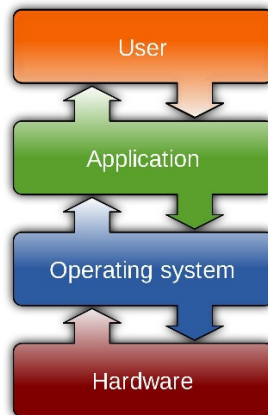


Operating Systems

Lab assignment 3: Inter-process Communication – Pipes



Objectives

1. To develop multi-process multi-task application programs
2. To demonstrate the use of pipes as an inter-process communication (IPC) mechanism

Guidelines

In Lab2, you have learned to develop multi-processing applications using `fork()`, and `exec()` respectively. With `fork()` the kernel creates and initializes a new process control block (PCB) with a new address space that copies the entire content of the address space of the parent process. The new process is a child process and it inherits the execution context of the parent (e.g. open files). These processes do not share their address space. However, processes can use an IPC mechanism to exchange information. The kernel provides three IPCs, namely: Pipes, Shared Memory, and Message Queues.

Pipes are traditional Unix inter-process communication. The `|` symbol is used in the command line to denote a pipe. Try the following:

- `who | sort`
- `ls | more`
- `cat /etc/passwd | grep root`

Pipes can be created in a C program using the `int pipe(int P[2])` system call, where:

- `P[1]`: file descriptor on upstream end of pipe
- `P[0]`: file descriptor on downstream end of pipe

Typically Process A (parent) creates a pipe, forks twice, creating B and C. Each process closes the ends of the pipe it does not need. Process B closes the downstream end, process C closes the upstream end, and process A closes both ends. Processes B and C execute other programs, using `exec` (file descriptors are retained). You may need to use `int dup2(int OldFileDes, int NewFileDes)`; for redirection of the standard file descriptors, (e.g. 0 (stdin), 1(stdout)).

With shared memory, one process creates a shared segment using `id = shmget(key, MSIZ, IPC_CREAT | 0666)`; system call, other process(es) get ID of the created segment `id = shmget(key, MSIZ, 0)`. Using the same key, each process attaches to itself the created shared segment using `ctrl = (struct info *) shmat(id, 0, 0)`; , then process can begin to read and write to the shared memory segment. At the end processes detach the segment.

With message queues, one process creates a message queue using `id = msgget(key, IPC_CREAT | 0644)`;

system call. Other process(es) get ID of the created segment using `id = shmget(key,0)` system call. Processes send a message to a queue and receive a message from a queue, using `v = msgsnd(msid, ptr, length, IPC_NOWAIT)`; and `v = msgrcv(msid,ptr,length,type,IPC_NOWAIT)`; system calls, respectively.

You will not use shared memory nor message queues for IPC in this assignment.

In this lab, you will practice and use pipes for IPC. Include the following libraries for IPC.

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

```
#include <sys/ipc.h>
```

C Program with pipe IPC

Demonstrate each of the following steps to the TA to get a grade on this part of the lab assignment

Step 1. Compile and run the following program

```
/*Sample C program for Lab assignment 3*/
```

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

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

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

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

```
//main
```

```
int main() {
```

```
    int fds[2];
```

```
    pipe(fds);
```

```
    /*child 1 duplicates downstream into stdin */
```

```
    if (fork() == 0) {
```

```
        dup2(fds[0], 0);
```

```
        close(fds[1]);
```

```
        execlp("more", "more", 0);
```

```
    }
```

```
    /*child 2 duplicates upstream into stdout */
```

```
    else if (fork() == 0) {
```

```
        dup2(fds[1], 1);
```

```
        close(fds[0]);
```

```
        execlp("ls", "ls", 0);
```

```
    }
```

```
    else { /*parent closes both ends and wait for children*/
```

```
        close(fds[0]);
```

```
        close(fds[1]);
```

```
        wait(0);
```

```
        wait(0);
```

```
    }
```

```
    return 0;
```

```
}
```

Step 2. Compile and run the following program

```
/*Sample C program for Lab assignment 3*/
```

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

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

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

```
#include <string.h>
```

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

```
// main
```

```
int main(int argc, char *argv[]){
```

```
    int fds[2];
```

```
    char buff[60];
```

```
    int count;
```

```
    int i;
```

```
    pipe(fds);
```

```
    if (fork()==0){
```

```
        printf("\nWriter on the upstream end of the pipe -> %d arguments \n",argc);
```

```
        close(fds[0]);
```

```
        for(i=0;i<argc;i++){
```

```
            write(fds[1],argv[i],strlen(argv[i]));
```

```
        }
```

```
        exit(0);
```

```
    }
```

```
    else if(fork()==0){
```

```
        printf("\nReader on the downstream end of the pipe \n");
```

```
        close(fds[1]);
```

```
        while((count=read(fds[0],buff,60))>0){
```

```
            for(i=0;i<count;i++){
```

```
                write(1,buff+i,1);
```

```
                write(1," ",1);
```

```
            }
```

```
            printf("\n");
```

```
        }
```

```
        exit(0);
```

```
    }
```

```
    else{
```

```
        close(fds[0]);
```

```
        close(fds[1]);
```

```
        wait(0);
```

```
        wait(0);
```

```
    }
```

```
    return 0;
```

```
}
```

Step 3. Write a C program that implements the shell command: `cat /etc/passwd | grep root`

Requirements to complete the lab

1. Show the TA correct execution of the C programs.
2. Submit any questions, observations, and notes as .txt file and upload to Canvas.
3. Submit the source code for all your programs as .c file(s) and upload to Canvas.

Be sure to retain copies of your .c and .txt files. You will want these for study purposes and to resolve any grading questions (should they arise).

Please start each program/ text with a descriptive block that includes minimally the following information:

```
# Name: <your name>
# Date: <date> (the day you turnin your lab)
# Title: Lab3 - task
# Description: This program computes ... <you should
# complete an appropriate description her
```

