Practical 4 Process Creation and Pipe Communication in Unix
CS240 - Dermot Kelly

You should be able to answer these quick questions from last week's lab:-

1. What is the difference between the string processing function scanf() and the system call function read() ?
2. What does the numeric value returned by the read() function represent?
3. What is a process descriptor table?
4. What is the purpose of the open() system call?
5. Why do you need to use the close() system call?

Process Creation

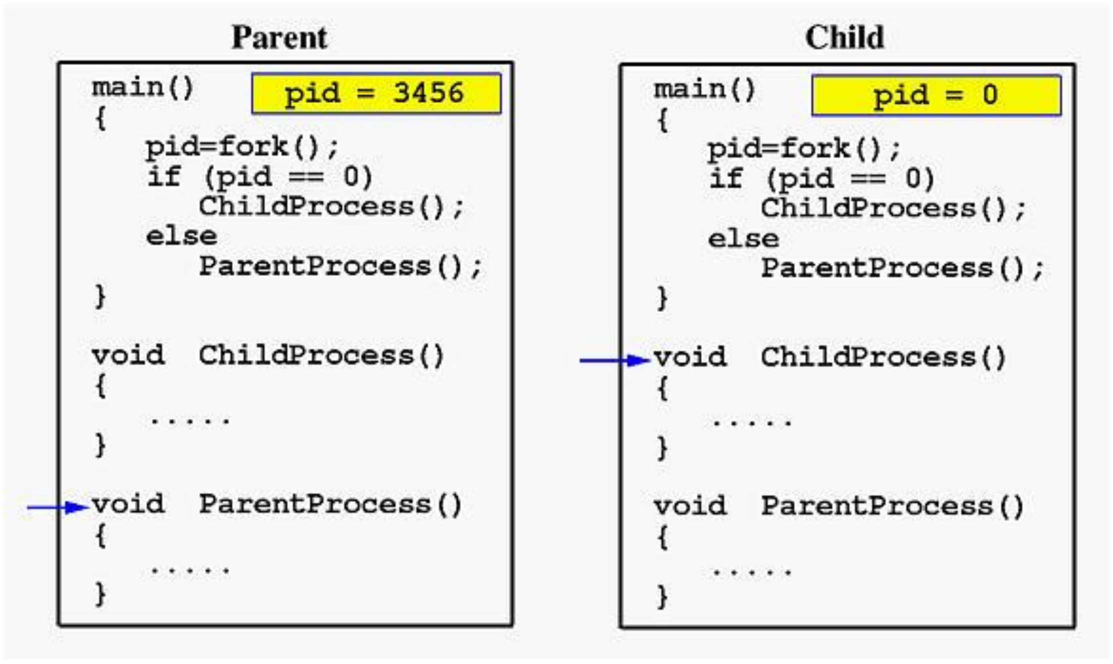
Processes are created using the Unix `fork()` system call discussed in lectures.

The syntax of the `fork` system call is as follows:-

```
    pid = fork();
```

where `pid` is an integer variable whose value after the call uniquely identifies the newly created process. Every process in the system has a unique process id (pid).

Fork causes the creation of a new process. In Unix, the new (child) process is an exact copy of the parent process. This means that it has its own copy of the parent's memory (containing the program code and data), its own copy of the parent's state including its descriptor table (pointing to various open files, pipes and other I/O devices). It receives a copy of the parent's processor state (including the program counter) and all other environmental attributes of the parent. As the program counter is copied, the child will begin execution at the same point in its code as the parent has currently reached.



```
main()
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```

When the child is created, two copies of the variable pid now exist. The original exists in the parent's address space and the other in the child's address space. The fork system call returns different values to the parent and the child. These values are assigned to the two pid variables and this is the only difference between the processes after the fork. However, it is an important difference because it allows the two identical processes to know who is the parent and who is the child and as a result they may perform alternative actions subsequently.

Specifically, the fork system call returns a value of 0 to the child process and returns a non zero positive value to the parent. This value will be the system process id of the new child.

If you want the child to behave differently to the parent, then the next statement after the fork should compare the value stored in the pid variable. If a process finds the value of pid to be 0, then this is only true for the child. If it is positive then the process knows it is the parent. If the value of pid is less than 0 then an error occurred during fork and a new process was not created.

In the following program, the main() process creates a new process. The original process subsequently tests the value of its pid and takes one course of action, i.e. writes "I am the parent". The child process tests its pid, finds that it is equal to 0 and so follows another course of action, i.e. writes "I am the child". Note that because both processes are exact copies of each other, they also share the same I/O devices initially until altered by either process subsequently. This means that the printf function in both processes will display on the same output window. Note that we don't know which messages will appear first as this depends on how the operating system schedules the processes.

Create a directory in your cs240 folder called practical4, and make it your current directory.
Save the program below as “p4a.c”, compile the program and run it using
“cc p4a.c –o p4a”.

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

int main() {
    int pid;

     pid = fork();

     if (pid == 0) {
          printf("I am the child\n");
          exit(0);  /* Terminate */
     }
     else {
          printf("I am the parent\n");
     }

     printf("Which process am I?\n");
}
```

In this program, which process prints out the message “Which process am I?”.

In the previous example, replace the line containing the `exit(0);` statement with the statement `sleep(5);`
Save the program as "`p4b.c`". Compile and run the program again and wait for a few seconds for all the output. Can you explain the program's behaviour from the code?

The next program demonstrates how to `fork()` a process with the purpose of executing a separate program using the `exec()` system call. Save the program below as "`p4c.c`", compile and run it.

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

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

   pid = fork(); /* create a child process */

/* Two processes now exist executing copies of this code
(unless fork failed) but with different pid values */

   if (pid < 0) {
      fprintf(stderr, "Fork failed");
      exit(-1);  /* program ends */
   }

   else if (pid == 0) { /* child process */
      execlp("/bin/ls", "ls", NULL);
   }
   else {  /* Parent process */
/* Wait suspends execution of current process until a
   child has ended. */
      wait(NULL);
      printf("Child %d complete", pid);
      exit(0); /* parent ends */
   }
}
```

Alter the program above so that the child process produces a list of user processes that are running rather than a list of files in the current directory. Save the modified program as "`p4d.c`"

Communication in Unix is handled in a very similar way to the file I/O which we have seen last week. There are many mechanisms available in Unix which allow processes to communicate, such as pipes, sockets, message queues, remote procedure call and remote method invocation through Java. With our understanding of basic I/O we will look at how some of these mechanisms work in this and in later practicals.

Pipe Communication

A pipe is a one way communication channel which transfers data between processes in a first-in-first-out manner. To send information, the `write` system call is used to put data into the pipe. To receive information, a process uses the `read` system call to get data out of the pipe. If the pipe is empty, the `read` function blocks the calling process until data can be read from the pipe. If a process attempts to read more data than is in the pipe, the `read` succeeds returning all data in the pipe and the count of bytes actually read.

The syntax of the `pipe` system call is as follows:

```
     pipe(fdptr);
```

where `fdptr` is an array of two integers. After the system call, this array will contain two file descriptors which identify both ends of the new pipe. `fdptr[0]` is the read descriptor for the pipe and `fdptr[1]` is the write descriptor. The file descriptor values correspond to the indices of the next two free entries found in the user descriptor table.

The following C program "`p4e.c`" demonstrates the creation and use of a pipe. Make sure you can follow the code before you execute it. See if you can anticipate what output it will produce.

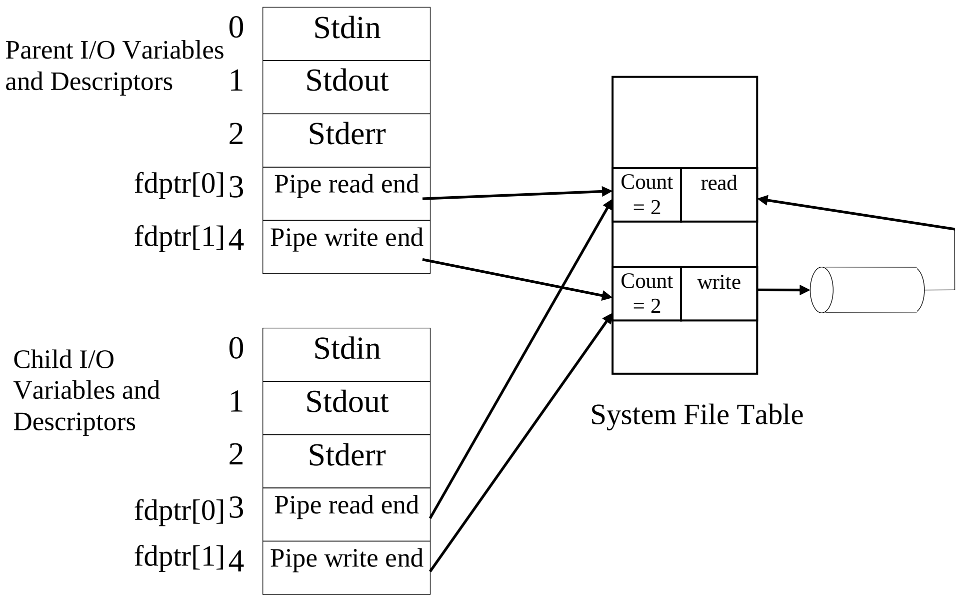
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main() {
int fdptr[2], n, buffersize=5;
char strbuff[buffersize+1];
char message[] = "Welcome to Unix pipes";

pipe(fdptr);  /* Create a new pipe */

/* Let's see what descriptors were allocated
   for the read and write ends of the pipe*/
printf("read descriptor = %d, write descriptor = %d,"
       "buffersize = %d \n", fdptr[0], fdptr[1], buffersize);

/* This process writes a string of chars into the pipe */
write(fdptr[1], message, strlen(message));

/* Let's read the data written to the pipe
   in blocks of buffersize and write it to standard output */
do {
   n = read(fdptr[0], strbuff, buffersize);
   /* Add null character to end of string to
      terminate string properly for printf */
   strbuff[n] = 0;
      printf (" Read =%s\n",strbuff);
   } while (n == buffersize);
}
```

Alter the last program (`p4e.c`) by changing the value of the integer `buffersize` at the top of the program from 5 to 10. Save the program as “`p4f.c`” and compile and run it. Note that fewer calls to `read()` are required with a bigger buffer.

Using pipe communication between parent and child processes

Next we want to combine the ideas of pipes and child processes to get a parent process to send a message to its child using a shared pipe.

If a process creates a pipe (with `pipe(fdptr)`) and later forks a child process, the child will have a copy of the array `fdptr` and a copy of the parent's descriptor table. This enables the child to access all open I/O objects (open at the time of child creation) of the parent using the same descriptor numbers



Write a C program “`p4g.c`” which causes a process to create a pipe. It should then fork a child process. The parent should write some data into the pipe forever. Whenever it writes into the pipe, it should print a message indicating this on its standard output and then sleep for 5 seconds. The child should read data from the pipe and display the data on its standard output forever, sleeping for 5 seconds after each read.

Note that because the standard output of both processes is the same, the output may be confusing due to various scheduling sequences. You might include the words *parent* or *child* at the start of each output message to allow you to distinguish who produced which output.

If you need help with the program “p4g.c”, a template of example code is given below.

You need to **replace all the comments** with specific code statements to do what is required in the description for the exercise. Use the earlier programs for example usage of the system calls used in this program.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

int main() {
int fdptr[2], n, buffersize=21;
char strbuff[buffersize+1];
char message[] = "Welcome to Unix pipes";
int pid;

/* Create a new pipe */

/* fork a process */

if (pid == 0) {
    while (true) {
       /* read from the pipe into strbuff */
       strbuff[n]=0;
       printf("Child read: %s\n", strbuff);
       /* sleep for 5 seconds */
       }
   }
else {
   while (true) {
      printf("Parent writing: %s\n", message);
      /* write message into the pipe */
      /* sleep for 5 seconds */
      }
   }

}
```

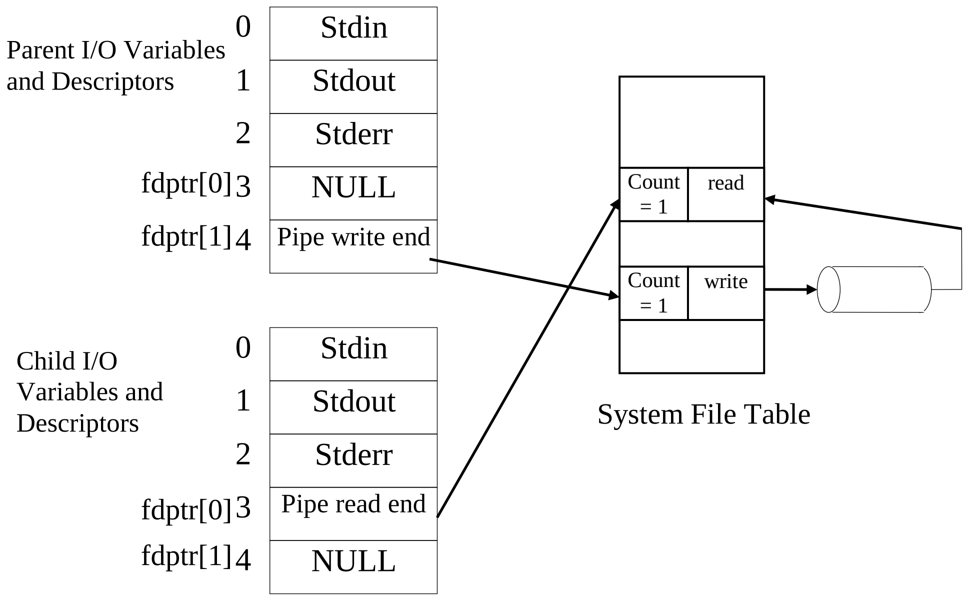
Final Words

In the last example, both parent and child have access to both ends of the pipe even though they each use only one end of it. It is normal practice for both processes to close the end not required before using the pipe.

For the child
```
    close(fdptr[1]);        /* Close the write end */
```
For the parent
```
    close(fdptr[0]);        /* Close the read end */
```

The diagram below demonstrates the new status of the system tables after performing the closes.

Modify the last program exercise to include this requirement and implement the situation described by the diagram below.



Save the program overleaf (which includes these changes) as "`p4h.c`" and compile and execute it.

Have your eight C programs ready and all compiled, and be sure you understand the material and can answer any questions your demonstrator may have before the end of the lab in order to get your mark for the session.

Sample code for “p4h.c”

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

int main() {

int fdptr[2], n, buffersize=21;
char strbuff[buffersize+1];
char message[] = "Welcome to Unix pipes";
int pid;


pipe(fdptr);  /* Create a new pipe */

pid = fork();

if (pid == 0) {
    close(fdptr[1]); /* Close the write end in child */
    while (true) {
       n=read(fdptr[0], strbuff, buffersize);
       strbuff[n]=0;
       printf("Child read: %s\n", strbuff);
       sleep(5);
       }
   }
else {
   close(fdptr[0]); /* Close the read end in parent */
   while (true) {
      printf("Parent writing: %s\n", message);
      write(fdptr[1], message, strlen(message));
      sleep(5);
      }
   }
}
```