CS180 Experiment 2

June 24, 2011

1 `write`, `open` and `close`

1. Examine the `write-system-call.c`. This C file contains 3 separate system calls. If you need to, look up the man pages for open, write and close. (Clicking on those words should open up the links to the man pages of these commands.)

2. What does `O_CREAT` or `O_RDWR` mean? (Refer to the man pages)

3. Cygwin emulates these system calls for POSIX compliance. (POSIX is a family of related standards of APIs for operating systems in the Unix/Linux family.) As mentioned in the lectures, "file" are referred to with a file descriptor.

2 `stdin`, `stdout` and `stderr`

1. Examine `012.c`. Compile the code using `gcc` and run. What happens when you uncomment line 13 and comment line 12? Is the output the same? Repeat the same for lines 15 and 16, and then for lines 18 and 19.

2. Note the order in which line 12 is evaluated. Why do we need to assign a null character to the `buffer` array?

3. Who determines what `stdin`, `stdout` and `stderr` refer to for a new process?

3 Posix Pipes

1. Examine `pipes.c`. `pipe` is a system call that fills in the array with 2 file descriptor. Invariably, the first file descriptor will refer to the reading end of the pipe while the latter file descriptor will refer to the writing end of the pipe.

2. Opened files are automatically inherited by child processes. Hence the child process is able to write into the writing end of the pipe directly, as shown in the code.

3. Use `gcc` to compile the program. Try running it with or without the comments in lines 35 and 45. What do you observe? Are both processes running concurrently?

4 Size of pipes

1. Examine `pipe-size.c`. Observe that the child writes the entire buffer into the pipe. Note also that the parent performs a `wait` before reading the reading end of the pipe.

2. Compile the program using `gcc -o pipe-size pipe-size.c` command. Try running the program with `pipe-size 1000`, `pipe-size 2000`, `pipe-size 4000` and so on and so forth. What do you eventually observe? Try to explain why this behaviour occurs.

3. Could you find out the exact size of your pipe in your system?

5 Exec

1. Examine `simple-exec.c`. Compile it with `gcc` and run it.

2. Try the different `exec` calls by commenting the others out while keeping the one you are trying. Observe the different outputs. Could you explain the difference between the outputs?

6 Infinite Exec

1. Examine `infinite-exec.c`. Compile it the command `gcc -o infinite-exec infinite-exec.c` and run it. What happens? Why is the PID printed always the same PID?

7 Parameter passing in `exec` calls

1. Examine `print-param.c`. Compile the program using `gcc -o print-param print-param.c`. Run the program using `./print-param.exe 1 2 3 4 5`. What is the output? Is it what you have expected?

2. Examine `exec-print-param.c`. Compile and run the program in the same directory containing `print-param.c`. What is the difference that you observe in the output? Why the difference in the two?

8 `dup` call

1. `simple-dup.c` is derived from a modification of `write-system-call.c`. Compile it with `gcc` and see what it prints.

2. Try the different options in lines 24-26 (i.e., comment out the other two) and observe that they result in the same `temp1.txt`. Try to explain the reason why this is so.

9 The `ls|sort` example

1. `ls_pipe_sort.c` is an example that performs the `ls|sort` in a C program. The code gives you a hint as to how it is done by the shell program. Read through the code to understand it. Please check that the program gives you the same output as what you see when you type `ls|sort` in the cygwin command line.

10 Code Listings

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


int main(void)
{

  int fd = open("./temp.txt", O_CREAT| O_RDWR);
  write(fd, "Hello World", 5);
  close(fd);
  return 0;
}
```

write–system–call.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
  char buffer[100];

  buffer[read(0, buffer, 100)]='\0';
  //gets(buffer);

  write(1, buffer, strlen(buffer));
  //fprintf(stdout, "%s\n", buffer);

  write(2, buffer, strlen(buffer));
  //fprintf(stderr, "%s\n", buffer);
}
```

012.c

```c
/* pipes.c
** -- using POSIX pipes
** cs180 1/08
*/

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


int main(int argc, char** argv)
{

  int pid;
  int descriptor[2];
  char buffer[1024];

  pipe(descriptor);

  /*
    descriptor[0] - reading end
    descriptor[1] - writing end
  */


  pid = fork();

  if (pid == 0)
  {
    gets(buffer);

    //while(1)
      write(descriptor[1],buffer,strlen(buffer));

    exit(0);
  }
  else
  {

    int bytes_read;

    //while(1)
    {
      bytes_read=read(descriptor[0],buffer, 10);
      buffer[bytes_read]='\0';
      if(bytes_read>0)
      {
        fprintf(stdout,"Parent reading message:\"%s\"\n", buffer);
      }
    }
  }

  close(descriptor[0]);
  close(descriptor[1]);
  return 0;
}
```

pipes.c

```c
/* pipes.c
** -- using POSIX pipes
** cs180 1/08
*/

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


int main(int argc, char** argv)
{

  int pid;
  int descriptor[2];
  char *buffer;
  int size_of_buffer, i;

  if(argc!=2)
  {
    fprintf(stderr, "Usage: %s <buffer-size>\n", argv[0]);
    return 1;
  }

  pipe(descriptor);
  pid = fork();

  size_of_buffer=atoi(argv[1]);
  buffer = malloc(size_of_buffer);

  if (pid == 0) {

    for(i=0;i<size_of_buffer;i++)
      buffer[i]='a'+rand()%26;

    write(descriptor[1],buffer,size_of_buffer);

    exit(0);
  }
  else
  {

    int bytes_read;
    wait(NULL);
    {
      bytes_read=read(descriptor[0],buffer, size_of_buffer);
      buffer[bytes_read]='\0';
      if(bytes_read>0)
      {
        fprintf(stdout,"Parent reading message:\"%s\"\n", buffer);
      }
    }
  }

  close(descriptor[0]);
  close(descriptor[1]);
  return 0;
}
```

pipe–size.c

```
#include "unistd.h"

int main()
{
  /*
    Calling the ls utility program:
    Comment out the others and keep only one
    un-commented for the experiments
  */
  execl("/bin/ls", "ls", "-l", 0);
  //execl("ls", "ls", "-l", 0);
  //execlp("ls", "ls", "-l", 0);
  //execlp("ls", "ls", "-l");

  printf("If execution reaches here, exec has failed.\n");
}
```

simple–exec.c

```
#include "unistd.h"
#include <stdio.h>
int main()
{
  /*
    Calling the ls utility program:
    Comment out the others and keep only one
    un-commented for the experiments
  */
  printf("My process ID is %d\n", getpid());
  execl("./infinite-exec.exe", "./infinite-exec.exe",  0);
}
```

infinite–exec.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int i;
  for(i=0; i<argc; i++)
    printf("Arg %i: %s\n", i, argv[i]);
}
```

print–param.c

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


int main(void)
{
  int fd = open("./temp1.txt", O_CREAT| O_RDWR);

  /*
      close(1) means making file descriptor 1 unused.
      dup(fd) means we are duplicating fd on lowest unused
      file descriptor, which is 1.
      So that both fd and 1 will point to the same file.
  */

  close(1);
  dup(fd);

  /*
      You should find that all of the statements do
      the same thing. Why?
  */
  write(1, "Hello World", 11);
  //printf("Hello World");
  //write(fd, "Hello World", 11);
  close(fd);
  return 0;
}
```

simple–dup.c

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

#define READ_END 0
#define WRITE_END 1

int main()
{
  int pid;
  char buffer[1024];
  int descriptor[2];
  int i;
  char c;

  /* descriptor[0] - for reading
  ** descriptor[1] - for writing
  */
  pipe(descriptor);

  printf("Parent's PID is %d\n", getpid());

  pid = fork();

  if (pid == 0)
  {
    //Child's code
    printf("1st child PID is %d\n", getpid());

    //close the reading end of the pipe for safety's sake.
    close(descriptor[READ_END]);

    //close 1, which is a file descriptor pointing to stdout
    close(1);

    //duplicate the writing end. Since 1 is the smallest file descriptor
    //available, 1 will point to the writing end after this.
    dup(descriptor[WRITE_END]);

    //exec call to run ls
    execlp("ls", "ls", "-l", NULL);
  }
  else
  {
    //Here we perform another fork to show that
    //a pipe can be shared between any 2 processes of the same ANCESTRY
    if(fork()==0)
    {
      //2nd child of the parent run here.
      printf("2nd child PID is %d\n", getpid());
      // Again, all file descriptors are inherited in this child.

      //close the writing end of the pipe for safety's sake
      close(descriptor[WRITE_END]);

      //close 0, a file descriptor originally pointing to stdin
      close(0);

      //duplicate the reading end, since 0 is smallest file decsriptor
      //available, 0 will point to the reading end after this.
      dup(descriptor[READ_END]);

      //exec call to run sort.
      execlp("sort", "sort", NULL);
    }
```

```
    // Parent can wait for both child to complete. No need to use the pipes
    // in parent. Parent can close the pipes while the pipes are
    // still being used in the children processes.
    close(descriptor[WRITE_END]);
    close(descriptor[READ_END]);
    wait(NULL);
    wait(NULL);
  }
}
```

ls_pipe_sort.c