COMP20200 Unix Programming

Lecture 18

Ravi Reddy Manumachu (ravi.manumachu@ucd.ie)

CS, University College Dublin, Ireland

12/04/2023

Lecture overview

- Overview of UNIX interprocess communication (IPC).
- Learn the oldest method of IPC, UNIX pipes.

UNIX Interprocess communication (IPC)

IPC are tools that processes and threads can use to communicate with one another and to synchronize their actions.

Communication	Synchronization	Signal
pipe, FIFO, stream socket	System V semaphore POSIX semaphore - named POSIX semaphore - unnamed	standard signal realtime signal
pseudoterminal	eventfd	
System V message queue, POSIX message queue, datagram socket	file lock (flock()) record lock (fcntl())	
System V shared memory POSIX shared memory Memory mapping - anonymous Memory mapping - mapped file	futex mutex (threads) condition variable (threads) barrier	
Covered in this Module		
stream and datagram sockets (Lectures 14-17), mutex (L10), pipe (L18)		

Figure: UNIX IPC tools.

UNIX Interprocess communication (IPC)

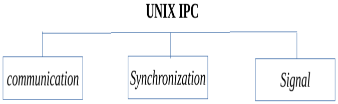

Figure: UNIX IPC features.

There are three broad functional categories in IPC:

- *Communication*: Tools concerned with exchanging data between processes.
- *Synchronization*: Tools concerned with synchronizing the actions of processes or threads.
- *Signals*: Can be used for synchronization or communication. *Review lecture on signals*.

IPC communication tools - Data transfer

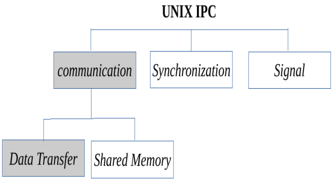

Figure: IPC data transfer facility.

There are two main categories under *communication*.
The first category is **Data transfer**.

- One process writes to the IPC facility, and another process reads the data.
- Two data transfers required between user memory and kernel memory.
- One transfer from user memory to kernel memory during writing.
- One transfer from kernel memory to user memory during reading.
- Synchronization between the reader and writer processes is automatic.

IPC communication tools - Shared memory

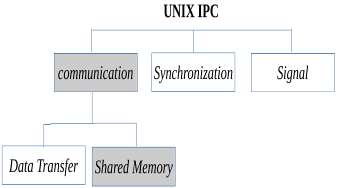

Figure: IPC shared memory facility.

The second category is **Shared memory**.

- Processes exchange information by placing it in a region of physical memory that is shared between the processes.
- The kernel does this by making page-table entries in each process point to the same pages of RAM.
- Communication doesn't require system calls or data transfer between user memory and kernel memory.
- So, this IPC can be fast but synchronizing reads and writes can be tricky.

IPC data transfer tools - Byte stream

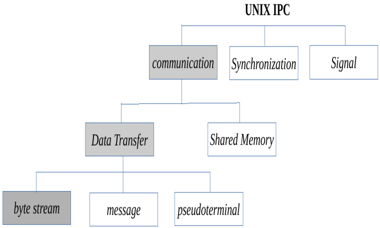

Figure: IPC data transfer facility, byte stream.

Following are the main categories under data transfer:

Byte stream:

- Data exchanged via stream sockets, pipes, and FIFOs is an undelimited byte stream.
- There is no concept of message or message boundaries.
- *We cover stream sockets and pipes in this module.*

IPC data transfer tools - Message

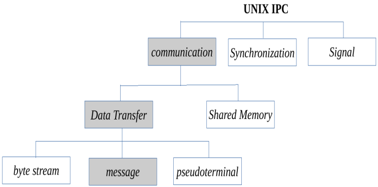

Figure: IPC data transfer facility, message.

Message:

- Data exchanged via datagram sockets and message queues are delimited messages.
- Each read operation reads a whole message, as written by the writer process.
- It is not possible to read part of a message.
- It is not possible to read multiple messages in a single read operation.

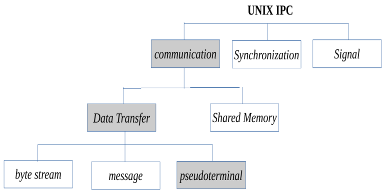

Figure: IPC data transfer facility, pseudoterminal.

Pseudoterminal:

- It is a pair of virtual devices, known as the master and slave.
- This pair of devices provides an IPC channel for data transfer between the two devices.
- Allows a user on one host to operate a terminal-oriented program on another host connected via a network.
- Employed in terminal emulators and *ssh*, which provides network login service.

IPC: Synchronization tools

Following are the main categories under synchronization:

Semaphores:

- A semaphore is a kernel-maintained integer whose value is never permitted to fall below 0.
- A process can increase or decrease the value of the semaphore.
- If an attempt is made to decrease the value of the semaphore below 0, then the kernel blocks the operation.
- A process decrements a semaphore in order to reserve exclusive access to some shared resource, and
- After completing work on the resource, a process increments the semaphore so that the shared resource is released for use.

File locks:

- Synchronize the actions of multiple processes operating on the same file.
- *fcntl()* system call provides record locking, allowing processes to place multiple read and write locks on different regions of the same file.

IPC: Synchronization tools

Mutexes and Condition variables:

- A mutex (short for mutual exclusion) ensures that only one thread at a time can access a shared variable.
- A mutex prevents multiple threads from accessing a shared variable at the same time.
- A condition variable allows one thread to inform other threads about changes in the condition/state of a shared variable, and
- Allows the other threads to wait (block) for such notification.
- Revisit lecture on threads (L11).

UNIX IPC using Pipes

IPC: Data transfer using pipes

Pipes can be used to pass data between *related* processes.

Consider the following:

```
$ cat iserver.c | wc -l
```

- *cat* prints the contents of the file iserver.c to standard output.
- *wc -l* outputs the numbers of lines in the input given to it.
- Shell creates two processes (using *fork()* and *exec()*).
- One process executes *cat* and the other, *wc*.
- The output produced by *cat* process is piped as input to *wc*.
- The vertical bar between the two programs represents a pipe.

IPC Pipe

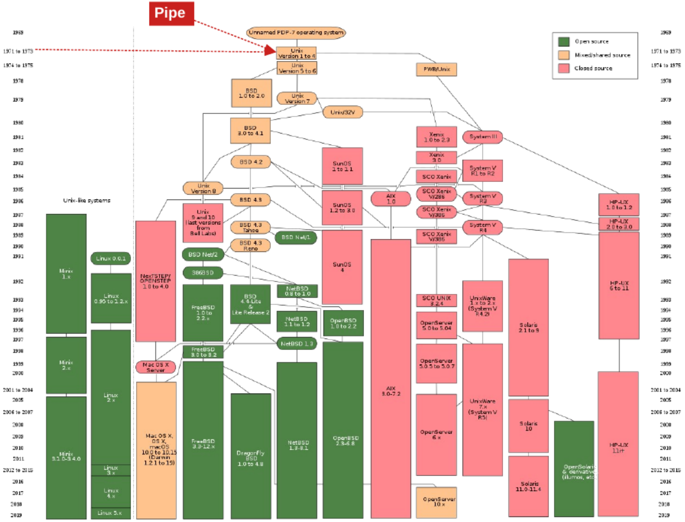

Figure: Pipes appeared in Second Edition UNIX in 1972.

IPC Pipe API

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

- Two open file descriptors are returned in *fd*, read end in *fd[0]* and the write end in *fd[1]*.
- The *read()* and *write()* system calls can be used to perform I/O on the pipe.
- Pipes are used for communication between related processes. For example:
  - Parent process and child process.
  - Parent process and its grandchild.
  - Two sibling processes (two child processes created by a parent).

IPC Pipe

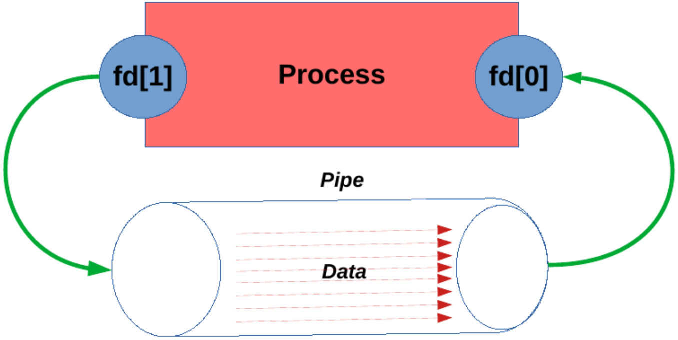

Figure: File descriptors after creation and direction of data flow.

IPC pipes

- A pipe is a byte stream. There is no concept of messages or message boundaries when using a pipe.
- Pipes are unidirectional. One end of the pipe is used for writing, and the other end is used for reading.
- Pipes have a limited capacity. A pipe is a buffer maintained in kernel memory.
- However, the application doesn't need to know the capacity (or maximum) of a pipe.
- Once a pipe is full, further writes to the pipe will block until the reader removes some data from the pipe.

IPC Pipe - Parent to child

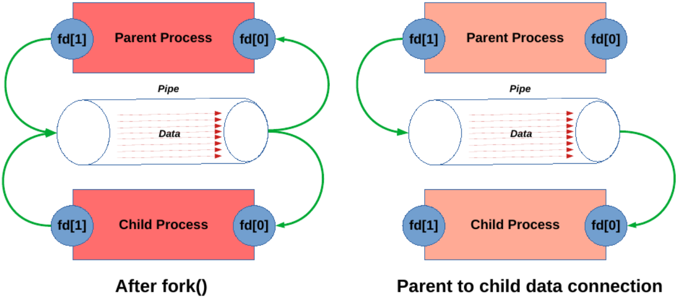

Figure: Simple pipe program creating a channel from parent to child.

Simple pipe program

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define BUFSIZE 10
int main(int argc, char *argv[])
{
    int pfd[2];
    char buf[BUFSIZE];

    if (pipe(pfd) == -1)
        exit(EXIT_FAILURE);
/* Cont'd... */
```

- The program takes a string as an input. The parent process writes the string to be received by the child process.
- *pipe()* system call is invoked to create the pipe.

Simple pipe program: Child process code

```
switch (fork()) {
case 0:
    if (close(pfd[1]) == -1)
        exit(EXIT_FAILURE);
/* Cont'd... */
```

The child process closes the write end of the pipe (*pfd[1]*) since it is required to read only from the parent process.

Simple pipe program: Child process code

```
switch (fork()) {
case 0:
    for (;;) {
        numRead = read(pfd[0], buf, BUFSIZE);
        if (numRead == -1)
            exit(EXIT_FAILURE);
        if (numRead == 0)
            break;
/* Cont'd... */
```

- The child enters a loop to read blocks of data (of size BUFSIZE) and writes it to the standard output.
- *numRead* of 0 represents end-of-file (EOF) condition.
- The child process exits the loop when it receives the end-of-file (EOF) condition.

Simple pipe program: Child process code

```
switch (fork()) {
case 0:

      if (write(1, buf, numRead) != numRead)
          exit(EXIT_FAILURE);
   } /* for loop end */

   write(1, "\n", 1);
/* Cont'd... */
```

- The child process writes what it has read to the standard output.
- Note the use of *write()* system call to write to standard output represented by the file descriptor 1.

Simple pipe program: Child process code

```
switch (fork()) {
case 0:
    if (close(pfd[0]) == -1)
        exit(EXIT_FAILURE);

    _exit(EXIT_SUCCESS);

/* Cont'd... */
```

- It then closes the read-end of the pipe (*pfd[0]*).
- A child process terminates by issuing an *_exit()* call.

Simple pipe program: Parent process code

```
switch (fork()) {
default:
    if (close(pfd[0]) == -1)
        exit(EXIT_FAILURE);
    if (write(pfd[1], argv[1], strlen(argv[1])) != strlen(argv[1]))
        exit(EXIT_FAILURE);
    ...
/* Cont'd */
```

- The parent process closes the read end of the pipe (*pfd[0]*).
- The string to be written to the pipe is passed to the program as the first argument (*argv[1]*).
- It then writes the input string to the write end of the pipe, *pfd[1]*.

Simple pipe program: Parent process code

```
switch (fork()) {
default:
    if (close(pfd[1]) == -1)
        exit(EXIT_FAILURE);
    wait(NULL);
    exit(EXIT_SUCCESS);
}
```

- The parent process then closes the write end of the pipe so that the child process encounters EOF condition.
- It finally waits for the child to finish before exiting.
- The *wait()* call blocks until the child terminates.
- *What happens if child process terminates before parent process calls wait()?*

Simple pipe program: Parent process code

```
switch (fork()) {
default:
  if (close(pfd[1]) == -1)
    exit(EXIT_FAILURE);
  wait(NULL);
  exit(EXIT_SUCCESS);
}
```

- As discussed in the previous lecture, *wait()* call allows parent to reap the child process and disallow it to become a zombie.
- The parent passes NULL to the *wait()* call since it is not interested in the exit status of the child.
- *Why no SIGCHLD signal handler?*

exit() and _exit() calls

- *exit()* is a library function; *_exit()* is a system call.
- Programs typically call the *exit()* library function.
- *exit()* performs the following actions:
  - Exit handlers.
  - stdio stream buffers are flushed.
  - *_exit()* system call is invoked.
- Programs also exit by issuing **return n**, which is the same as *exit(n)*.

exit() and _exit() calls

- In an application with parent and child processes, typically only one of the parent and child terminate by calling *exit()*.
- The other should terminate by calling *_exit()*.
- This is done so that only one process calls exit handlers and flushes *stdio* buffers.
- When this is not done, you observe duplication of printf strings to the standard output.

Simple pipe program: Execution

spipe.c - Contains the simple pipe program code.

```
$ gcc -o spipe spipe.c
$ ./spipe "UNIX was initially developed at Bell Labs and became
    operational on a PDP-7 in 1970."
Parent: Writing 'UNIX was initially developed at Bell Labs and
    became operational on a PDP-7 in 1970.' to child.
Child received
'UNIX was initially developed at Bell Labs and became operational
    on a PDP-7 in 1970.'
```

Lectures 19-21: Bash scripting language

We will focus on Bash scripting language in the next three lectures.

Q & A