Operating systems. Lecture 8

Michał Goliński

2018-11-20

Introduction

Recall

- CPU protection rings
- system calls
- pointers
- reading and writing files

Plan for today

- Reading and writing wiles in the C/C++ standard libraries
- System calls managing processes (`fork`, `exec` etc.)

Files – the C standard library

Library functions vs. system calls

Although reading and writing files (even standard input and output) from the OS point of view must happen through system calls, programmers do not usually use the raw system calls because of performance and portability reasons – system calls are expensive and use no buffering, moreover system calls and their semantics might differ between different operating systems. Library functions are universal and because they are buffered they are usually much faster.

Reading files in C

The basic structure that hold a reference to an open file is called `FILE`. A pointer to an initialized structure of this type is usually obtained by calling `fopen`:

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char *mode);
```

The argument are the file path and open mode (read/write etc.). The function returns `NULL` on error and set `errno` appropriately.

`stdin` etc.

The standard library gives initialized `FILE` structures: `stdin`, `stdout` and `stderr` for standard input, output etc.

Unformatted read/write

To read/write binary data one uses the functions `fread`/`fwrite`:

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE
 ↪  *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE
 ↪  *stream);
```

Semantics is very similar to `read` and `write`, but both can act on units larger than one byte.

Formatted read/write

To read/write formatted data one uses the functions `(f)printf`/`(f)scanf`:

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

Hopefully we all know how to use these functions by now.

Flushing the buffer

Calling `fwrite` does not necessarily mean that data is actually written into the file. The library buffers data and writes larger chunks at a time. Sometime we want to empty the buffer and write it immediately to the file. This is called *flushing* and is performed by `fflush`:

```
#include <stdio.h>
int fflush(FILE *stream);
```

In principle an **abrupt termination** of the program (e.g., because of a segmentation fault) may leave the data unflushed in the buffer.

Closing the file

To close a file one uses `fclose`:

```
#include <stdio.h>
int fclose(FILE *stream);
```

Closing a file flushes the associated buffer.

Files – the C++ standard library

Streams

C++ uses the idea of a *stream* together with operators » and «. To read and write files one would use `ifstream` ad `ofstream`. Constructing these object requires a file path. By choosing a proper type one decides between reading and writing (or both).

```
#include <fstream>
ofstream myfile("example.tx+t");
```

Formatted read/write

One can have formatted read and write using the stream operators « and ». Contrary to the C library there is an official mechanism to extend the stream operators to user defined types (be overloading the operators for the type).

Unformatted read/write

The C++ library streams support the methods `read` and `write` that allow to read/write a requested number of bytes.

Manipulating streams

The library supplies several *stream manipulators* – these are objects that do not necessarily output data themselves but change the state of the stream, e.g. change the precision of float numbers, minimal width of numbers etc. The most often used are `endl` and `flush`.

Synchronizing with `stdio.h`

Normally the standard text streams `cin`, `cout`, `cerr` are synchronized with `stdin`, `stdout` and `stderr`. This allows to mix C and C++ without problems, but this can impact performance. If one uses only the C++ library, then calling `cin.sync_with_stdio(false)` can improve performance.

Examples

`cp` clone with system calls

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

#define BUFSIZE 1

int main() {
    int src = open("/dev/zero", O_RDONLY);
    int dst = open("/dev/null", O_WRONLY);
        char buf[BUFSIZE];
    ssize_t s;
    ssize_t total = 0;
    while ((s = read(src, buf, BUFSIZE)) > 0) {
        write(dst, buf, s);
        total += s;
        if (total >= 20*1024*1024)
            break;
    }
    return 0;
}
```

`cp` clone with `stdio.h`

```c
#include <stdio.h>

#define BUFSIZE 1

int main() {
    FILE *src = fopen("/dev/zero", "r");
    FILE *dst = fopen("/dev/null", "w");
    char buf[BUFSIZE];
    size_t s;
    size_t total = 0;
    while ((s = fread(buf, 1, BUFSIZE, src)) > 0) {
        fwrite(buf, 1, BUFSIZE, dst);
        total += s;
        if (total >= 20*1024*1024)
            break;
    }
```

```
    return 0;
}
```

cp clone with C++ streams

```
#include <fstream>

#define BUFSIZE 4096

int main() {
    std::ifstream src("/dev/zero");
    std::ofstream dst("/dev/null");
    char buf[BUFSIZE];
    std::streamsize s;
    std::streamsize total = 0;
    while (src.read(buf, BUFSIZE)) {
        s = src.gcount();
        dst.write(buf, s);
        total += s;
        if (total >= 20*1024*1024)
            break;
    }
    return 0;
}
```

Processes

Introduction

After the Linux kernel finishes starting it starts a designated process that gets ID 1: `init`. It is the `init`'s responsibility to start all the other processes: from servers to shells. The kernel helps by providing the necessary system calls used for creating and supervising processes. All the other processes are descendants of `init`. Currently the most often used `init` implementation under Linux is systemd.

Getting the process ID

To get the ID of the current process (PID) one uses the `getpid` function:

```
#include <unistd.h>

pid_t getpid(void);
```

The ID does not change as long as the process exists.

Getting the parent process ID

Each process has a parent process (except `init`). The ID of the parent cat be obtained by calling `getppid`:

```
#include <unistd.h>
pid_t getppid(void);
```

If a parent process dies before its child, then the child is adopted, usually by `init`.

Creating new processes

This is a very peculiar system call: it creates an almost exact copy of the calling process as its child.

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

pid_t fork(void);
```

Incidentally this is (roughly) the only way to create new processes.

Return value

The `fork` function returns **twice**: in the parent and in the newly created child. In the parent it returns the PID of the child, in the child it returns 0.

Most resources are inherited from parent in the child: memory content, file descriptors etc.

Both parent and child resume execution at the same spot: the place where `fork` was called.

Copy-on-write

Calling `fork` on Linux is relatively cheap. This is because most data is not really copied, e.g., memory pages are used by both processes as long as neither of them writes to it. A true copy is created only when one of the processes writes data to the memory page.

Example 1

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

```c
int main() {
    printf("Before fork()\n");
    fork();
    printf("After fork()\n");
    return 0;
}
```

Example 2

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

int main() {
    printf("The PID of the parent: %jd\n", (intmax_t) getpid());
    pid_t child_id;
    if(child_id = fork()) {
        printf("In the parent, child has PID: %jd\n",
                (intmax_t) child_id);
    } else {
        printf("In the child, my PID is %jd, my parent has PID
↪   %jd\n",
                (intmax_t) getpid(), (intmax_t) getppid());
    }
    return 0;
}
```

Example 3

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

int main() {
    int i = 0;
    pid_t child_id;
    if(child_id = fork()) {
        i = 1;
        printf("In parent: i = %d\n", i);
    } else {
        i = 2;
        printf("In child: i = %d\n", i);
```

```
    }
    return 0;
}
```

`exec` – substitute another process

Of course copying a running process is rarely our goal. We may want to run a lengthy calculation side by side (multithreading) or we may wish to run a totally unrelated process. The `exec` function help with the latter. These functions (which differ in the way the arguments are passed) start a new process **in the current one**.

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ...
                /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...
                /* (char  *) NULL */);
int execle(const char *path, const char *arg, ...
                /*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
                char *const envp[]);
```

Calling conventions

The three `execl*` functions use a variable length list of arguments, while the `execv*` pass the arguments in an array. The functions with `p` use the `PATH` when looking for executables. All the functions do not normally return – a completely new process is started. File descriptors are normally inherited across an `exec` call.

Exiting a process

The C library gives a function that gracefully finishes a process:

```
#include <stdlib.h>
void exit(int status);
```

The call will flush all the buffers etc. returning the status to the parent. This function does not return. The underlying system call is much more abrupt:

```c
#include <unistd.h>
void _exit(int status);
```

This won't flush buffers etc.

Why do we need `exit`?

One might think that calling `exit` is unnecessary and the program finishes when the last instruction finishes. But at memory level there is no way to tell the CPU that the code has ended. Without `exit` the program would just continue trying to execute random data as code. Writing such a program is impossible in C, we need assembly.

```asm
; nasm -f elf64 -o hello.o hello.s
global _start
section .text
_start:
  mov rax, 1          ; write(
  mov rdi, 1          ;   STDOUT_FILENO,
  mov rsi, msg        ;   "Hello, world!\n",
  mov rdx, msglen     ;   sizeof("Hello, world!\n")
  syscall             ; );

;  mov rax, 60         ; exit(
;  mov rdi, 0          ;   EXIT_SUCCESS
;  syscall             ; );

section .rodata
  msg: db "Hello, world!", 10
  msglen: equ $ - msg
```

Waiting for a child

If a parent process wants to wait until the child process finishes its execution, it must use the `wait` function:

```c
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

This function will block until any child finishes its work. It returns the PID of the child that finished work and the child's status code. The function will finish immediately if the child has already finished or there is no process to wait for.

Sleeping

To pause the current process we can use one of these functions:

```c
#include <unistd.h>
unsigned int sleep(unsigned int seconds);

#include <time.h>
int nanosleep(const struct timespec *req, struct timespec *rem);
```

As an argument the take the time to sleep. The process will be preempted from the CPU for the specified time, unless a signal is delivered (both functions return the remaining time if that happens).

Example

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

int main() {
    pid_t child;
    if (child = fork()) {
        int wstatus;
        wait(&wstatus);
        printf("My IP address is: ");
        fflush(stdout);
        execl("/usr/bin/cat", "cat", "/tmp/internet", (char *)
 ↪  NULL);
        return 127;
    } else {
        execl("/usr/bin/wget", "wget", "http://ipinfo.io/ip",
 ↪  "-O", "/tmp/internet", "-q", (char *) NULL);
        return 126;
    }
}
```