Pipes

Process pipes

- A simple pipe

  cmd1—cmd2

  The standard input to cmd1 from the terminal keyboard.
  The standard output from cmd1 is fed to cmd2 as its standard input.
  The standard output from cmd2 is connected to the termianl screen.

- popen and pclose

```
#include <stdio.h>

FILE *popen(const char *command,
            const char *open_mode);

int pclose(FILE *stream_to_close);
```

  Parameters:

command string: is the name of the program to run;
open mode: can be either "r" or "w"
stream to close: the value(a pointer) returned by popen(), be passed to pclose().

- Examples

1. Readin from a program

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

int main()
{
  FILE *read_fp;
  char buffer[BUFSIZ];
  int char_read;

/*use popen call to invoke the command
 * ‘‘ps -ax’’, return the file stream */
```

```
  read_fp = popen("ps -ax", "r");

  if(read_fp != NULL){
 /* read data from the file stream
  * into the buffer, it returns
  * number of items in the file stream */
    char_read = fread(buffer,sizeof(char),
                        BUFSIZ, read_fp);
    if(char_read > 0){
     printf("Output was:- %s\n", buffer);
    }
    pclose(read_fp);
    exit(EXIT_SUCCESS);
  }
  exit(EXIT_FAILURE);
}
```

2. sending out to a program

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

```
int main()
{
  FILE *write_fp;
  char buffer[BUFSIZ];

  /* write to buffer */
  sprintf(buffer, "Once upon a time,
                   there was ...\n");

/* popen invokes the command ‘‘od -c’’,
 * which write the data in ASCII format
 * to the standard output.

  write_fp = popen("od -c", "w");

  if(write_fp != NULL){
/* take data from the buffer, and write
 * into the output file stream */
    fwrite(buffer, sizeof(char),
           strlen(buffer), write_fp);
    pclose(write_fp);
    exit(EXIT_SUCCESS);
  }
  exit(EXIT_FAILURE);
}
```

Pipe call

1. Function prototype

   ```
   #inlclude <unistd.h>
   int pipe(int file_descriptor[2]);
   ```

   The pipe call passes an array of two integer file descriptors. It fills the array with two new file descriptors and returns 0 on success, -1 on failure.

   The two file descriptors returned are connected in a special way. Any data written to file descriptor[1] can be read back from file descriptor[0]. The data is processed in a “first in, first out” basis.

2. Example

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

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

int main()
{
  int data_processed;
  int file_pipes[2];

  const char some_data[]= "123";
  char buffer[BUFSIZ];

  pid_t pid;

  if(pipe(file_pipes) == 0){

    pid = fork();/*fork child*/
    if(pid < 0 ){
      fprintf(stderr, "fork failure");
      exit(EXIT_FAILURE);
    }

    if(pid == 0){/* this is child */
   /*read from pipe to the buffer */
```

```
      data_processed = read(file_pipes[0],
                            buffer, BUFSIZ);
      printf("Read %d bytes: %s \n",
              data_processed, buffer);
      exit(EXIT_SUCCESS);
    }
    else{ /* this is parent */
    /*write the string to the pipe  */
      data_processed = write(file_pipes[1],
                some_data, strlen(some_data));
      printf("Wrote %d bytes: \n",
                      data_processed);
      exit(EXIT_SUCCESS);
    }
  }
  exit(EXIT_SUCCESS);
}
```

This program firstly creates a pipe with pipe() call. Then creates a new process use fork() call. The parent write the data to the pipe, while child read from the pipe. Both parent and child exit after a write() and read() call.

Named pipe: FIFO

So far, we only have been able to pass data between programs that are related(e.g. partnt and child). We would like to pass data between different programs that are not related. We need a named pipe, which is a special type of file that exists as a name in UNIX file system.

1. create a named pipe: FIFO

   The function call is mkfifo() as followed:

   ```
   #include <sys/types.h>
   #include <sys/stat.h>
   int mkfifo(const char *filename, mode_t mode);
   ```

   The mkfifo takes a name(as what you like to name your fifo), and the creation mode(user's permission), normally read and write for all(0755).

It returns 0 on success, otherwise indicate failure.

Example

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

int main()
{
  int res = mkfifo("myfifo", 0755);
  if (res == 0)
    printf("FIFO created.\n");
  exit(EXIT_SUCCESS);
```

Run this program, and check the result use “ls -lF myfifo”, see the reult.

Remove the pipe use “rm myfifo”.

2. opening a FIFO with open

Since FIFO is a file exists in the file system, we can use open() and close() which are used for regular files to open and close the FIFO.

```
open(const char *path, open_flag);
```

The open flag can be four legal combinations of

```
O_RDONLY, O_WRONLY, O_NONBLOCK
```

There are four cases of open() as followed:

```
open(const char *path,O_RDONLY);
```

This open call will block, i.e. not return untill a process opens the same FIFO for writing.

```
open(const char *path,O_RDONLY|O_NONBLOCK);
```

This open call will now succeed and return immediately, even if the FIFO had been opened but not written by any proecess.

```
open(const char *path,O_WRONLY);
```

This open call will block untill a process open the same FIFO for reading.

```
open(const char *path,O_WRONLY|O_NONBLOCK);
```

This open call will return immediately, but if no process has the FIFO open for reading, open will return -1 indicating error, and the FIFO won't be opened. If a process does have the FIFO open for reading, the file descriptor returned can be used for writing to the FIFO.

3. reading and writing FIFOs

A read on an empty, full blocking FIFO will wait untill some data can be read.

A read on an empty, non-blocking FIFO will return 0 bytes.

A write on a full blocking FIFO will wait untill the data can be written.

A write on a FIFO that can't accept all of the bytes being written will either:

• Fail if the request is the PIPE-BUF bytes or less and the data can't be written.

• Write part of the data if the request is more than PIPE-BUF bytes, returning the number of bytes actually been written.

The size of the FIFO is important consideration. There is a system-imposed limitation on how much data can be in FIFO at any one time. This is defined in $< limits. >$:

```
#define PIPE_BUF 4096
```

On some systems, it can be as low as 512 bytes. The size limitation is not so important for single FIFO writer and a single FIFO reader. In case of several programs try to write to the FIFO at the same time, it's usually bital the blocks of data from different programs don't get interleaved. If you can ensure all your write requests are to a blocking FIFO and are size of the PIPE-BUF, then the processes will never get interleaved.

Examples

(a) A writer to a FIFO firstly create the FIFO, write onto the FIFO byte by byte.

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

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

#define FIFO_NAME "mynewfifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
  int pipe_fd;
  int res;
  int open_mode = O_WRONLY;
  int bytes_sent = 0;
  char buffer[BUFSIZ];

 if(access(FIFO_NAME, F_OK) == -1){
  res = mkfifo(FIFO_NAME, 0755);
  if(res != 0){
    printf("couldn't create a fifo.\n");
    exit(EXIT_FAILURE);
  }
```

```
}

printf("process %d opening FIFO WRONLY\n",
                                getpid());

pipe_fd = open(FIFO_NAME, open_mode);

printf("process %d result %d\n",getpid(),
                                pipe_fd);

if(pipe_fd != -1){
  while (bytes_sent < TEN_MEG){
    res= write(pipe_fd, buffer,
                           BUFFER_SIZE);
    if (res == -1){
      printf("write error on pipe\n");
      exit(EXIT_FAILURE);
    }
    bytes_sent += res;
  }
  (void)close(pipe_fd);
}else{
  exit(EXIT_FAILURE);
```

```
     }
     printf("process %d finished\n",getpid());
     exit(EXIT_SUCCESS);
    }
```

(b) A reader reads and discards data from the FIFO

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

#define FIFO_NAME "mynewfifo"
#define BUFFER_SIZE PIPE_BUF

int main()
{
  int pipe_fd;
  int res;
```

```
 int open_mode = O_RDONLY;
 int bytes_read = 0;
 char buffer[BUFSIZ];

printf("process %d opening FIFO RDONLY\n",
                                getpid());

pipe_fd = open(FIFO_NAME, open_mode);

printf("process %d result %d\n", getpid(),
                                 pipe_fd);

 if(pipe_fd != -1){
   do{
      res= read(pipe_fd, buffer,
                          BUFFER_SIZE);
      bytes_read +=res;
   }while(res > 0);
      (void)close(pipe_fd);
 }
 else{
    exit(EXIT_FAILURE);
 }
```

```
  printf("process %d finished and %d bytes
              read\n", getpid(), bytes_read);
  exit(EXIT_SUCCESS);
}
```

THE END