Process & Pipe

Matt Warner

1 Unit Overview

- Process Management
  - create new process
  - change what process is doing
- Pipe concept
  - enables inter-process communication

2 Process Management System Calls

- **fork**
  - create a new process
- **wait**
  wait for a process to terminate
- **exec**
  - execute a program

2.1 System Call: fork

- creates new process that is duplicate of current process
- new process is ***almost*** the same as current process

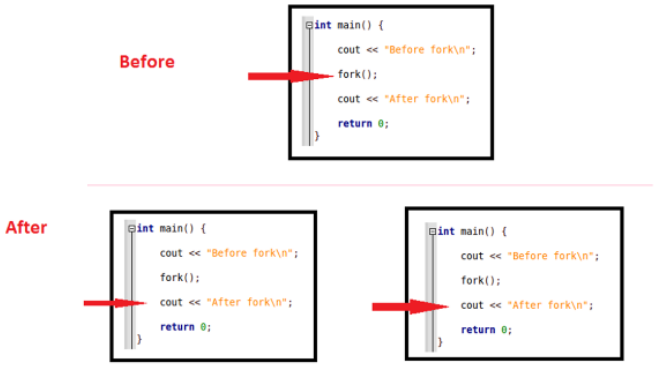
- new process is ***child*** of current process
- old process is ***parent of new process***

- after call to fork, both processes run concurrently

Example:

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

int main() {

  std::cout << "Before fork\n";

  fork();

  std::cout << "After fork\n";

  return 0;

 /*
    ***Output***
    Before fork
    After fork
    After fork
 */
}
```

Note: "After fork" gets printed twice because fork() duplicates the current process. Also, fork() child process will execute the remaining part of the code after the call to fork() so "Before fork" only gets printed once.



Its also worth mentioning that the pid of the child process is 0, indicating a newly created child process, so if we had:

```
pid_t p = fork;

std::cout << p << std::end;
```

The output would be something like:

4412
0

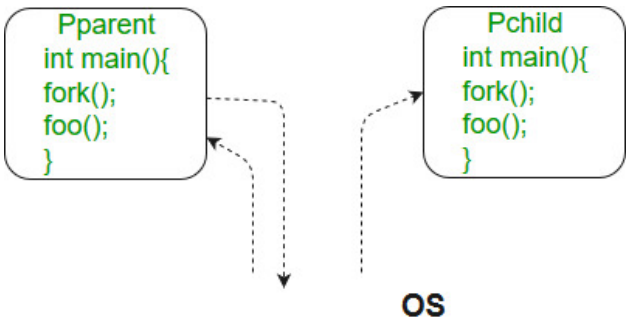
3 Fork() in depth

The Fork system call is used for creating a new process in Linux, and Unix systems, which is called the ***child process***, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call.

The child process uses the same pc(program counter), same CPU registers, and same open files which use in the parent process. It takes no parameters and returns an integer value.

Below are different values returned by fork().

- ***Negative Value:*** The creation of a child process was unsuccessful.
- ***Zero:*** Returned to the newly created child process.
- ***Positive value:*** Returned to parent or caller. The value contains the process ID (PID) of the newly created child process



Note:-

fork() is threading based function, to get the correct output run the program on a local system

Example:

```
pid_t p = fork();
if (p < 0) {
  perror("fork fail");
  exit(1);
}
```

```
std::cout << "Hellow world!" << "process_id(pid) = " <<  getpid() << std::endl;
```

There are a couple things to note here. For starters, the fork system call is included in the unistd.h lib.

Secondly, we call the fork() function and set its return value equal to a variable of type ***pid_t***, which is used to store the process id (PID) returned from the function call.

Note:-

The type of ***pid_t*** is a ***signed int***

The header file which is needed to to used ***pid_t*** is ***sys/types.h***

After obtaining the process id, we check for failure (if the pid is negative value)

We then print out the pid using the getpid function found in ***sys/types.h***

3.1 PID functions

- ***getpid()*** - this function returns the process id of the calling process.
- ***getppid()*** - this function returns the process id of the parent process.

Since we know that the returned value of fork() for the child process is 0, we can implement a section of code that will only execute within the child process by comparing pid to 0.

That would look something like this:

```
{
    pid_t p;
    p = fork();
    if(p<0)
    {
      perror("fork fail");
      exit(1);
    }
    // child process because return value zero
    else if ( p == 0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

4 Systel Call: wait

Syntax:

```
pid_t wait(int *status)
```

- lets parent process wait untill a child process terminates
    - parent is resumed once a child process terminates
- returns process id of terminated child
    - return -1 if there is no child to wait for
- **status** holds exit status of child
    - can be examined with **WEXITSTATUS(status)**

Example:

```
int pid, status;

std::cout << "Before fork\n" << std::endl;

fork();

pid = wait(&status);

if (pid == -1) {
  std::cout << "nothing to wait for\n";
  return 255;
} else {
  std::cout << "done waiting for " << pid << ". ended with: " << WEXITSTATUS(status) << std::endl;
}

std::cout << "After fork\n";

/*
  *** Output ***
  Before fork
  nothing to wait for
  done waiting for: 23550, ended with: 255
  After fork
*/
```

5 System Call: exec

- family of functions that replace current process image with a new process image
    - actual system call: **execve**
    - library functions
        - **execl, execlp, execle**
        - **execv, execvp**
- arguements specify new executable to run, plus its arguments and enviroment.

5.1 C Library Function: execl

Syntax:

```
int execl(const char *path, const char *arg, ...)
```

- starts executable for command spefified in **path**
- new executable runs in current process
- **path** is specified as absolute path
- arguments are specified as list, starting at argv[0],
  terminated with (char *NULL)
- new executable keeps same enviroment
- returns -1 on error

Example:

```
#include <unistd.h>
#include <sys/types.h>
#include <cstdio>
#include <cstdlib>
#include <iostream>

int main() {
  int rs;

  cout << "Program started in process " << getpid() << endl;

  rs = execl("/bin/ps", "ps", (char *)NULL);

  if (rs == -1) {
    perror("execl");
    exit(rs);
  }
  cout << "Maybe we see this ?\n";
  return 0;
}
```

5.2 C Library Functions: exec family

- **execl, execlp, execle**
  - specify arguments and enviroment as list
- **execv, execvp**
  - specify arguments and enviroment as array of string values
- **execlp, execvp**
  - look for new executable via PATH

Example:

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

int rs; // return status

cout << "program started in process: " << getpid();

rs = execvp("ls", argv);
if (rs == -1) {
  perror("execvp");
  exit(rs);
}

cout << "Maybe we see this ?\n";
```

6 Together: fork and exec

- UNIX does not have a single system call to spawn a new additional process with a new executable
- Instead, there are two steps that must be used:
  - fork to duplicate current process
  - execo to morph child process into new executable

Example:

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

#include <cstdio>
#include <cstdlib>
#include <iostream>

int main(int argc, char **argv) {
  int rs, pid, status;

  // Create child process
  pid = fork();

  // Error checking
  if (pid == -1) {
    perror("fork");
    exit(pid);
  }
  if (pid == 0) { // child process
    rs = execvp("echo", argv)
    if (rs == -1) {
      perror ("execvp");
      exit(rs);
    }
  } else {
    cout << "done waiting for: " << wait(&status) << endl;
  }
  return 0;
}
```

7 UNIX Pipe

- Can create a software ***pipline:***

  set of processes chained by their standard IO

- Output of one process becomes input of second process

command line example:

```
ls | wc
```

8 System Call: pipe

```
int pipe(int pipefd[2])
```

- Creates a ***channel*** to transport data
- Has direction: one side to write, one side to read
  - available via 2 file descriptors **pipefd[2]**
  - read side **pipefd[0]**
  - write side **pipefd[1]**

Example:

```
#include <unistd.h>
#include <iostream>
#include <cstdlib>
int main() {
  cout << "Before pipe\n";

  int pipefd[2], rs;

  rs = pipe(pipefd);
  if (rs == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
  }

  write(pipefd[1], "Hello", 6);

  char buffer[256];
  read(pipefd[0], buffer, sizeof(buffer));

  cout << "pip contained: "  << buffer << endl;

  return 0;

}
```