CS2850 Operating System Lab

Week 8: Pipes

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

Inter-process communication

Anonymous pipes

C implementation

Processes

A *process* is the operating system's abstraction of a running program.

When multiple processes run concurrently, each process appears to have *exclusive* use of the hardware.

The process context is the *state information* the process needs to run.

The OS keeps the context associated with concurrent processes separate.

Inter-process communication

Can two processes communicate?

To connect the context of two processes you need an IPC *system call*.

Example: IPC occurs when a parent process *waits* for a child to terminate (by calling `wait`).

The child sends a *termination* signal to the parent that *can be decoded* through the macro `WEXISTATUS`

example

IPC between a child and its parent.

```c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
#include <stdlib.h>
int main() {
  int status, pid;
  int N = getchar()- '0';
  int *vOut = malloc(sizeof(int) * N);
  int *vPid = malloc(sizeof(int) * N);
  pid_t pid;
  for (int j = 0; j < N; j++) {
    if (!(pid=fork())) return rand() % (1 + j);
    *(vPid + j) = wait(&status);
    *(vOut + j) = WEXITSTATUS(status);
  }
  for (int k = 0; k < N; k++) printf("%d-pid=%d returns %d \n",
        k, *(vPid + k), *(vOut+k));
  free(vOut);
  free(vPid);
}
```

More general IPC

There exist different types of IPC:

- *Signals*: simple messages that are not used to transfer data,
- *Anonymous pipes*: unidirectional data channels between *related processes*,
- *Named pipes*: more flexible pipes that are treated like a file,
- *Shared memory*: a memory block that can be accessed by multiple concurrent processes,
- ...

See Section 6 of the Linux online manual or Wikipedia page on IPC for more details about UNIX and general IPC.

Linux channels

A channel is an IPC model for message passing.

It has a shareable reference that allows more processes to access it.

Processes can access the channel to write a message *in* it or read a message *from* it.

Channels normally have two distinct references, one for *reading* or *writing*.

They behave like one-way tunnels with two *gates*: an *entry* on one side and an *exit* on the other.

Writing and reading

Processes connected through a channel know the reference of (at least) one end of the channel so that

- one of them can *push* data into the channel and
- one of them can *pull* data out of the channel

i.e. reading and writing happen at different ends.

Written data *travel* through the tunnel before they are read and arrive in the order they are sent (FIFO).

Data written into the channel are *buffered* by the OS until they are read from the other end.

Implementation

In C, you can implement 3 types of channels:

- *half-duplex UNIX pipes*: communication between related processes, e.g. a parent and a child or the children of the same parent,
- *FIFOs named pipes*: communication between two independent processes,
- *sockets*: communication between different computers,
- ...

UNIX half-duplex (*anonymous*) pipes

Anonymous pipes are the eldest of the IPC tools.

A half-duplex pipe connects processes that *share a related ancestor*, i.e.

- a parent and child or
- children of the same parent.

Defining pipes requires kernel-level operations, i.e. a pipe is created by making a *system call*, e.g. by calling `pipe()` in a C program.

2-way pipes can be created by opening two pipes.

Example

Pipes are used in the UNIX shell to connect the *output* of one process to the *input* of another one, e.g. in

```
ls  sort
```

Intuitively, you can imagine data flow through the channel from the *left* to the *right* both the `ls` and `sort` processes are *children of the same process*, i.e. the shell.

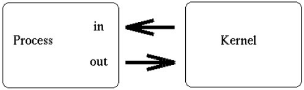
Creating a pipe: step 1

The system call `pipe()` creates two file descriptors associated with the *reading* and *writing* ends:

- `fd0` for *reading*,
- `fd1` for *writing*.

The new pipe initially connects a process to itself.

Data *written to* and *read from* the pipe travels through the kernel.



Step 1: code

```c
#include <stdio.h>
#include <unistd.h>
#define MAXCHARS 100
int main() {
  int fd[2];
  pipe(fd);
  printf("fd[0]=%d, fd[1]=%d\n", fd[0], fd[1]);
  char buf[MAXCHARS];
  *buf = '\0';
  printf("buf = %s\n", buf);
  int nbytes = write(fd[1], "hello world", 12);//write to fd[1]
  printf("%d written to fd[1]\n", nbytes);
  nbytes = read(fd[0], buf, nbytes);
  printf("%d read from fd[0]\n", nbytes);// and read from fd[0]
  printf("buf = %s\n", buf);
  return 0;
}
```

Step 1: output

The output of the program is

```
fd[0]=3, fd[1]=4
buf =
12 bytes written to fd[1]
12 bytes read from fd[0]
buf = hello world
```

File descriptors

File descriptors, e.g. `fd[0]` and `fd[1]`, are used by *low-level* I/O functions as `read` and `write`. [1]

The *standard library* functions refer to files through file *handles* or streams, i.e. *structures* that contain more information about the file.

Given a file descriptor, you can obtain the corresponding stream and *vice versa* through.

```
FILE *streamPointer = fdopen (f0, openMode)
int f1 = fileno(streamPointer)
```

where `f0` and `f1` are (equal) integers, `openMode` a string, e.g. `"w"` or `"r"`, and `streamPointer` a pointer to a `FILE` structure.

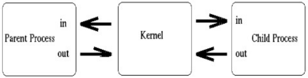
[1] See Slides 4 or Section 13.4 of the GNU online manual for more details about low-level I/O

Creating a pipe: step 2

What happens if the process generates a child process *after creating* the pipe?

As `fork` makes an *exact copy* of the parent process, the pipe file descriptors are copied from the parent into the child,

Both processes have then access to the pipe and can use it to *communicate*.

]


Step 2: code

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int copyString(char *out, char *in);
int main() {
  int fd[2], nBytes;
  pipe(fd);
  char out[100], in[100], author[100];
  if (!fork()) {//child process
    sleep(1);
    nBytes=copyString(in, "world");
    copyString(author, "child");
  }
  else { //parent process
    nBytes=copyString(in, "hello");
    copyString(author, "parent");
  }
  nBytes = write(fd[1], in, nBytes);//write to fd[1]
  printf("the %s writes to fd[1]: %s (%d)\n", author, in, nBytes);
  wait(NULL);
  nBytes = read(fd[0], out, nBytes);
  printf("the %s reads from fd[0]: %s (%d)\n",author, out, nBytes);
}
```

Step 2: copyString and program output

copyString is defined by

```
int copyString(char *out, char *in) {
  int k = 0;
  while (*(in + k) != '\0') {
    *(out + k) = *(in + k);
    k++;
  }
  *(out +k) = '\0';
  return k;
}
```

The program produces the following output

```
the parent writes to fd[1]: hello (5)
the child writes to fd[1]: world (5)
the child reads from fd[0]: hello (5)
the parent reads from fd[0]: world (5)
```

Problems

Why do we get such an output?

The child process inherits both file descriptors, `fd[0]` and `fd[1]` both processes can write to and read from the channel.

While the child is sleeping the parent writes to the channel.

The order of the messages is kept in the channel (channels are FIFO containers) so messages written by the parent and the child may get mixed.

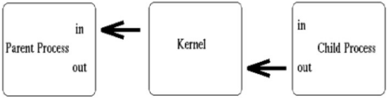
Creating a pipe: step 3

You must decide *who writes* and *who reads* i.e. in which direction data will travel.

The processes should *close* the unused end of the pipe, i.e.

- close `fd[0]` if the process will be *writing* and
- close `fd[1]` if the process will be *reading*

Example: The child *writes* and the parent *reads*.



Pseudo-code

`createPipe.c` creates a message-passing system between a parent process and its child in a few steps.

- The parent creates a half-duplex pipe. (3)
- The parent generates a child process by calling `fork`.(5)
- Both processes close the *unused* ends of the pipe (6 and 15).
- The child's sends messages to the parent by calling `writeMessages`. (9)
- The parent starts reading the child messages and prints them on `stdout` by calling `readMessages`.(19)
- The child sends a signal, e.g. a message containing the *keyword* `quit`, to tell the parent that the *last message* has been sent. (10)
- The child and the parents close the channel. (11 and 20)

createPipe.c (main)

```
int main() {
  int fd[2];
  pipe(fd);
  char  process[MAXCHARS];
  if (!fork()) {
    close(fd[0]);
    copyString(process, "the child");
    writeMessage(fd[1], "hello", process);
    writeMessage(fd[1], "world", process);
    writeMessage(fd[1], "stop", process);
    close(fd[1]);
    return 0;
  }
  else {
    close(fd[1]);
    copyString(process, "the parent");
    int end = 0;
    while(end == 0)
      end = readMessage(fd[0], process);
    close(fd[0]);
  }
}
```

createPipe.c (Headers and subroutines declaration)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#define MAXCHARS 100
int readMessage(int fd, char *reader);
int writeMessage(int fd, char *buf, char *author);
int compareString(char *s1, char *s2);
int copyString(char *in, char *out);
int stringLength(char *s);
```

createPipe.c (Read and write functions)

```
int writeMessage(int fd, char *buf, char *author) {
  int nbytes = stringLength(buf);
  printf("%s writes to fd[1]: %s (%d) \n", author, buf, nbytes);
  nbytes = write(fd, buf, nbytes + 1);
  return 0;
}
int readMessage(int fd, char *reader) {
  char c = ' ';
  int i = 0;
  char buf[MAXCHARS];
  while (c != '\0') {
    read(fd, &c, 1);
    buf[i++] = c;
  }
  buf[i] = '\0';
  if (compareString(buf, "stop") == 0) return -1;
  printf("%s read from fd[0]: %s (%d) \n", reader, buf, stringLength(buf));
  return 0;
}
```

Step 3: code (string functions)

```
int copyString(char *out, char *in) {
  int k = 0;
  while (*(in + k) != '\0') {
    *(out + k) = *(in + k);
    k++;
  }
  *(out + k) = '\0';
  return k;
}
int stringLength(char *s) {
  int i = 0;
  while (*(s + i) != '\0')
    i++;
  return i;
}
int compareString(char *s1, char *s2) {
  int i = 0;
  while (*(s1 + i) != '\0' && *(s2 + i) != '\0')
    if (*(s1 + i) != *(s2 + i)) return -1;
    else i++;
  if (*(s1 + i) == *(s2 + i)) return 0;
  else return -1;
}
```

Step 3: `createPipe.c` (output)

```
the child writes to fd[1]: hello (5)
the child writes to fd[1]: world (5)
the child writes to fd[1]: stop (4)
the parent read from fd[0]: hello (5)
the parent read from fd[0]: world (5)
```

Note: The implementation of `stringLength` makes the program print the actual size of a word, excluding the null-termination character.

Would it be possible for the parent to read `hello` *before the child writes* `world`?