CS342: OS Lab 3

System Calls

Shifts the control from user-mode to kernel mode.

Examples of few system calls:

File Related: Read, Write, Close etc.

Information: getpid, getppid, get system time and date etc.

Process control: Load, execute, abort, fork, wait, signal, allocate etc.

fork() : A technique for multi-processing

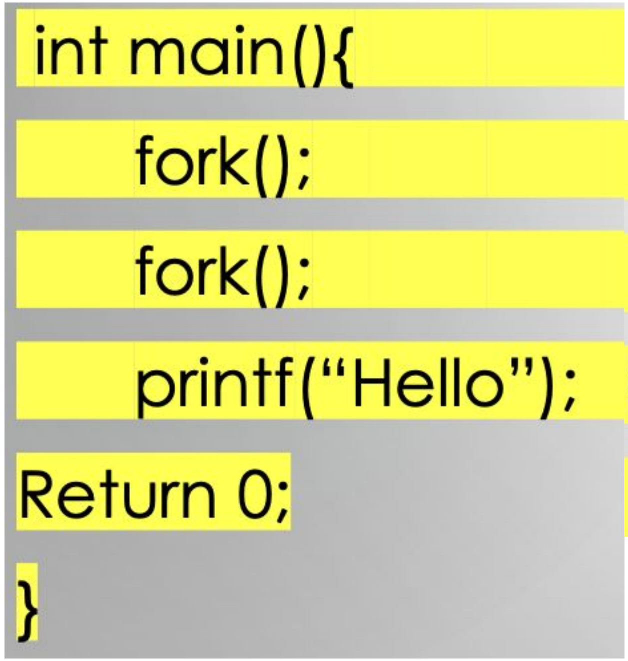
To create child process from parent process.

The fork() call is unusual in that it returns twice: It returns in both, the process calling fork() and in the newly created process.

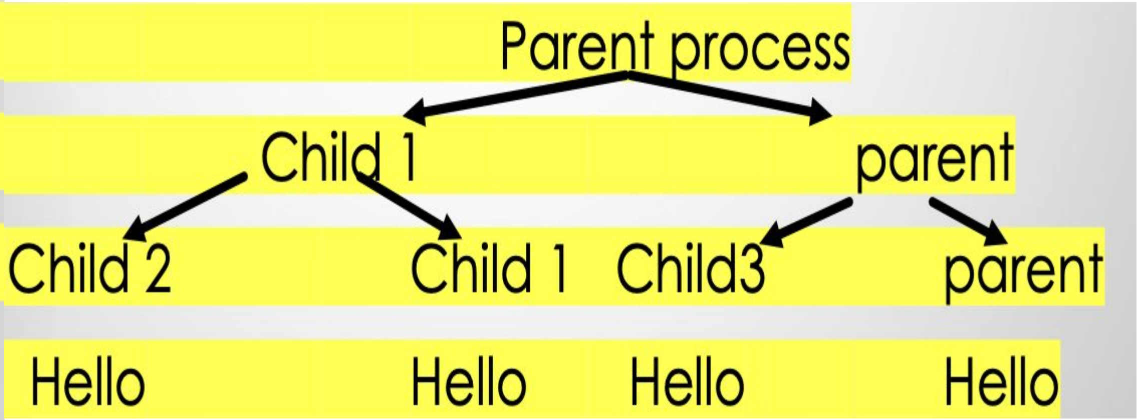
The child process returns zero and the parent process returns a number greater then zero.

If fork() fails then its return value will be less than zero.

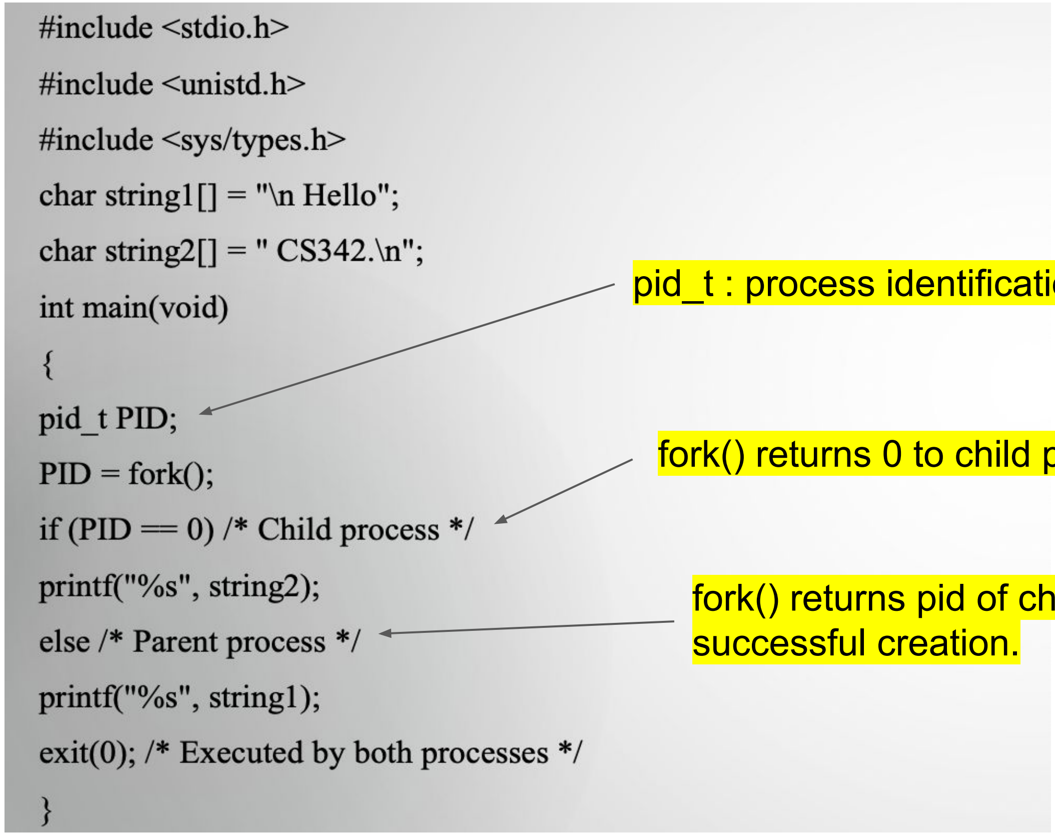
fork() : A technique for multi-processing



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

pid_t : process identification data-type

fork() returns 0 to child process on successful creation

fork() returns pid of child process to parent process on successful creation.

```
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.

Fetches process id of parent process

Fetches process id of the calling process

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, -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");
   }
 }
```

`ls -l /path/to/file : print files in long listing format`

Array of char pointers

contains pathname of a file that contains the new program to be executed

Last argument is always a NULL pointer

This shouldn’t be executed, since we have loaded a new process 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
```