CS2850 Operating System Lab

Week 4: processes and IO

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

outline

System calls

File system

Processes

`fork`

System calls

The operating system provides its services through a set *system calls*.

Accessing to the OS is mainly needed for

- input/output,
- file handling,
- memory allocation,
- ...

A portable interface

Different OS's may offer similar services in *different ways*.

In a C program, system calls are architecture-independent, e.g. `printf` performs a system call and its syntax is machine independent.

The portable interface is a set of *standard library functions*.

These routines *depend on the host OS*, i.e. they are written in terms of the facilities of a *specific* OS.

System calls from a C program

From a C program, you can make

- Task-dedicated UNIX system calls, e.g.

  ```
  int n_read = read(int fd, char *buf, int n);
  ```

- Direct system calls, through a general function declared in `unistd.h`. (for system calls that have no *wrapper* in the standard library), e.g.

  ```
  syscall(3, ... );
  ```

  where 3 is the number of the *read* system call.

`syscall`

`syscall` is a flexible function with a variable number of parameters,

```
long int sys_return = syscall(long int SYS_NO, ... );
```

- `SYS_NO` is the system call number, which identifies each kind of system call,
- the remaining *parameters* depend on the system call and the architecture,
- the return value is the return value of the specific system call.

`syscall` is useful for system calls that have no wrapper in the C library.

Example

The following program prints `hello, world` on screen.

```
#include <unistd.h>
int main() {
  syscall(1, 1, "hello world \n", 20);
}
```

What is the meaning of the first two arguments? What happens if you change the last argument to 6?

Low-level input-output

You can use `syscall` to implement low-level IO operations.

```
n_read = syscall(SYS_read, fd, buf, n)
n_written = syscall(SYS_write, fd, buf, n)
```

where `SYS_read` and `SYS_write` are the numbers associated with the *read* and *write* system calls in `sys/syscall.h`.

Primitive input-output

Reading and *writing* is easier with

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

- `n_read` or `n_written` is the number of bytes actually transferred (if an error occurs, `n_read` $<$ `n`, idem `n_write`),
- `fd` is a *file descriptor* (see below),
- `char *buf` is a character array,
- `n` is the number of bytes to be transferred.

`read` and `write` appear in *higher-level* functions that read or write, e.g. `getchar` or `printf`.

Example (1)

This program copies its input into its output.

```
#include <unistd.h>
int main() {
  char buf[20];
  int n;
  while((n = read(STDIN_FILENO, buf, 20))> 0)
    write(STDOUT_FILENO, buf, n);
}
```

The file descriptors `STDIN_FILENO` and `STDOUT_FILENO` are defined in `unistd.h` and refer to *standard input* (the keyboard) and *standard output* (the terminal).

Example (2)

This program is equivalent to the previous one.

```
#include <unistd.h>
int main() {
  char buf[20];
  int n;
  while((n = syscall(0, 0, buf, 20))> 0)
    syscall(1, 1, buf, n);
}
```

Read more about `read` and `write` on this page and more about `syscall` on this page of the The GNU C Library Reference Manual.

syscall codes

Run the following program to print on screen the syscall codes for a few IO and process control operations.

```
#include <stdio.h>
#include <sys/syscall.h>
int main() {
  printf("SYS_read=%d\n",SYS_read);
  printf("SYS_write=%d\n",SYS_write);
  printf("SYS_open=%d\n",SYS_open);
  printf("SYS_close=%d\n",SYS_close);
  printf("SYS_fork=%d\n",SYS_fork);
  printf("SYS_getpid=%d\n",SYS_getpid);
}
```

Why do you need to include stdio.h here? Why can you use file-IO facilities to read from and write on the terminal?

Streams

A file is a *stream of data*, i.e. a sequence of bytes.

What happens when you open a file?

1. the kernel checks your right to do so (Does the file exist? Do you have permission to access it?),
2. the kernel returns to the program a small non-negative integer called a file descriptor, and
3. the kernel keeps track of all information about the open file, e.g. a file position (offset from the beginning of a file).

File references

Direct system calls, e.g. `read`, `write`, and `syscall`, refers files through their file descriptor.

Higher-level IO facilities defined in the standard library refer to files through their file handle.

A file handle is an object of type `struct` containing

- a pointer to a buffer,
- the number of characters left in the buffer,
- a pointer to the next position in the buffer,
- the file descriptor,
- various flags, e.g. describing the read/write *mode*,
- ...

Unix IO

In a C program, the easiest way to open a file is to call

```
FILE *fopen(char *name, char *mode);
```

- FILE is the file handle-type.
- The name parameter is the file name (a string constant).

  The mode parameter controls how the file is to be opened ("r" for *read-access*, "w" for *write-access* and "a" for *append access*).

File descriptors and file handles

File handles, i.e. `struct` of type `FILE`, are composite objects and contain all the information you need to access an open file.

In most cases, file handles are *passed to functions* by passing a pointer to `FILE`.

File descriptors are simple integers. You can obtain the file descriptor of an open file from the file handle of an open file with

```
int fd = fileno(FILE *fh);
```

Normally, IO functions defined in `stdio.h`, e.g. `fopen` and `fileno`, refer to open files through file handles.

Lower-level file handling

The *lower-level* versions of `fopen` are defined in `fcntl.h` and return a file descriptor.

```
int open(char *name, int flag, int perms)
int creat(char *name, int perms)
```

See The GNU C Library Reference Manual for more details about the arguments `flags`, a parameter that controls how the file is opened, and `perms`, a number encoding the user's permissions.

`stdin` and `stdout`

In UNIX, all IO operations are done by reading or writing files all peripheral devices, even the keyboard and the screen, are files in the file system.

Printing on the terminal or reading from the keyboard is a special case.

All processes automatically open three files called `stdin` (*standard input*), `stdout` (*standard output*), and `stderr` (*standard error*), with fixed file descriptors 0, 1, and 2.

`stdin`, `stdout`, and `stderr` are file handles defined in `stdio.h`.

High-level reading and writing

In a C program, the easiest way to read and write on file is to use

```
int fscanf(FILE *f, char *format, ...)
int fprintf(FILE *f, char *format, ...)
```

Example: the following two calls are equivalent:

```
fprintf(stdout, "hello world");
printf("hello world);
```

Read more about `fscanf` and `fprintf` on the The GNU C Library Reference Manual.

Processes

A process is the OS abstraction of a program in execution.

Each program runs in the context of some process.

The context includes all state information needed to run the process: program code, data, stack, program counter, environment variables, ...

Concurrent processes

You always have the *illusion* that your program is the only one running on the system.

This happens because a process has

- an *independent* control flow and
- a *private* address space, i.e. the memory associated with a process cannot be read or written by any other process.

Multitasking is needed for running multiple *concurrent* processes in an apparently simultaneous way.

Process handling

Unix provides the basic system calls for manipulating processes from C programs.

A process has a unique process identifier (PID) obtained by calling

```
pit_t getpid(void);
```

defined in `unistd.h`.

The type of the return value (a positive integer), `pid_t`, is defined in `types.h`.

You can print the process identifier of your running program at any time by writing

```
printf("pid=%d\n", getpid());
```

States of a process

A process can be in three states:

i) *running*: executing on the CPU,

ii) *stopped*: the execution is suspended (until it receives a signal), or

iii) *terminated*: the process is stopped permanently

Child processes

A process *can create a new running process* by calling

```
pid = fork();
```

which is defined in `unistd.h` and returns twice:

1. `pid` $= 0$ in the child process, and
2. `pid` $=$ `pid_child`, where `pid_child` is the *pid of the generated child process* in the parent process.

`fork` returns -1 if the process creation failed.

Process termination

A process can be terminated by writing (in `main`)

- `return i;`

  which *returning the integer* `i`,

- `exit();`

  which has no return value.

Note: you need to include `stdlib.h` to use `exit`.

Child and parent are quasi-identical

`fork` creates a child process that is almost identical to its parent.

The child gets a *separate copy* of the parent's (user level) address space: code, data, heap, user stack.

The child gets copies of all parent's open file descriptors, e.g. the child can read and write any files that are *open in the parent when* `fork` *is called*.

This is why they can both print on `stdout`.

Parent and child are distinct

The parent and the child have different PID's.

`fork` is called once by the parent, but it returns twice, once in the parent and once in the child.

The parent and the child processes run concurrently changes made after calling `fork` are private i.e not reflected in the other process.

Read more about `fork` on The GNU C Library Reference Manual.

Waiting for a child process to terminate

Unix provides a few more system calls for manipulating processes from C programs.

A parent process waits for a child to terminate by calling the

```
pid_t wait(int *status)
```

which is defined in `sys/wait.h`.

The parameter `status` can be used to encode some *information about the termination of the child process*, [1], e.g. its return value.

The return value is *the PID of the child that terminates* or -1 if the process has no children

[1] Set `&status` to `NULL` if you do not need this information

Suspending a process

A process can be suspended by calling

```
unisgned int sleep(unsigned int secs)
```

which is defined in `unistd.h`.

`secs` is the number of seconds the process will sleep.

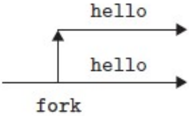
The return value is the number of seconds left to sleep (if it returns prematurely).

We suggest you read more about `wait` and `sleep` on The GNU C Library Reference Manual.

Examples (1)

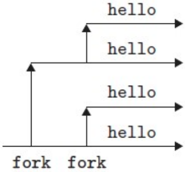
```
#include <stdio.h>
#include <unistd.h>

int main(){
        fork();
        printf("hello\n");
}
```




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

int main(){
        fork();
        fork();
        printf("hello\n");
}
```



Examples (2)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
  if (!fork())
    printf("the PID of the child is %d\n", getpid());
  else{
    printf("the return value of wait(NULL) is %d\n",
        wait(NULL));
    printf("the PID of the parent is %d\n", getpid());
  }
}
```

The program above prints

```
the PID of the child is 3384654
the return value of wait(NULL) is 3384654
the PID of the parent is 3384653
```