C/UNIX Seminar - January 2022

Day 3: I/O, signals and Introduction to IPC

Assistants Salles Machine 2024

I/O in C

- High level buffered I/O
- Low-level I/O
- Working with files

Inter Process Communication

I/O in C

I/O in C

- Two types on I/O: high and low level.
- Available in <stdio.h> or <unistd.h>.
- Concept of buffers for high level functions like *printf*.

High level buffered I/O

The high level I/O is the most common one: *printf*, *puts*, etc... "*Buffered*" functions, meaning the content given to these functions is not always printed. The input is collected in buffers to write as much as possible at once. Then, everything gets outputed when:

- The buffer is full.
- A newline character is encountered.

There is actually a third one, but not relevant here.

Formatted operations

- *printf(3) and scanf(3)* family.
- Variadic functions: can take an infinite amount of arguments.
- Format strings: *%d*, *%s*, ...

Feel free to look at their respective man page.

Printing values

```c
#include <stdio.h>
int main()
{
    int i = 42;
    float f = 3.141592;

    printf("i with a leading space = |%3d|\n", i);
    printf("i with a leading zero = |%03d|\n", i);
    printf("f = %g\n", f);
    fprintf(stderr, "Printing on stderr...\n");

    return 0;
}
```

Reading values

```c
#include <stdio.h>
int main()
{
    int age;

    puts("What is your age?");
    scanf("%d", &age);

    printf("You are %i years old.\n", age);

    return 0;
}
```

Low-level I/O

The first notable difference with low-level I/O functions is that they require a *file descriptor*.
A file descriptor, or *fd* is a unique identifier - a number - for a file or other I/O resource, like network sockets or pipes.

The three FDs that interest us here are:

- 0: standard input *STDIN_FILENO*.
- 1: standard output *STDOUT_FILENO*.
- 2: standard error *STDERR_FILENO*.

Syscalls

Syscalls are "particular" functions. They are actually wrappers around existing OS operations and can be used as any other C function. Their man-page differ though as they are featured in section 2 of the manual, instead of the usual section 3.
Also, these functions are usually **blocking**, meaning that your program will wait for them to be finished before moving on.

read(2)

```c
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

- Reads at most *count* bytes from *fd*. *count* should be lower or equal to the size of the buffer.
- Puts the read input in *buf*.
- Returns the number of bytes - characters in most cases - read. If it is equal to 0, reached *end-of-file*, everything was read; else if -1, an error occurred.

write(2)

```c
#include <unistd.h>
ssize_t write(int fd, void *buf, size_t count);
```

- Writes at most *count* bytes from *buf*. *count* should be lower or equal to the size of the buffer.
- Writes the content of the buffer in *fd*.
- Returns the number of bytes written. If it is equal to -1, an error occurred.

Example

```c
#include <err.h>
#include <unistd.h>

#define BUF_SIZE 128

int main()
{
    ssize_t r;
    char buf[BUF_SIZE];
    while ((r = read(STDIN_FILENO, buf, BUF_SIZE)) != 0)
    {
        if (r == -1)
            errx(3, "Reading failure.");
        r = write(STDOUT_FILENO, buf, r);
        if (r == -1)
            errx(3, "Writing failure.");
    }
    return 0;
}
```

Error handling

As usual, you are expected to handle all possible errors. To do so, introducing: <errno.h> and its variable, *errno*.
A failing syscall will set *errno* to a specific value, and the possible errors are described in the different syscalls' man-pages. The explicited error can be obtained with *strerror(3)* or *err(3)*.

Also, there are two ways to notify errors:

- *err(3)* family: puts a message on stderr and exits the program.
- *warn(3)* family: puts a message on stderr but does not stop the program.

Working with files

```c
#include <fcntl.h>
int open(const char *path, int flags);
int open(const char *path, int flags, mode_t mode);
```

- Returns a *fd* to the given *path*.
- *flags* is used to specify the opening mode (reading, writing, etc)
- *mode* sets the permissions of a file when creating it.
- Returns -1 in case of an error.

```c
#include <fcntl.h>

/* ... */
  int fd;
  if ((fd = open("foo.txt", O_RDONLY)) == -1)
    err(3, "error while opening %s", "foo.txt");
/* Do something with the content of the file */
  close(fd);
```

Getting file information: stat(2)

```c
#include <sys/stat.h>
int stat(const char *path, struct stat*buf);
```

Allows to get:

- the size of a file,
- its permissions,
- the date of creation, modification, ...

Feel free to read the man-page to learn more about these functions and what they permit.

Seeking in a file: seek(2)

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

In a sense, this function allows to move the *offset* of the resource designed by *fd*. An easier explanation would be like moving a cursor inside of a file.
whence specifies how the offset is to be interpreted:

- SEEK_SET: file offset set to *offset* bytes.
- SEEK_CUR: file offset set to its current location plus *offset*.
- SEEK_END: file offset set to file size plus *offset*.

Inter Process Communication

Process

- An instance of a running program is called a process
- It can be identified by its PID (Process ID)
- Each instance of a running program has a unique PID

Example

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

int main(void)
{
    pid_t pid = getpid();

    printf("The pid of the current program is %d\n", pid);

    return 0;
}
```

Or...

Even the terminal you use is a process that has it's own PID.
You can access the PID using the $$ variable:

```
echo $$
```

(This will display the PID of the current terminal)

IPC (Inter Process Communication)

Each process is independent which means each process has its own memory pool. That is why we need IPCs:

Mechanisms an operating system provides to allow the **processes to manage shared data** and "communicate" between each other

Types of IPCs

1. Pipes
2. Shared Memory
3. Message Queue
4. Direct Communication (targeted)
5. Indirect communication (uses intermediary mailbox)
6. Message Passing (i.e.: sockets)
7. FIFO

Signals

- Simple asynchronous Inter Process Communication (IPC)
- Designed mostly for exceptional behavior
- Most signals kill the process by default
- Most signals are catchable

Signals

To send signal: kill(2) syscall (also kill(1), a command!)

- Needs a signal number
- Needs a pid to send the signal to
- Requires ownership or enough privileges

Find the list of existing signal numbers with man 7 signal or using 'kill -l on a terminal

Basic Signal Handling

- Define functions to be called when signal arrives
- Not all signals can be handled

Some Common Signals

- SIGHUP
- SIGINT
- **SIGKILL**
- **SIGSTOP**
- SIGTERM
- SIGSEGV
- SIGCHLD
- SIGCONT
- SIGBUS
- SIGILL

More Flexible: sigaction(2)

- Provides more possibilities than signal(2)
- Provides finer control over signal handling
- Can give a lot more information to your handlers

asm 2024