Interprocess Communication

Based on book chapter 12.6

How do two processes communicate? We will look at:

- Pipes
- Sockets

Pipes

An interprocess communication mechanism allowing two or more processes to send information to each other.

They are commonly used from within shells to connect the standard output of one utility to the standard input of another.
Interprocess Communication

- Consider `$ who | wc -l`
  - option -l outputs total number of lines in input

![Diagram of Interprocess Communication]

- both processes run concurrently; pipe buffers and protects against overflow; suspends reader until more data becomes available...

Unnamed Pipes

- System Call: `int pipe (int fd [2])`
  - pipe () creates an unnamed pipe and returns two file descriptors;
  - the descriptor associated with the "read" end of the pipe is stored in fd[0],
  - the descriptor associated with the "write" end of the pipe is stored in fd[1].
Unnamed Pipes

The following rules apply to processes that read from a pipe:

- If a process reads from a pipe whose write end has been closed, the read() returns a 0, indicating end-of-input.
- If a process reads from an empty pipe whose write end is still open, it sleeps until some input becomes available.
- If a process tries to read more bytes from a pipe than are present, all of the current contents are returned and read() returns the number of bytes actually read.

Unnamed Pipes

The following rules apply to processes that write to a pipe:

- If a process writes to a pipe whose read end has been closed, the write fails and the writer is sent a SIGPIPE signal.
- The default action of this signal is to terminate the writer.
- If a process writes fewer bytes to a pipe than the pipe can hold, the write() is guaranteed to be atomic; that is, the writer process will complete its system call without being preempted by another process.
- If a process writes more bytes to a pipe than the pipe can hold, no similar guarantees of atomicity apply.
Unnamed Pipes

executing the following code will create the data structure shown

```c
int fd [2];
pipe (fd);
```

Unnamed pipes are usually used for communication between a parent process and its child, with one process writing and the other process reading. The typical sequence of events is as follows:

- The parent process creates an unnamed pipe using `pipe()`.
- The parent process forks.
- The writer closes its read end of the pipe, and the designated reader closes its write end of the pipe.
- The processes communicate by using `write()` and `read()` calls.
- Each process closes its active pipe descriptor when finished with it.
Unnamed Pipes

Example that allows parent to read message from child via a pipe

- note that child includes NULL terminator as part of the message

```c
#include <stdio.h>
#define READ   0       /* The index of the read end of the pipe */
#define WRITE  1       /* The index of the write end of the pipe */
char* phrase = "Stuff this in your pipe and smoke it";
main ()
{
    int fd [2], bytesRead; /* Parent process' message buffer */
    pipe (fd); /*Create an unnamed pipe */
    if (fork () == 0) /* Child, writer */
    {
        close(fd[READ]); /* Close unused end */
        write (fd[WRITE], phrase, strlen (phrase) + 1); /* include NULL*/
        close (fd[WRITE]); /* Close used end*/
    }
    else /* Parent, reader*/
    {
        close (fd[READ]); /* Close unused end */
        bytesRead = read (fd[READ], message, 100);
        printf ("Read %d bytes: %s\n", bytesRead, message); /* Send */
        close (fd[READ]); /* Close used end */
    }
}
$ ./talk                         ...run the program.
Read 37 bytes: Stuff this in your pipe and smoke it
$ _
Unnamed Pipes

Example of chaining two programs

- parent creates pipe
- each end attaches its stdin or stdout to the pipe (via dup2)
- both processes exec

```c
#include <stdio.h>
#define READ   0
#define WRITE  1
main (argc, argv)
int argc;
char* argv[];
{
    int fd [2];
    pipe (fd); /* Create an unnamed pipe */
    if (fork () != 0) /* Parent, writer */
    {
        close (fd[READ]); /* Close unused end */
        dup2 (fd[WRITE], 1); /* Duplicate used end to stdout */
        close (fd[WRITE]); /* Close original used end */
        execvp (argv[1], argv[1], NULL); /* Execute writer program */
        perror ("connect"); /* Should never execute */
    }
    else /* Child, reader */
    {
        close (fd[WRITE]); /* Close unused end */
        dup2 (fd[READ], 0); /* Duplicate used end to stdin */
        close (fd[READ]); /* Close original used end */
        execvp (argv[2], argv[2], NULL); /* Execute reader program */
        perror ("connect"); /* Should never execute */
    }
}
```

$ who                       ...execute "who" by itself.
glass    pts/1      Feb 15 18:45   (:0.0)
$ ./connect who wc          ...pipe "who" through "wc".
    1 6 42     ...1 line, 6 words, 42 chars.
$ _
Named Pipes

- Named pipes (FIFOs) are less restricted than unnamed pipes, and offer the following advantages:
  - They have a name that exists in the file system.
  - They may be used by unrelated processes.
  - They exist until explicitly deleted.

- All of the pipe rules mentioned for unnamed pipes apply.

Because named pipes exist as special files in the file system, processes using them to communicate need not have a common ancestry as when using unnamed pipes.

A named pipe (FIFO) may be created in one of two ways:

- by using the Linux mkfifo utility or the mkfifo() system call
- Utility: `mkfifo fileName`

  `mkfifo` creates a named pipe called `fileName`.
Named Pipes

- example

```bash
$ mkfifo myPipe              ...create pipe.
$ chmod ug+rw myPipe         ...update permissions.
$ ls -l myPipe              ...examine attributes.
  prw-rw----   1 glass   cs   0 Feb 27 12:38 myPipe
$ _
```

- Note the type of the named pipe is "p" in the `ls` listing.

```c
mkfifo ("myPipe", 0660);   /* Create a named pipe */
```

Named Pipes

- Named Pipes operation:
  - a special file is added into the file system
  - once opened by open(),
    - write() puts data into the FIFO queue
    - read() removes data at end of FIFO queue
  - process closes pipe using close()
  - when no longer needed remove pipe from file system using unlink()
Example using a reader and a writer

```c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

main ()
{
    int fd;
    char str[100];
    mkfifo ("aPipe", 0660); /* Create named pipe */
    fd = open ("aPipe", O_RDONLY); /* Open it for reading */
    while (readLine (fd, str)) /* Display received messages */
        printf("%s\n", str);
    close (fd); /* Close pipe */
}

readLine (fd, str)
int fd;
char* str;
/* Read a single NULL-terminated line into str from fd */
/* Return 0 when the end-of-input is reached and 1 otherwise */
{
    int n;
    do /* Read characters until NULL or end-of-input */
    { n = read (fd, str, 1); /* Read one character */
    }
    while (n > 0 && *str++ != 0);
    return (n > 0); /* Return false if end-of-input */
}
```

The writer.c program looks like this:

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

main ()
{
    int fd, messageLen, i;
    char message [100];
    /* Prepare message */
    sprintf (message, "Hello from PID %d", getpid ());
    messageLen = strlen (message) + 1;
    do /* Keep trying to open the file until successful */
    { fd = open ("aPipe", 0_WRONLY); /*Open named pipe for writing */
        if (fd == -1) sleep (1); /* Try again in 1 second */
    }
    while (fd == -1);
    for (i = 1; i <= 3; i++) /* Send three messages */
    { write (fd, message, messageLen); /* Write message down pipe */
        sleep (3); /* Pause a while */
    }
    close (fd); /* Close pipe descriptor */
}
```