İŞLETİM SİSTEMLERİ UYGULAMA

Outline

- Process management -> Section-1
  - creation, differentiation, termination
- Inter-process communcation (IPC) -> Section-1
  - Unix-based
  - Posix
- Threads -> Section-2
- Thread synchronization -> Section-2

Section Outline

- Process definition
- Process handling
- Process creation
- Process differentiation
- Process termination
- Process synchronization

What is a Process

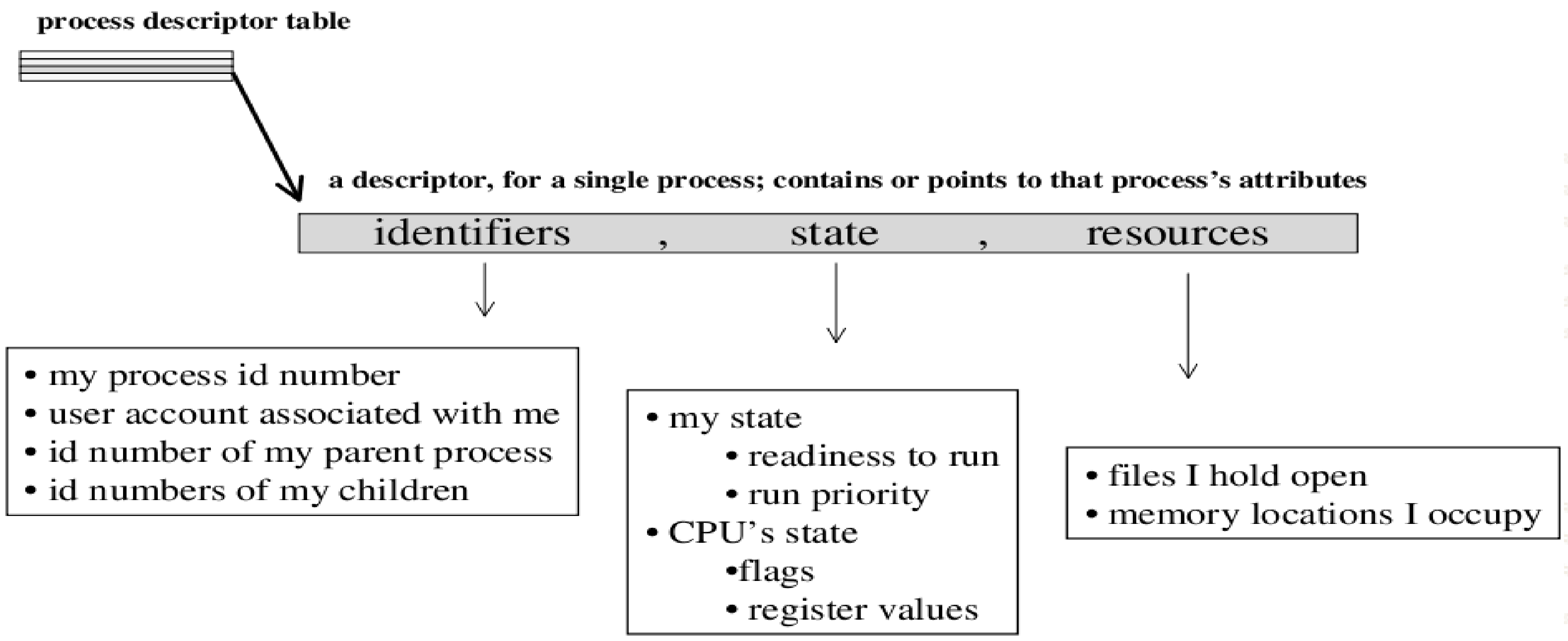
- **Definition ;**
  - A process is an instance of a running program.
- Not the same as "**program**" or "**processor**"
  - A program is a set of instructions and initialized data in a file, usually found on a disk.
- A **process** is an instance of that **program** while it is running, along with the state of all the CPU registers and the values of data in memory.

Process Handling

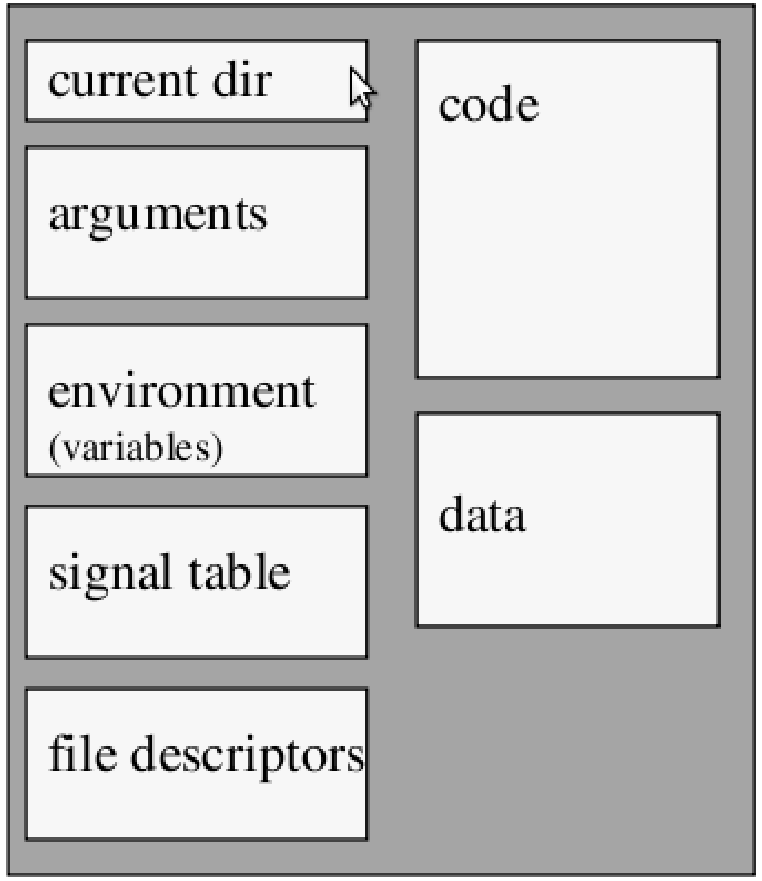
- Constituents of a process
  - Its code
  - Data
    - its own
    - OS’s data used by/for process
  - Various attributes OS needs to manage it

Process Handling

- OS keeps track of all processes
  - Process table/array/list
  - Elements are process descriptors (aka control blocks)
  - Descriptors reference code & data


process descriptor table
a descriptor, for a single process; contains or points to that process's attributes
identifiers , state , resources
• my process id number
• user account associated with me
• id number of my parent process
• id numbers of my children
• my state
• readiness to run
• run priority
• CPU's state
•flags
• register values
• files I hold open
• memory locations I occupy

Single process in Unix (consolidated view)

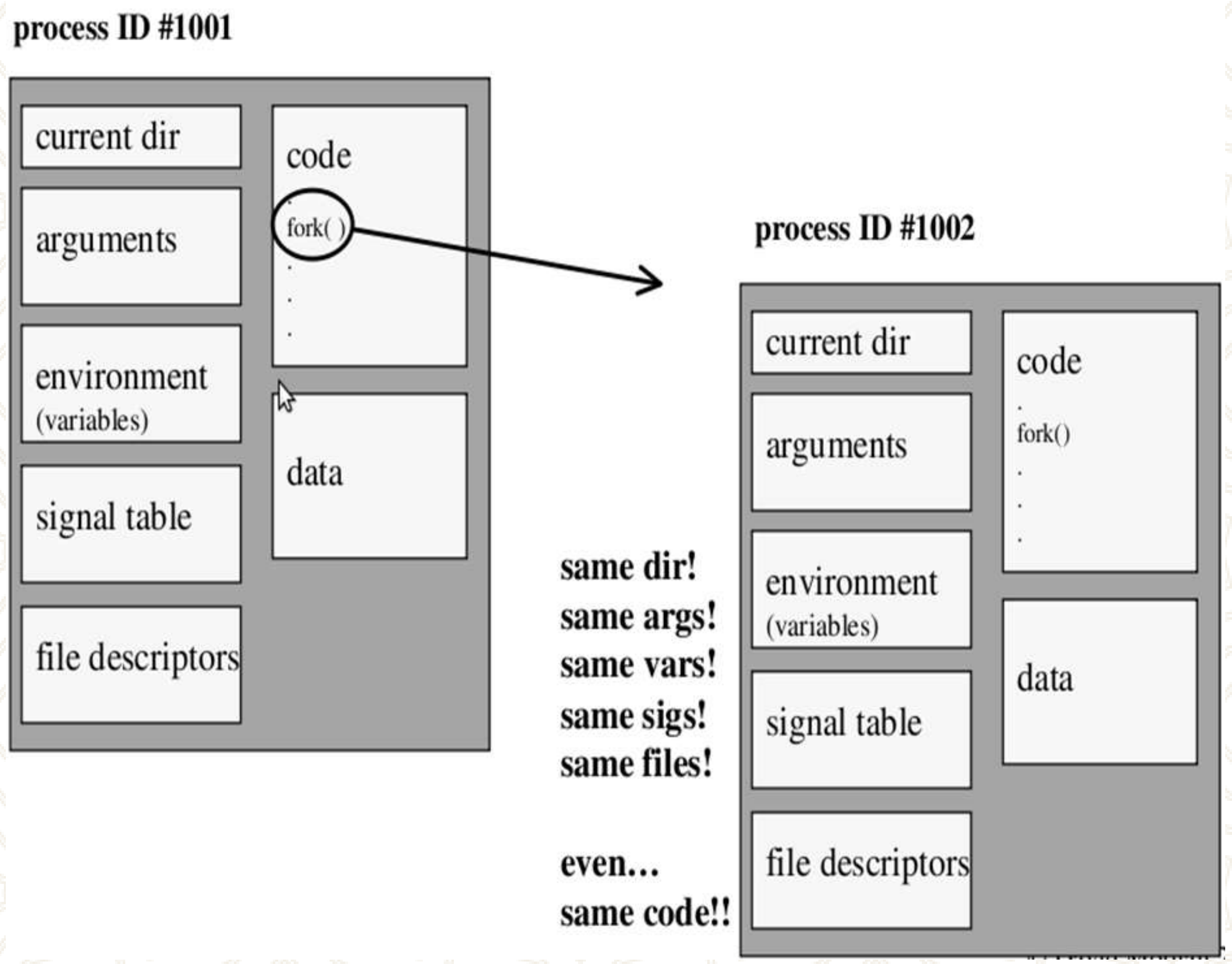


- Some important properties
  - code
  - data
  - current directory
  - argument list
  - environment list
  - responses to signals
  - list of open files

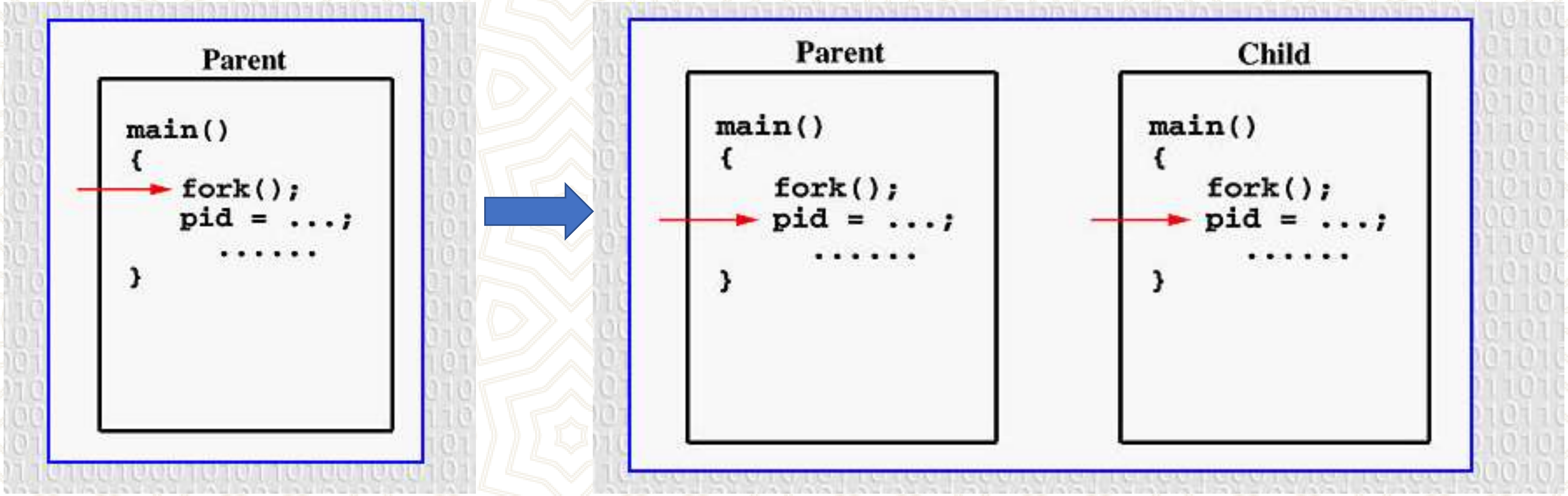
Process Creation

- OS perspective
  - find empty slot in process table
  - write a process descriptor and
  - put it there
  - read in program code from disk
- User perspective
  - System calls
    - **fork(), exec()**

fork() system call

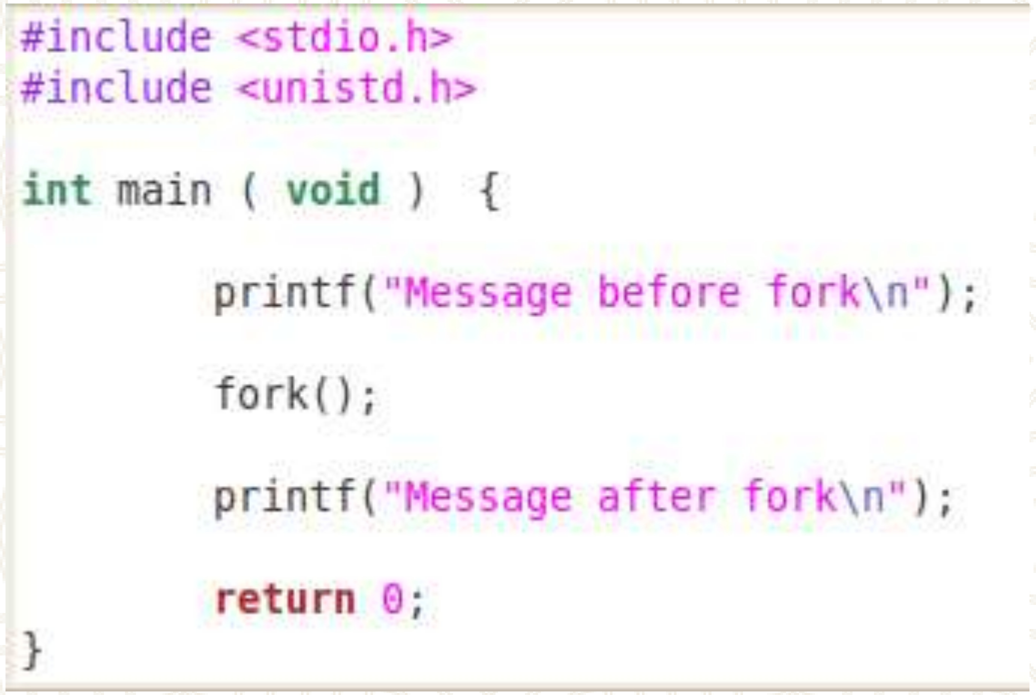


fork() system call



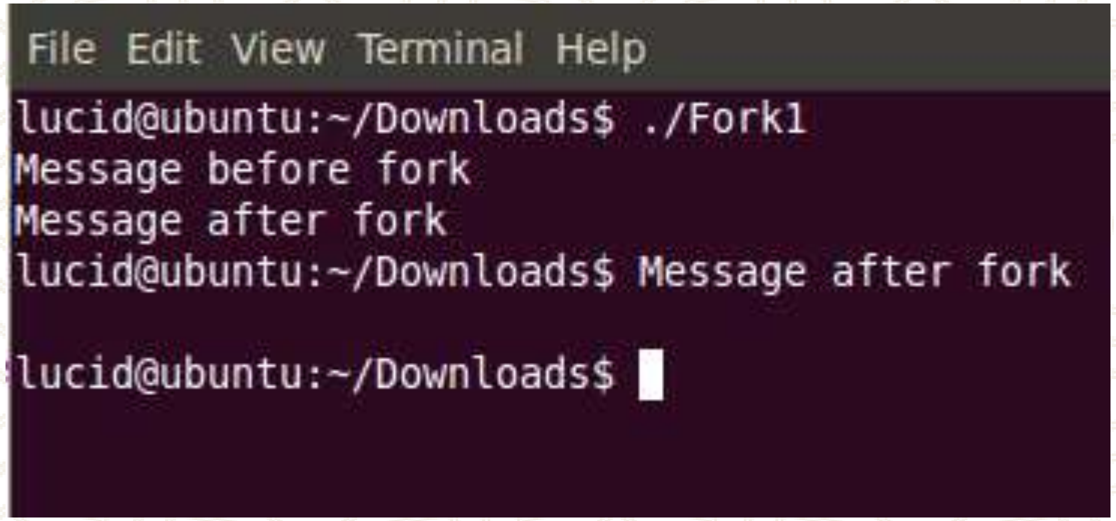
```
main()
{
    fork();
    pid = ...;
      ......
}
```

A Simple fork() Example



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

int main ( void )  {

        printf("Message before fork\n");

        fork();

        printf("Message after fork\n");

        return 0;
}
```

- a simple fork example
- Message after fork is printed **twice** !!

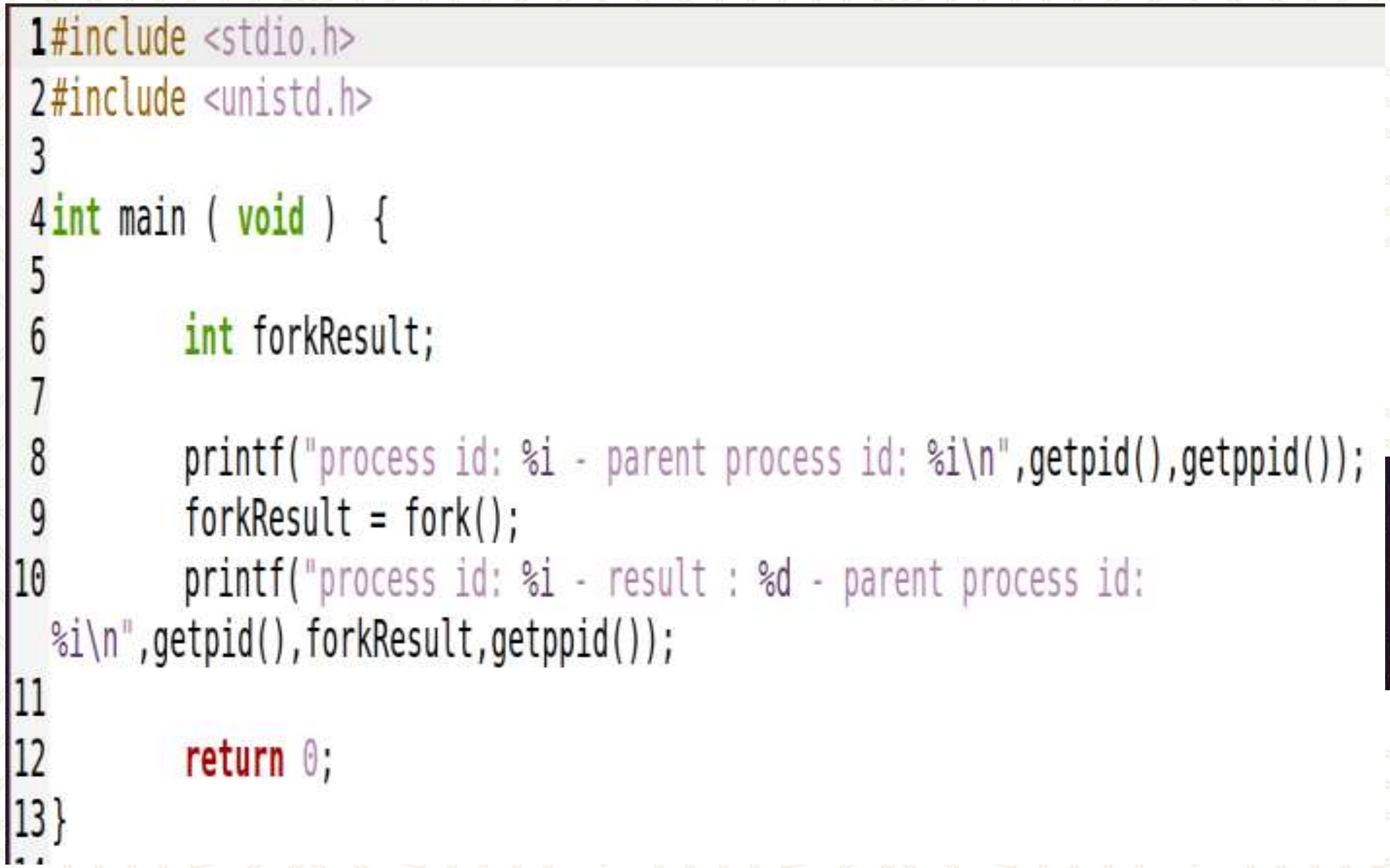


```
lucid@ubuntu:~/Downloads$ ./Fork1
Message before fork
Message after fork
lucid@ubuntu:~/Downloads$ Message after fork

lucid@ubuntu:~/Downloads$ █
```

- **Ex_1_fork1.c**

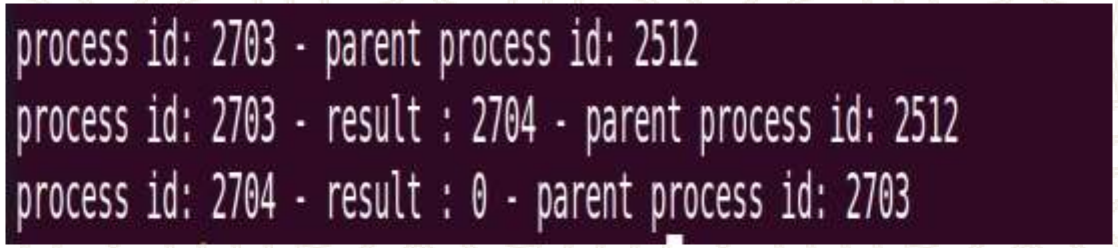
Self Identification

- **Ex_2_fork2.c**



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

int main ( void )  {

        int forkResult;

        printf("process id: %i - parent process id: %i\n",getpid(),getppid());
        forkResult = fork();
        printf("process id: %i - result : %d - parent process id: 
%i\n",getpid(),forkResult,getppid());

        return 0;
}
```

- for the parent process fork returns **child's pid**
- for the child process fork returns **0**

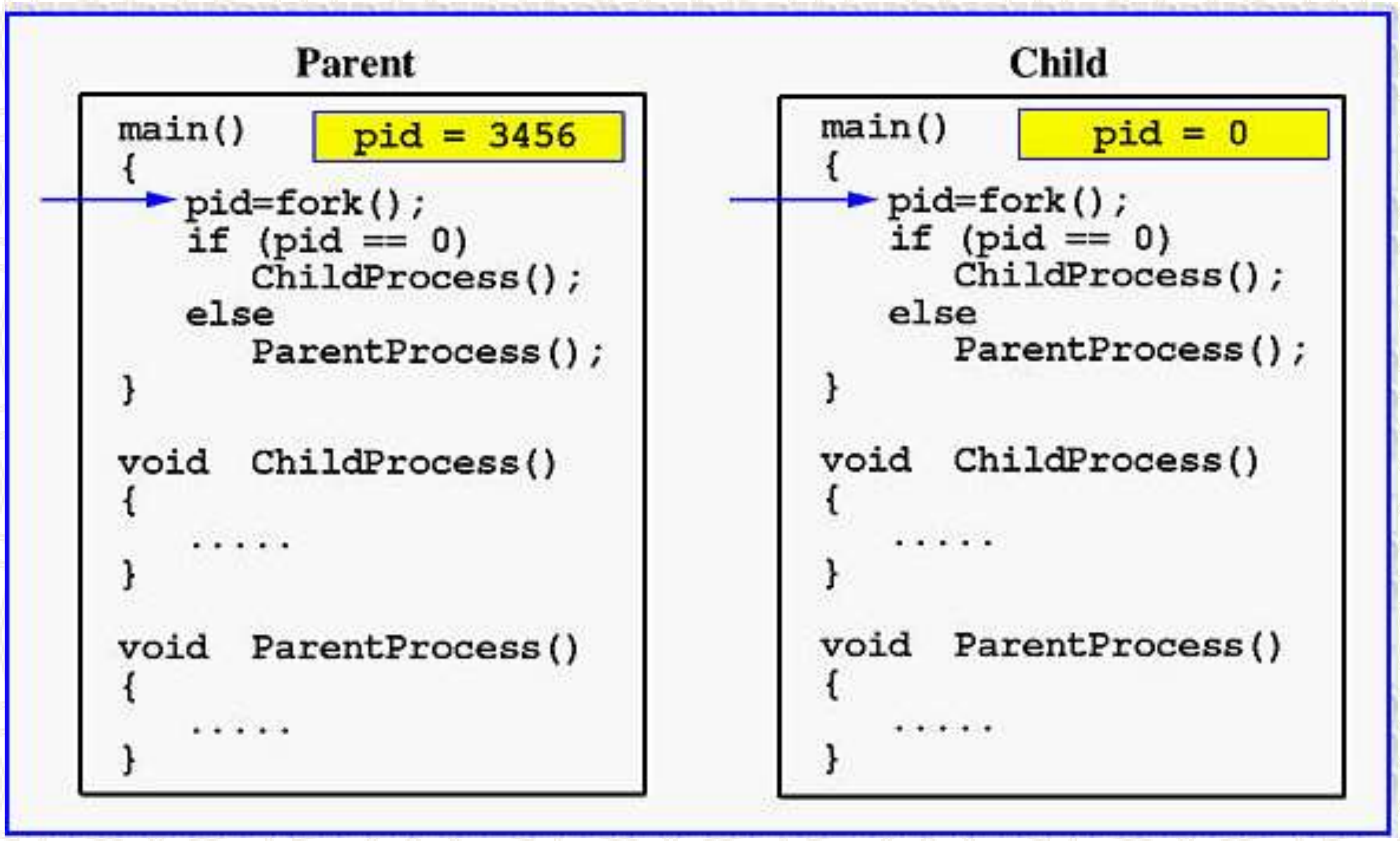


```
process id: 2703 - parent process id: 2512
process id: 2703 - result : 2704 - parent process id: 2512
process id: 2704 - result : 0 - parent process id: 2703
```

Process Differentiation

- **identical?** not what we had in mind!
- more useful if child does different stuff
- can we give it **different** behaviour?
  - in the form of source code
  - in the form of an existing binary executable
    - exec() family of functions

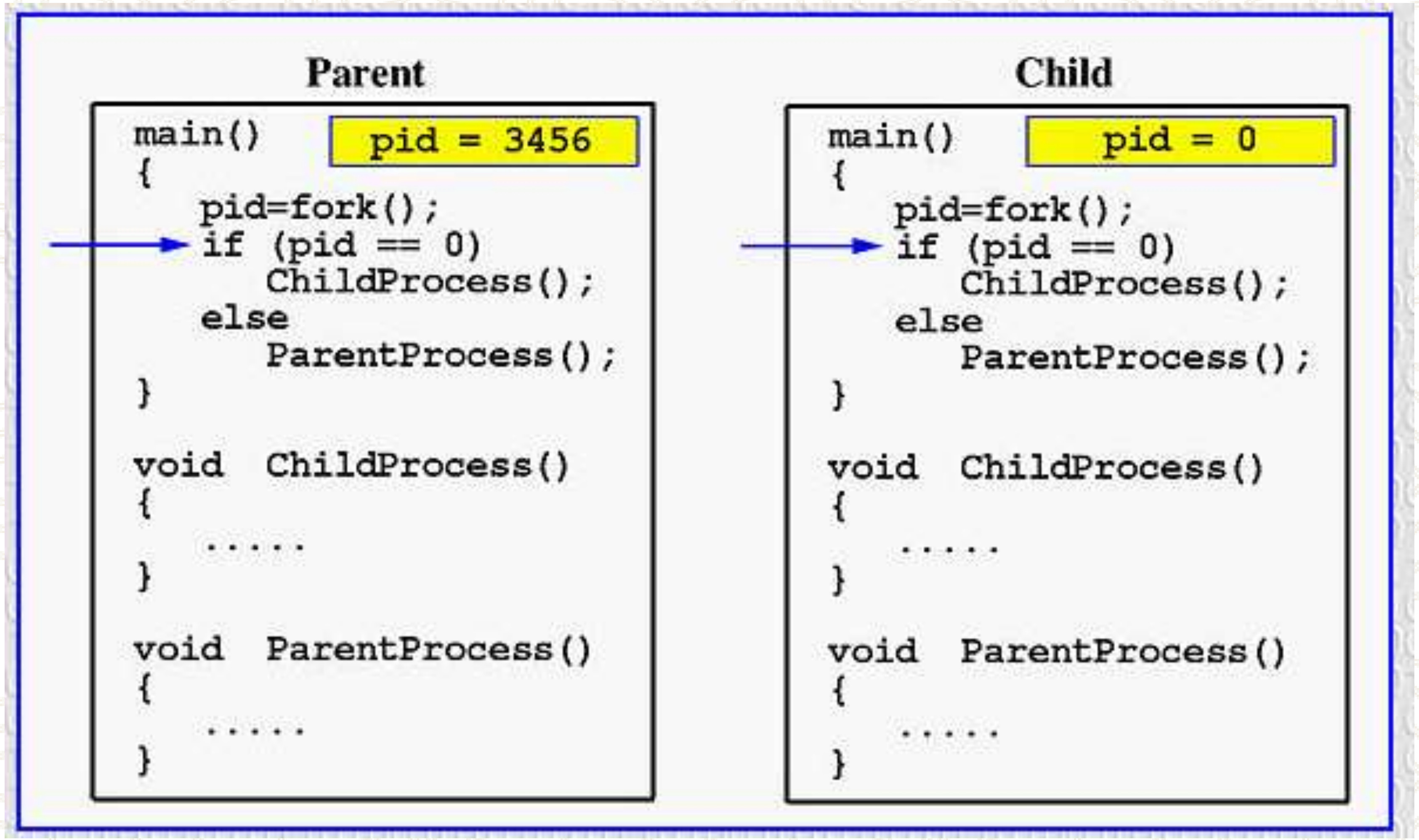
Process Differentiation



```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess()
{
    .....
}

void  ParentProcess()
{
    .....
}
```

```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess()
{
    .....
}

void  ParentProcess()
{
    .....
}
```

Process Differentiation



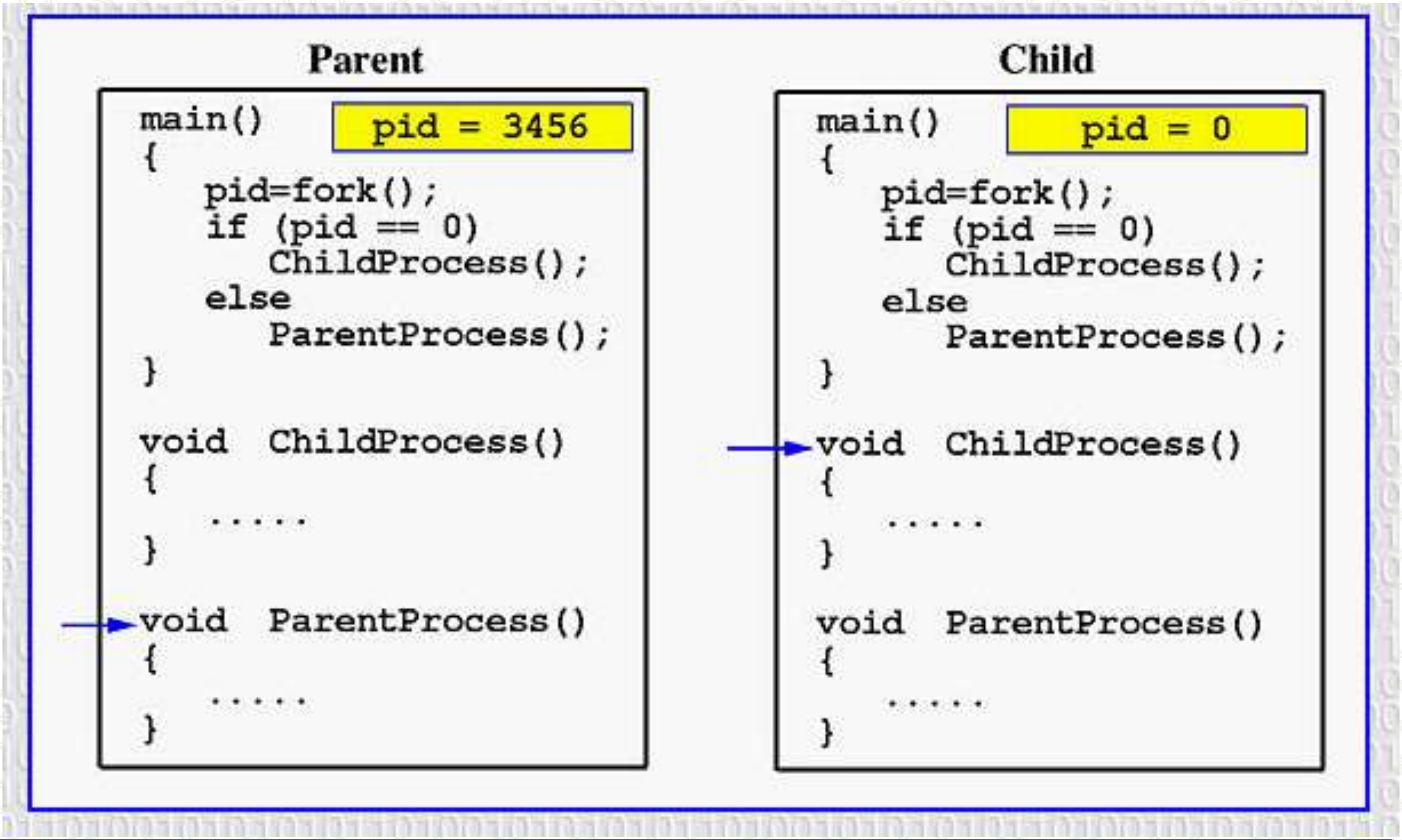
Parent:

```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess()
{
    .....
}

void  ParentProcess()
{
    .....
}
```

Child:

```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess()
{
    .....
}

void  ParentProcess()
{
    .....
}
```

Process Differentiation



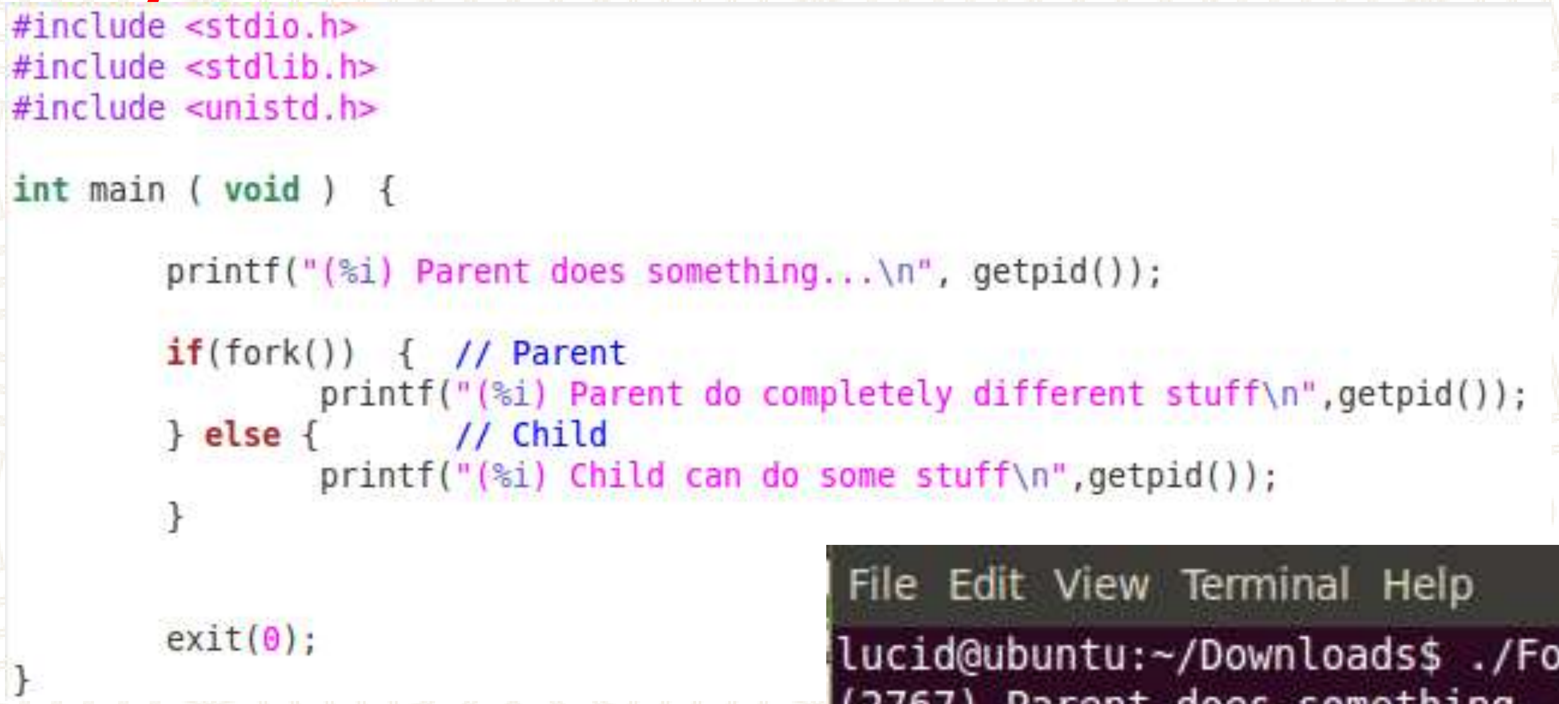
Parent:

```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess()
{
    .....
}

void  ParentProcess()
{
    .....
}
```

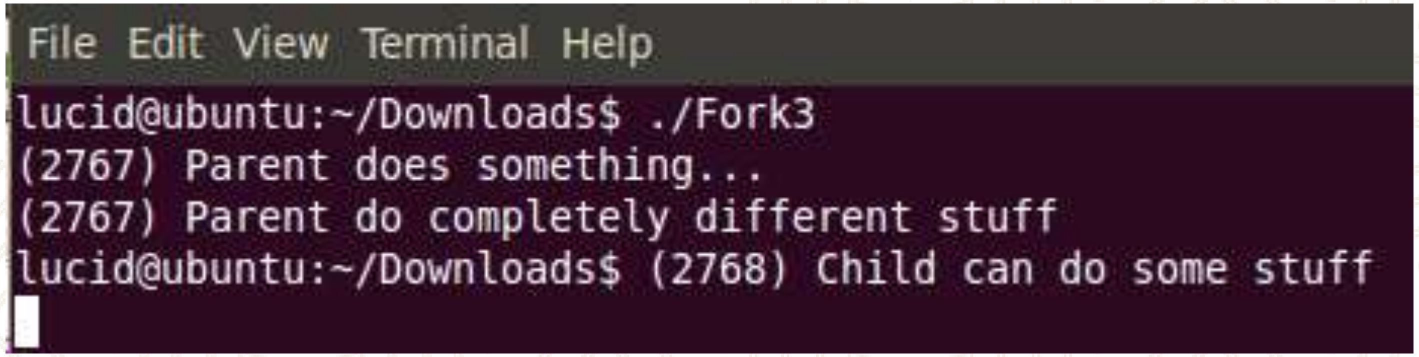
Child:

```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess()
{
    .....
}

void  ParentProcess()
{
    .....
}
```

Process Differentiation
by source code



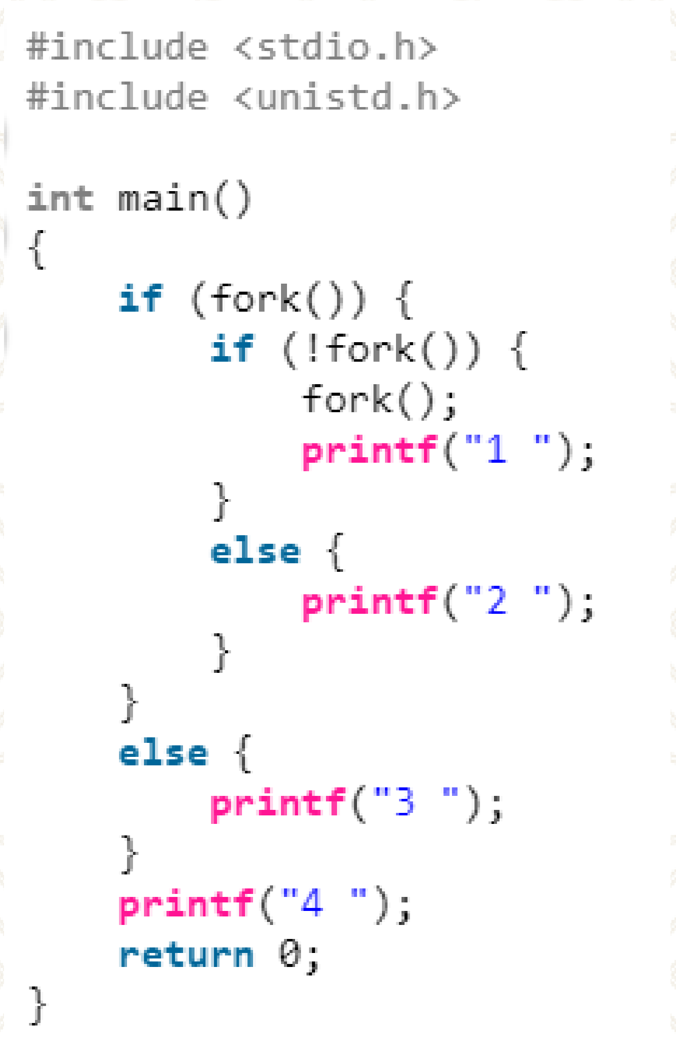
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ( void )  {

        printf("(%i) Parent does something...\n", getpid());

        if(fork())  {  // Parent
                printf("(%i) Parent do completely different stuff\n",getpid());
        } else {        // Child
                printf("(%i) Child can do some stuff\n",getpid());
        }


        exit(0);
}
```



```
lucid@ubuntu:~/Downloads$ ./Fork3
(2767) Parent does something...
(2767) Parent do completely different stuff
lucid@ubuntu:~/Downloads$ (2768) Child can do some stuff
```

- **Ex_3_fork3.c**

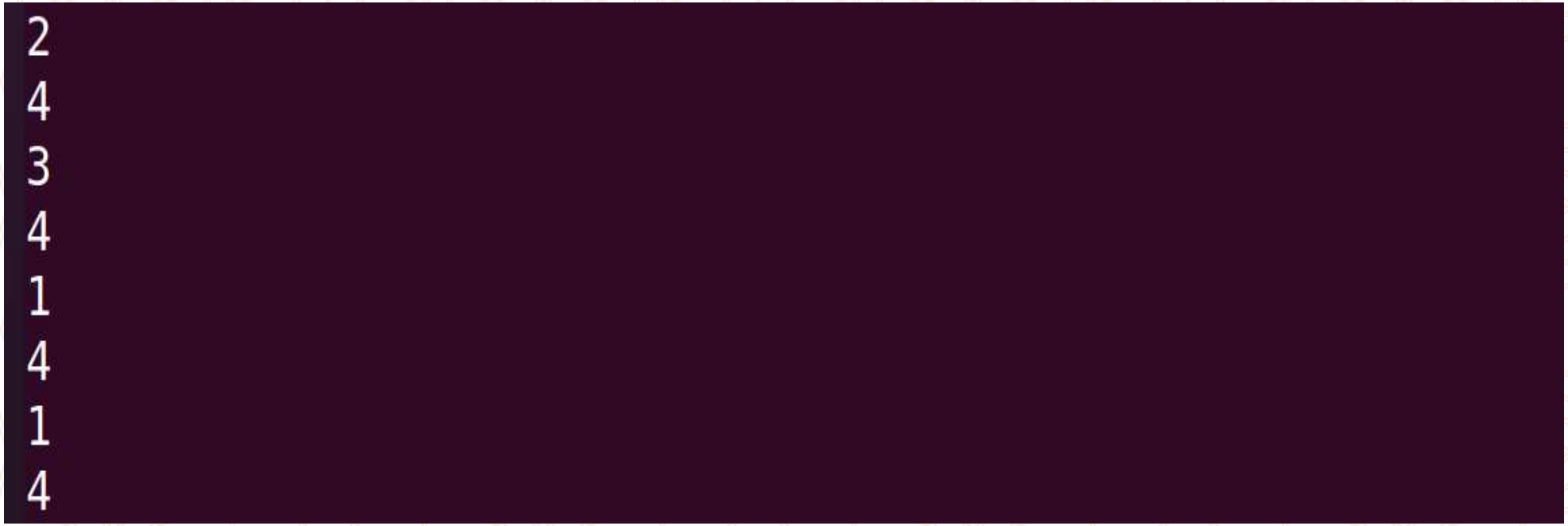
Fork Practice Question



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

int main()
{
    if (fork()) {
        if (!fork()) {
            fork();
            printf("1 ");
        }
        else {
            printf("2 ");
        }
    }
    else {
        printf("3 ");
    }
    printf("4 ");
    return 0;
}
```

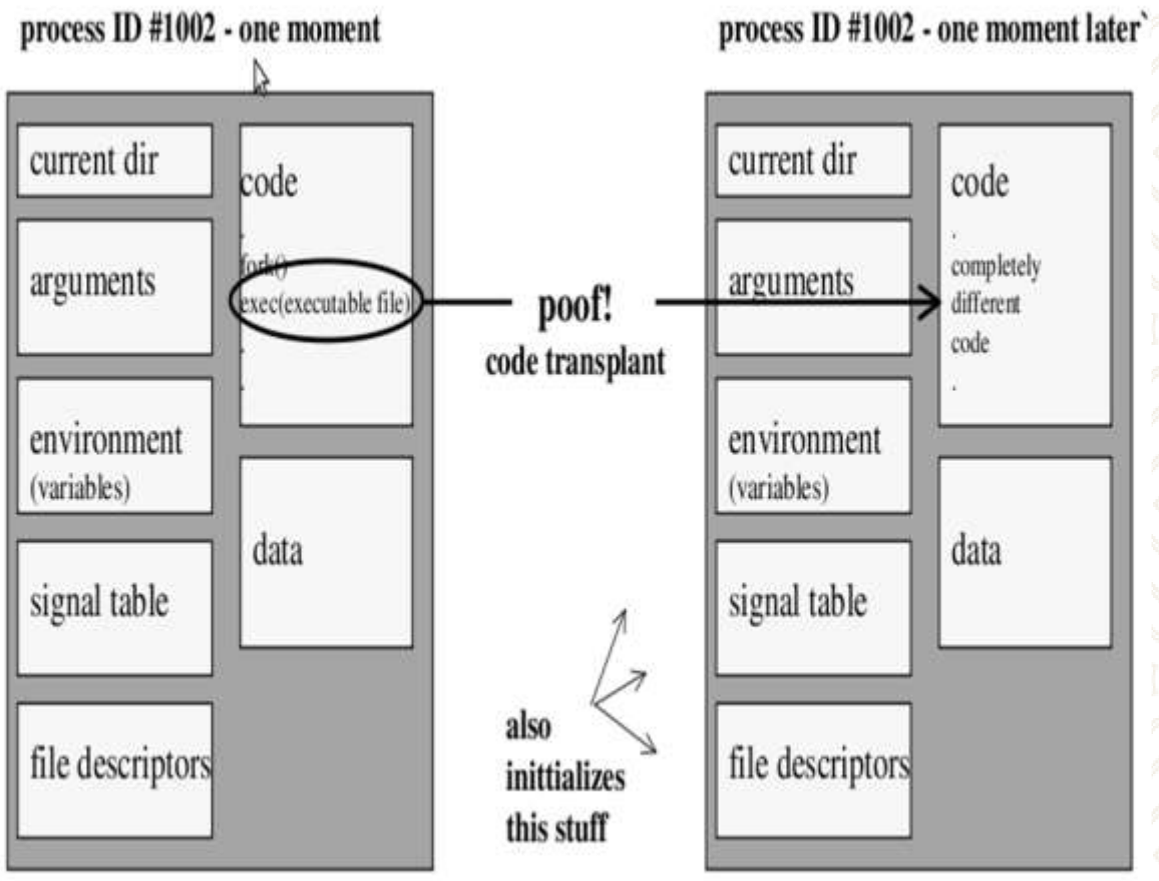
- **Ex_4_fork4.c**

Output



```
2
4
3
4
1
4
1
4
```

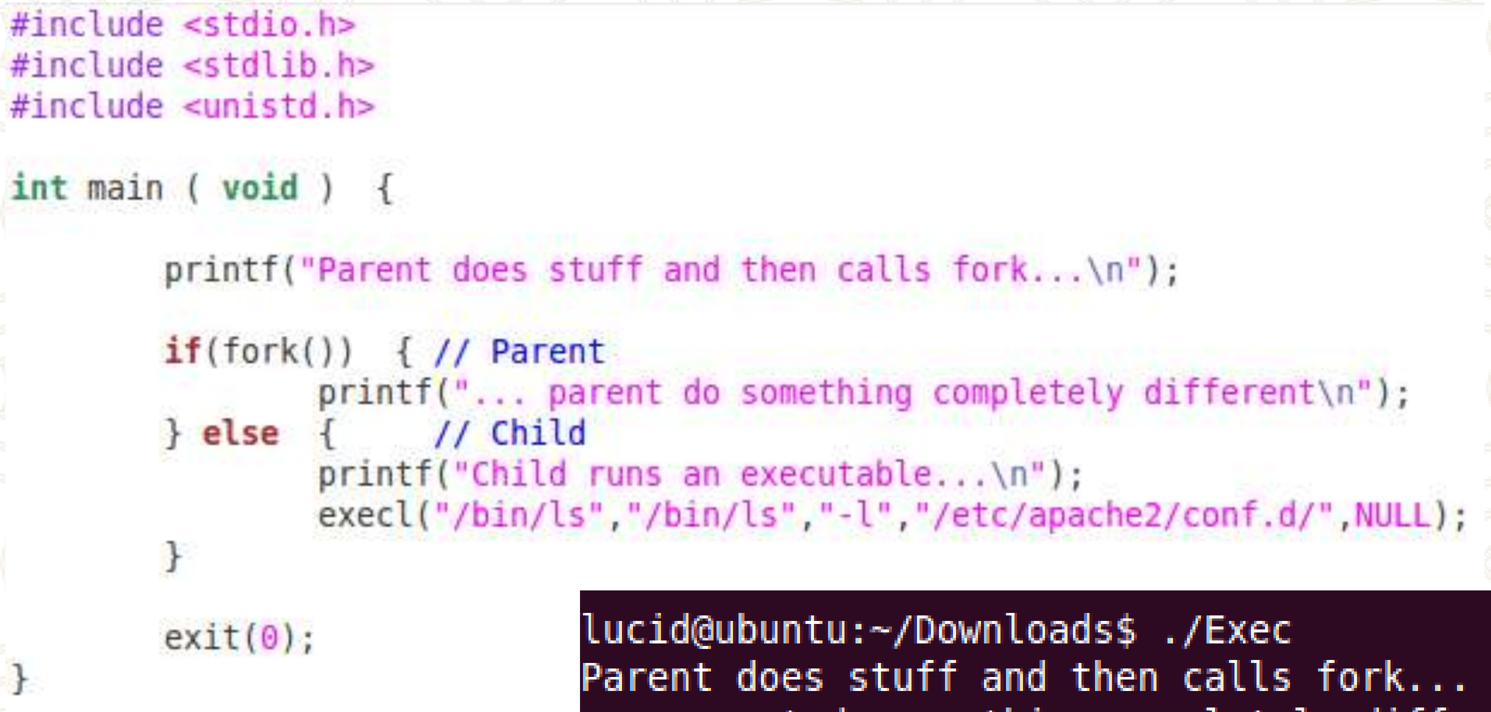
Process Differentiation
by exec() function



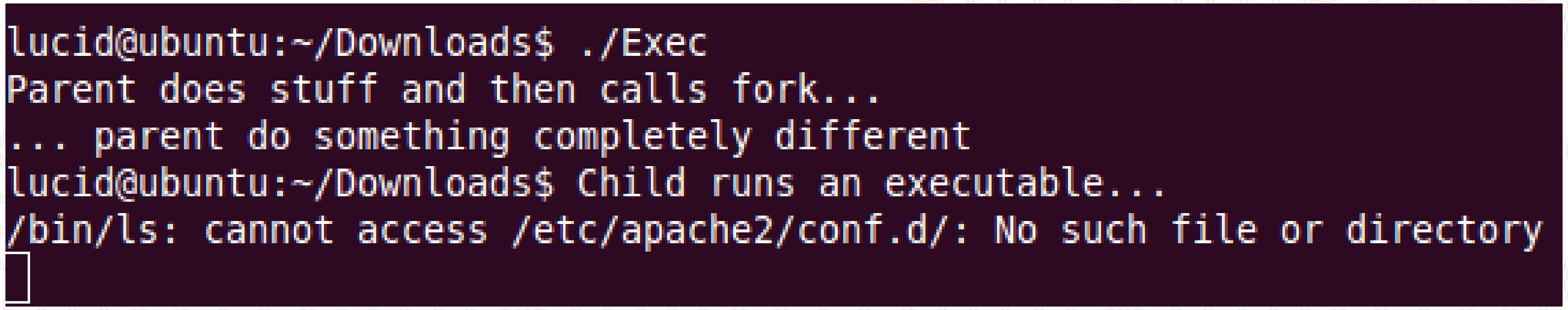
Process Differentiation by exec() function

- **exec() family of functions**
  - int **execl** (const char *pathname, const char *arg0, …);
  - int **execv** (const char *pathname, char *const argv[]);
  - int **execle** (const char *pathname, const char *arg0, ..., 0, char *const envp[]);
  - int **execlp** (const char *filename, const char *arg0, …);
  - int **execvp** (const char *filename, char *const argv[]);
  - int **execve** (const char *pathname, char *const argv[],   char *const envp[]);

A Simple exec() Example



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ( void )  {

        printf("Parent does stuff and then calls fork...\n");

        if(fork())  { // Parent
                printf("... parent do something completely different\n");
        } else  {       // Child
                printf("Child runs an executable...\n");
                execl("/bin/ls","/bin/ls","-l","/etc/apache2/conf.d/",NULL);
        }

        exit(0);
}
```



```
lucid@ubuntu:~/Downloads$ ./Exec
Parent does stuff and then calls fork...
... parent do something completely different
lucid@ubuntu:~/Downloads$ Child runs an executable...
/bin/ls: cannot access /etc/apache2/conf.d/: No such file or directory
```

- **Ex_5_exec.c**

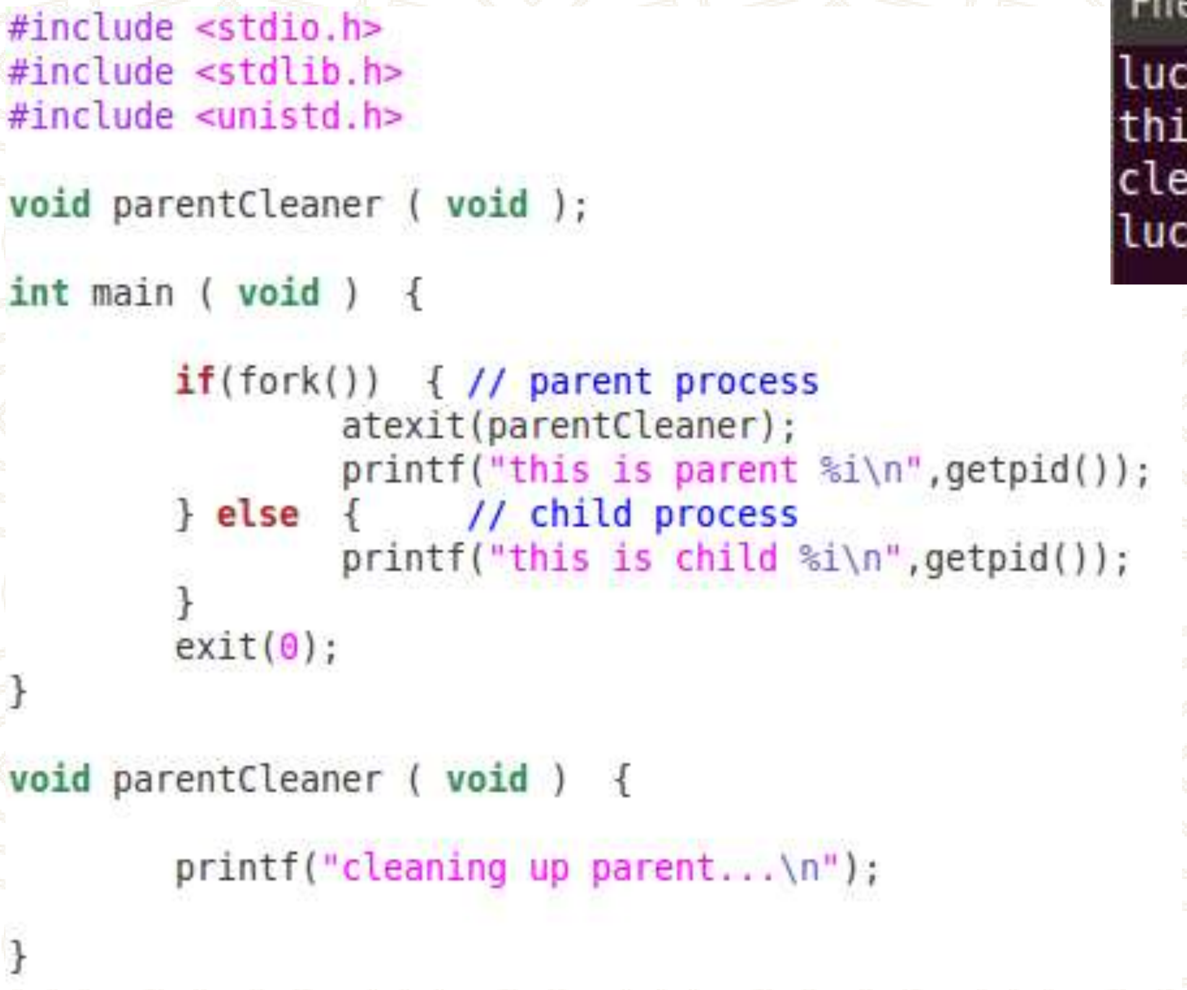
Process Termination

- void **exit** (int status);
  - exits a process
  - normally return with status 0
- int **atexit** (void (*function)(void));
  - registers function to be executed on exit
- int **wait** (int *child_status)
  - suspends current process until one of its children terminates

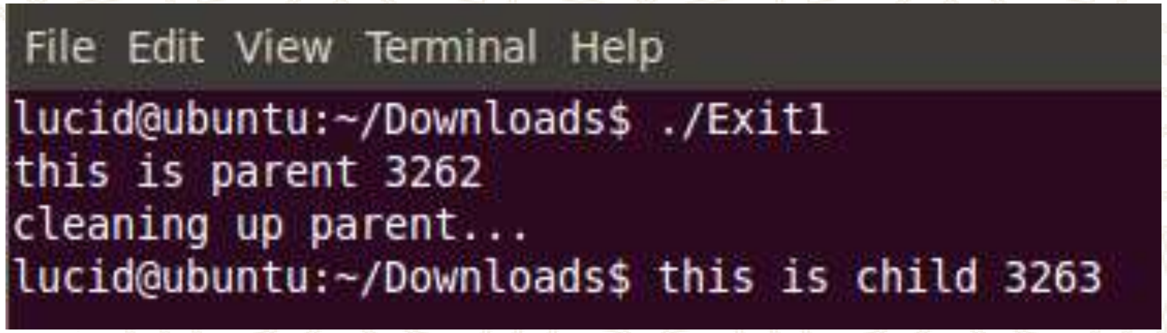
exit() vs return

- **return**
  - is an instruction of the language that returns from a function call.
- **exit**
  - is a system call (not a language statement) that terminates the current process.

atexit() example



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void parentCleaner ( void );

int main ( void )  {

        if(fork())  { // parent process
                atexit(parentCleaner);
                printf("this is parent %i\n",getpid());
        } else  {       // child process
                printf("this is child %i\n",getpid());
        }
        exit(0);
}

void parentCleaner ( void )  {

        printf("cleaning up parent...\n");

}
```



```
lucid@ubuntu:~/Downloads$ ./Exit1
this is parent 3262
cleaning up parent...
lucid@ubuntu:~/Downloads$ this is child 3263
```

- registers a function to clean up resource at process termination

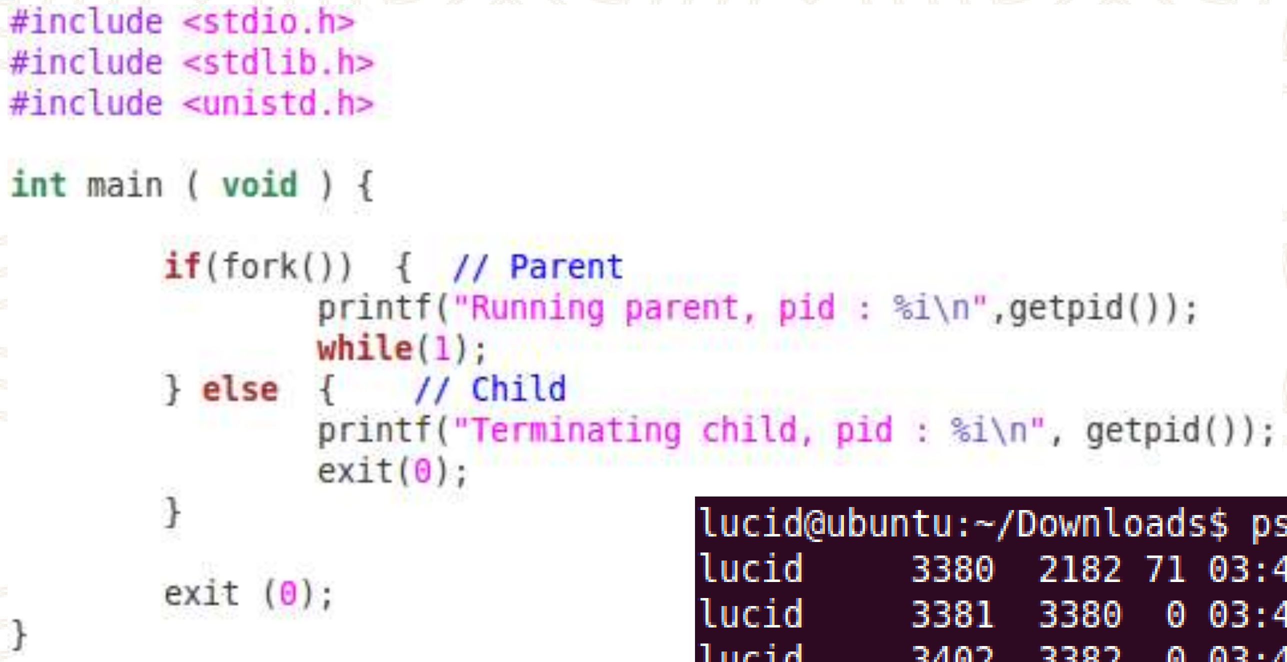
- **Ex_6_atexit.c**

Zombie Process

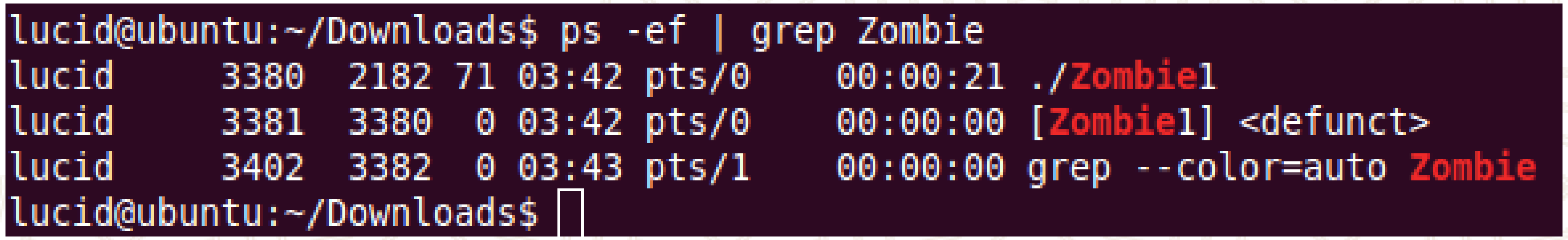
- When process terminates, still consumes system resources
  - Various tables maintained by OS
  - Called a zombie; living corpse, half alive, half dead
- **Reaping**
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process
- What if parent does not reap ?
  - if any parent terminates without reaping a child, then child will be reaped by **“init”** process
  - so, only need explicit reaping in long-running processes

Zombie example

non-terminating parent



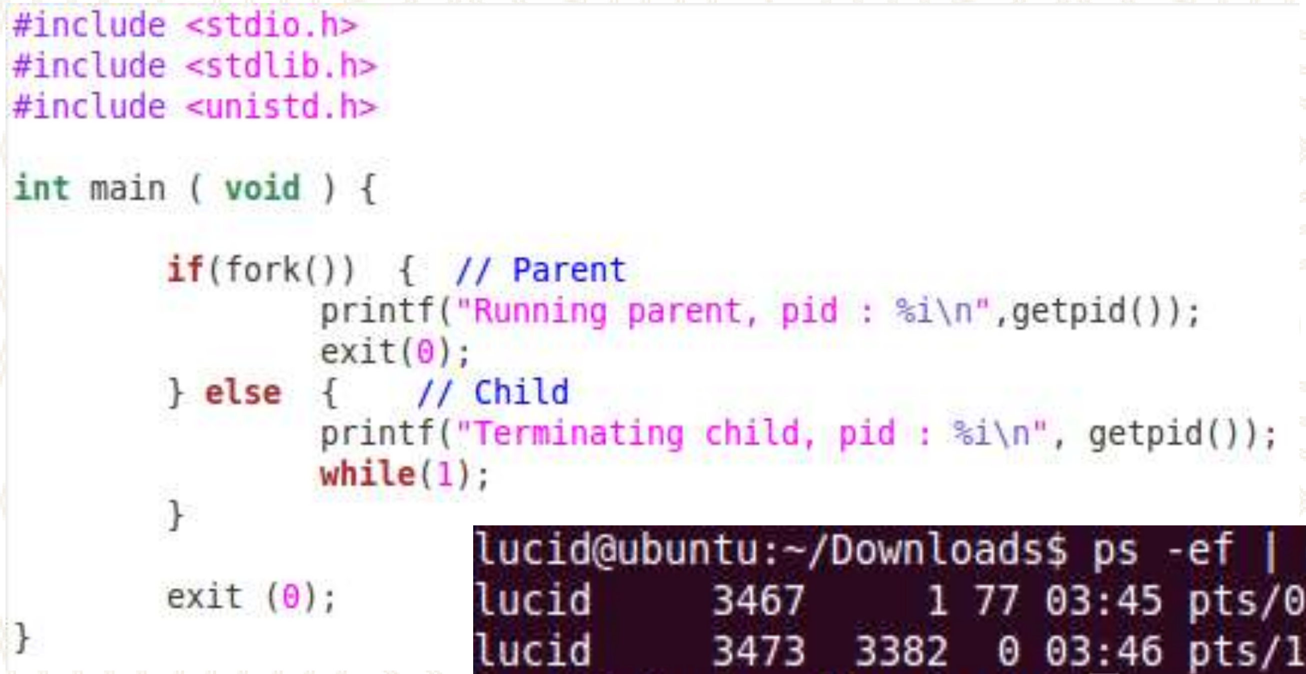
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ( void ) {

        if(fork())  {  // Parent
                printf("Running parent, pid : %i\n",getpid());
                while(1);
        } else  {     // Child
                printf("Terminating child, pid : %i\n", getpid());
                exit(0);
        }

        exit (0);
}
```



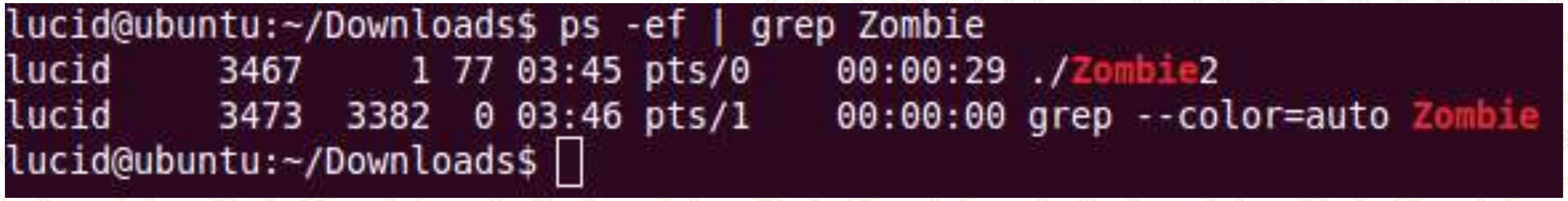
```
lucid@ubuntu:~/Downloads$ ps -ef | grep Zombie
lucid     3380  2182 71 03:42 pts/0    00:00:21 ./Zombie1
lucid     3381  3380  0 03:42 pts/0    00:00:00 [Zombie1] <defunct>
lucid     3402  3382  0 03:43 pts/1    00:00:00 grep --color=auto Zombie
lucid@ubuntu:~/Downloads$ 
```

- **Ex_8_zombie1.c**

Zombie example
non-terminating child



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

int main ( void ) {

        if(fork())  {  // Parent
                printf("Running parent, pid : %i\n",getpid());
                exit(0);
        } else  {     // Child
                printf("Terminating child, pid : %i\n", getpid());
                while(1);
        }

        exit (0);
}
```



```
lucid@ubuntu:~/Downloads$ ps -ef | grep Zombie
lucid      3467      1 77 03:45 pts/0    00:00:29 ./Zombie2
lucid      3473   3382  0 03:46 pts/1    00:00:00 grep --color=auto Zombie
lucid@ubuntu:~/Downloads$ 
```

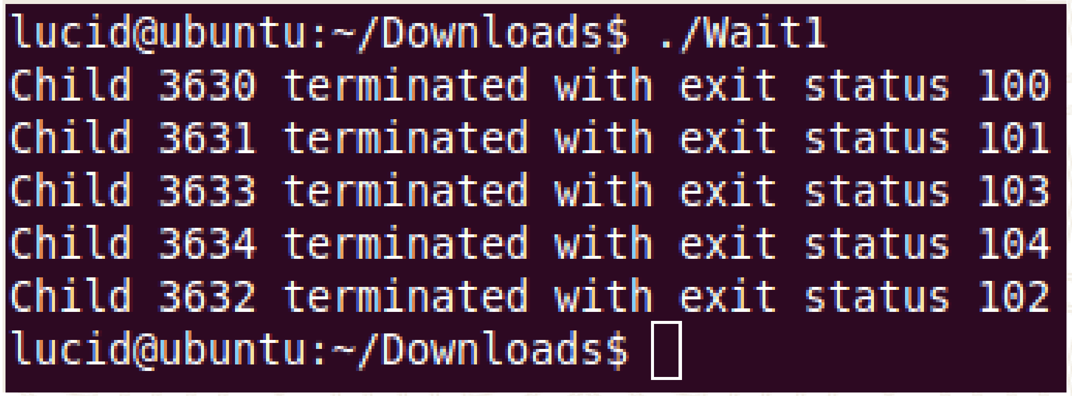
- **Ex_9_zombie2.c**

Synchronizing with child

- int wait(int *child_status)
  - suspends current process until one of its children terminates
  - return value is the pid of the child process that terminated
  - If the child has already terminated, then wait returns its pid immediately
  - If child_status != NULL, then the object it points to will be set to a status indicating why the child process terminated

wait() Example

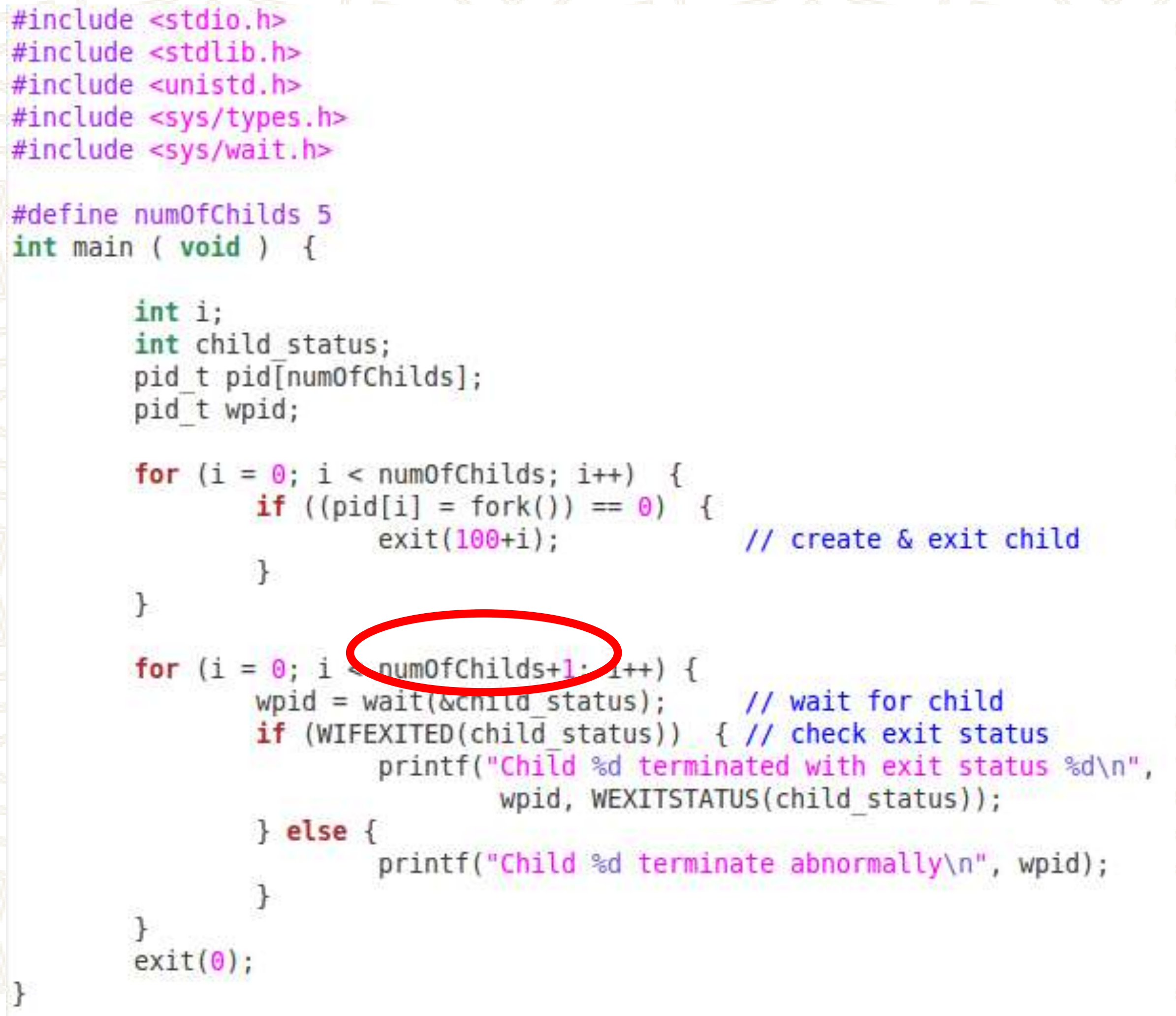
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define numOfChilds 5
int main ( void )  {

        int i;
        int child_status;
        pid_t pid[numOfChilds];
        pid_t wpid;

        for (i = 0; i < numOfChilds; i++)  {
                if ((pid[i] = fork()) == 0)  {
                        exit(100+i);            // create & exit child
                }
        }

        for (i = 0; i < numOfChilds; i++) {
                wpid = wait(&child_status);     // wait for child
                if (WIFEXITED(child_status))  { // check exit status
                        printf("Child %d terminated with exit status %d\n",
                                wpid, WEXITSTATUS(child_status));
                } else {
                        printf("Child %d terminate abnormally\n", wpid);
                }
        }
        exit(0);
}
```



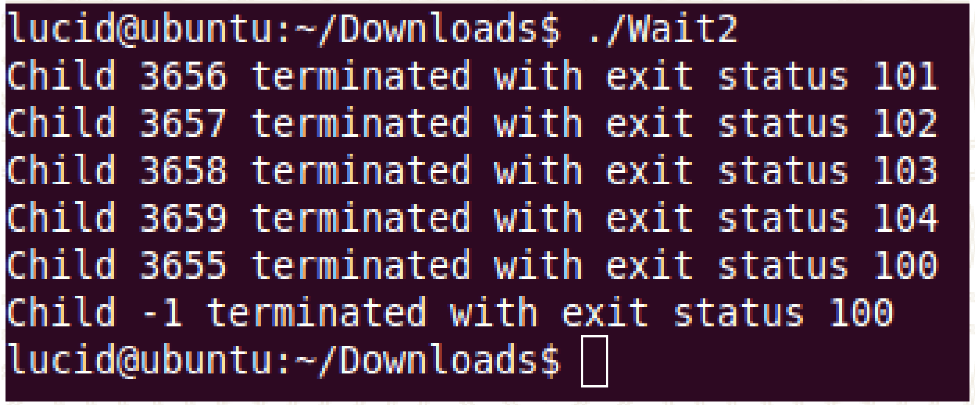
```
lucid@ubuntu:~/Downloads$ ./Wait1
Child 3630 terminated with exit status 100
Child 3631 terminated with exit status 101
Child 3633 terminated with exit status 103
Child 3634 terminated with exit status 104
Child 3632 terminated with exit status 102
lucid@ubuntu:~/Downloads$ 
```

- **Ex_10_wait1.c**

wait() Example



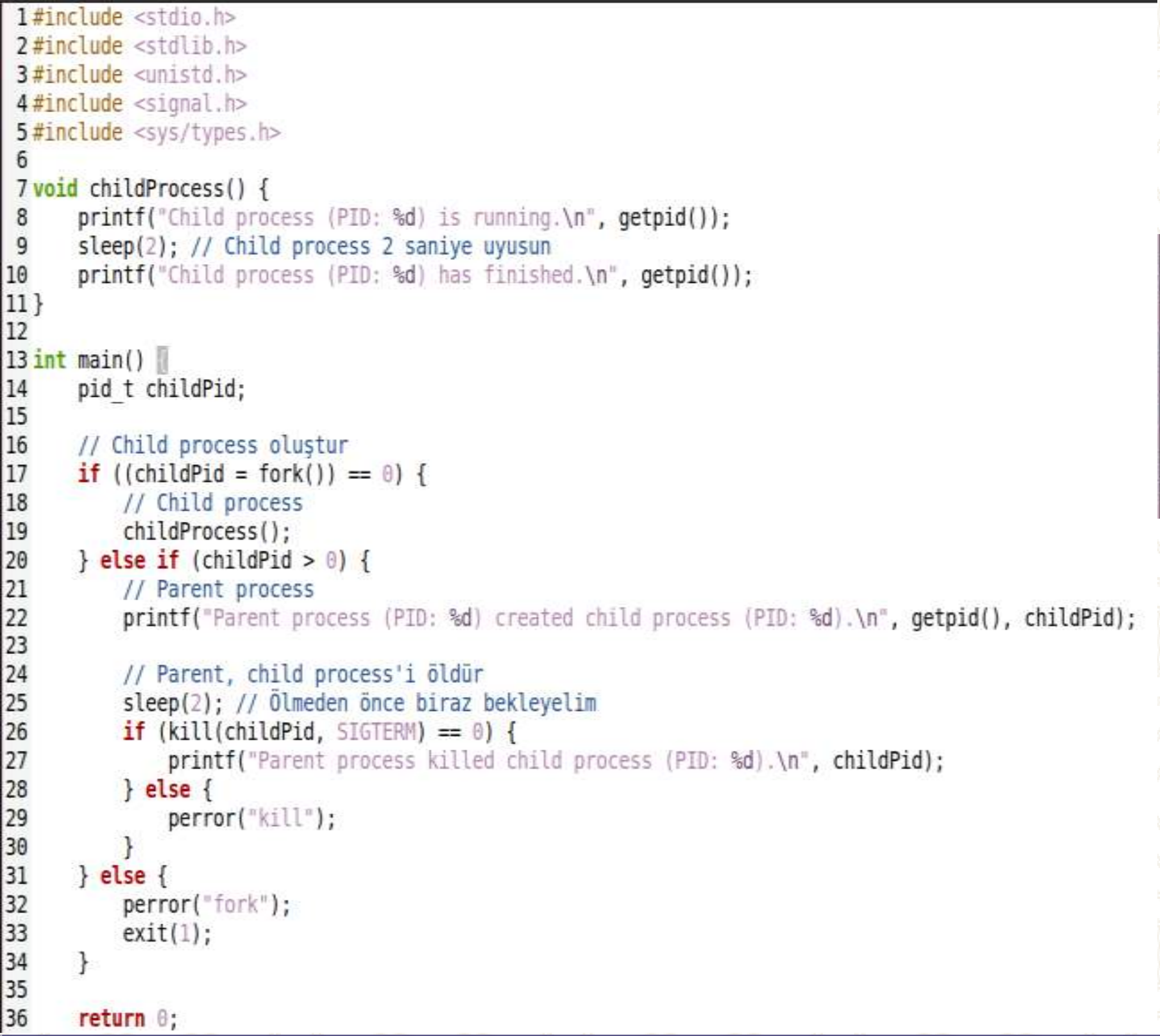
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define numOfChilds 5
int main ( void )  {

        int i;
        int child_status;
        pid_t pid[numOfChilds];
        pid_t wpid;

        for (i = 0; i < numOfChilds; i++)  {
                if ((pid[i] = fork()) == 0)  {
                        exit(100+i);            // create & exit child
                }
        }

        for (i = 0; i < numOfChilds+1; i++) {
                wpid = wait(&child_status);     // wait for child
                if (WIFEXITED(child_status))  { // check exit status
                        printf("Child %d terminated with exit status %d\n",
                                wpid, WEXITSTATUS(child_status));
                } else {
                        printf("Child %d terminate abnormally\n", wpid);
                }
        }
        exit(0);
}
```



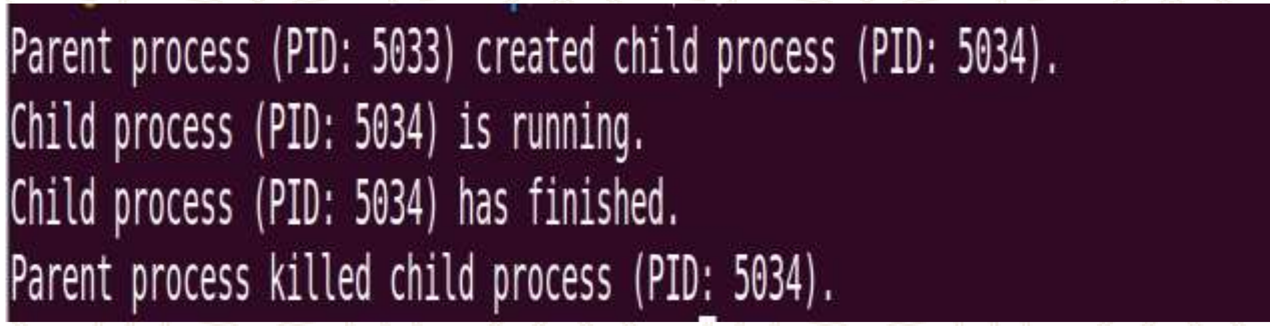
```
lucid@ubuntu:~/Downloads$ ./Wait2
Child 3656 terminated with exit status 101
Child 3657 terminated with exit status 102
Child 3658 terminated with exit status 103
Child 3659 terminated with exit status 104
Child 3655 terminated with exit status 100
Child -1 terminated with exit status 100
lucid@ubuntu:~/Downloads$ 
```

- **Ex_11_wait2.c**

kill() Example



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

void childProcess() {
    printf("Child process (PID: %d) is running.\n", getpid());
    sleep(2); // Child process 2 saniye uyusun
    printf("Child process (PID: %d) has finished.\n", getpid());
}

int main() {
    pid_t childPid;

    // Child process oluştur
    if ((childPid = fork()) == 0) {
        // Child process
        childProcess();
    } else if (childPid > 0) {
        // Parent process
        printf("Parent process (PID: %d) created child process (PID: %d).\n", getpid(), childPid);

        // Parent, child process'i öldür
        sleep(2); // Ölmeden önce biraz bekleyelim
        if (kill(childPid, SIGTERM) == 0) {
            printf("Parent process killed child process (PID: %d).\n", childPid);
        } else {
            perror("kill");
        }
    } else {
        perror("fork");
        exit(1);
    }

    return 0;
```



```
Parent process (PID: 5033) created child process (PID: 5034).
Child process (PID: 5034) is running.
Child process (PID: 5034) has finished.
Parent process killed child process (PID: 5034).
```

- **Ex_12_kill.c**

References

- **man pages**
- http://www.cs.princeton.edu/courses/archive/fall01/cs217/slides/process.pdf
- http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15213-f08/www/lectures/lecture-11.pdf
- http://csapp.cs.cmu.edu
- http://homepage.smc.edu/morgan_david/linux/a12-processes.pdf
- https://www.geeksforgeeks.org/fork-system-call/amp/