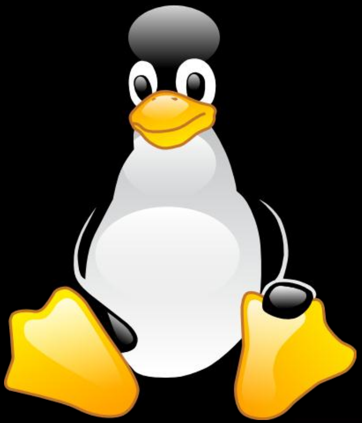
Tushar B Kute,
Assistant Professor,
Sandip Institute of Technology and Research Centre, Nashik (INDIA)
tbkute@gmail.com

University of Pune
T.E. I.T.
Subject code: 314441

OPERATING SYSTEM

Part 17 : Binary Semaphore, Pipe and Files



Semaphore

- It is the type of semaphore that takes only two values i.e. a and 1.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int
value);
int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);
int sem_destroy(sem_t * sem);
```

sem_init( )

- It creates a binary semaphore.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- This function initializes a semaphore object pointed to by *sem*, sets its sharing option and gives it an initial integer *value*.
- The *pshared* parameter controls the type of semaphore. If the value of *pshared* is 0, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes.

sem_post( )

```
int sem_post(sem_t * sem);
```

- It atomically increases the value of the semaphore by 1. Atomically here means that if two threads simultaneously try to increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment, and write a value to a file at the same time.
- If both programs try to increase the value by 1, the semaphore will always be correctly increased in value by 2.

sem_wait( )

```
int sem_wait(sem_t * sem);
```

- It atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first. If sem_wait is called on a semaphore with a value of 0, the function will wait until some other thread has incremented the value so that it is no longer 0.

sem_destroy( )

```
int sem_destroy(sem_t * sem);
```

- This function takes a pointer to a semaphore and tidies up any resources that it may have. If we attempt to destroy a semaphore for which some thread is waiting, we will get an error.
- Like most Linux functions, *all* these functions all return 0 on success.

Low Level file access

- Each running program, called a process, has a number of file descriptors associated with it. These are small integers that we can use to access open files or devices. How many of these are available will vary depending on how the system has been configured. When a program starts, it usually has three of these descriptors already opened. These are:
  - 0: Standard input
  - 1: Standard output
  - 2: Standard error
- We can associate other file descriptors with files and devices by using the open system call. The file descriptors that are automatically opened, however, already allow us to create some simple programs using write.

The write( ) system call

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
```

- It arranges for the first *nbytes* bytes from *buf* to be written to the file associated with the file descriptor *fildes*.
- It returns the number of bytes actually written. This may be less than *nbytes* if there has been an error in the file descriptor. If the function returns 0, it means no data was written; if it returns –1, there has been an error in the write call.

The read( ) system call

```
#include <unistd.h>
size_t read(int fildes, void *buf, size_t nbytes);
```

- It reads up to *nbytes* bytes of data from the file associated with the file descriptor *fildes* and places them in the data area *buf*.
- It returns the number of data bytes actually read, which may be less than the number requested. If a read call returns 0, it had nothing to read; it reached the end of the file. Again, an error on the call will cause it to return –1.

Pipe

- We use the term pipe to mean connecting a data flow from one process to another. You attach, or pipe, the output of one process to the input of another.
- For linking shell commands together so that the output of one process is fed straight to the input of another.
- For shell commands, this is done using the pipe character to join the commands, such as

```
cmd1 | cmd2
```

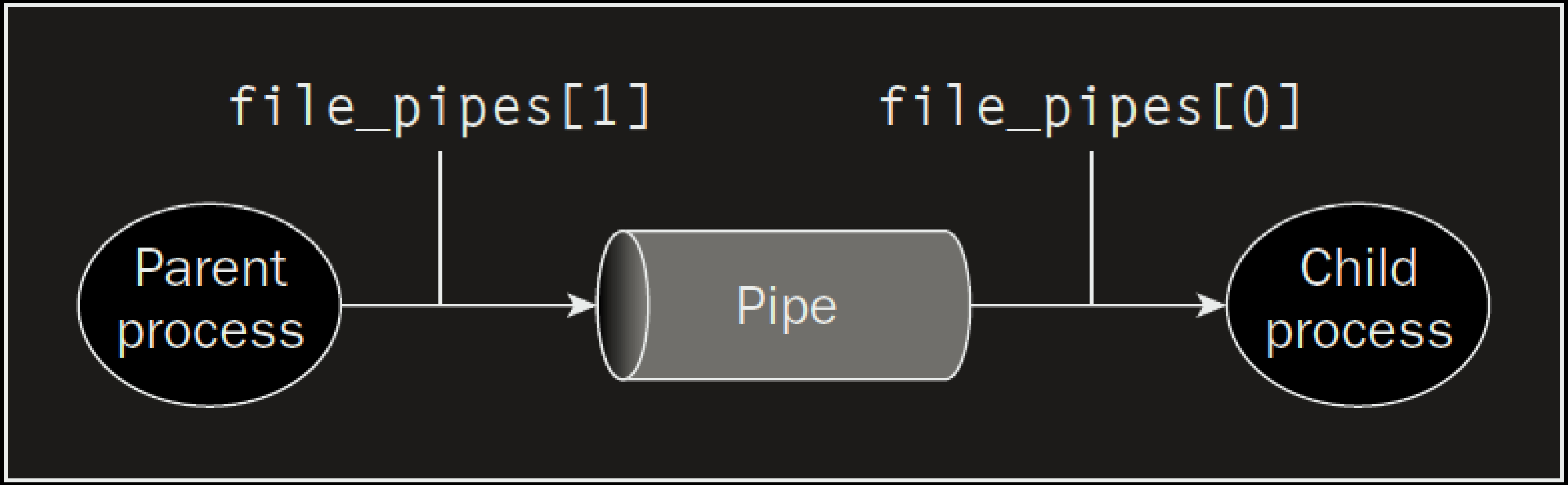
The pipe( ) call

- This is the lower-level pipe function which provides a means of passing data between two programs, without the overhead of invoking a shell to interpret the requested command. It also gives us more control over the reading and writing of data.

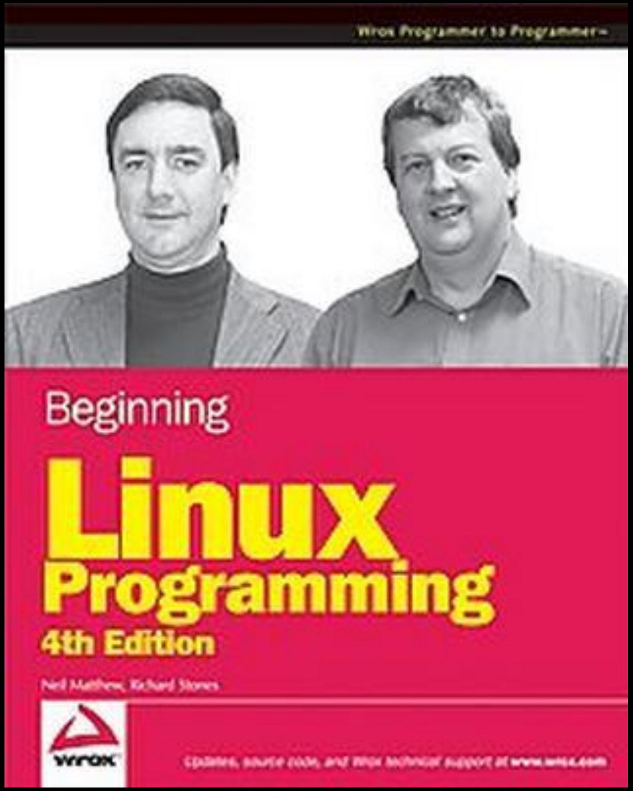
```
#include <unistd.h>
int pipe(int file_pipe[2]);
```

- It is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero. On failure, it returns -1.
- The two file descriptors returned are connected in a special way. Any data written to *file_pipe[1]* can be read back from *file_pipe[0]*. The data is processed in a FIFO. This means that if we write the bytes 1, 2, 3 to *file_pipe[1]*, reading from *file_pipe[0]* will produce 1, 2, 3.

How it works?



Reference Books



- "Beginning Linux Programming", 4th Edition, Neil Mathew, Richard Stones, Wrox Publication.
- Rating: 🐧🐧🐧🐧🐧