of the program, while exit(integer) means an end due to some error;
- This system call:
  - takes the "result" that the program returns as an argument and places it in the PCB where the parent process can access it;
  - closes all open files, links, etc.;
  - de-allocates the memory;
  - de-allocates most of the OS structures used for the process, but not the PCB;
  - checks if the parent process is alive;

Normal Execution of exit()

- If the parent process is alive and has called wait() → the PCB is removed
- If the parent process is alive, but has not yet called wait() → the PCB record is kept in memory until the parent needs it
  - the child process keeps the result value, until the parent process asks for it;
  - the child process does not die, but enters a zombie / defunct status;
  - the child process is still present in the table of processes;
- If the parent process is not alive → the child process becomes an orphan
  - and is “adopted” by the init process;

wait() and exit()



Example Task #1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        // Child process
        execlp("/bin/ls", "ls", NULL);
        perror("exec failed");
        exit(EXIT_FAILURE);
    }
    else {
        // Parent process
        wait(NULL);
        printf("Child process
                     completed.\n");
    }
    return 0;
}
```

The parent process creates a child process which executes the ls (list) command, to list the content of the current directory.

Example Task #2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    pid_t pid1 = fork();
    if (pid1 < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    } else if (pid1 == 0) { // Child #1
        execlp("/bin/ls", "ls", "-l", NULL);
        perror("exec failed");
        exit(EXIT_FAILURE);
    } else { // Parent process
        wait(NULL);
        printf("First child process 
completed.\n");
        pid_t pid2 = fork();
        if (pid2 < 0) {
            perror("fork failed");
            exit(EXIT_FAILURE);
        } else if (pid2 == 0) { // Child #2
            execlp("/bin/ps", "ps", "-ef",
                                        NULL);
            perror("exec failed");
            exit(EXIT_FAILURE);
        } else { // Parent process
            wait(NULL);
            printf("Second child process 
completed.\n");
        }
    }
    return 0;
}
```

The parent process creates two child process, but the second one is created only after the first one finishes.

Example Task #3

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#define NUM_CHILDREN 2
#define MAX_DEPTH 3
void create_process_tree(int depth) {
    pid_t pid;
    int i;
    printf("Process ID: %d, Parent ID: %d,
Depth: %d\n", getpid(), getppid(), depth);
    if (depth >= MAX_DEPTH) {
        return;
    }
 for (i = 0; i < NUM_CHILDREN; i++) {
        pid = fork();
        if (pid < 0) {
            perror("Unsuccessful fork");
            exit(EXIT_FAILURE);
        } else if (pid == 0) {
            // Child process
            create_process_tree(depth + 1);
            exit(EXIT_SUCCESS); // Child exits
        }
    }
}
int main() {
    create_process_tree(0);
    // Wait for all child processes
    while (wait(NULL) > 0);
    printf("All child processes finished, now I
can exit as well...\n");
    return 0;
}
```

A more complex tree of processes.

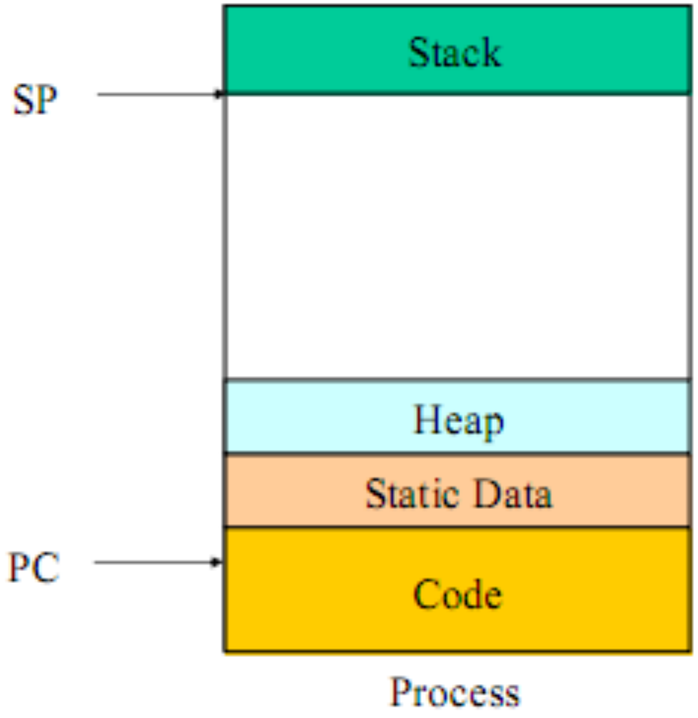
Process Features

- Processes have two features:
  - They own resources;
  - They have a thread of execution;

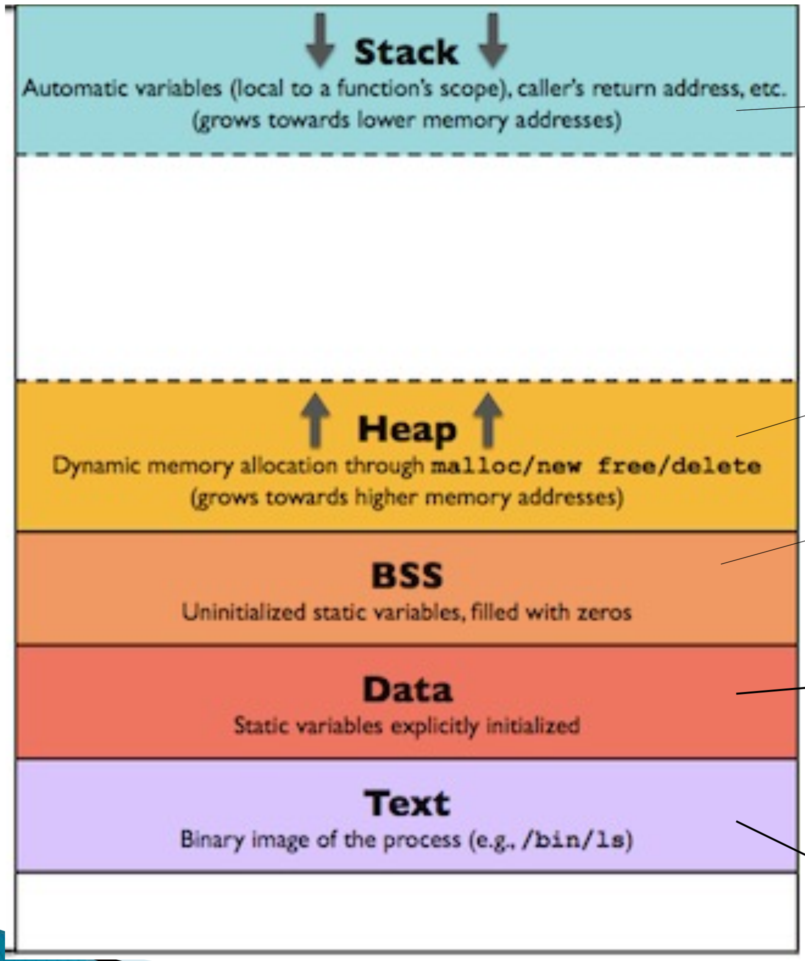
- These two features of processes are treated independently by the OS

Process

- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers
- Code
- Data
- Stack



Process in Memory



Stack
Automatic variables (local to a function's scope), caller's return address, etc.
(grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete
(grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)

Stack segment

Segment for dynamically created variables

BSS (Block Started by Symbol)

Initialized static variables

Code segment

Threads

- Parallelism in one process (Multithreading)
  - Multiple “threads of execution” in the same process;
  - Parallel execution in the same address space;
  - The process represents resource grouping;
- Cooperation between the threads (no system mechanisms for protection)
- The PCB is expanded to include information for each thread of a process
  - Threads have states, just as processes do;

Threads

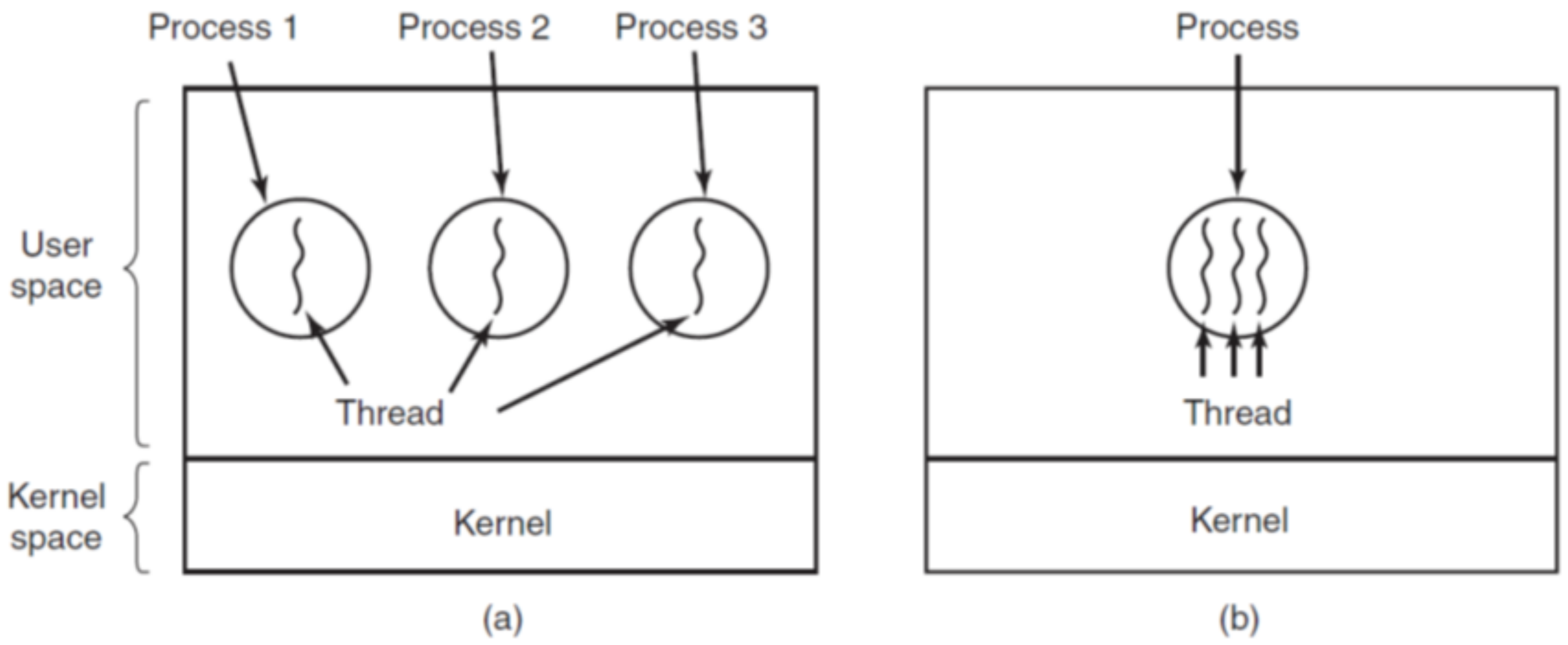


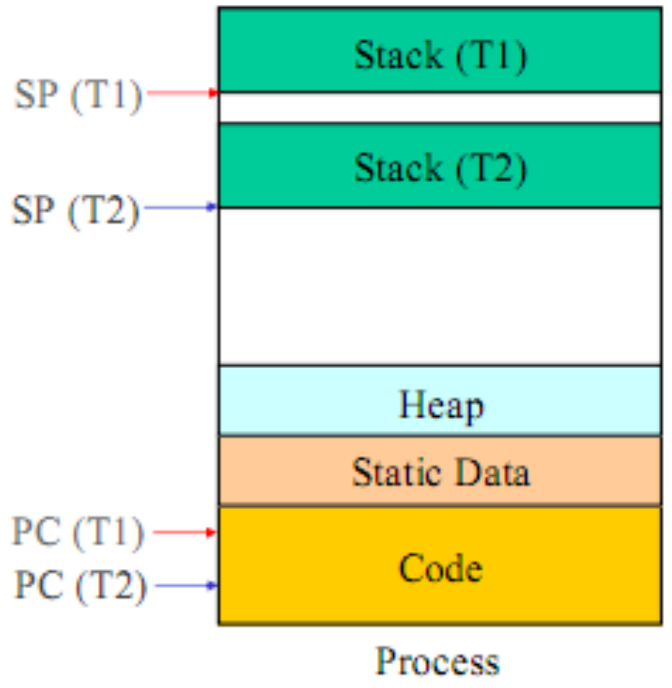
Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.

Threads

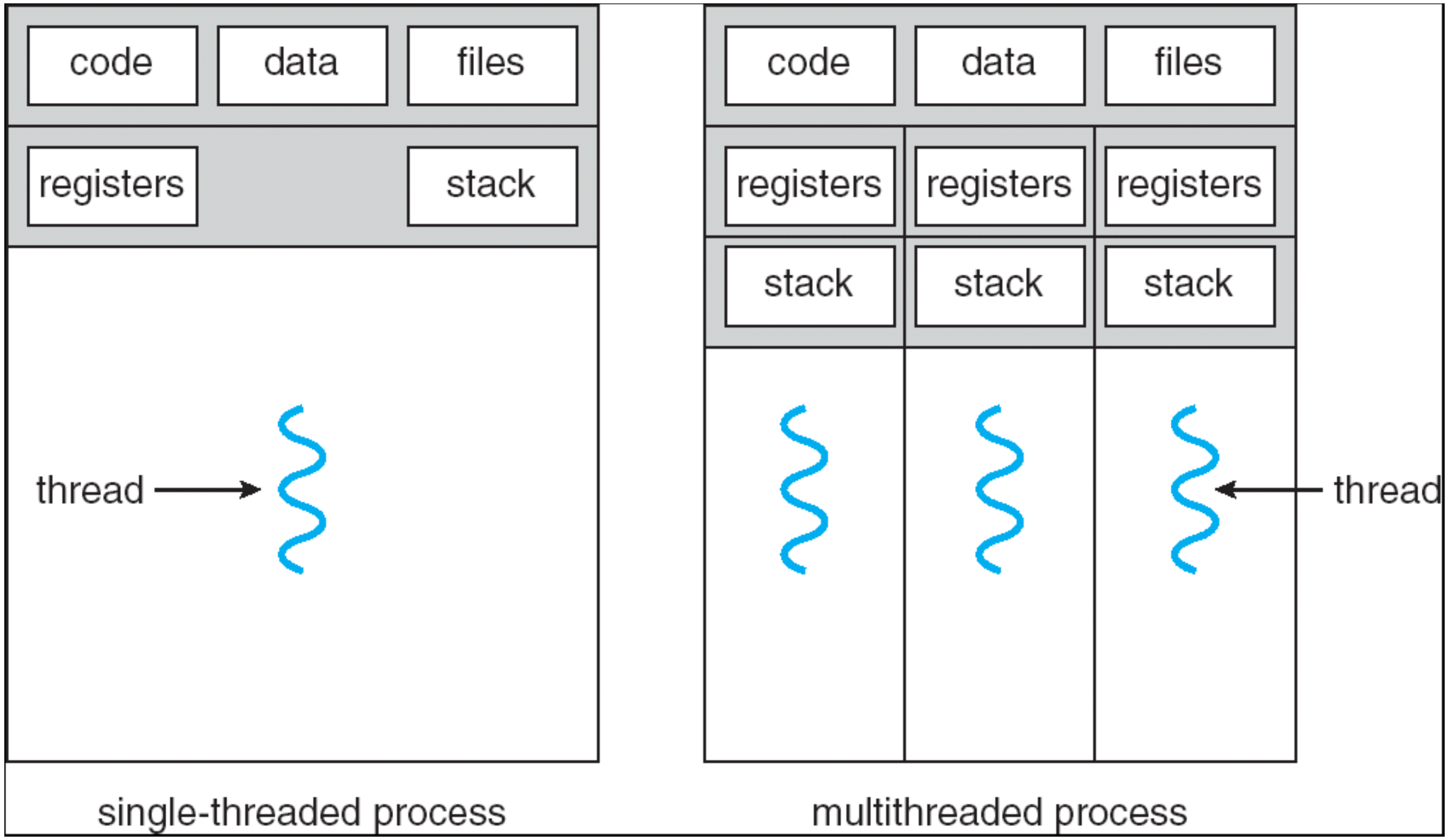
Per-process items	**Per-thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Process with Multiple Threads

- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers



Process with One and Multiple Threads



Multithreading

- It allows the OS to support multiple concurrent execution flows in the same process

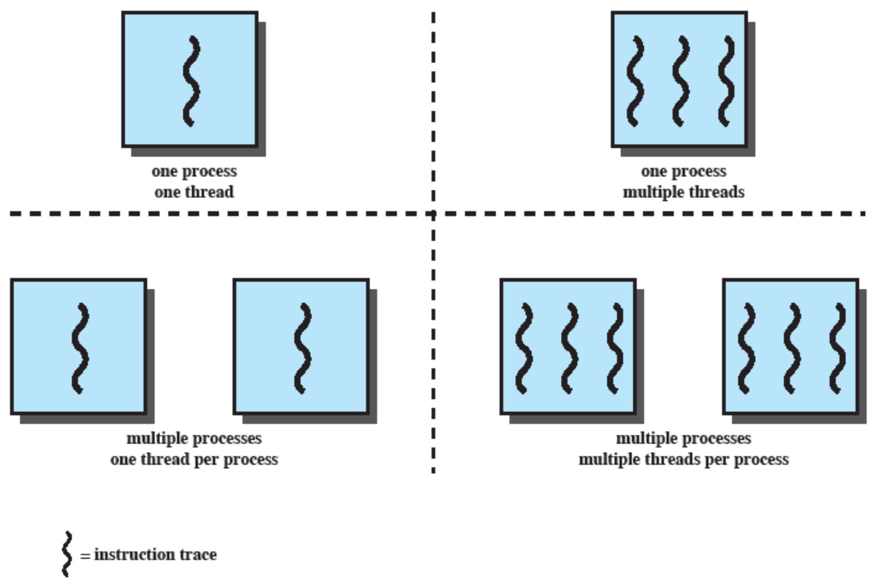


Figure 4.1 Threads and Processes [ANDE97]

Single-Threaded Approach

- MS-DOS supported one user process, with one thread
- Some old UNIX versions support multiple user processes, with only one thread per process

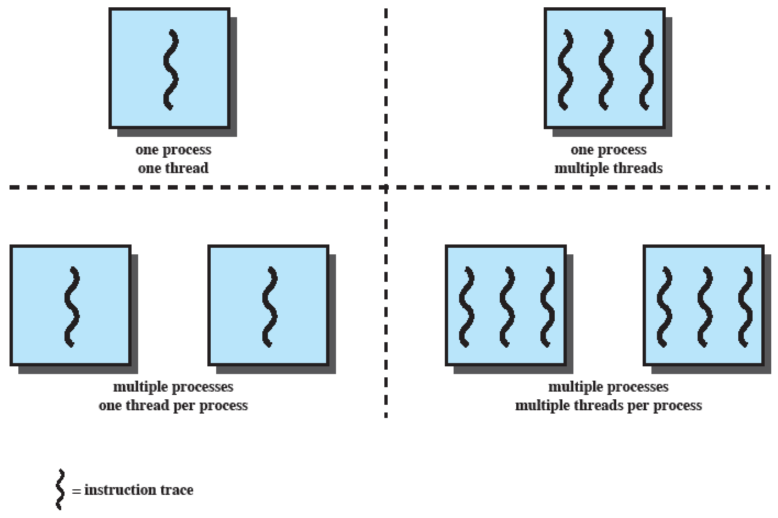


Figure 4.1 Threads and Processes [ANDE97]

Multi-Threaded Approach

- Java run-time environment supports one process, but multiple threads
- Multiple processes and threads are supported in Windows, Solaris, and modern versions of UNIX

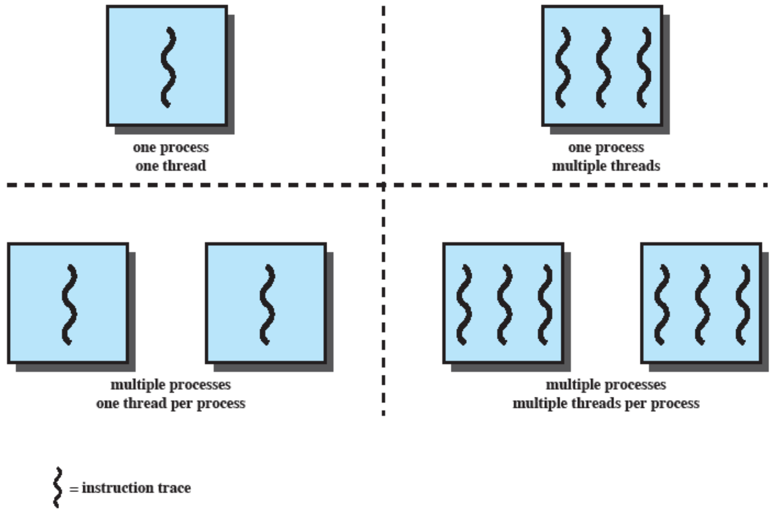


Figure 4.1 Threads and Processes [ANDE97]

Thread Advantages

- Simpler and faster creation and termination compared to processes.
- Thread switching is much faster than process switching.
- Threads communicate without the kernel.
- Application speedup: in situations with lots of processor activities and significant I/O at the same time.
- True parallelism in systems with multiple CPUs.

Example: Writing a Book

- If the user writes a book and they have to change some part of the text.
- If there is only one process (with one thread), then at the same time the system has to:
  - Follow user commands (from keyboard and mouse);
  - Reformat the text;
  - Save the new content on disk;

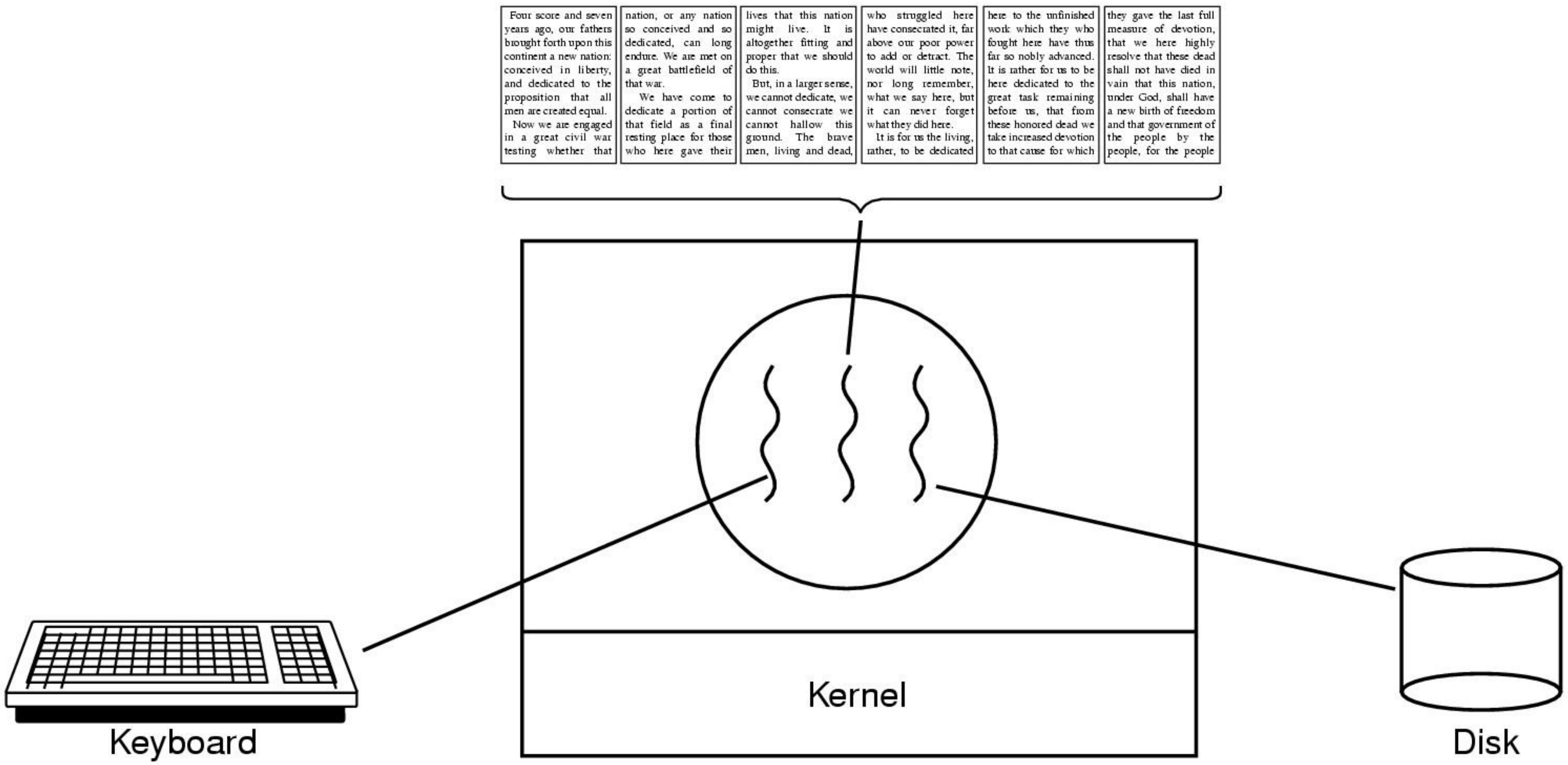
- Process solution: Complex program mode based on interrupts!

Example: Writing a Book

- Solution: A process with three threads:
  1. Used for user interaction;
  2. Used for text reformatting;
  3. Used for content saving;

- The three threads share the same memory, so they can access the working document.
- Note: An option with three processes will not work!

Example: Writing a Book (Threads)



Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.

We have come to dedicate a portion of that field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.

It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people

Thread vs. Process

- Threads share the address space of the process that created them; Processes have their own address space.
- Threads have direct access to the data segment of their process; Processes have their own copy of the data segment of the parent process.
- Threads can directly communicate with other threads from their process; Processes must use inter-process communication to communicate with sibling processes.
- Threads have almost no overhead; Processes have considerable overhead.
- There is no clock interrupt to actually enforce multiprogramming as there is with processes – Threads have to cooperate.

Thread vs. Process

- New threads are easily created; New processes require duplication of the parent process.
- Threads can exercise considerable control over threads of the same process; Processes can only exercise control over child processes.
- Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process; Changes to the parent process do not affect child processes.
- All threads in a program must run the same executable. A child process may run a different executable by calling an exec() function.

Thread vs. Process

- During a thread switch, the virtual memory space remains the same. It does not during a process switch.
- Both types involve handing control over to the OS kernel to perform the context switch. The process of switching in and out of the OS kernel along with the cost of switching out the registers, is the largest fixed cost of performing a context switch.
- In a process context switch, the processor's Translation Lookaside Buffer (TLB) and the cache get flushed, making memory accesses much more expensive for a while. This does not happen during a thread switch.

Questions?