Operating System Lab

Lect-2

Low Level Function Calls

- Input and output uses the read and the write system calls, which are accessed from C program through functions *read* and *write*.
- For both, the first argument is a file descriptor.
- The second argument is a character array where the data is to go or come from.
- The third argument is the number of bytes to be transferred.

Low Level Function Calls

- Syntax:

```
int n_read = read(int fd, char* buf, int n);
int n_written = write(int fd, char* buf, int n);
```

Low Level Function Calls

- Any number of bytes can be read or written in one call.
  - The most common value is 1, which means one character at a time (unbuffered), and a number like 1024 or 4096 that corresponds to a physical block size on a peripheral device.

Example: Create a File and Write Data

```
#include<stdio.h>
…
int main(){ char buf[] = "Sentence to be written in file";
   int handle = 0, len = strlen(buf);
   handle = creat(“A.Txt", O_WRONLY | _S_IREAD | _S_IWRITE);
   if( -1 == handle)
   {       printf("Error Creating File\n"); exit(-1);         }
   if(len == write(handle, buf, len)) /* write Date to the file */
   {       printf("Wrote %d Byte\n", len);         }
   else
   {       printf("Unsuccessful in writing\n");   }
   close(handle); /* close the file */
   return 0;}
```

The *open* Function

- If we want to write data to an file, we need to open the file in write mode, if the file exists or create a new file. This can be done with the *open* function.
- Once we have finished writing to the file, the file is closed using the *close* function.
- The syntax for the open function is:

  int open(char* name, int flags, int perms);

  – The first argument is the name of the file.

The *open* Function

- The second argument is an int that specifies how the file is to be opened, the values are:
  - O_RDONLY open for reading only
  - O_WRONLY open for writing only.
  - O_RDWR open for both reading and writing
- These constants are defined in *fcntl.h*
- The third parameter specifies the permission of the file.

Read/Write File

- For reading and writing , use the *read* and *write* function
- *read*/*write* takes 3 arguments
  - From where to read or write
  - A buffer variable for data
  - How many bytes to write
- Each call returns a count of number of bytes transferred.
  - On reading , the number of bytes may be less than the number requested. A return value of 0 bytes implies end of file, and –1 indicates an error of some sort
  - On writing, the return value is the number of bytes written; an error has occurred if this isn’t equal to the number requested

Example (*read*)

```
#include <stdio.h>
int main()
{
        char buf[10];
        int fd = 0, len = 1, res = 0;
        fd = open("A.Txt", O_RDONLY, 0       );
        if( -1 == handle)
        {    printf("Error Creating File\n");    exit(-1);    }

        while(res = read(fd, buf, len)!= 0)
        {        buf[res] = '\0';            printf("%s", buf);       }
        close(fd); /* close the file */
        return 0;
}
```

Interprocess Communication (pipes)

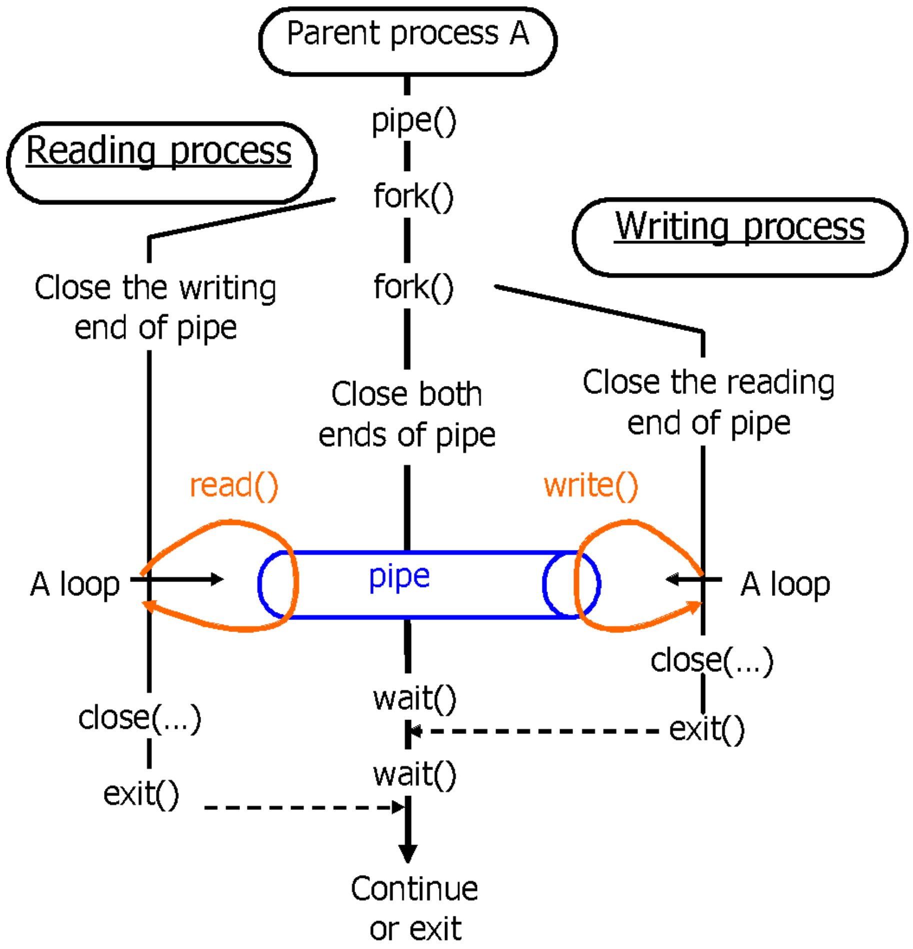
- Pipe is the most traditional Unix inter-process communication
- The pipe function creates a communication buffer that the caller can access through the file descriptors.
- The data written to one file descriptor and read from the other on a first-in-first-out basis.

- Pipes are typically used to communicate between two different processes:
    - Process A (parent) creates a pipe
    - Process A forks twice, creating B and C.
    - Each process closes the ends of the pipe it does not need.
        - **Process B** closes downstream end
        - **Process C** closes upstream end
        - **Process A** closes both ends
    - Processes B and C execute other programs, using exec, where file descriptors are retained.

Synchronization Using Pipes

- Have finite capacity (few hundred bytes)
- This imposes loose synchronization between up and down stream processes:
  - Upstream process blocks if pipe is full
    - Until downstream consumes some
  - Downstream process blocks if pipe is empty
    - Until upstream writes some
- If upstream closes descriptor, a downstream read operation will return EOF (0)

Using Pipes



Parent process A
pipe()
fork()
fork()
Reading process
Writing process
Close the writing end of pipe
Close both ends of pipe
Close the reading end of pipe
read()
write()
A loop
pipe
A loop
close(…)
wait()
close(…)
exit()
wait()
exit()
Continue or exit

- Closing upstream end of pipe is essential for 1st process otherwise it will never see EOF

- A pair of wait() are there to ensure that parent will not return before both children have finished

How to Create Pipes

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

int fd[2];
if (pipe(fd) == -1)
  perror("Failed to create the pipe");
```

Redirection

```
#include <unistd.h>
int dup2(int OldFileDes, int NewFileDes);
```

- takes an existing file descriptor (OldFileDes) and duplicates it into (NewFileDes)
- Example:

```
fd = open("my.file",O_RDRW)
dup2(fd,1)
```

file descriptor table

[0]	*standard input*
[1]	*standard output*
[2]	*standard error*

file descriptor table

[0]	*standard input*
[1]	*write to* `my.file`
[2]	*standard error*

Example

who | sort

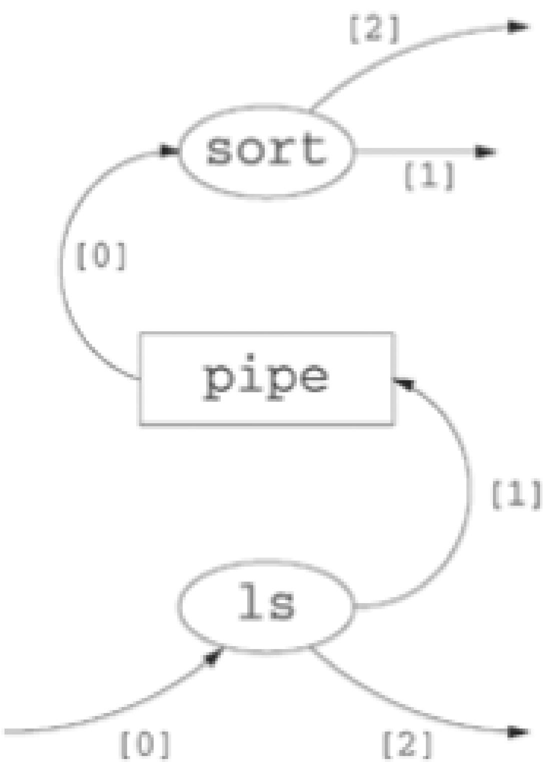
- The shell gets the output of who connected to the upstream end of the pipe, and the input to sort connected to the downstream end.
- Shell uses dup2 to do the plumbing:
  - dup2(old, new):
  - takes an existing file descriptor (old) and duplicates it into (new).

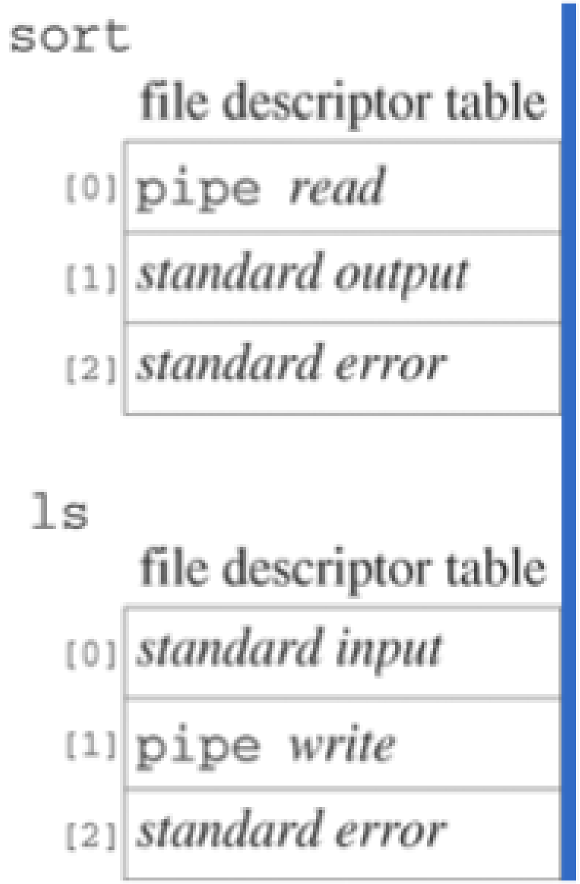
```
int p[2];
pipe(p);
dup2(p[1], 1);
/*standard output connected upstream end of pipe*/
```

Example

```
main() {
   int fds[2];
   pipe(fds);
   /*child 1 duplicates downstream into stdin */
   if (fork() == 0) {
        dup2(fds[0], 0);
        close(fds[1]);
        execlp("sort", "sort", 0); }
   /*child 2 duplicates upstream into stdout */
   else if fork() == 0) {
        dup2(fds[1], 1);
        close(fds[0]);
        execlp("who", "who", 0); }
   else{  /*parent closes both ends and wait for children*/
        close(fds[0]);
        close(fds[1]);
        wait(0);
        wait(0); }
}
```

Example


[2]
sort
[1]
[0]
pipe
[1]
ls
[0]
[2]


sort
file descriptor table
[0] pipe *read*
[1] *standard output*
[2] *standard error*
ls
file descriptor table
[0] *standard input*
[1] pipe *write*
[2] *standard error*

Pipes Characteristics

- Unidirectional
- Pipes can only be used between processes that have a common ancestor (named pipes)
- No mechanism to authenticate
- Do not work across the network
- They are the easiest of IPC mechanisms. Simple, easy to understand and easy to implement as well.