

OPERATING SYSTEMS

Module1_Part5

Textbook : Operating Systems Concepts by Silberschatz



System boot process

- The procedure of starting a computer by loading the kernel is known as *booting* the system.
- On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel, loads it into main memory, and starts its execution.
- When a CPU receives a reset event-for instance, when it is powered up or rebooted -the instruction register is loaded with a predefined memory location, and execution starts there
- . At that location is the initial bootstrap program.
 - Small piece of code – bootstrap loader, BIOS, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it
 - Sometimes two-step process where boot block at fixed location loaded by ROM code, which loads bootstrap loader from disk

Fork() system call

fork() system call is used to create child processes in a C program.

It takes no arguments and returns a process ID.

After a new child process is created, **both** processes will execute the next instruction following the **fork()** system call.

Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**.

If **fork()** returns a negative value, the creation of a child process was unsuccessful.

fork() returns a zero to the newly created child process.

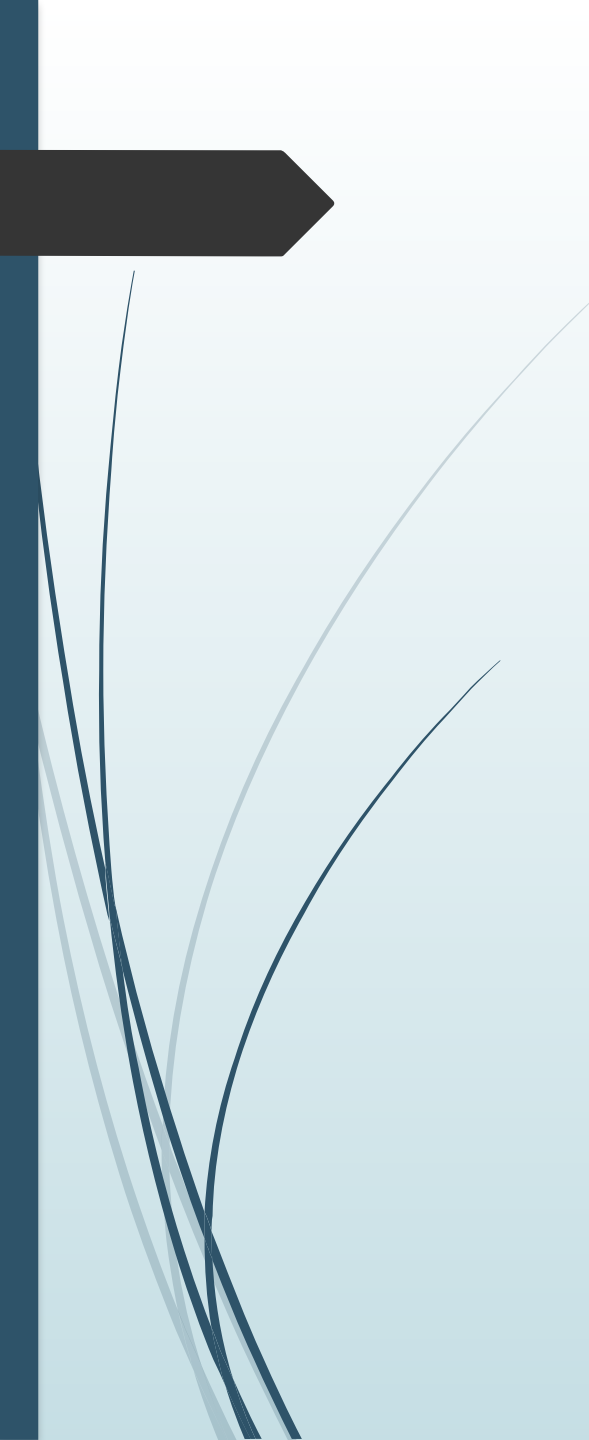
fork() returns a positive value, the **process ID** of the child process, to the parent.

A process can use function **getpid()** to retrieve the process ID assigned to this process

The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer.

make two identical copies of address spaces, one for the parent and the other for the child.

both processes have identical but separate address spaces



exit() system call

When a process terminates it executes an exit() system call.

The prototype for the exit() call is: `#include <stdlib.h> void exit(int status);`

Macro: `int EXIT_SUCCESS`

This macro can be used with the exit function to indicate successful program completion

opendir() system call

`DIR * opendir (const char * dirname)`

dirname

The path of the directory to be opened. It can be relative to the current working directory, or an absolute path.

Returns a pointer to DIR structure

readdir() system call

`struct dirent *readdir(DIR *dirp);`

The *readdir()* function shall return a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument *dirp*.

A **dirent structure** contains the character pointer `d_name`, which points to a string that gives the name of a file in the directory.



Exec system call

The `exec()` system call is used to execute a file which is residing in an active process. When `exec()` is called the previous executable file is replaced and new file is executed. process id will be the same.

Stat system call

Stat system call is a system call in Linux to check the status of a file. The `stat()` system call actually returns file attributes.

the type of the file, the size of the file, when the file was accessed (modified, deleted) that is time stamps, and the path of the file, the user ID and the group ID, etc



Wait() system call

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.

Close() system call

A close system call is a system call used to close a file descriptor by the kernel. For most file systems, a program terminates access to a file in a filesystem using the close system call.

```
int close (int fildes)
```



fork() system call

fork() system call is used to create child processes in a C program.

It takes no arguments and returns a process ID.

After a new child process is created, **both** processes will execute the next instruction following the **fork()** system call.

Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**.

If **fork()** returns a negative value, the creation of a child process was unsuccessful.

fork() returns a zero to the newly created child process.

fork() returns a positive value, the **process ID** of the child process, to the parent.

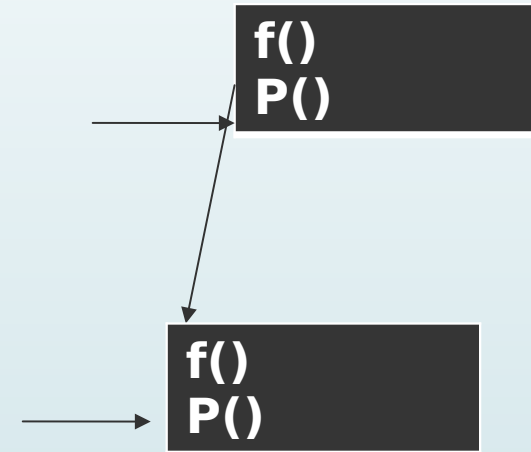
fork() system call

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // make two process which run
    // same
    // program after this
    // instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

output

```
Hello world!
Hello world!
```

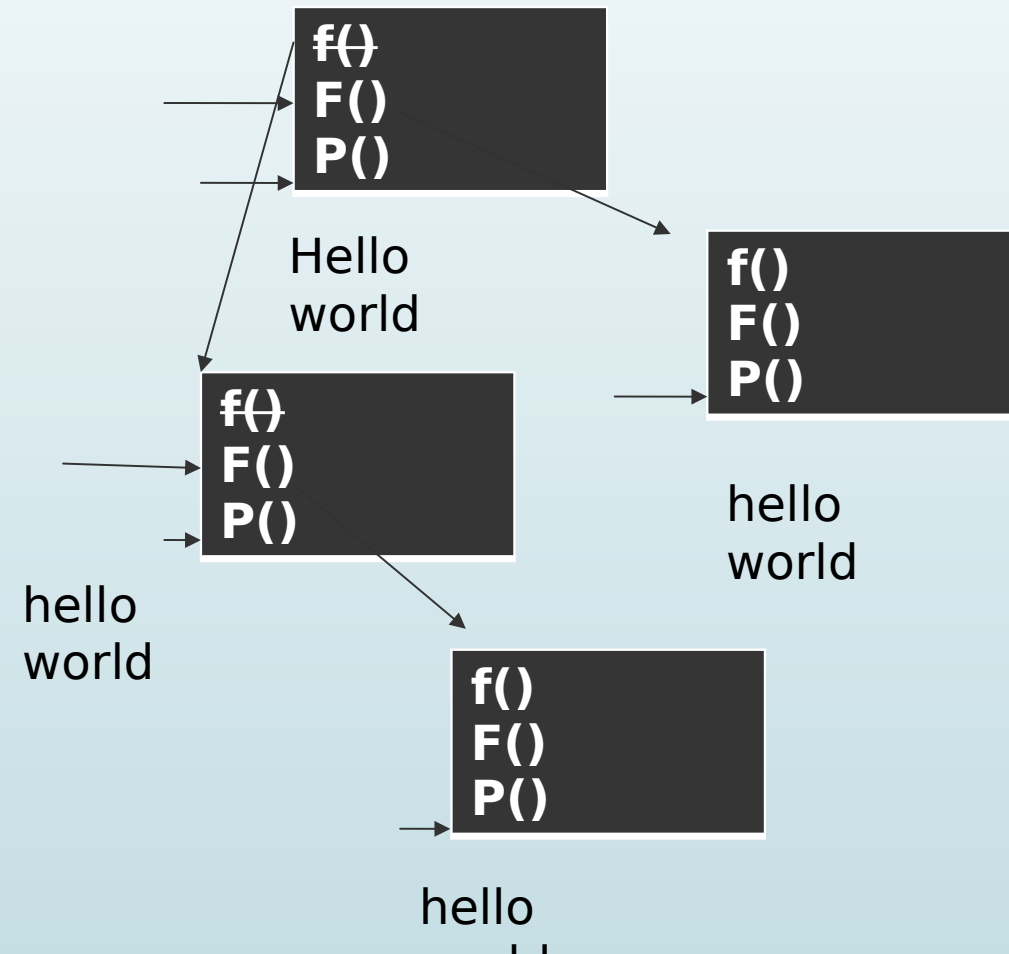


fork() system call

□ Consider this code

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
main()
{
    fork();
    fork();
    printf("hello world");
}
```

Four times hello world will be printed



getpid() system call

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
Int main(void)
{
//variable to store calling function's process id
pid_t process_id; // pid_t unsigned integer type
//variable to store parent function's process id
pid_t p_process_id;

//getpid() - will return process id of calling function
process_id = getpid();
//getppid() - will return process id of parent function
p_process_id = getppid();

//printing the process ids
printf("The process id: %d\n",process_id);
printf("The process id of parent function: %d\n",p_process_id);
return 0;
}
```

Output

The process id: 31120

The process id of parent function: 31119



Exec system call

The `exec()` system call is used to execute a file which is residing in an active process. When `exec()` is called the previous executable file is replaced and new file is executed.
process id will be the same.

Exec() system call

▮ There are two programs ex1.c ex2.c

Ex1.c

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

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

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

```
Int main(int argc,char *argv[])
```

```
{
```

```
printf("Pid of ex1.c=%d\n",getpid());
```

```
Char *args[] ={"hello",NULL};
```

```
execv("./ex2",args);
```

```
printf("Back to Ex1.c");
```

```
Return 0;
```

```
}
```



▮ Ex2.c

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

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

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

```
Int main(int argc,char *argv[])
```

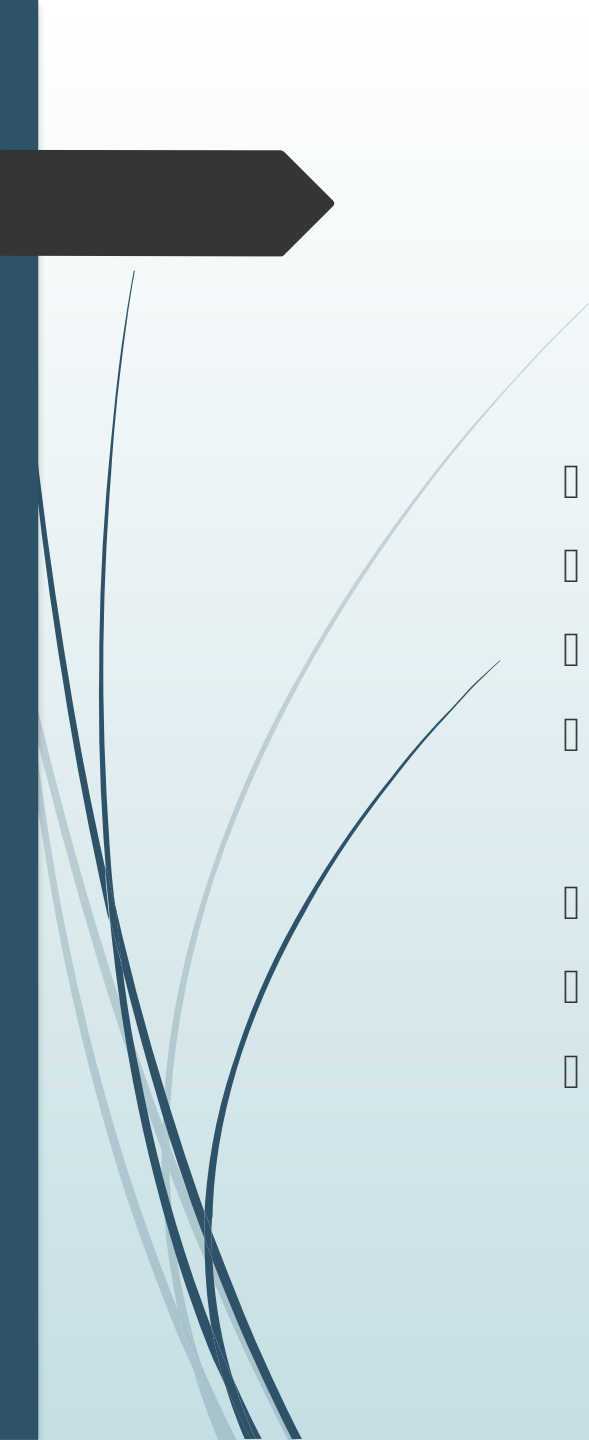
```
{
```

```
printf("We are in ex2.c\n");
```

```
printf("Pid of ex2.c=%d\n",getpid());
```

```
Return 0;
```

```
}
```

- 
- ▮ Compile these two programs
 - ▮ `gcc ex1.c -o ex1`
 - ▮ `gcc ex2.c -o ex2`
 - ▮ Run the first program `./ex1`

 - ▮ Pid of ex1.c=5962
 - ▮ We are in ex2.c
 - ▮ Pid of ex2.c=5962