Process & Pipe I

1. Process Pipe I

1.1 CSCI 330

1.2 Unit Overview

Unit Overview

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

1.3 Process Management System Calls

Process Management System Calls

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

1.4 System Call: fork

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

1.5 Example: fork system call

Example: fork system call

forkSimple.cxx ×

```
/*
 * forkSimple.cxx
 *
 *  Example Program for CSCI 330
 *  shows fork system call
 *
 */
#include <unistd.h>
#include <iostream>
using namespace std;

int main() {

    cout << "Before fork\n";

    fork();

    cout << "After fork\n";

    return 0;
}
```

1.6 Timeline: fork system call

Timeline: fork system call

```
int main() {
    cout << "Before fork\n";
    fork();
    cout << "After fork\n";
    return 0;
}
```

1.7 Return value: fork system call

Return value: fork system call

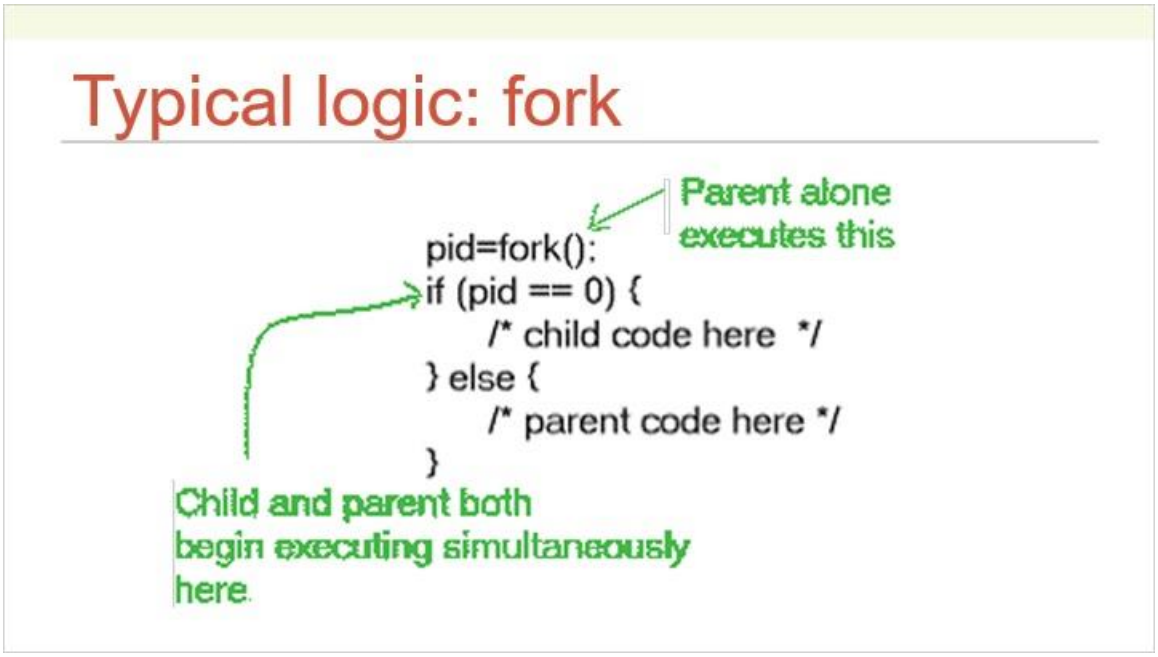
- new process is almost the same as current process

```
pid_t fork(void)
```

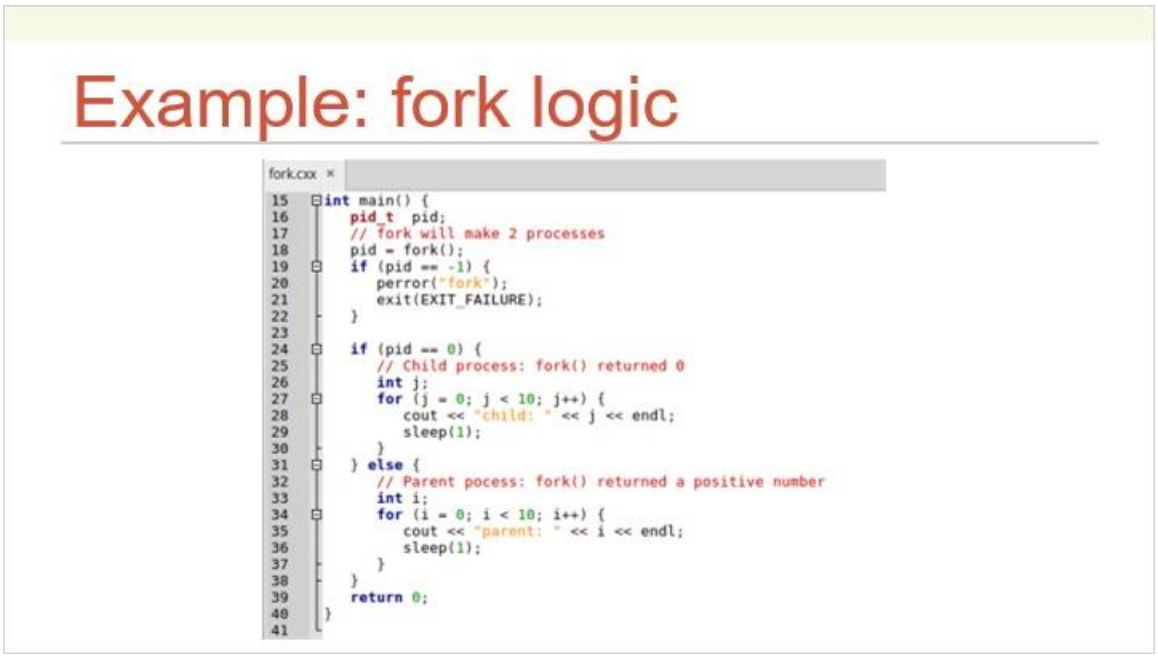
return value of fork is different:
parent: fork returns process id of child process
child: fork returns 0

- fork returns -1 on failure

1.8 Typical logic: fork



1.9 Example: fork logic



1.10 System Call: wait

System Call: wait

`pid_t wait(int *status)`

- lets parent process wait until 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)`**

1.11 Example: wait system call

Example: wait system call

forkWait.cxx

```
#include <unistd.h>
#include <iostream>
using namespace std;

int main() {
    int pid, status;

    cout << "Before fork\n";

    fork();

    pid = wait(&status);
    if (pid == -1)
        cout << "nothing to wait for\n";
    else
        cout << "done waiting for: " << pid << endl;

    cout << "After fork\n";

    return 0;
}
```

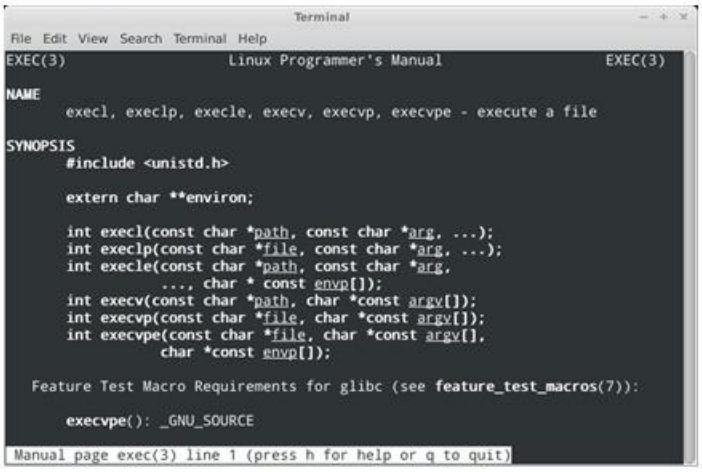
1.12 System Call: exec

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`**
- arguments specify new executable to run, plus its arguments and environment

1.13 C Library Functions: exec

C Library Functions: exec



1.14 C Library Function: execl

C Library Function: execl

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

- starts executable for command specified 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 environment
- returns -1 on error

1.15 Example: execl

Example: execl

execSimple.cxx

```
3
4   #include <cstdio>
5   #include <cstdlib>
6   #include <iostream>
7   using namespace std;
8
9   int main() {
10
11      int rs;
12
13      cout << "program started in process: " << getpid() << endl;
14
15      rs = execl("/bin/ps", "ps", (char *)NULL);
16      if (rs == -1) {
17          perror("execl");
18          exit(rs);
19      }
20      cout << "Maybe we see this ?\n";
21
22      return 0;
23  }
```

1.16 C Library Functions: exec family

C Library Functions: exec family

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

1.17 Example: execvp

Example: execvp

execVP.cxx ×

```
16  int main(int argc, char* argv[]) {
17
18      int rs;
19
20      cout << "program started in process: " << getpid() << endl;
21
22      rs = execvp("ls", argv);
23      if (rs == -1) {
24          perror("execvp");
25          exit(rs);
26      }
27      cout << "Maybe we see this ?\n";
28
29      return 0;
30  }
31
```

1.18 Together: fork and exec

Together: fork and exec

- UNIX does not have a single system call to spawn a new
  additional
  process with a new executable

- instead: 2 steps
  - fork to duplicate current process
  - exec to morph child process into new executable

1.19 Example: fork and exec

Example: fork and exec

forkExec.cxx

```
10  int main(int argc, char* argv[]) {
11
12      int rs, pid, status;
13
14      pid = fork();
15      if (pid == -1) {
16          perror("fork");
17          exit(pid);
18      }
19      if (pid == 0) { //child process
20          rs = execvp("echo", argv);
21          if (rs == -1) {
22              perror("execvp");
23              exit(rs);
24          }
25      } else {    // parent process
26          cout << "done waiting for: " << wait(&status) << endl;
27      }
28
29      return 0;
30  }
```

1.20 UNIX Pipe

UNIX Pipe

- can create a software pipeline:
  set of processes chained by their standard IO
- output of one process becomes input of second process

command line example:

```
ls | wc
```

implemented via **pipe** system call

1.21 System Call: pipe

System Call: pipe

```
Terminal
File  Edit  View  Search  Terminal  Help
PIPE(2)                    Linux Programmer's Manual                    PIPE(2)

NAME
       pipe, pipe2 - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);

DESCRIPTION
       pipe() creates a pipe, a unidirectional data channel that can be used
       for interprocess communication.  The array pipefd is used  to  return
       two  file  descriptors  referring to the ends of the pipe.  pipefd[0]
       refers to the read end of the pipe.  pipefd[1] refers  to  the  write
       end  of  the  pipe.  Data  written  to  the write end of the pipe is
       buffered by the kernel until it is read from  the  read  end  of  the
 Manual page pipe(2) line 1 (press h for help or q to quit)
```

1.22 System Call: pipe

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]`**
- can be used synchronize producer and consumer of data

1.23 Example: pipe

Example: pipe

pipeSimple.cxx ×

```
15  int main() {
16
17      cout << "Before pipe\n";
18
19      int pipefd[2], rs;
20
21      rs=pipe(pipefd);
22      if (rs == -1) {
23          perror("pipe");
24          exit(EXIT_FAILURE);
25      }
26
27      write(pipefd[1],"Hello", 6);
28
29      char buffer[256];
30      read(pipefd[0], buffer, sizeof(buffer));
31
32      cout << "pipe contained: " << buffer << endl;
33
34      return 0;
35  }
36
```

1.24 Process communication: pipe & fork

Process communication: pipe & fork

- Idea: read and write end of pipe in different processes
  - fork creates two processes
- parent process:
  - close read end of pipe
  - write to write end of pipe
- child process:
  - close write end of pipe
  - read from read end of pipe

1.25 Example: pipe & fork

Example: pipe & fork

pipeFork.cxx ×

```
int main() {

    int pipefd[2], rs;
    char buffer[256];

    // create pipe
    rs = pipe(pipefd);
    if (rs == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    cout << "pipe created\n";

    // fork into 2 processes
    rs = fork();
    if (rs == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
```

1.26 Summary

Summary

- Process Management
  - create new process (fork)
  - wait for child process (wait)
  - change what a process is doing (exec)
- Pipe concept
  - Pipe for inter-process communication