Operating Systems 2017/2018

TP Class 07 – Pipes, Named Pipes and I/O Multiplexing

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Slides based on previous versions from Bruno Cabral, Paulo Marques and Luis Silva.

operating system
noun
the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

Abbreviation: OS

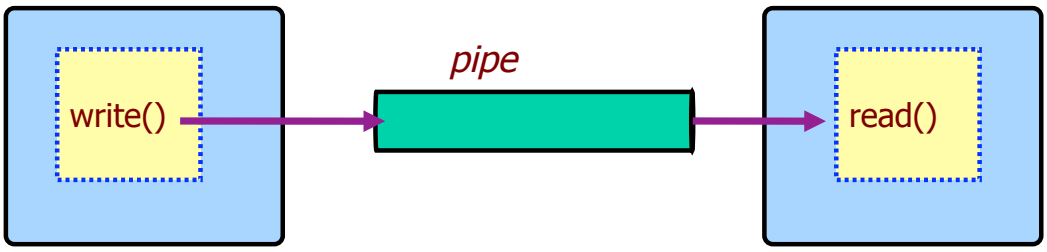
Source: Dictionary.com

· U C · FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE DE COIMBRA

Updated on 26 October 2017

Stream mode of communication

- Pipes and Named Pipes allow processes to communicate using "streams of data"
  - A "pipe" is a connection between two processes. You can send things through the pipe, you can try to receive things from the pipe.



- A pipe acts like a synchronous finite buffer.
  - If a process tries to write to a pipe that is full, it blocks
  - If a process tries to read from a pipe that is empty, it blocks

Pipes (unnamed pipes)

- Provides for communication amount processes that are hierarchically related (i.e. father-child)
  - Pipes must be created prior to creating child processes
- Whenever a pipe is created, using `pipe()`, two file descriptors are opened: one for reading (fd[0]), one for writing (fd[1])
  - Unused file descriptors should be closed!
- Pipes are unidirectional

```
int fd[2];

pipe(fd);
```

Example
demo01-pipes.c

```
typedef struct {
  int a;
  int b;
} numbers;

// File descriptors for the pipe channel
int channel[2];
(...)

int main() {
  // Create a pipe
  pipe(channel);

 // Create the processes
  if (fork() == 0) {
    worker();
    exit(0);
  }
  master();
  wait(NULL);

  return 0;
}
```

Example (cont.)

```
void worker() {
  numbers n;

  close(channel[1]);

  while (1) {
    read(channel[0], &n, sizeof(numbers));
    printf("[WORKER] Received (%d,%d) from master to add. Result=%d\n",
           n.a, n.b, n.a+n.b);
  }
}

void master()
{
  numbers n;

  close(channel[0]);

  while (1) {
    n.a = rand() % 100;
    n.b = rand() % 100;

    printf("[MASTER] Sending (%d,%d) for WORKER to add\n", n.a, n.b);
    write(channel[1], &n, sizeof(numbers));
    sleep(2);
  }
}
```

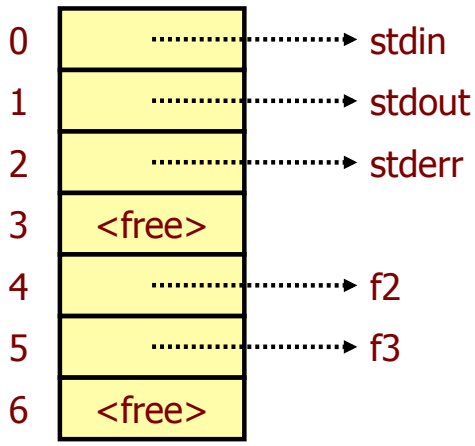

Be careful!

- A pipe is a finite buffer. If you try to write too much too quickly into it, the process will block until some space clears up.
- Atomicity is something to be dealt with
  - If you try to write less that **PIPE_BUF** bytes into a pipe, you are guaranteed that it will be written atomically (**PIPE_BUF** is a system variable defined in the **limits.h** file).
  - It you try to write more, you have no guaranties! If several processes are writing at the same time, the writes can be interleaved
  - Also, when a process tries to read from a pipe, you are not guaranteed that it will be able to read everything
- Meaning...
  - You must synchronize your writes when you're writing a lot of data!
  - You must ensure that you read complete messages!

```
struct person p;

int n, total = 0;
while (total < sizeof(p)) {
  n = read(fd[0], (char*)p + total, sizeof(p)-total);
  total+= n;
}
```

Controlling File Descriptors

- Each process has a file descriptor table. By default, entries 0, 1 and 2 are: *stdin*, *stdout*, *stderr*.
- Each time a file is opened, an entry is added to this table. Each time a file is closed, the corresponding entry becomes available.
- The process descriptor table, in fact, contains only references to the OS global file descriptor table.



File Descriptor Table after:
open("f1")
open("f2")
open("f3")
close("f1")

Controlling File Descriptors (2)

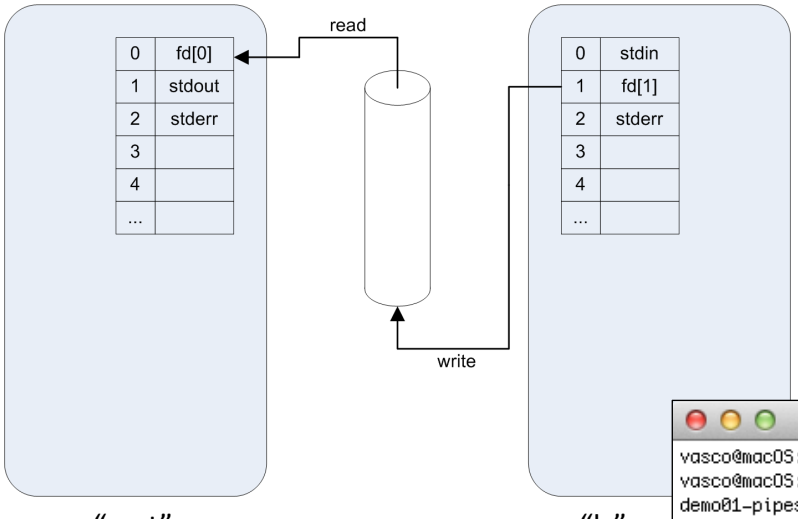
- Two routines are useful for controlling file descriptors:
  - int dup(int fd)
    - Duplicates file descriptor "fd" on the first available position of the file descriptor table.
  - int dup2(int fd, int newfd)
    - Duplicates file descriptor "fd" on the "newfd" position, closing it if necessary.
- Note that after a file descriptor is duplicated, the original and the duplicate can be used interchangeably. They share the file pointers, the buffers, locks, etc.
  - Careful: Closing one file descriptor doesn't close all other that have been duplicated!

Implementing a pipe between two processes

- Implementing a pipe between two processes is quite easy. It's only necessary to associate the standard output of one process with the standard input of another.
- Simple example: "ls | sort".
- Note: closing one file descriptor doesn't close all other that have been duplicated!

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

int main()
{
  // Create a pipe for associating "ls" with "sort"
  int fd[2];
  pipe(fd);

  if (fork() == 0) {
    // Redirect stdout to the input of the pipe and
    // close unneeded file descriptors
    dup2(fd[1], fileno(stdout));
    close(fd[0]);
    close(fd[1]);

    // Become ls
    execlp("ls", "ls", NULL);
  }
  else {
    // Redirect stdout to the exit of the pipe and
    // close unneeded file descriptors
    dup2(fd[0], fileno(stdin));
    close(fd[0]);
    close(fd[1]);

    // Become sort
    execlp("sort", "sort", NULL);
  }

  return 0;
}
```


Implementing a pipe between two processes (2)



- Create process
- Create pipe
- Create child
- Redirect fds
- Close fds not needed
- Execute "ls" and "sort"

```
Terminal — bash — 50×11
vasco@macOS:demos$ gcc -Wall demo_dup2.c -o pipe
vasco@macOS:demos$ ./pipe
demo01-pipes.c
demo01-pipes_select_signals.c
demo01-unnamed_pipes_select.c
demo_dup2.c
np_client.c
np_server.c
pipe
printerd.c
vasco@macOS:demos$
```

Named Pipes (also known as FIFOs)

- Similar to pipes but allow communication between unrelated processes.
  - Each pipe has a name (string).
  - The pipe is written persistently in the file system.
  - For creating a named pipe, use the "mkfifo" command or call mkfifo(const char* filename, mode_t mode);

- Typically, like pipes, they are half-duplex
  - Means that they must be open read-only or write-only
  - They are opened like files, but they are not files
  - You cannot `fseek()` a named pipe; `write()` always appends to the pipe, `read()` always returns data from the beginning of the pipe.
  - After data is read from the named pipe, it's no longer there. It's not a file, it's an object in the Unix kernel!

Unrelated client/server program

np_server.c

```
#define PIPE_NAME    "np_client_server"
(...)

int main()
{
  // Creates the named pipe if it doesn't exist yet
  if ((mkfifo(PIPE_NAME, O_CREAT|O_EXCL|0600)<0) && (errno!= EEXIST))  {
    perror("Cannot create pipe: ");
    exit(0);
  }

  // Opens the pipe for reading
  int fd;
  if ((fd = open(PIPE_NAME, O_RDONLY)) < 0) {
    perror("Cannot open pipe for reading: ");
    exit(0);
  }

  // Do some work
  numbers n;
  while (1) {
    read(fd, &n, sizeof(numbers));
    printf("[SERVER] Received (%d,%d), adding it: %d\n",
           n.a, n.b, n.a+n.b);
  }
  return 0;
}
```

Unrelated client/server program
np_client.c

```
#define PIPE_NAME   "np_client_server"
(...)

int main()
{
  // Opens the pipe for writing
  int fd;
  if ((fd = open(PIPE_NAME, O_WRONLY)) < 0) {
    perror("Cannot open pipe for writing: ");
    exit(0);
  }

  // Do some work
  while (1) {
    numbers n;
    n.a = rand() % 100;
    n.b = rand() % 100;
    printf("[CLIENT] Sending (%d,%d) for adding\n", n.a, n.b);
    write(fd, &n, sizeof(numbers));
    sleep(2);
  }

  return 0;
}
```


Some interesting issues...

- If you get a SIGPIPE signal, this means that you are trying to read/write from a closed pipe
- A named pipe is a connection between two processes. A process blocks until the other party opens the pipe...
  - Being it for reading or writing.
  - It's possible to bypass this behaviour (open it non-blocking – O_NONBLOCK), but be very, very careful: if not properly programmed, it can lead to busy waiting. If a named pipe is open non-blocking, EOF is indicated when `read()` returns `0`.
  - When designing a client/server multiple client application, this means that either the pipe is re-opened after each client disconnects, or the pipe is open read-write.
  - If opened "read-write", the server will not block until the other party connects (since, he itself is also another party!)

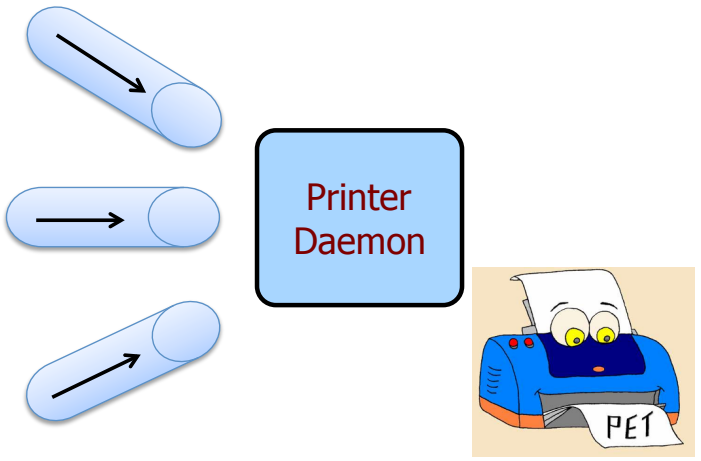
Interesting Problem

- A printer daemon is connected to a physical printer
- There are 3 named-pipes which allow automatic formatted printing

/printer/a4_double_sided

/printer/a4_single_sided

/printer/a3_single_sided



Interesting Problem

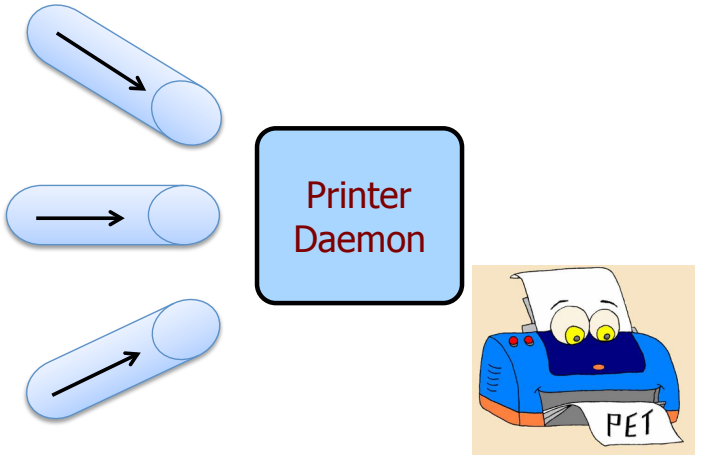
- Pooling?

Pipes are blocking, so this doesn't work!!!

/printer/a4_double_sided

/printer/a4_single_sided

/printer/a3_single_sided



I/O Multiplexing

- I/O Multiplexing: The ability to examine several file descriptors at the same time
  - select() and pselect()

```
int select(int n,
           fd_set* readfd,
           fd_set* writefd,
           fd_set* exceptfd,
           struct timeval* timeout)
```

- Greatest fd plus one
- For reading activity
- For writing activity
- For out-of-band activity

Blocks until activity is detected or a timeout occurs.

The fd_set variables are input/output. Upon return, they indicate if there was activity in a certain descriptor or not.

select()

- Careful: **n** is the number of the highest file descriptor added of one.
  - It's not the number of file descriptors

fd_set

A bit set representing file descriptors

- FD_ZERO(fd_set* set)
  - Cleans up the file descriptor set
- FD_SET(int fd, fd_set* set)
  - Sets a bit in the file descriptor set
- FD_CLEAR(int fd, fd_set* set)
  - Clears a bit in the file descriptor set
- FD_ISSET(int fd, fd_set* set)
  - Tests if a file descriptor is set

Example
printerd.c

```
(...)

#define BUF_SIZE                    4096
#define NUM_PRINTERS                3

const char* PRINTER_NAME[] = {
  "printer1", "printer2", "printer3"
};

// The printer file descriptors
int printer[NUM_PRINTERS];

void create_printers() {
  for (int i=0; i<NUM_PRINTERS; i++) {
    unlink(PRINTER_NAME[i]);
    mkfifo(PRINTER_NAME[i], O_CREAT|O_EXCL|0666);
    printer[i] = open(PRINTER_NAME[i], O_RDONLY|O_NONBLOCK);
    assert(printer[i] >= 0);
  }
}

int main(int argc, char* argv[]) {
  create_printers();
  accept_requests();
}
```


Example (2)
printerd.c

```
void accept_requests() {
  while (1) {
    fd_set read_set;
    FD_ZERO(&read_set);
    for (int i=0; i<NUM_PRINTERS; i++)
      FD_SET(printer[i], &read_set);

    if ( select(printer[NUM_PRINTERS-1]+1, &read_set, NULL, NULL, NULL) > 0 ) {
      for (int i=0; i<NUM_PRINTERS; i++)  {
        if (FD_ISSET(printer[i], &read_set)) {
          printf("[<%s> PRINTING]: ", PRINTER_NAME[i]);

          char buf[BUF_SIZE];
          int n = 0;
          do {
            n = read(printer[i], buf, BUF_SIZE);
            if (n > 0) {
              buf[n] = '\0';
              printf("%s", buf);
            }
          } while (n > 0);

          close(printer[i]);
          printer[i] = open(PRINTER_NAME[i], O_RDONLY|O_NONBLOCK);
        }
  } } }
}
```

INTRODUCTION TO ASSIGNMENT 07 – "SIGNALS AND PIPES"

Thank you! Questions?

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.
—Rudyard Kipling