# CHAPITRE III ORDINARY AND NAMED PIPES

A pipe acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems and typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. In implementing a pipe, four issues must be considered:

1. Does the pipe allow unidirectional communication or bidirectional communication?
2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
3. Must a relationship (such as *parent-child)* exist between the communicating processes?
4. Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

In the following sections, we explore two common types of pipes used on both UNIX and Windows systems: **ordinary pipes** and **named pipes.**

## PART I: ORDINARY PIPES

Ordinary pipes allow two processes to communicate in standard producer-consumer fashion; the producer writes to one end of the pipe (the write-end), and the consumer reads from the other end (the read-end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction. We next illustrate constructing ordinary pipes on both UNIX and Windows systems. In both program examples, one process writes the message Greetings to the pipe, while the other process reads this message front the pipe.

On UNIX systems, ordinary pipes are constructed using the function pipe (int fd []).

This function creates a pipe that is accessed through the int fd [] file descriptors: fd [0] is the read-end of the pipe, and fd [1] is the write end.

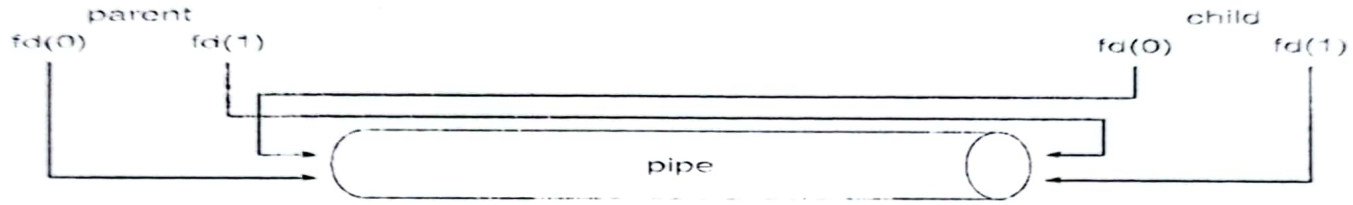


*Figure 1: File descriptors for and ordinary pipe*

UNIX treats a pipe as a special type of file; thus, pipes can be accessed using ordinary read() and write() system calls. An ordinary pipe cannot be accessed from outside the process that creates it. Thus, typically a parent process creates a pipe and uses it to communicate with a child process it creates via fork(). Recall that a child process inherits open files from its parent. Since a pipe is a special type of file, the child inherits the pipe from its parent process. Figure 1 illustrates the relationship of the file descriptor *fd* to the parent and child processes. In the UNIX program shown in Figure 2, the parent process creates a pipe and then sends a fork() call creating the child process. What occurs after the fork() call depends on how the data are to flow through the pipe. In this instance, the parent writes to the pipe and the child reads from it. It is important to notice that both the parent process and the child process initially close their unused ends of the pipe. Although the program shown in Figure 2 does not require this action, it is an important step to ensure that a process reading from the pipe can detect end-of-file (read() returns 0) when the writer has closed its end of the pipe.

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

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

/* create the pipe */
if (pipe(fd) == -1) {
   fprintf(stderr,"Pipe failed");
   return 1;
}

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
}

if (pid > 0) { /* parent process */
   /* close the unused end of the pipe */
   close(fd[READ_END]);

   /* write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

   /* close the write end of the pipe */
   close(fd[WRITE_END]);
}
else { /* child process */
   /* close the unused end of the pipe */
   close(fd[WRITE_END]);

   /* read from the pipe */
   read(fd[READ_END], read_msg, BUFFER_SIZE);
   printf("read %s",read_msg);

   /* close the write end of the pipe */
   close(fd[READ_END]);
}

return 0;
}
```

*Figure 3: Ordinary pipe in UNIX*

Ordinary pipes on Windows systems are termed *anonymous pipes* and they behave similarly to their UNIX counterparts: they are unidirectional and employ parent-child relationships between the communicating processes. In addition, reading and writing to the pipe can be accomplished with the ordinary *ReadFile ()* and *WriteFile ()* functions. The Win32 API for creating pipes is the *CreatePipe ()* function, which is passed four parameters: separate handles for (1) reading and (2) writing to the pipe, as well as (3) an instance of the STARTUPINFO structure, which is used to specify that the child process is to inherit the handles of the pipe. Furthermore, (4) the size of the pipe (in bytes) may be specified.

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;

/* set up security attributes allowing pipes to be inherited
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
  fprintf(stderr, "Create Pipe Failed");
  return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL,NULL,
 TRUE, /* inherit handles */
 0, NULL,NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL)
  fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

*Figure 4: Windows anonymous pipes-parent process*

Figure 3 illustrates a parent process creating an anonymous pipe for communicating with its child. Unlike UNIX systems, in which a child process automatically inherits a pipe created by its parent, Windows requires the programmer to specify which attributes the child process will inherit. This is accomplished by first initializing the SECURITY--ATTRIBUTES structure to allow handles to be inherited and then redirecting the child process's handles for standard input or standard output to the read or write handle of the pipe. Since the child will be reading from the pipe, the parent must redirect the child's standard input to the read handle of the pipe. Furthermore, as the pipes are half duplex, it is necessary to prohibit the child from inheriting the write end of the pipe. Before writing to the pipe, the parent first closes its unused read end of the pipe. The child process that reads from the pipe is shown in Figure 5. Before reading from the pipe, this program obtains the read handle to the pipe by invoking GetStdHandle (). Note that ordinary pipes require a parent-child relationship between the communicating processes on both UNIX and Windows systems. This means that these pipes can be used only for communication between processes on the same machine.

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

   /* get the read handle of the pipe */
   ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

   /* the child reads from the pipe */
   if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
     printf("child read %s",buffer);
   else
     fprintf(stderr, "Error reading from pipe");

   return 0;
}
```

*Figure 5: Windows anonymous pipes-child process*

# PART I: NAMED PIPES

Ordinary pipes provide a simple communication mechanism between a pair of processes. However, ordinary pipes exist only while the processes are communicating with one another. On both UNIX and Windows systems, once the processes have finished communicating and terminated, the ordinary pipe ceases to exist.

Named pipes provide a much more powerful communication tool; communication can be bidirectional, and no parent-child relationship is required. Once a named pipe is established, several processes can use it for communication. In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished. Both UNIX and Windows systems support named pipes, although the details of implementation vary greatly. Next, we explore named pipes in each of these systems.

Named pipes are referred to as FIFOs in UNIX systems. Once created, they appear as typical files in the file system. A FIFO is created with the mkfifo () system call and manipulated with the ordinary open(), read(), write(), and close () system calls. It will continue to exist until it is explicitly deleted from the file system. Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted. If data must travel in both directions, two FIFOs are typically used. Additionally, the communicating processes must reside on the same machine; sockets must be used if intermachine communication is required.

Named pipes on Windows systems provide a richer communication mechanism than their UNIX counterparts. Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines. Additionally, only byte-oriented data may be transmitted across a UNIX FIFO, whereas Windows systems allow either byte- or message-oriented data. Named pipes are created with the CreateNamedPipe () function, and a client can connect to a named pipe using ConnectNamedPipe (). Communication over the named pipe can be accomplished using the ReadFile () and WriteFile () functions.

Named pipes are used for inter-process communications. Features:

- Exist as special files in the physical file system
- Any unrelated processes can access a named pipe and share data through it
- Access to named pipes is regulated by the usual file permissions
- Pipe data is accessed in a FIFO style
- Once created, a named pipe remains in the file system until explicitly deleted
- Creation: By UNIX shell commands.

Example: *mknod <filename> p*

*mkfifo a=rw <filename>*

Example:

mknod(char *pathname, mode_t mode, dev_t dev);

mknod("/tmp/myfifo", S_IFIFO | 0660, 0);

Same I/O operations style on named pipes and regular files – open(), read() and write() calls. Implementation of I/O operations:

- By system calls.
- By library functions.

Semantics of open() call:

- Blocking. The process that opens the named pipe for reading, sleeps until another process opens it for writing, and v.v.
- Non-blocking. Flag O_NONBLOCK, used in open() call disables default blocking. Pipes have size limitations.

Example

**Named pipes. Example**

Client-server communication through a pipe

**Server**


```
#include <fcntl.h>
...
#define PIPE "fifo"
int main(){
  int fd;
  char readbuf[20];
  mknod(PIPE, S_IFIFO | 0660, 0);   // create pipe
  fd = open(PIPE, O_RDONLY, 0);     // open pipe
  for (;;) {
    if (read(fd, &readbuf, sizeof(readbuf)) < 0){   //read from pipe
      perror("Error reading pipe");
      exit(1);
    }
    printf("Received string: %s\n", readbuf);   // process data
  }
  exit(0);
}
```

**Client**

```
#include <stdio.h>
...
#define PIPE "fifo"

int main(){
  int fd;
  char writebuf[20] = "Hello"; // open pipe
  fd = open(PIPE, O_WRONLY, 0);
  // write to pipe
  write(fd, writebuf, sizeof(writebuf));
  exit(0);
}
```

## PIPES IN PRACTICE

Pipes are used quite often in. the UNIX command-line environment for situations in which the output of one command serves as input to the second. For example; the UNIX *ls* command produces a directory listing. For especially long directory listings; the output may scroll through several screens. The command *more* manages output by displaying only one screen of output at a time; the user must press the space bar to move from one screen to the next. Setting up a pipe between the ***ls*** and ***more*** commands (which are running as individual processes) allows the output of *ls* to be delivered as the input to ***more***, enabling the user to display a large directory listing a screen at a time. A pipe can be constructed on the command line using the '|' character. The complete command is ***ls |more.*** In this scenario; the ***ls*** command serves as the producer, and its output is consumed by the *more* command. Windows systems provide a ***more*** command for the DOS shell with functionality similar to that of its UNIX counterpart. The DOS shell also uses the |character for establishing a pipe. The only difference is that to get a directory listing, DOS uses the ***dir*** command rather than ***ls***. The equivalent command in DOS to what is shown above is ***dir | more.***

## FILE HANDLE

A number that the operating system assigns temporarily to a file when it is opened. The operating system uses the file handle internally when accessing the file. A special area of main memory is reserved for file handles and the size of this area determines how many files can be opened at once.