CS-323 Exercises Week 2

01 Mar 2019

1) Process trees

a) A user is running a shell *csh* and types *make* to *csh*. The *makefile* contains the following two lines:

```
foo &
bar
```

Draw the Linux process tree below *csh* step-by-step as the *make* command is executed.

b) Draw the Linux process tree for the following code sample:

```
int main() {
      for (int i=0; i<10; i++) {
            fork();
            while(1);
      }
}
```

2) Linux fork

What will be an output printed by these programs? If there is more than one output possible, write them all.

A – test1.cpp compiled to test1:

```
int main() {
      for(int i=0; i<3; i++) {
            printf("%d", i); fflush(stdout); fork();
      }
}
```

B – test2.cpp compiled to test2:

```
 int main() {
      for(int i=0; i<3; i++) {
            printf("%d", i);
            fflush(stdout);
            execl("test2", "test2");
      }
}
```

3) I/O Redirection under the hood

Using bash, users can redirect a file to a process' stdin, or redirect a process' stdout/stderr to a file or other file descriptor. The two commands below are examples of using redirection. They redirect stdout from the ls command to the file output.txt and stderr from the ls command to errors.txt.

```
ls –la > output.txt
ls –la 2> errors.txt
```

What happens under the hood when bash wants to run `ls`, as in the examples above?

1. The main process (e.g. bash) forks itself using the fork libc wrapper.
2. The forked process sees that output redirection was entered on the command line and opens the specified file using the open system call.
3. The forked process calls dup2 to copy the freshly opened file descriptor over stdin/stdout/stder.
4. The forked process proceeds as normal by calling the execve syscall or something similar to replace the executable image with that of the process to be run (e.g. l s)

Before explaining the dup/dup2 and open system calls, it is necessary to be aware of the process file descriptor table, a crucial data structure used in unix systems for file manipulation. The process file descriptor table is **local to every process** and contains information such as the identifiers of the files opened by the process. Whenever, a process creates a file, it gets an index from this table called a file descriptor. **File descriptors 0, 1 and 2 are by default assigned to the standard input/output and error.** In addition, the kernel also manages a global file table. The global file table contains information that is **global to the kernel** e.g. the byte offset in the file where the users' next read/writes will start and the access rights allowed to the opening process.

The open system call returns an integer called the *file descriptor*. Under the hood, the kernel parses the pathname given in the input and finds the file corresponding to that pathname. The kernel then allocates an entry with a reference to the file in the private process file-descriptor table and notes the index of this entry. This index itself is returned to the user. The entry in the file descriptor table points to the file. The dup system call copies a file descriptor **into the first free slot** in the private file descriptor table and then returns the new file descriptor to the user.

Example: Say the user opens a file "/var/file1" (fd = 3 because fd 0, 1, and 2 correspond to stdin, stdout and stderr respectively), then opens a file "/var/file2" (fd = 4) and again opens "/var/file1" (fd = 5). Now, if the user calls dup(3), the kernel follows the pointer from the user file descriptor table for the fd entry '3', and increments the count value in the global file table. Then, it searches for the next available free entry in file descriptor table and returns that value to the user (6 in this case).

Question 1: What does the program below print, assuming /var/file1 contains “aaaaabbbbb”? Explain your answer.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
      int i,j;
      char buf1[5],buf2[5];

      i = open("/var/file1", O_RDONLY);
      j = dup(i);

      read(i, buf1, sizeof(buf1));
      read(j, buf2, sizeof(buf2));
      close(i);

      for (int k=0; k<5; k++)
            printf("%c",buf1[k]);
      printf("\n");

      for (int k=0; k<5; k++)
            printf("%c",buf2[k]);

      return 0;
}
```

Question 2: What does the program below print, assuming /var/file1 contains “aaaaabbbbb”?

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

int main()
{
      int pid, fd;
      char buf[5];

      fd = open("foo.txt", 'r');

      pid = fork();
      if (pid) {
            wait(NULL);
            read(fd, buf, 5);
            for (int i=0;i<5;i++)
                  printf("%c", buf[i]);
      } else {
            read(fd, buf, 5);
            for (int i=0;i<5;i++)
                  printf("%c", buf[i]);
      }
}
```

Question 3: Your task is to implement the command `ls –la > output.txt` in C. In the code sample below, complete the corresponding lines that implement output redirection, using the dup, open and close system calls. Try both implementations with dup and dup2.

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

int main(void)
{
	char *argv[] = { "/bin/ls", "-la", 0 };
	char *envp[] =
	{
		"HOME=/",
		"PATH=/bin:/usr/bin",
		0
	};
	int fd = ...
	...
	...
	// execute ls command, with redirected output
	execve(argv[0], &argv[0], envp);
	return -1; //something went wrong
}
```

4) Linux pipes

Regular files are not a satisfactory communication medium for processes running in parallel. Take for example a reader/writer situation in which one process writes data and the other reads them. If a file is used as the communication medium, the reader can detect that the file does not grow any more (read returns zero), but it does not know whether the writer is finished or simply busy computing more data. Moreover, the file keeps track of all the data transmitted, requiring needless disk space. **Pipes** provide a mechanism suitable for this kind of communication. A pipe is made of two file descriptors. The first one represents the pipe's output. The second one represents the pipe's input. Pipes are created by the system call pipe.

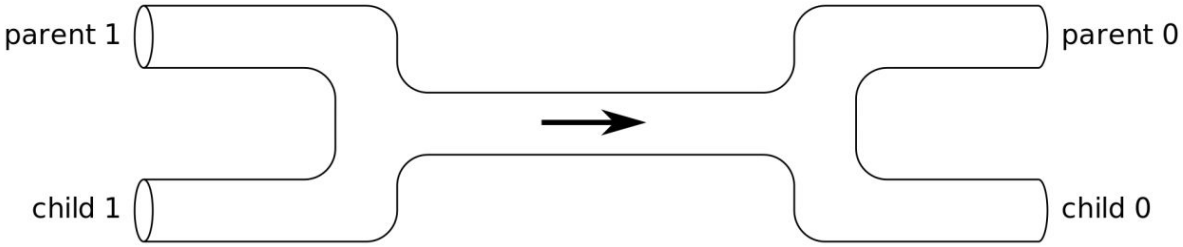
The code sample below shows how to create, write to, and read from a pipe. Feel free to try it out on your machine. pipe() takes an array of two integers as an argument and fills that array with two file descriptors. The first element of the array is the reading-end of the pipe, and the second element in the array is the writing end. When any bytes are written to pipefds[1],the OS makes them available for reading from pipefds[0].

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

int main(int argc, char* argv[])
{
        int pipefds[2];
        char buf[30];
        //create pipe
        if (pipe(pipefds) == -1) {
                perror("pipe");
                exit(EXIT_FAILURE);
        }
        //write to pipe
        printf("writing to file descriptor #%d\n", pipefds[1]);
        write(pipefds[1], "CS323", 9);
        //read from pipe
        printf("reading from file descriptor #%d\n", pipefds[0]);
        read(pipefds[0], buf, 9);
        printf("read \"%s\"\n", buf);

        return 0;
}
```

However, it is not very useful for a single process to use a pipe to talk to itself. In typical use, a process creates a pipe just before it forks one or more child processes. The pipe is then used for communication either between the parent or child processes, or between two sibling processes.

The goal of this exercise is to modify the program above such that the parent process forks and writes "CS323" to the pipe, and its child reads from the pipe one byte at a time, until the pipe is empty.

After forking, the pipe file descriptors are shared between the parent and child, as shown below.



To ensure the pipe works properly, the ends of the pipe that are not used should be closed. For instance, if the child receives data from the parent, the parent should close pipefds[0] and the child should close pipefds[1].