CENG328 Operating Systems
Laboratory Chapter 3

1 Interprocess Communication 1

Remember that each process in memory has its own address space. These address spaces are independent from each other, thus a process may not access variables in address spaces of other processes. This is valid for parent-child processes as well. Remember the fork2.c code in Chapter 2:

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

int main() {
    pid_t child_pid;
    int a = 5;
    printf("a before fork: %d\n", a);
    child_pid = fork();
    if (child_pid == 0)
        a = 7;
    printf("a after fork: %d\n", a);
    return 0;
}
```

When you execute this program, you will see that you get two different values for the variable a after fork operation; because now there are two different memory locations which are accessed by variable “a”:

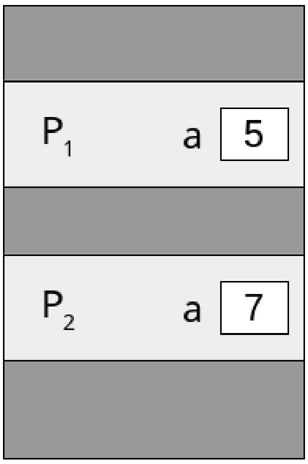

Figure 1: Variables in memory after fork

With the help of interprocess communication methods, it is possible to share information between different processes.

1.1 Communicating With Pipes

A pipe is an interprocess communication method which can be used in program codes to carry information between two processes. It acts as a file with two file descriptors; data can be written to writing-end file descriptor of a pipe and data can be read from reading-end file descriptor of a pipe. It is a unidirectional communication method; that means data can not be sent in both ways. If bidirectional communication is required, multiple pipes must be used. Study the given code below (pipe1.c):

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

int main() {
  pid_t pid1;
  int a, b;
  int pfd[2]; // pipe variable
  pipe(pfd);  // convert variable to "pipe"
              // it must be done before fork()

  pid1 = fork();
  if (pid1 > 0) {
    close(pfd[0]); // parent will only write to the pipe,
                   // therefore close the read-end
    while (1) {
      printf("Enter a number: ");
      scanf("%d", &a);
      write(pfd[1], &a, sizeof(int)); // write to write-end of pipe
      if (a < 0)
        break;
      sleep(1);
    }
    waitpid(pid1, NULL, 0);
  } else if (pid1 == 0) {
    close(pfd[1]); // child will only read from the pipe,
                   // therefore close the write-end
    while (1) {
      read(pfd[0], &b, sizeof(int));  // read from read-end of pipe
      if (b < 0)
        break;
      printf("Child received from pipe: %d\n", b);
    }
  }
  return 0;
}
```

A pipe always has to be defined as an integer array of size 2. This variable will be converted to a pipe after using it as parameter of **pipe()** system call. This conversion must be done prior to any fork() call, so that all processes will have access to this pipe later. If pipe() is used after fork(), multiple seperate pipes (which will be unable to communicate) will be created.

Figure 2: Pipe

Data must be written to a pipe with **write()** system call and it must be read with **read()** system call. These system calls take three parameters:

1. File descriptor to read/write,
2. Variable address to read/write the value,
3. How many bytes will be read/written starting from the address in second parameter.

1.2 Redirecting Standard Input/Output to Pipes

Every process has access to preset file descriptors. These are:

- File 0: Standard Input
- File 1: Standard Output
- File 2: Standard Error

Unless especially overridden, file 0 reads input from keyboard and files 1 and 2 print messages to terminal. When needed, these standard files can be overridden to perform different tasks; such as sending the screen output of a process to an another process as standard input. Study the following code (dup1.c):

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

int main() {
  pid_t pid1;
  int pfd[2];

  pipe(pfd);
  pid1 = fork();
  if (pid1 > 0) {
    char buffer;
    int lines = 0;
    close(pfd[1]);      // close write-end
    close(0);           // close stdin
    dup(pfd[0]);        // attach read-end of pipe
                        // to the first available file
    while (read(0, &buffer, 1) > 0) {
      if (buffer == '\n')
        lines++;
    }
    printf("Read %d lines.\n", lines);
  } else if (pid1 == 0) {
    close(pfd[0]);      // close read-end
    close(1);           // close stdout
    dup(pfd[1]);        // attach write-end of pipe
                        // to the first available file
    execl("/bin/ls", "ls", "-l", NULL);
  }
  return 0;
}
```

dup() system call works by attaching the specified parameter to the first available file in file descriptor table of a process. In the code above, parent closes standard input and child closes standard output files.

Therefore when dup() is used in both processes, read and write ends of the pipe get attached to stdin and stdout of mentioned processes respectively.

It is possible to use **dup2()** to specify which file descriptor should be attached to which file descriptor, instead of attaching to the first available file. Study the following code (dup2.c):

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

int main() {
  pid_t pid1;
  int pfd[2];

  pipe(pfd);
  pid1 = fork();
  if (pid1 > 0) {
    char buffer;
    int lines = 0;
    close(pfd[1]);      // close write-end
    dup2(pfd[0], 0);    // attach file 0 to read-end
    while (read(0, &buffer, 1) > 0) {
      if (buffer == '\n')
        lines++;
    }
    printf("Read %d lines.\n", lines);
  } else if (pid1 == 0) {
    close(pfd[0]);      // close read-end
    dup2(pfd[1], 1);    // attach file 1 to write-end
    execl("/bin/ls", "ls", "-l", NULL);
  }
  return 0;
}
```

This program creates a child process, which attaches its standard output to the pipe with the help of **dup2()** system call. So, whenever the child process wants to print output on screen, this output will instead be redirected to the write-end of the pipe. On the other hand, the parent process is continuously waiting for input coming from the pipe. Parent process increases a counter whenever it reads a newline character from the pipe, which will be printed on screen when the pipe is broken (child terminated).

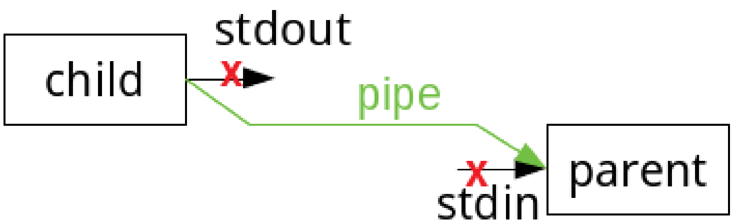

Figure 3: Proper use of pipe as stdin / stdout

2 Exercises

1. Read man pages for the following system calls: pipe, read, write, close, dup, dup2.
2. Modify pipe1.c such that child sends number+1 back to the parent as well.
3. Modify dup1.c such that the output of child process is sent to an another child process now.