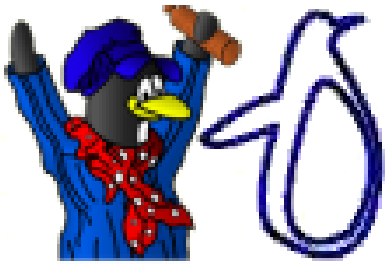
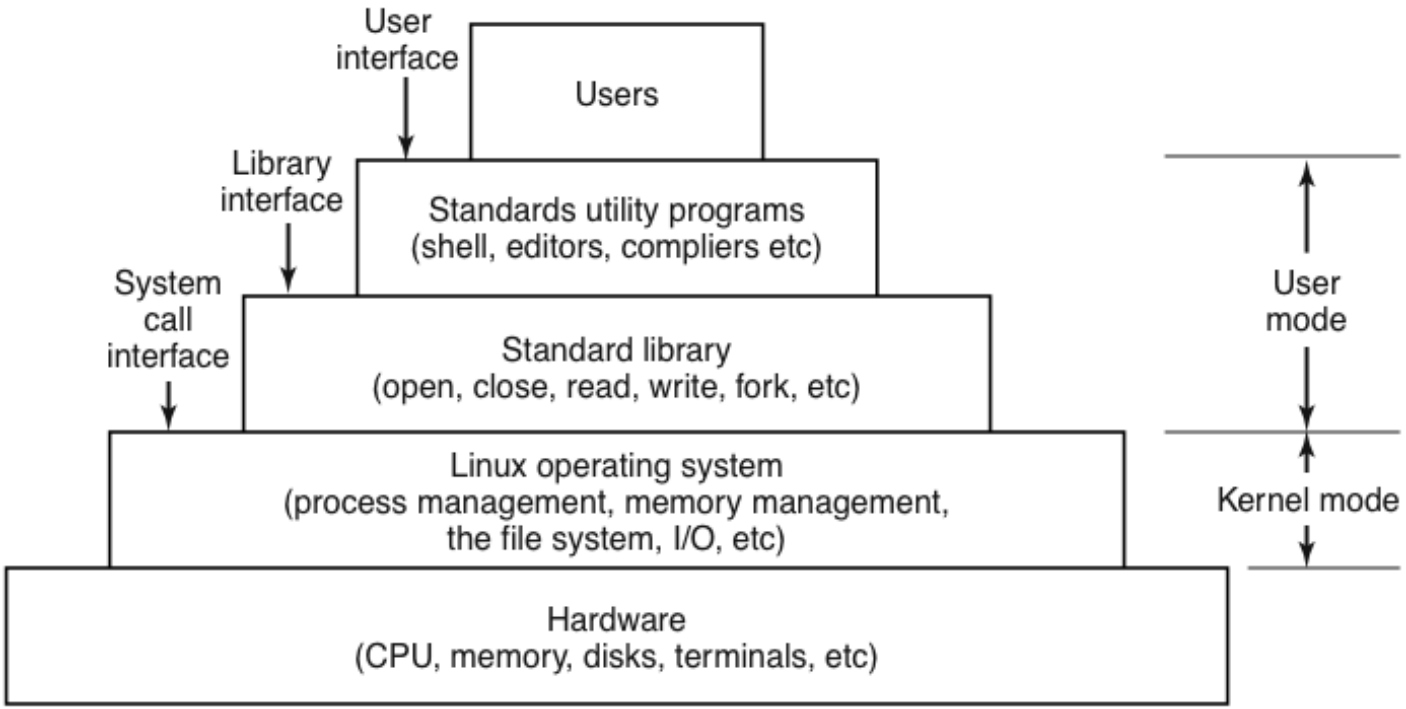
CS 153 Lab2

Kishore Kumar Pusukuri



Outline

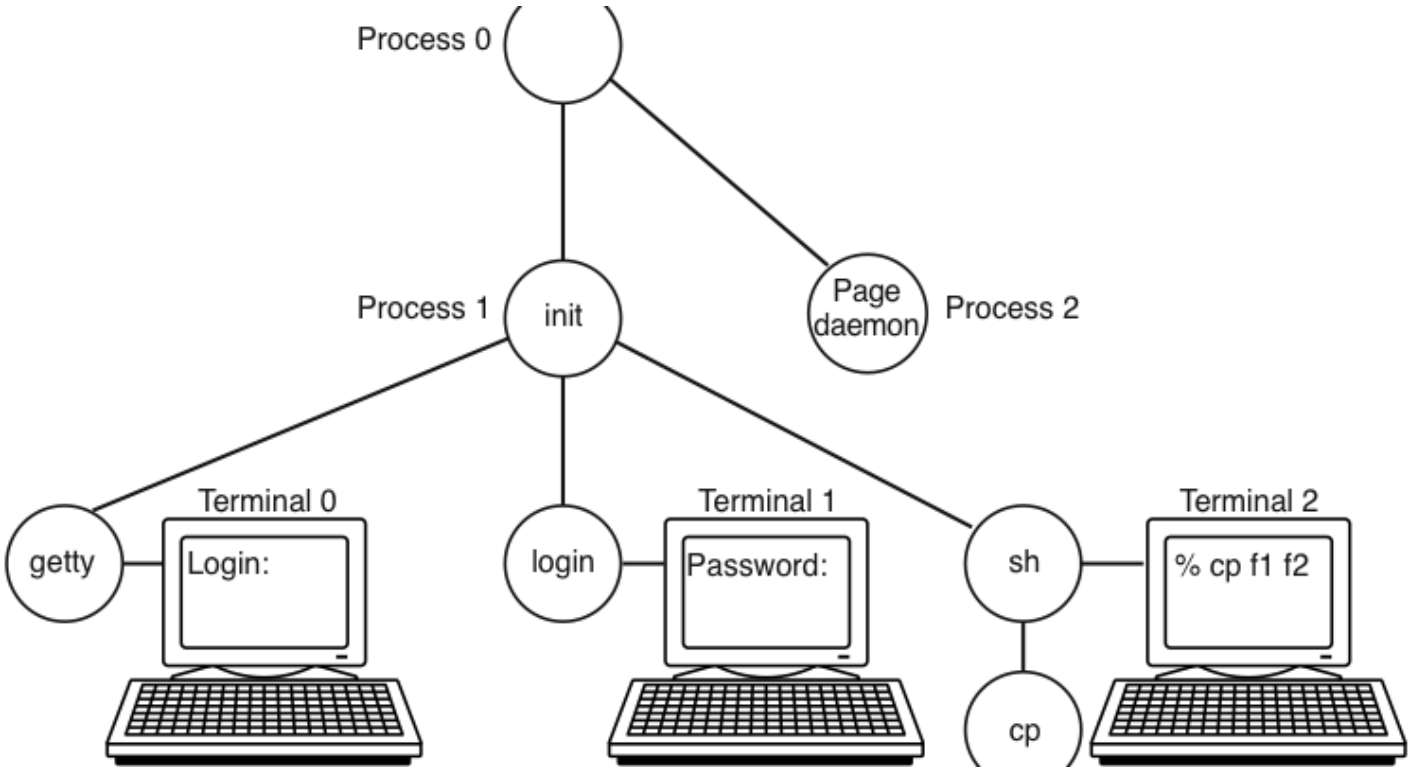
Operating System Structure
Program vs Process
Creating a process : fork()
Executing a program with exec()
How to implement a simple shell ?
Interprocess communication (IPC) - pipe


User interface
Users
Library interface
Standards utility programs (shell, editors, compliers etc)
System call interface
Standard library (open, close, read, write, fork, etc)
Linux operating system (process management, memory management, the file system, I/O, etc)
Hardware (CPU, memory, disks, terminals, etc)
User mode
Kernel mode

Program and Process

- A program is an executable file, and a process is an instance of the program in execution.
- Many processes can execute simultaneously on LINUX system.
- The main active entities in a Linux system are the processes.

Process is an active entity



Special Processes

- Every process has a unique process ID, a nonnegative integer.
- The process ID 0 is a scheduler process and is often known as the swapper. No program on disk corresponds to this - it is part of the kernel and is known as system_process. This is the only one process not created via fork().
- The process ID 1 is the init process and is invoked by the kernel at the end of the bootstrap process.
- The process ID 2 is the pagedaemon. This process is responsible for supporting the paging of virtual memory system. Like swapper, this is a system/kernel process.

init process

- The program file for this process was /sbin/init.
- This process is responsible for bringing up a Linux/Unix system after the kernel has been boot strapped.
- init usually reads the system-dependent initialization files (the /etc/rc* files) and brings the system to a certain state (such as multiuser).
- The init process never dies.
- It is a normal user process (not like swapper and pagedaemon) but run with supervisor/root privileges.
- Later we will see how init becomes the parent process of any orphaned child process.

Daemons

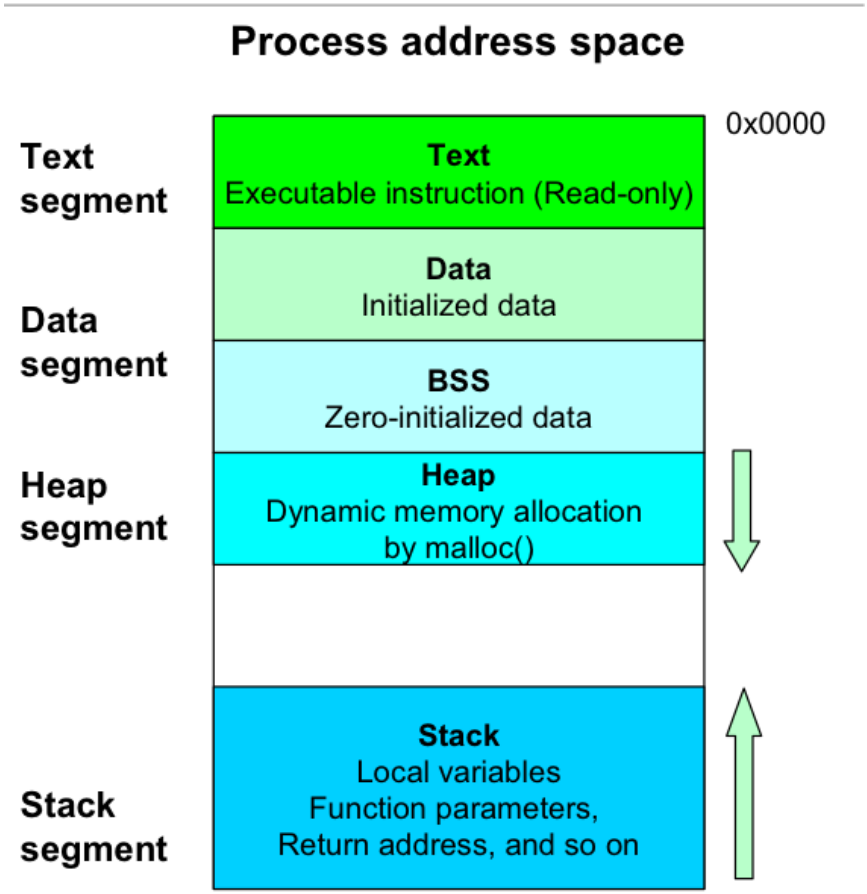
- On a large system there may be hundreds or even thousands of processes running.
- Even when the user absent, several processes, called "daemons", are running. These are started by a shell script when the system is booted.
- A typical daemon is the cron daemon. It wakes up at specified time and checks if there is any work to do. If do it does the work. Then it goes back to sleep until it is time for the next check.
- The daemon is needed because it is possible in Linux to schedule activities minutes, hours, days, or even months in the future.

Process context

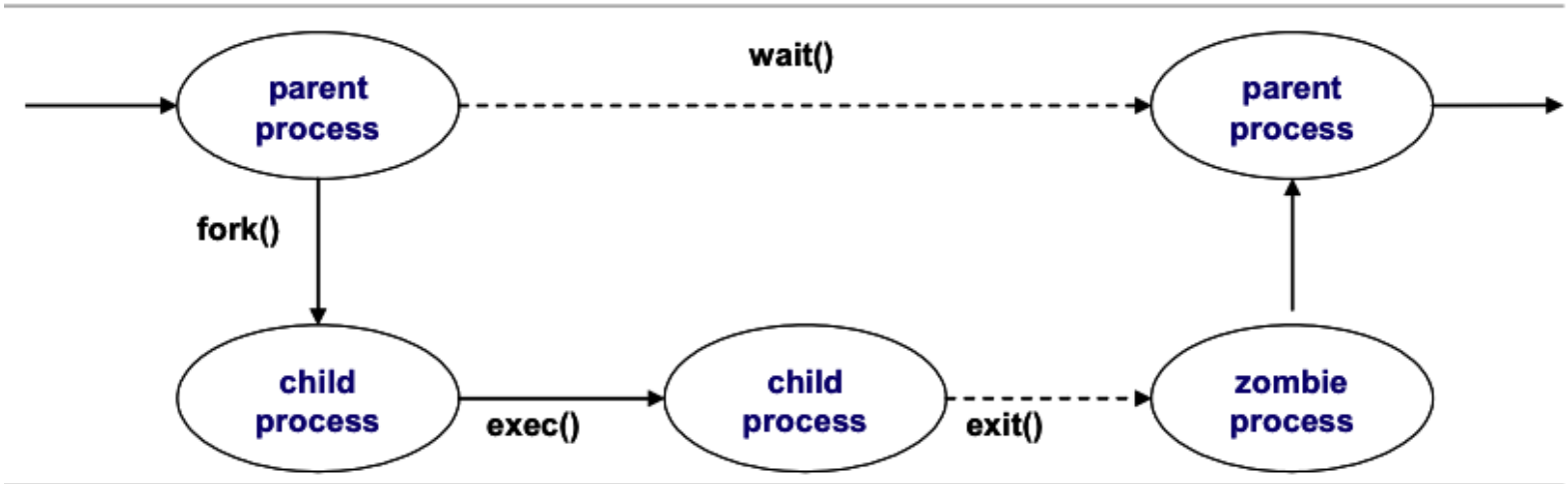
The context of a process consists of its (user) address space and the contents of hardware registers and kernel data structures that relate to the process.

- The program counter (PC) specifies the address of the next instruction the CPU will execute.
- The processor status register (PS) specifies the hardware status of the machine.
- Stack pointer.
- General purpose registers contain data generated by the process during its execution.

Process address space



Process life-cycle



The process that invokes fork is called parent process, and the newly created process is called child process.

cont...

The kernel does the following sequence of operations for fork.

- It allocates a slot in the process table for the new process.
- It assigns an unique ID number to the child process.
- It makes a logical copy of the context of the parent process.
- It returns child process ID to the parent process, and 0 to the child process.
- Some operations with files. We will study later.

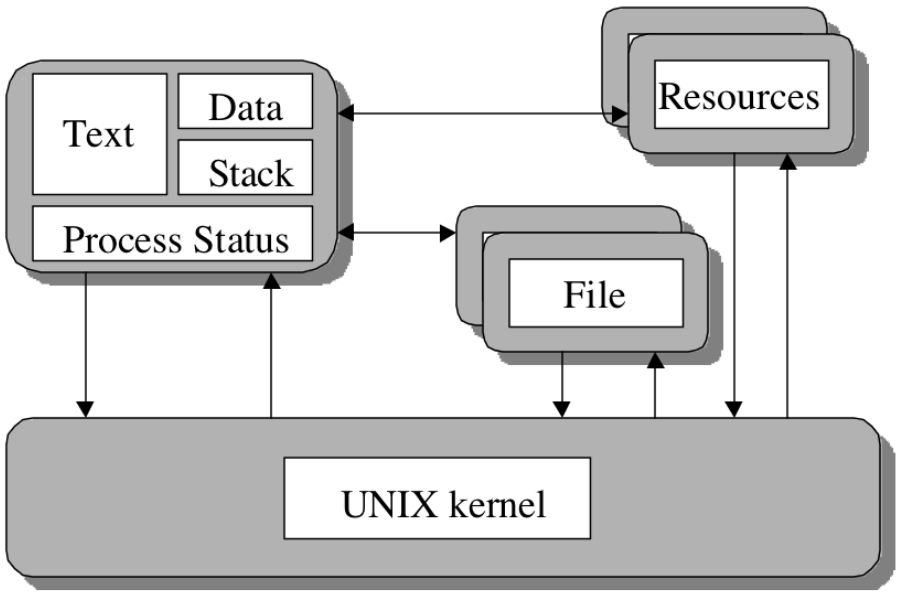
```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

Returns:   0 in child,
           Process ID of a child in parent,
           -1 on error.
In addition to the process ID, there are other identifiers
for every process. The following functions return these
identifiers.
pid_t getpid(void);   process ID of calling process.
pid_t getppid(void);  parent process ID of calling process
Note: None of these has an error return.
```

fork1.c

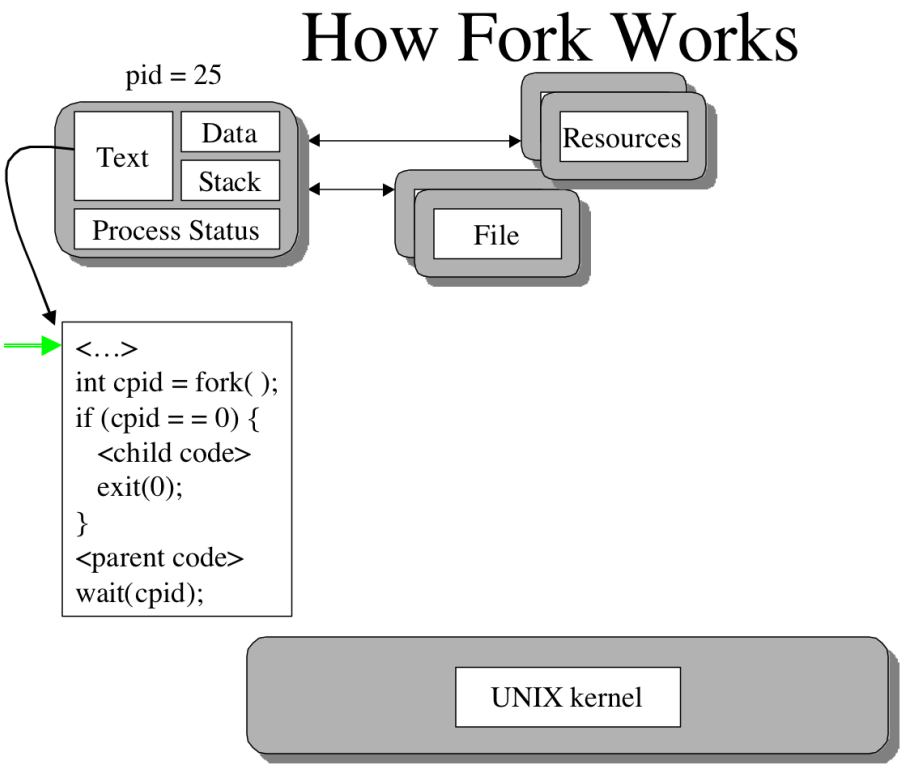
Study the program fork1.c to understand how fork works.

cont..

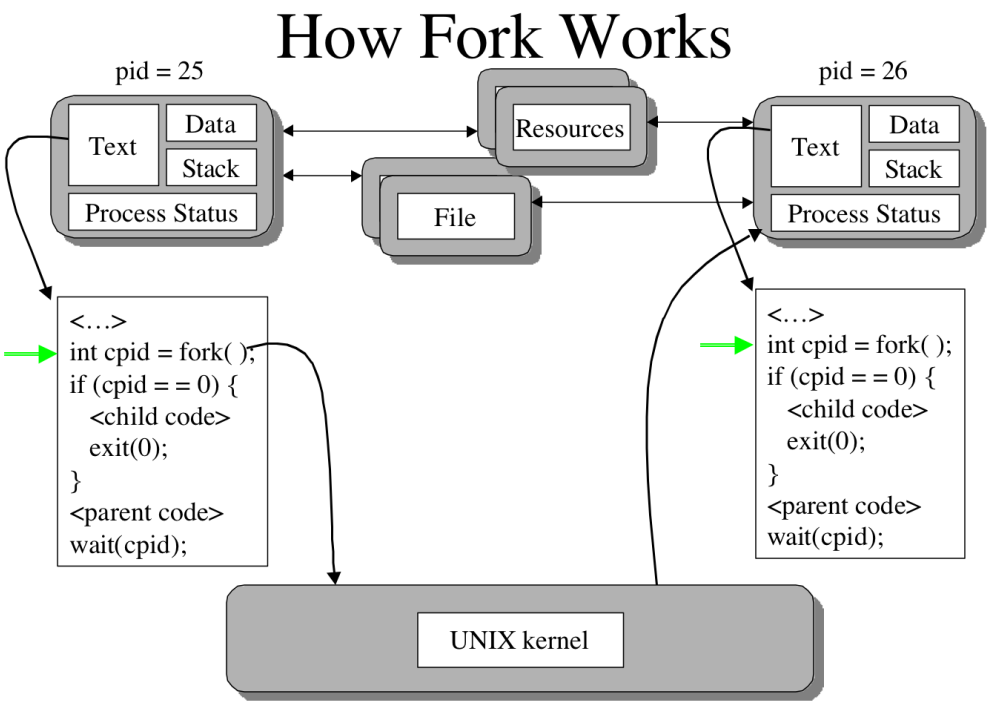
A Unix Process



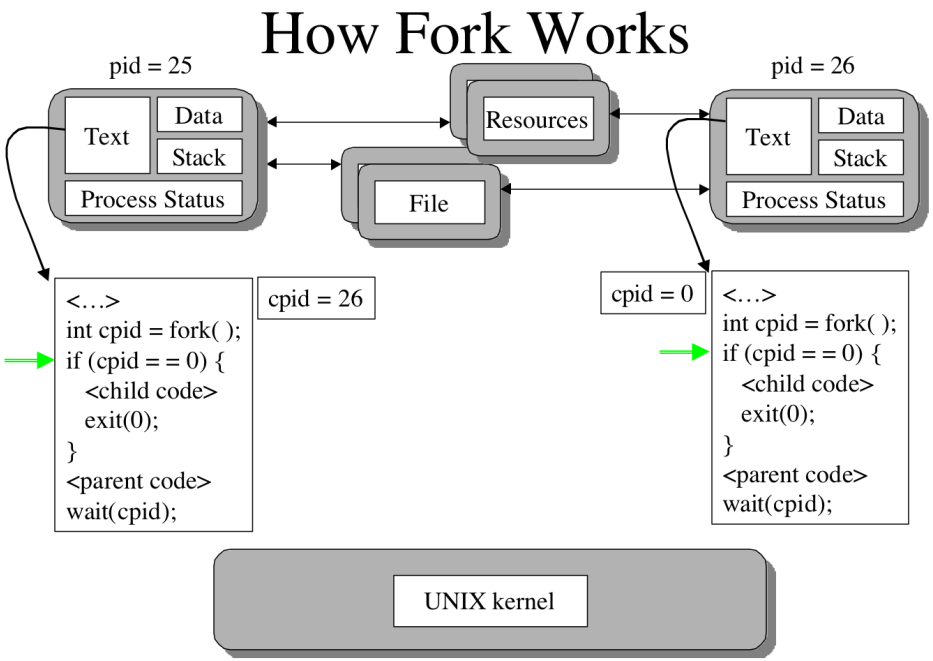
cont..


How Fork Works
pid = 25
Text
Data
Stack
Process Status
Resources
File
<...>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
UNIX kernel

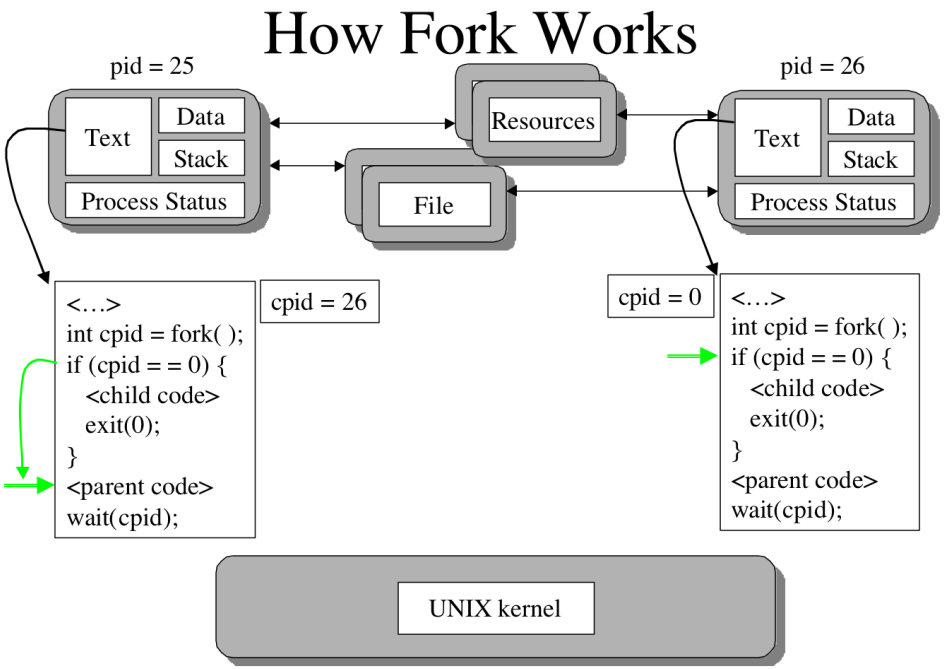
cont..


How Fork Works
pid = 25
pid = 26
Text
Data
Stack
Process Status
Resources
File
Text
Data
Stack
Process Status
<...>
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);
UNIX kernel

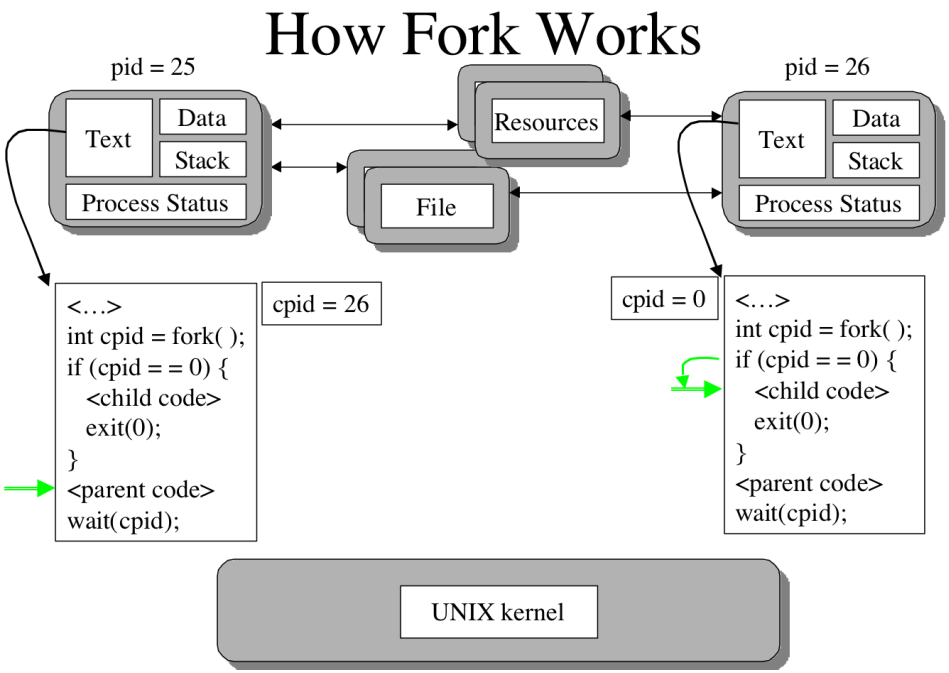
cont..


How Fork Works
pid = 25
Text
Data
Stack
Process Status
Resources
File
pid = 26
Text
Data
Stack
Process Status
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
cpid = 26
cpid = 0
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
UNIX kernel

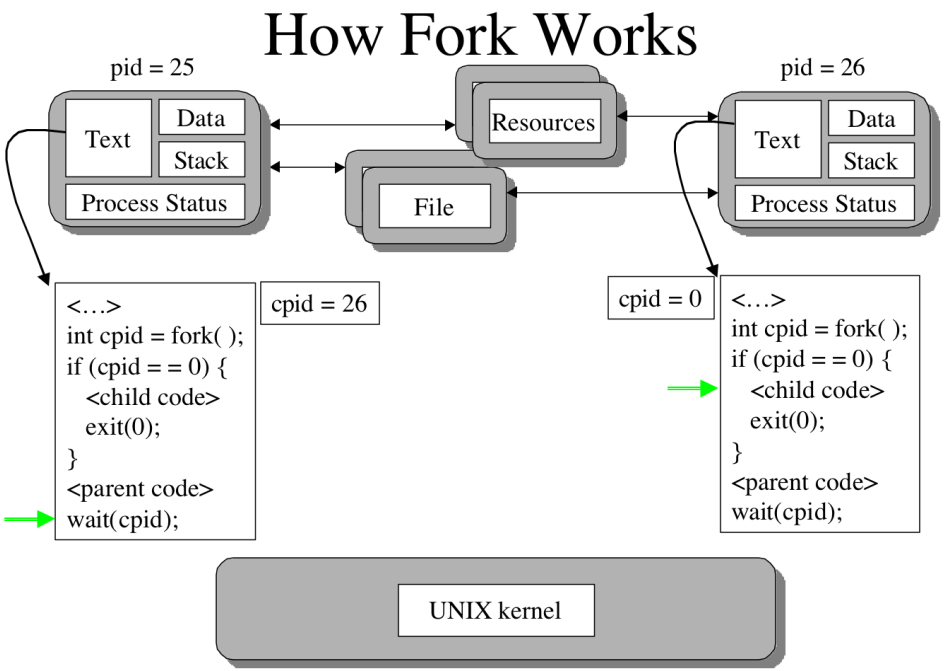
cont..


How Fork Works
pid = 25
pid = 26
Text
Data
Stack
Process Status
Resources
File
Text
Data
Stack
Process Status
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
cpid = 26
cpid = 0
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
UNIX kernel

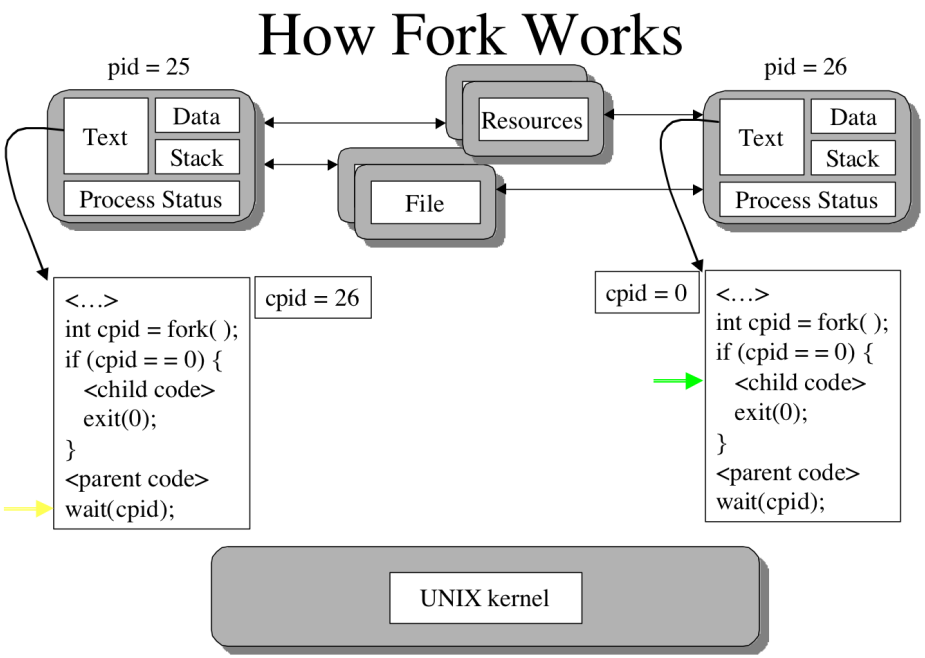
cont..


How Fork Works
pid = 25
Text
Data
Stack
Process Status
Resources
File
pid = 26
Text
Data
Stack
Process Status
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
cpid = 26
cpid = 0
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
UNIX kernel

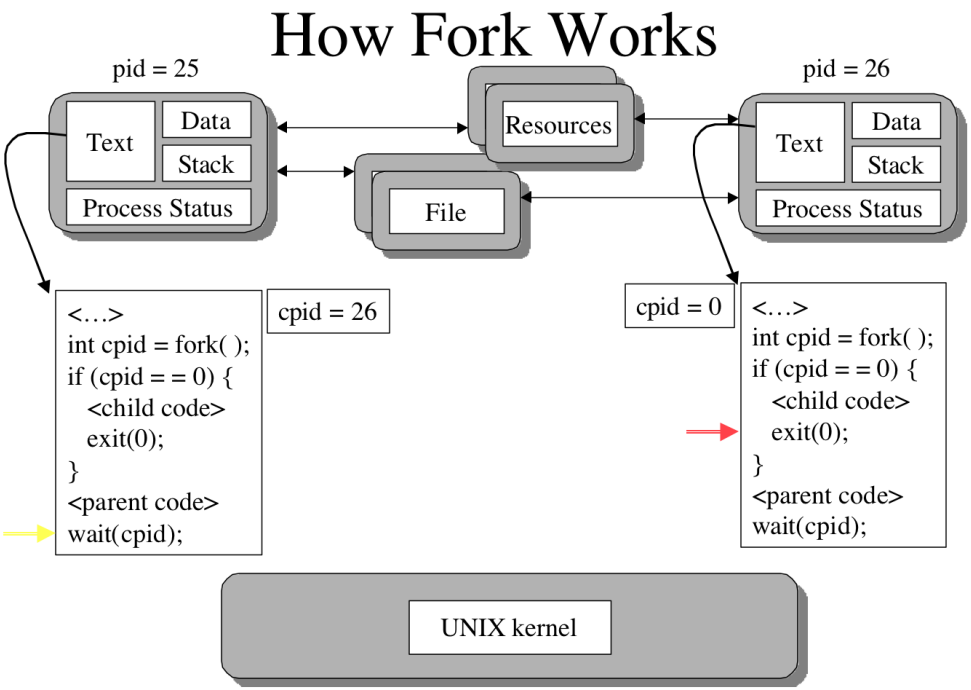
cont..


How Fork Works
pid = 25
Text
Data
Stack
Process Status
Resources
File
pid = 26
Text
Data
Stack
Process Status
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
cpid = 26
cpid = 0
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
UNIX kernel

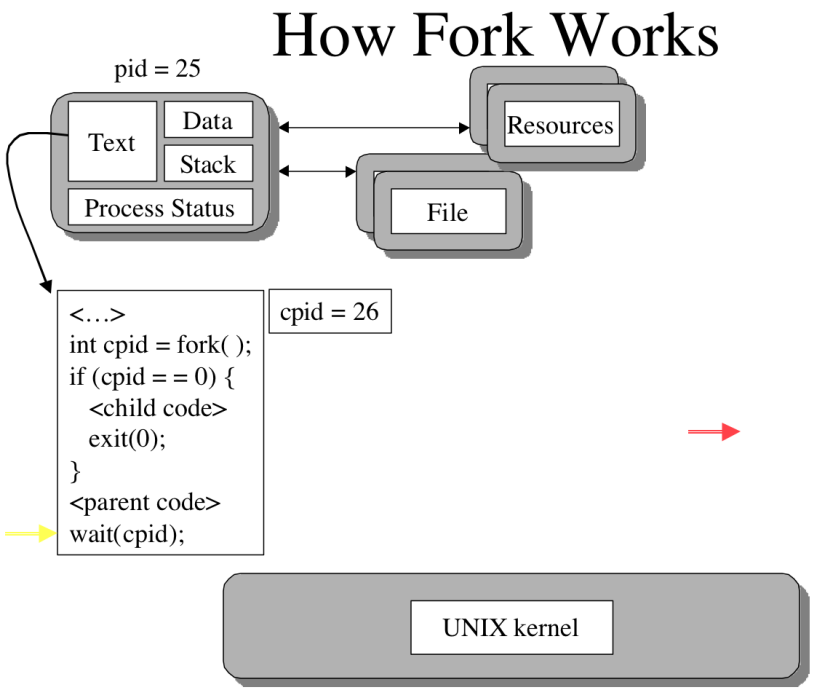
cont..


How Fork Works
pid = 25
Text
Data
Stack
Process Status
Resources
File
pid = 26
Text
Data
Stack
Process Status
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
cpid = 26
cpid = 0
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
UNIX kernel

cont..


How Fork Works
pid = 25
pid = 26
Text
Data
Stack
Process Status
Resources
File
Text
Data
Stack
Process Status
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
cpid = 26
cpid = 0
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
UNIX kernel

cont..


How Fork Works
pid = 25
Text
Data
Stack
Process Status
Resources
File
<…>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
cpid = 26
UNIX kernel

What are the differences between parent and child ?

Differences between parent and child

- The return value from fork().
- The process IDs are different.
- The two processes have different parent IDs.
- File locks set by parent are not inherited by the child.
- and .....some more differences

Can you tell me some uses of fork() ?
Think about some applications you are using in daily life...

Where you are using fork ?

- Network servers ?
- Shell

There are six exec functions, but we will often just refer to ‘‘the exec function", which means we could use any of the six different functions.

```
#include <unistd.h>

int execl (const char *pathname, const char * arg0,
 ... /* (char *) 0 */);
int execlp(const char *filename, const char *arg0,
... /* (char *) 0 */);
int execv (const char *pathname, char * const argv[]);
execve(...), execle(..),  execvp(....) ..
```

All six return: -1 on error, no return on success.

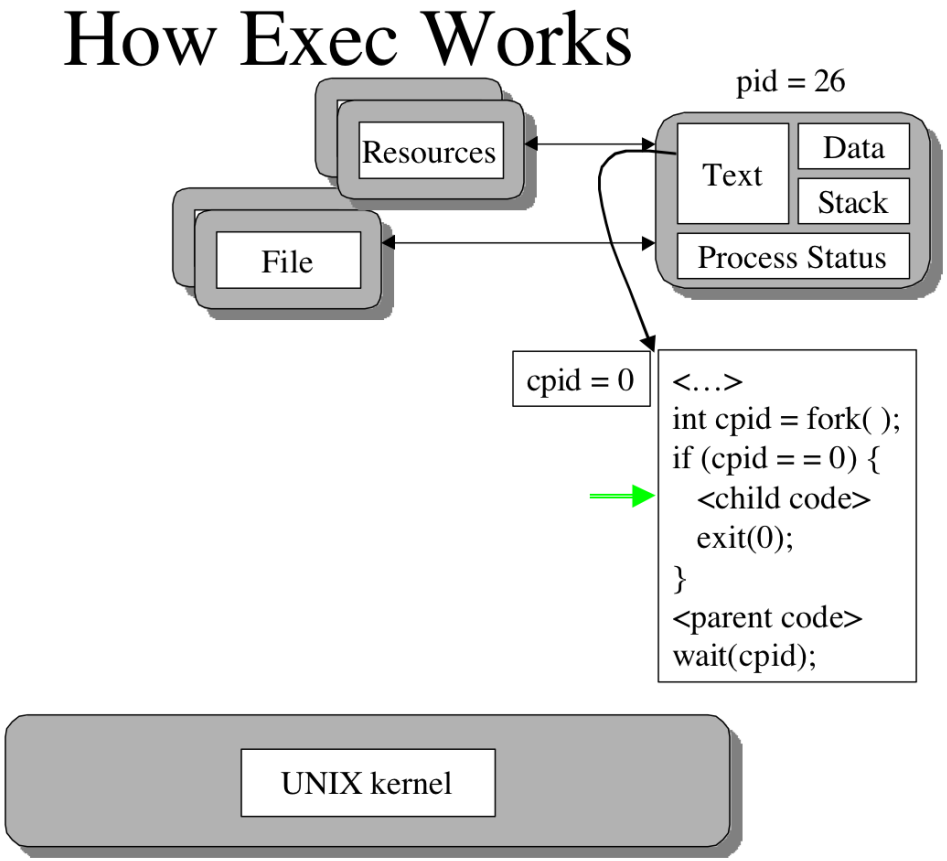
exec()

- fork() is to create a new process (the child) that then causes another program to be executed by calling one of exec functions.
- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID does not change across an exec because a new process is not created.
- exec merely replaces the current process (its text, data, heap and stack segments) with a brand new program.

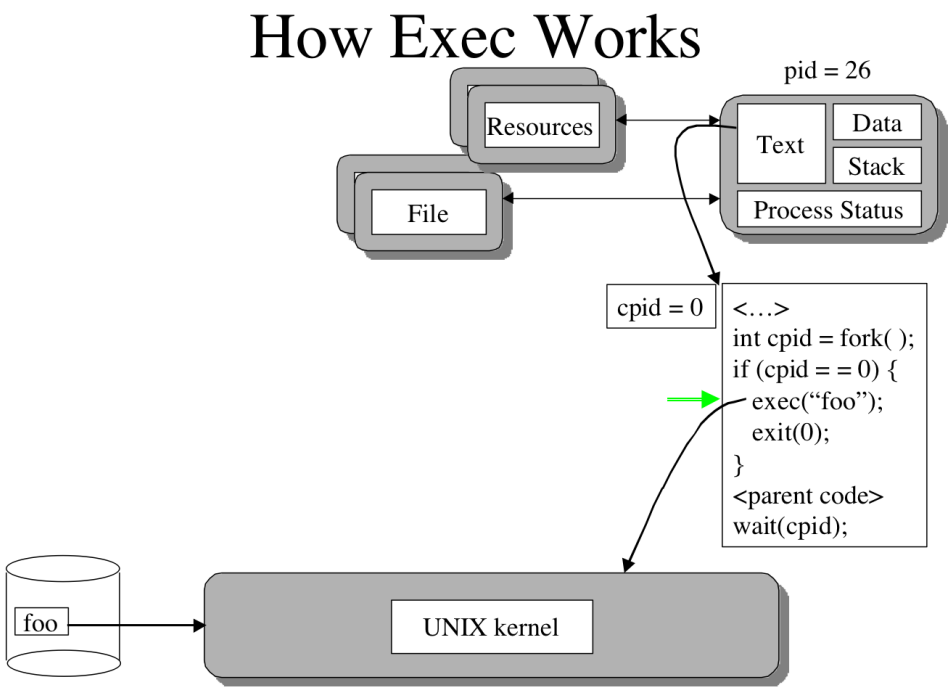
exec1.c and create.c

Study the programs exec1.c and create.c to understand how exec works.

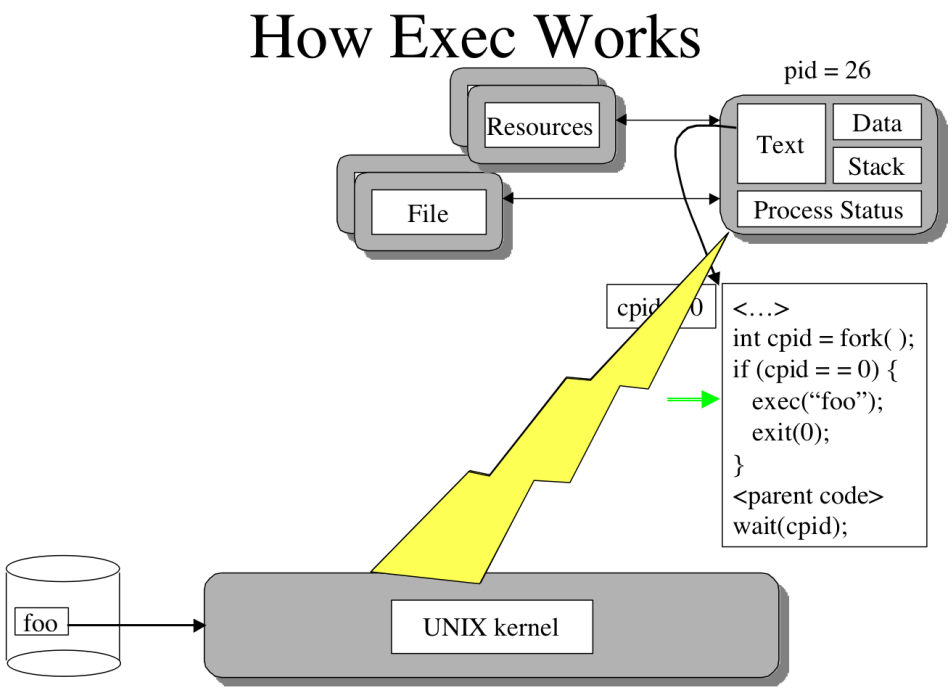
cont..


How Exec Works
pid = 26
Resources
File
Text
Data
Stack
Process Status
cpid = 0
<...>
int cpid = fork( );
if (cpid = = 0) {
<child code>
exit(0);
}
<parent code>
wait(cpid);
UNIX kernel

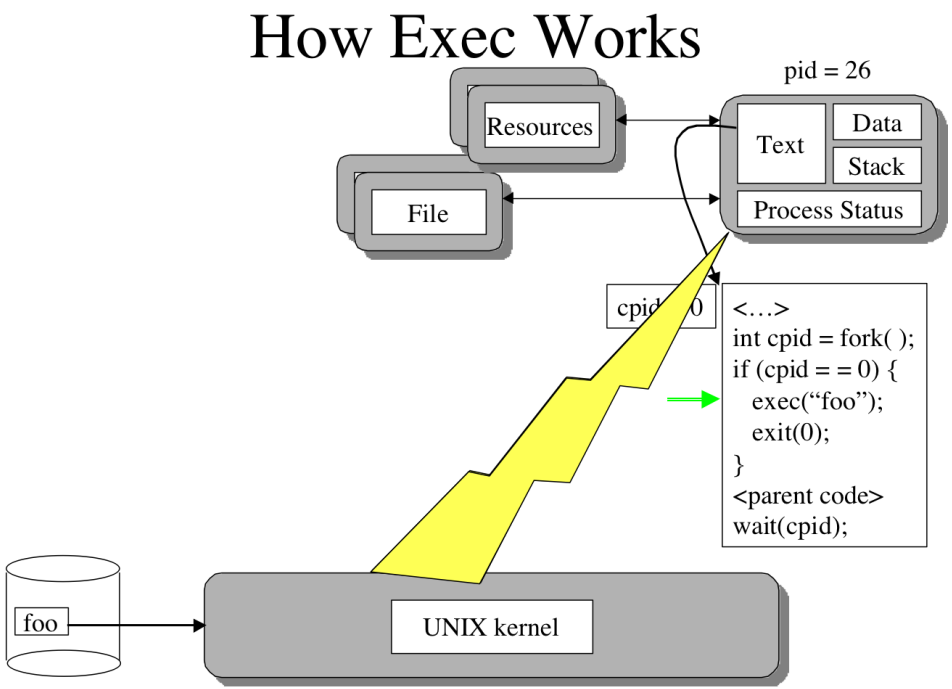
cont..


How Exec Works
pid = 26
Resources
File
Text
Data
Stack
Process Status
cpid = 0
<…>
int cpid = fork( );
if (cpid = = 0) {
exec("foo");
exit(0);
}
<parent code>
wait(cpid);
foo
UNIX kernel

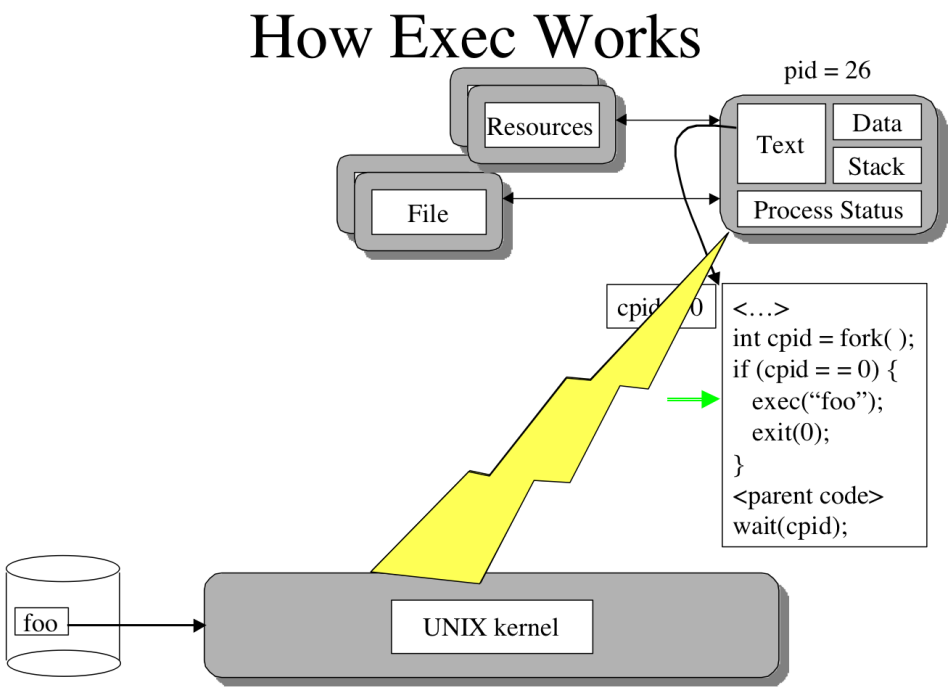
cont..


How Exec Works
pid = 26
Resources
File
Text
Data
Stack
Process Status
cpid = 0
<…>
int cpid = fork( );
if (cpid = = 0) {
exec("foo");
exit(0);
}
<parent code>
wait(cpid);
foo
UNIX kernel

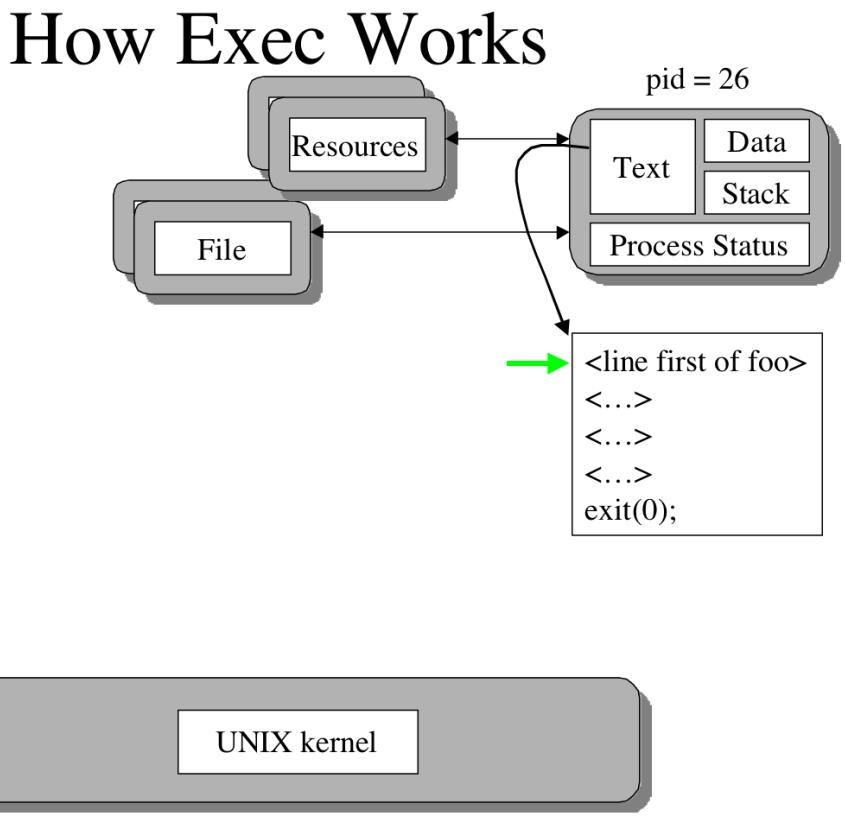
cont..


How Exec Works
pid = 26
Resources
File
Text
Data
Stack
Process Status
cpid = 0
<…>
int cpid = fork( );
if (cpid = = 0) {
exec("foo");
exit(0);
}
<parent code>
wait(cpid);
foo
UNIX kernel

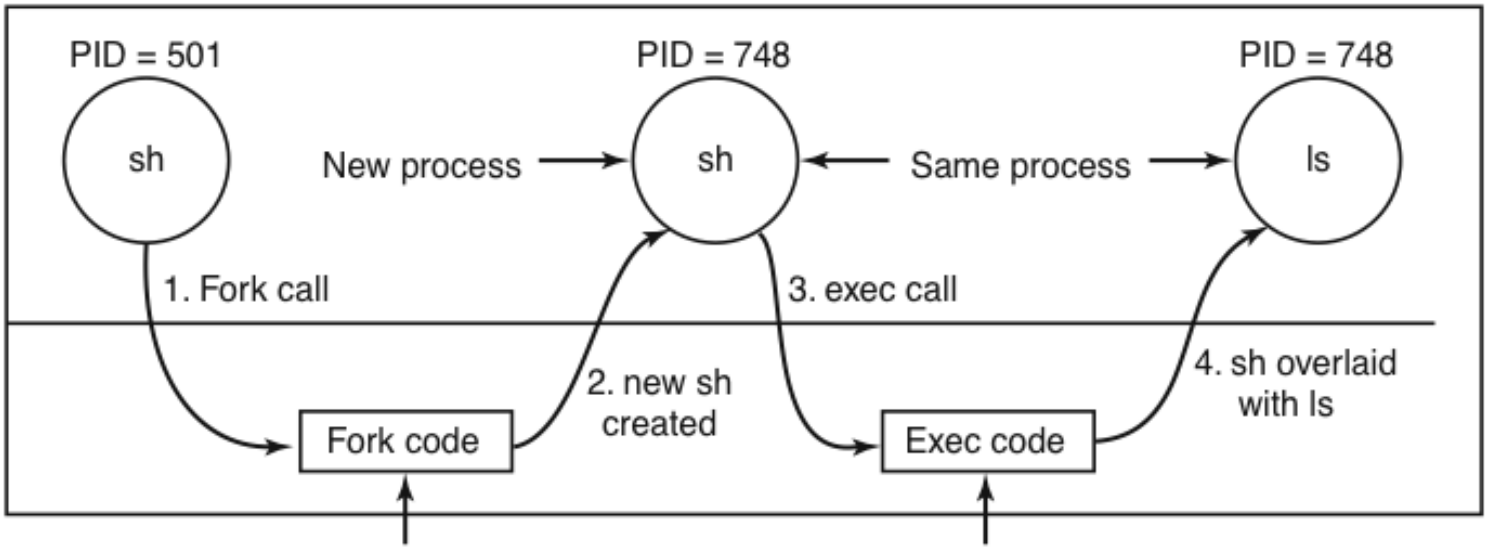
cont..


How Exec Works
pid = 26
Resources
File
Text
Data
Stack
Process Status
cpid = 0
<…>
int cpid = fork( );
if (cpid = = 0) {
exec("foo");
exit(0);
}
<parent code>
wait(cpid);
foo
UNIX kernel

cont..


How Exec Works
pid = 26
Resources
File
Text
Data
Stack
Process Status
<line first of foo>
<…>
<…>
<…>
exit(0);
UNIX kernel

How shell executes commands ?



Allocate child's task structure
Fill child's task structure from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

How to implement a simple shell

```
while (TRUE) {                          /* repeat forever /*/
        type_prompt( );                 /* display prompt on the screen */
        read_command(command, params);  /* read input line from keyboard */

        pid = fork( );                  /* fork off a child process */
        if (pid < 0) {
                printf("Unable to fork0);       /* error condition */
                continue;                       /* repeat the loop */
        }

        if (pid != 0) {
                waitpid (–1, &status, 0);       /* parent waits for child */
        } else {
                execve(command, params, 0);     /* child does the work */
        }
}
```

Interprocess communication

- Processes frequently need to communicate with other processes.
- For example shell pipeline, the output of the first process must be passed to the second process, and so on down to the line.
- So, there is a need for communication between processes.
- There are three issues here :
  - How one process can pass information to another.
  - Making sure two or more processes do not get in each other's way (Race condition problem).
  - Proper sequencing when dependencies are present (Critical Section problem or Critical Region problem).

pipe

- Pipes are the oldest form of UNIX IPC and are provided by all Unix systems.
  They have two limitations :
- They are half-duplex. Data flows only in one direction.
- They can be used only between processes that have a common ancestor.
- Normally a pipe is created by a process, that process calls fork, and the pipe is used between the parent and child.

pipe function

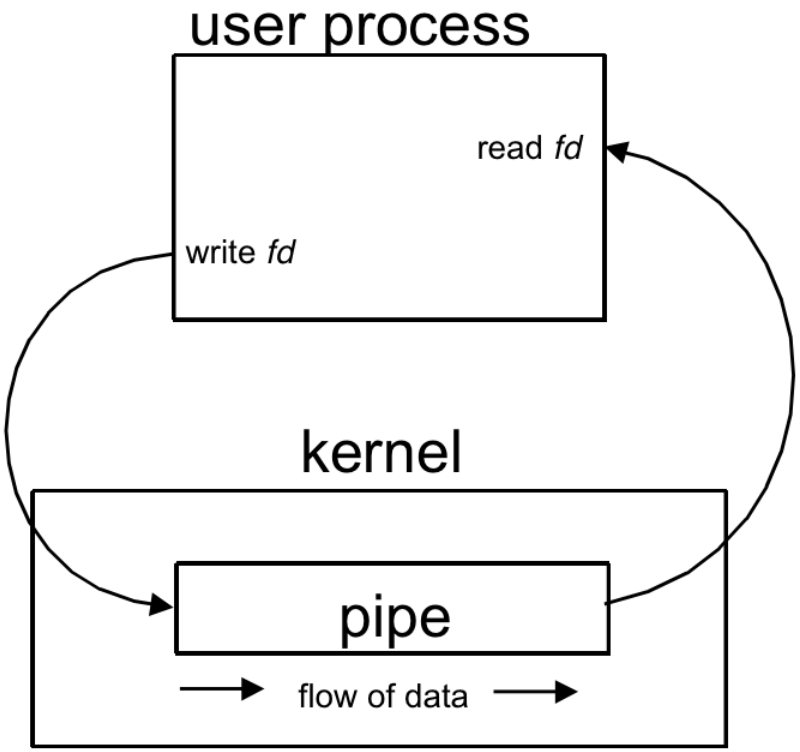
A pipe is created by calling the pipe function.

```
#include <unistd.h>

int pipe (int fields[2]);
```

Returns: 0 if OK, -1 on error.

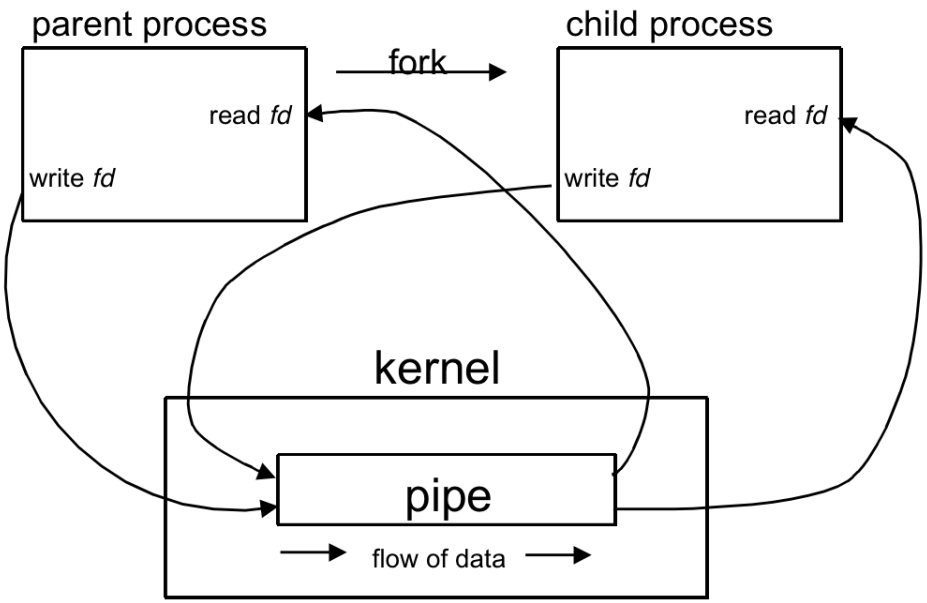
Data in the pipe flows through the kernel

Two file descriptors are returned through the fields argument. fields[0] for reading and fields[1] is open for writing.

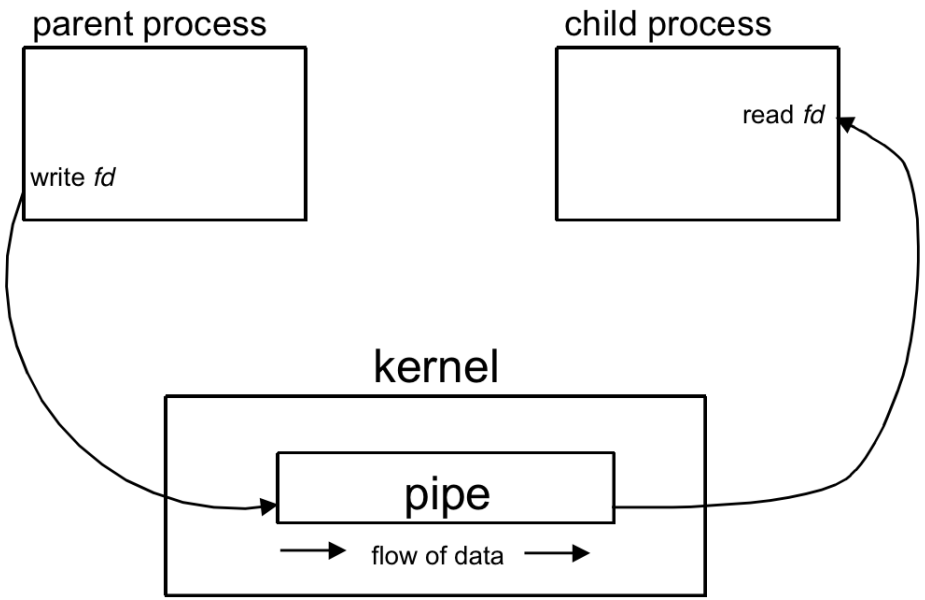


Creating IPC channel

A pipe in a single process is next to useless.
Normally the process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice-versa.



Data flow from parent to child



Study the program pipe1.c


Thaaaank
Yoooouuu!!!