CSci 4061: Introduction to Operating Systems

Recitation 7
Pipes
23 Oct 2017

TA – Manu Khandelwal

Today's Agenda

- pipe() and exercises
- dup(), dup2() and exercises
- Signals

pipe()

- *man pipe*
- pipe – create descriptor pair for inter-process communication
- A pipe is a unidirectional communication channel between UNIX processes. A pipe can be written to on one end and read from at the other.
- Usage

```
#include <unistd.h>
int pipe(int fildes[2]);
```

pipe() – Contd.

- pipe() - creates a pipe and place two file descriptors, one each into the arguments fildes[0] and fildes[1], that refer to the open file descriptions for the read and write ends of the pipe.
- Their integer values shall be the two lowest available at the time of the pipe() call.
- Reference:
- http://www.opengroup.org/onlinepubs/009695399/functions/pipe.html

Establishing inter-process communication between parent and child process through pipe mechanism.

Problem Statement

Write a program to create parent and child process where parent process write a string to pipe which is read by child process

Resources: parentwritepipe.c

parentwritepipe.c

```
int main(void) {
    char bufin[BUFSIZE] = "empty";
    char bufout[] = "hello";
    int bytesin;
    pid_t childpid;
    int fd[2];
    int pipe_creation_status;  //System call to create a pipe
    pipe_creation_status = ~~~~~~;
    if ( pipe_creation_status == -1)  {
            perror("Failed to create the pipe");
            return 1;
    }
```

parentwritepipe.c

```
bytesin = strlen(bufin);
//System call to create a child process
childpid = ~~~~~~;
if (childpid == -1)  {
     perror("Failed to fork");
     return 1;
}
// Check whether it is parent or child
 if (~~~~)  {                              /* parent code */
     write(~~~~, ~~~~, ~~~~~);
}   else    {                                        /* child code */
     bytesin = read(~~~~, ~~~~, ~~~~);
}
fprintf(stderr, "[%ld]:my bufin is {%.*s}, my bufout is {%s}\n",  (long)getpid(), bytesin,
     bufin, bufout);
     return 0;
}
```

dup()

Creates a copy of the file descriptor

Usage: int dup(int oldfd)
oldfd: File descriptor being aliased

On success returns the value (lowest-numbered unused descriptor) of the new file descriptor

On failure -1 integer value is returned with errno set
man dup

dup2()

Same as dup but instead of using the lowest-numbered unused file descriptor, it uses the descriptor number specified in newfd

Usage: int dup2(int oldfd, int newfd)
   fides: File descriptor being aliased

On success returns newfd

On failure -1 integer value is returned with errno set
man dup

Problem Statement

Create a parent-child process where parent execute ls –l command. The output is received by child process and it executes sort -nr -k5 on it.

Idea: You will need to create pipe between these 2 process and dup to overwrite STDIN and STDOUT

Resources: simpleredirect.c

simpleredirect.c

```
int main(void) {
	pid_t childpid;
	int fd[2];
	int pipe_creation_status;
	int dup_status;
	int close_desc_status;

	 // System call to create a pipe
	pipe_creation_status = ~~~~;
	 if ( pipe_creation_status == -1) {
			perror("Failed to create the pipe");
			return 1;
	}

	// System call to create a child process
	childpid = ~~~~;
	if (childpid == -1) {
			perror("Failed to fork");
			return 1;
	}
```

simpleredirect.c

```
// Check whether it is parent or child
if (~~~~) {                             /* ls is the child */
      dup_status = ~~~~; //Duplicate the file descriptor fd[1] with STDOUT_FILENO
      if (dup_status == -1) {
            perror("Failed to redirect stdout of ls");
      }

      //Close the file descriptors created by pipe system call
      close_desc_status = ~~~~;      //Close descriptor fd[0]
      if (close_desc_status == -1) {
            perror("Failed to close fd[0] descriptors on ls");
      }

      close_desc_status = ~~~~);     //Close descriptor fd[1]
      if (close_desc_status == -1) {
            perror("Failed to close fd[1] descriptors on ls");
       }
      execl(~~~~); //Execute the ls -l command using execl system call
      perror("Failed to exec ls");
}
```

simpleredirect.c

```
else {          /* sort is the parent */
        dup_status = ~~~~;              //Duplicate the file descriptor fd[0] with STDIN_FILENO
if (dup_status == -1) {
       perror("Failed to redirect stdout of ls");
 }

 //Close the file descriptors created by pipe system call
close_desc_status = ~~~~;                       //Close descriptor fd[0]
if (close_desc_status == -1) {
       perror("Failed to close fd[0] descriptors on ls");
 }

close_desc_status = ~~~~;                       //Close descriptor fd[1]
if (close_desc_status == -1) {
       perror("Failed to close fd[1] descriptors on ls");
}
        execl(~~~~);                    //Execute the sort -nr -k5 command using execl system call
       perror("Failed to exec wc");
 }
return 1;    }
```

Signals

A **signal** is a software interrupt delivered to a process.

You can define a handler function and tell the operating system to run it when that particular type of signal arrives.

One process can send a signal to another process helping in communication and synchronization.

Program to send SIGINT signal to process by pressing Ctrl+C during its execution and writing custom handler for that event

signal.c

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

void sig_handler(int signum) {
        printf("Received signal %d\n", signum);
}

int main() {
        signal(SIGINT, sig_handler);
        sleep(10); // This is your chance to press CTRL-C
        return 0;
}
```

Program where parent process terminates all the child process in case any child process return non-zero exit status

child_termination.c

```
void sig_handler(int signo){
        if (~~~~) {
                printf("received SIGINT and exiting\n");
        }
        exit(1);
}

void main ( void ) {
        int status;
        pid_t waitreturn;
        pid_t first_childpid;
        pid_t second_childpid;

        signal(SIGINT, sig_handler);

        //Parent creating first child process
```

child_termination.c

```
first_childpid = ~~~~; // Parent creating first child

if( first_childpid == 0) { // Inside the first child process

        printf("Child 1: Sleeping for 10000 seconds\n");
        ~~~~;
} else { //Inside the parent

        printf("Parent: Created child 1 with  process ID %d \n", first_childpid);
        second_childpid = ~~~~; // Parent creating second child
        if( second_childpid == 0) {  // Inside the second child process

                printf("Child 2: Sleeping for 5 seconds\n");
                ~~~~; // Sleep for 5 seconds
                printf("Child 2: Woke up and exit with status 3\n");
                ~~~~;
        } else { //Inside the parent
                printf("Parent: Created child 2 with process ID %d\n",
                second_childpid);
        }
}
```

child_termination.c

```
waitreturn = ~~~~; //Waiting for any child termination

if( WEXITSTATUS(status) != 0 ) {
        printf("Parent: Received exit status %d \n", WEXITSTATUS(status));
        printf("Parent: Sending kill signal to first child\n");
        kill(~~~~);
        printf("Parent: Sending kill signal to second child\n");
        kill(~~~~);
}

return;
}
```