

System Calls:

Shifts the control of the program from user-mode to kernel-mode.

```
Int main(){  
    int a=2;  
    int b=2;  
    int sum = a+b;  
    printf("%d",sum); —————→ program control shifts to kernel mode  
    retrun 0;  
}
```

System calls:

- ▶ File related: `open()`; `Read()`; `Write()`; `Close()` etc.
- ▶ Device related: `Read`, `Write`, `Reposition`, `ioctl`, `fcntl` etc.
- ▶ Information: `getpid`, `getppid`, attributes, get system time and data etc.
- ▶ Process control: `Load`, `execute`, `abort`, `fork`, `wait`, `signal`, `allocate` etc.

Fork(): A technique for multi-processing

- ▶ To create child process from parent process
- ▶ The fork() call is unusual in that it returns twice: It returns in both, the process calling fork() and in the newly created process. The child process returns zero and the parent process returns a number greater than zero. If fork() fails then its return value will be less than zero.

```
int main(){  
    fork();  
    fork();  
    printf("Hello");  
    Return 0;  
}
```

The diagram illustrates the execution flow of a program using fork(). It shows a parent process that calls fork() twice, creating three child processes (Child 1, Child 2, Child 3) and one parent process. All processes print "Hello".

```
graph TD
    PP[Parent process] --> C1[Child 1]
    PP --> P1[parent]
    C1 --> C2[Child 2]
    C1 --> C3[Child 3]
    P1 --> P2[parent]
```

Execution flow:

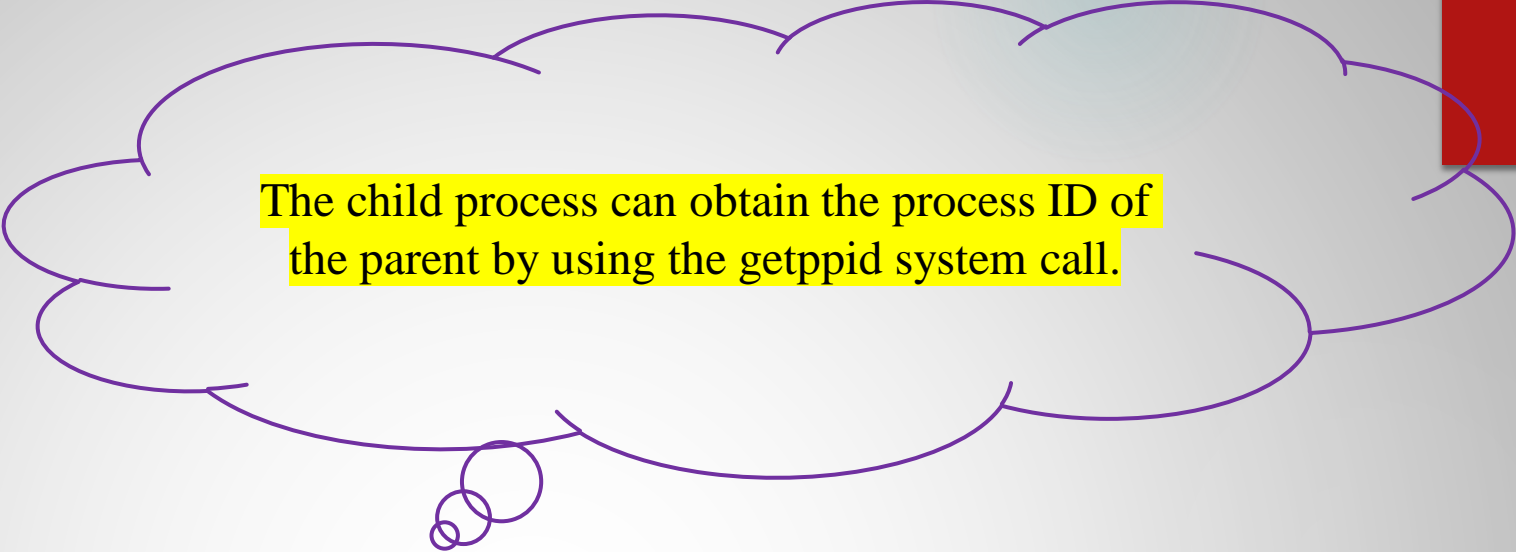
- Parent process calls fork().
- Parent process calls fork() again.
- Parent process prints "Hello".
- Child 1 prints "Hello".
- Child 2 prints "Hello".
- Child 3 prints "Hello".
- Parent process returns 0.

Fork(): example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
char string1[] = "\n Hello";
char string2[] = " CS342.\n";
int main(void)
{
    pid_t PID;
    PID = fork();
    if (PID == 0) /* Child process */
        printf("%s", string2);
    else /* Parent process */
        printf("%s", string1);
    exit(0); /* Executed by both processes */
}
```

Here, the parent process prints Hello to stdout, and the new child process prints CS342. Note that the order of printing is not guaranteed. Without some method of synchronizing the processes execution, Hello may or may not be printed before CS342.

```
main(void){
int childpid;
if( (childpid = fork() ) == -1)
{ printf("\n Can't fork.\n");
exit(0);}
else
if(childpid == 0)
{ /* Child process */
printf("\n Child: Child pid = %d, Parent pid = %d \n", getpid(), getppid());
exit(0);
}
else
{ /* Parent Process */
printf("\n Parent: Child pid = %d, Parent pid = %d \n", childpid, getpid());
exit(0);
}}
```



The child process can obtain the process ID of the parent by using the getppid system call.

Why Fork() ???

- ▶ A process wants to make a copy of itself so that one copy can handle an operation while the other copy does another task.
- ▶ A process wants to execute another program. Since the only way to create a new process is with the fork operation, the process must first fork to make a copy of itself, then one of the copies issues an exec system call operation to execute a new program. This is typical for programs such as shells.

Execv():

- ▶ `execv` replaces the calling process image with a new process image.
- ▶ This has the effect of running a new program with the process ID of the calling process.
- ▶ The `execv` function is most commonly used to overlay a process image that has been created by a call to the `fork` function.
- ▶ A successful call to `execv` does not have a return value because the new process image overlays the calling process image. However, a -1 is returned if the call to `execv` is unsuccessful.

use of execv to execute the ls shell command:

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

main()
{
    pid_t pid;
    char *const parmList[] = {"/bin/ls", "-l", "/u/userid/dirname", NULL};

    if ((pid = fork()) == -1)
        printf ("fork error");
    else if (pid == 0) {
        execv("/bin/ls", parmList);
        printf("Return not expected. Must be an execv error.n");
    }
}
```


Running a shell script using execv

prog.sh shell script which contain a simple sum of arguments:

```
expr $1 + $2 + $3
```

Sample.c to run the prog.sh file using execv

```
void main(int argc, char *argv[]) {  
    char* file=argv[1];  
    char* arguments[] = { "sh", file, argv[2],argv[3],argv[4],NULL };  
    execv("/bin/sh", arguments);  
}
```

```
./main prog.sh 1 2 3
```