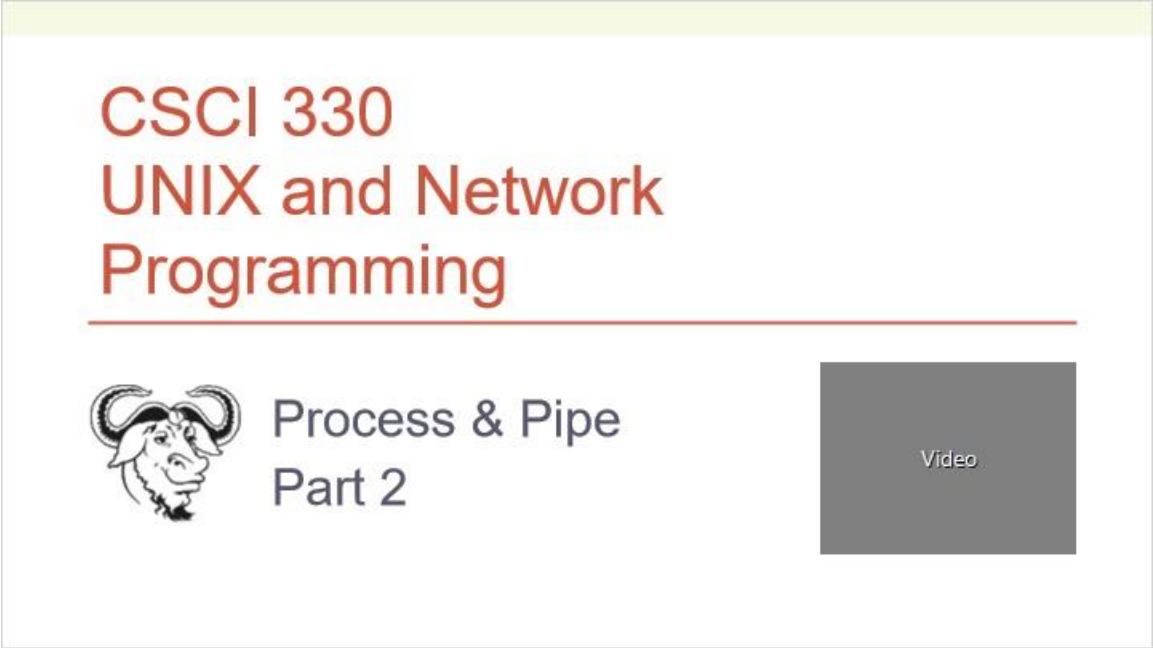
Process & Pipe II

1. Process Pipe II

1.1 CSCI 330



1.2 Unit Overview

Unit Overview

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

1.3 System Calls

System Calls

- fork
  - create a new process
- wait
  - wait for a process to terminate
- exec
  - execute a program
- pipe
  - establish communication channel
- dup
  - duplicate file descriptor

1.4 System Call: fork example

System Call: fork example

forkLoopWait.cxx ×

```
int main() {
   pid_t  pid;
   // fork will make 2 processes
   pid = fork();
   if (pid == -1) { perror("fork"); exit(EXIT_FAILURE); }

   if (pid == 0) {
      // Child process: fork() returned 0
      for (int j = 0; j < 5; j++) {
        cout << "child: " << j << endl;
      }
   } else {
      // Parent pocess: fork() returned a positive number
      // wait for child to end
      wait(NULL);
      for (int i = 0; i < 5; i++) {
        cout << "parent: " << i << endl;
      }
   }
   cout << "end of process:" << getpid() << endl;
   return 0;
}
```

1.5 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 and its arguments and environment

1.6 C Library Function: execlp

C Library Function: execlp

```
int execlp(const char *cmd, const char *arg, ...)
```

- starts executable for command specified in **cmd**
- new executable runs in current process
- **cmd** executable is found via path
- arguments are specified as list, starting at argv[0], terminated with **(char *NULL)**
- return -1 on error

1.7 Together: fork and exec

Together: fork and exec

- UNIX does not have a system call to spawn a new <u>additional</u> process with a new executable
- instead:
  - fork to duplicate current process
  - exec to morph child process into new executable

1.8 Together: fork and exec

Together: fork and exec

ForkRunCommand.cxx ×

```
int main() {

   string command, argument;
   // prompt user for command to run
   cout << "Enter command to run (with one argument): ";
   cin >> command >> argument;

   int pid, rs, status;
   // fork will make 2 processes
   pid = fork();
   if (pid == -1) { perror("fork"); exit(EXIT_FAILURE); }

   if (pid == 0) {
      // Child process: exec to command with argument
      cout << "before exec to: " << command << " " << argument << endl;

      rs = execlp(command.c_str(), command.c_str(), argument.c_str(), (char*) NULL);

      if (rs == -1) { perror("execlp"); exit(EXIT_FAILURE); }
   } else {
      // Parent process: wait for child to end
      wait(&status);
   }
   cout << "end of process: " << getpid() << endl;
   return 0;
}
```

1.9 UNIX Pipe

UNIX Pipe

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

<u>command line example:</u>

```
ls | wc
```

implemented via **`pipe`** system call

1.10 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

1.11 System Calls: pipe and fork

System Calls: pipe and fork

- Idea: read and write end of pipe in different processes
  - single process creates pipe
  - 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

1.12 System Call: pipe & fork example

System Call: pipe & fork example

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("fork"); exit(EXIT_FAILURE); }

    if (rs == 0) { // child process
        // close write end of pipe
        close(pipefd[1]);
        // read from read end of pipe
        read(pipefd[0], buffer, sizeof(buffer));
        cout << "pipe contained: " << buffer << endl;
    } else {  // parent process
        // close read end of pipe
        close(pipefd[0]);
        // write to write end of pipe
        write(pipefd[1],"Hello", 6);
        wait(NULL);
    }
    return 0;
}
```

1.13 Question: how to implement "|"

Question: how to implement "|"

- shell allows pipe on command line
  - connects output of one command to input of second command

  Example: `ls | wc`

- implemented with system calls:
  - `fork, pipe, dup, exec`

1.14 Command One: ls

Command One: ls

- shell creates pipe
- shell forks:
  - parent keeps going
  - child
    - closes standard output
    - dups pipefd[1]
    - execs to command one: ls

1.15 Command Two: wc

Command Two: wc

- shell forks again:
  - parent keeps going
  - child
    - closes standard input
    - dups pipefd[0]
    - execs to command two: wc
- parent (shell) waits for children to complete

1.16 Illustration: shell pipe implementation

Illustration: shell pipe implementation

- Idea:
  - create 2 child processes
  - 2 processes communicate via pipe
  - each process exec's into new executable
- Example: **`ls | wc`**
  - child process 1: runs **`ls`**
  - child process 2: runs **`wc`**

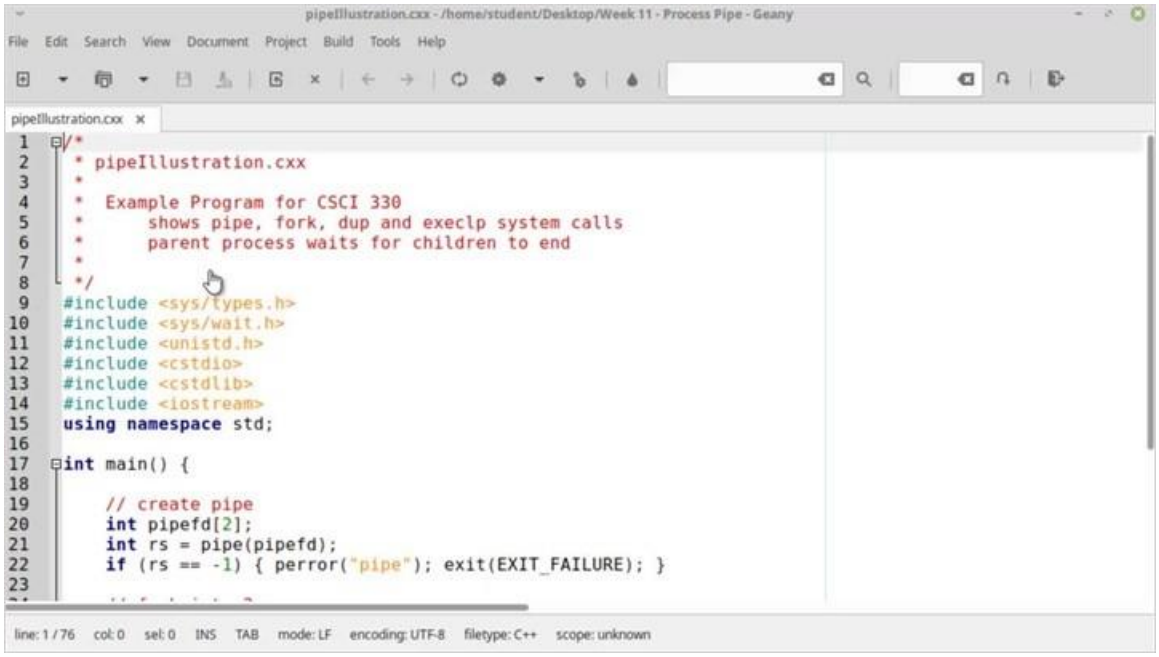
1.17 Illustration

Illustration

- create pipe
- run command 1 in child
  - with pipefd[1] as output
- run command 2 in child
  - with pipefd[0] as input
- wait for children

```
int main() {

    // create pipe
    int pipefd[2];
    int rs = pipe(pipefd);
    if (rs == -1) { perror("pipe"); exit(EXIT_FAILURE); }

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

    if (rs == 0) { // child process 1

        // close read end of pipe, keep write end open
        close(pipefd[0]);
        // close standard output
        close(1);
        // duplicate write end of pipe into standard output
        dup(pipefd[1]);
        // close write end of pipe
        close(pipefd[1]);
        // run first command
        rs = execlp("ls", "ls", (char*)NULL);
        if (rs == -1) { perror("execlp"); exit(EXIT_FAILURE); }

    } else {  // parent process

        // close write end of pipe, keep read end open
        close(pipefd[1]);

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

        if (rs == 0) { // child process 2
            // close standard input
            close(0);
            // duplicate read end of pipe into standard input
            dup(pipefd[0]);
            // close read end of pipe
            close(pipefd[0]);

            // run second command
            rs = execlp("wc", "wc", (char*)NULL);
            if (rs == -1) { perror("execlp"); exit(EXIT_FAILURE); }

        } else { // parent process

            // close read end of pipe
            close(pipefd[0]);

            wait(NULL); // for first child
            wait(NULL); // for second child
            cout << "Parent Donezo";

        }
    }
    return 0;
}
```

1.18 Pipe Illustration



1.19 Summary