10.3 Run a program

```
1 #include <unistd.h>
2 int execve(const char *file, char *const argv[], char *const env[]);
```

- The `execve`(2) function **executes** the program stored in `file` (no `$PATH` search), passing arguments `argv` and environment `env`.
  - **Note** that arrays `argv` and `env` must be `NULL`-terminated!
  - By convention `argv[0]` shall be the **filename** of the program.
- On success, `execve` **does not return**.
- The **calling program is replaced** by the program loaded from `file`, replacing code, data, and stack of the caller.
- Most process attributes are preserved, most notably:
  - The **process ID** stays the same.
  - By default, **file descriptors** remain open.

  See `execve`(2) for a list of exceptions!

The exec family

- Setting up `argv[]` and `env[]` sometimes is cumbersome. The `exec`(3) family of functions facilitates this task:

Example `execlp`, One of the seven `exec` functions:

```
1 int execlp(const char *file, const char *arg0, ..., NULL);
```

- Consumes a variable list of **arguments** passed as `argv` to `main`. The **last one** must be `NULL`!
- If there's no slash `/` in the string `file`, then a `$PATH` search is performed, just like the shell would do.
  Warning: Insecure defaults if `$PATH` is unset!
- `execl` (without `p`) is similar, but does not perform a `$PATH` search.

```c
int main(void)
{
    int status;
    pid_t pid = fork();

    if (pid == 0) {
        execl("/usr/bin/sleep", "sleep", "100", NULL);
        err(1, "execl");
    }

    if (pid < 0) err(1, "fork");
    printf("Child %u\n", pid);

    if (wait(&status) < 0) err(1, "wait");
    if (WIFEXITED(status))
        printf("Child exit status was: %d\n", WEXITSTATUS(status));
    else
        printf("Child terminated without exit status.\n");
    return 0;
}
```

```
$ ./a.out
#...
-rwx------ 1 marcel users 8624 Jan  7 13:25 a.out
-rw------- 1 marcel users  432 Jan  7 13:24 exec.c
#...
Child exit status was: 0
```

10.4 Duplicating file descriptors

```
1 #include <unistd.h>
2 int dup(int oldfd);
3 int dup2(int oldfd, int newfd);
```

- Both functions return -1 on error, or a **new** file descriptor referring to the **same open file description** as `oldfd`.
  - They share offset and other flags.
- `dup`(2) uses the **lowest unused** number for the new file descriptor.
  - Old and new descriptor can be used **interchangably**.
  - Closing one does **not close** the other.
- `dup2`(2) uses the **given** `newfd` for the new file descriptor.
  - If `newfd` is an open file descriptor, it is closed first, ignoring errors!
  - Careful programming: Check `close(newfd)` for errors before using `dup2`.
- Both functions can be replaced with `fcntl`(2) and additional work.

Note Only **closing the last file descriptor** referring to the underlying open file description frees the associated resources, *cf.* `close`(2).

Redirection of *stdout* to a file

```
1  int main(void)
2  {
3      int fd = open("data.txt", O_WRONLY|O_CREAT|O_EXCL|O_TRUNC, 0666);
4      if (fd < 0) err(1, "open");
5      pid_t pid = fork();     if (pid < 0) err(1, "fork");
6
7      if (pid == 0) {  /* the child */
8          printf("C: Writing to data.txt\n");
9          close(STDOUT_FILENO);
10         if (dup2(fd, STDOUT_FILENO) < 0) err(1, "dup2");
11         close(fd);  /* not needed any more */
12         execlp("date", "date", "+%Y-%m-%d %H:%M:%S", NULL);   /* see date(1) */
13         err(1, "execlp");
14     }
15
16     /* the parent */
17     close(fd);  /* not needed any more */
18     printf("P: Waiting for child %d\n", pid);
19     int status;
20     wait(&status);
21     if (WIFEXITED(status))
22         printf("P: Child exited with %d\n", WEXITSTATUS(status));
23     return 0;
24 }
```

Running the previous program:

```
$ ./a.out
Parent: Waiting for child 4333
Child: Writing to foo
Parent: Child exited with 0
$ cat foo
2014-01-05 17:18:19
```

11
Inter-Process Communication

aka. IPC

11.1 Pipes & FIFOs

- Pipes are the **oldest** form of UNIX System IPC.
- An **overview** discussion of pipes & FIFOs is in `pipe`(7).

Pipes (detailed discussion on next slides)

- A pipe has two **file descriptors**, one for **reading** and one for **writing**. *I.e.*, access is done via `read`(2) and `write`(2).
- Pipes can be used only between processes that have a **common ancestor**.

FIFOs Like pipes, but with a **name** in the file system.

- Created with C function `mkfifo`(3), or command line tool `mkfifo`(1).
- The FS is just a convenient place to put the FIFO's name, no data is stored.
- Any process may `open`(2) a FIFO through its **file system name** (subject to permissions).
- Open with `O_RDONLY` to get the **reading end** of the FIFO, and `O_WRONLY` for the **writing end**.

Using pipes

```c
#include <unistd.h>
int pipe(int pipefd[2]);
```

- `pipe`(2) creates a pipe, **returning** the file descriptors in `pipefd`.
- `pipefd[0]` is the read end, `pipefd[1]` is the write end.
- Returns `0` on success. On error `-1` is returned, and `errno` is set.

```c
#define MAXLINE 100

int main(void)
{
    int fd[2];
    if (pipe(fd) < 0)
        err(1, "pipe");

    pid_t pid = fork();
    if (pid < 0)
        err(1, "fork");
    if (pid > 0) {                          /* parent */
        close(fd[0]);
        write(fd[1], "hello world", 11);
        close(fd[1]);

    } else {                                /* child */
        close(fd[1]);
        char line[MAXLINE];
        ssize_t n = read(fd[0], line, MAXLINE-1);
        line[n] = '\0';
        printf("From parent: %s\n", line);
        close(fd[0]);
    }
    return 0;
}
```

Further remarks

- Data written to a pipe can be **read only once**.
  - Multiple readers may see only parts of the information.
- Writing big data chunks may be **non-atomic**:
  - Multiple `read`s may be necessary to get **all data**.
  - Data written by different processes may be **interleaved**.
- Pipes have **limited capacity**:
  - 64kiB since Linux 2.6.11.
  - **Writing blocks** if the pipe is full, **reading blocks** if the pipe is empty.
  - If `O_NONBLOCK` is set for the according file descriptors, these operations fail instead (*cf.* `fcntl`(2)).
- Properties of pipe descriptors:
  - `fstat`(2) returns a file type of FIFO for both ends of a pipe.
  - We can test for a pipe with the `S_ISFIFO` macro, *cf.* `sys_stat.h`(0).
- What happens if the **other end** has been closed?
  - **Reading** returns 0 after all the data has been read.
  - **Writing** generates **signal** `SIGPIPE`. If we either ignore the signal or catch it and return from the signal handler, `write` returns -1 with `errno` set to `EPIPE`.

11.2 Signals

Definition A signal is a message notifying a process of an event.

- A signal is identified by an **integer**, and traditional signals carry **no extra data**.
  - Newer implementations allow to pass an additional `int`, and provide more information to the recipient.
- Linux knows 32 "standard" signals (plus 32 more).
- An overview of Linux' signal interface is in `signal`(7).

How to react to a signal

- **Ignore** the signal — not recommened in general.
- The predefined **default action** — termination in most cases.
- Catch a signal by installing a **handler** function.
- **Block** the signal — it becomes pending, until unblocked.

Some signals

Name	Description	Default Action
SIGINT	terminal interrupt (*e.g.*, pressed C-c)	terminate
SIGALRM	timer expired	terminate
SIGSEGV	invalid memory reference	terminate
SIGCHLD	change in status of a child	ignore
SIGUSR1	user-defined	terminate
SIGUSR2	user-defined	terminate
SIGTERM	termination	terminate
SIGSTOP	stop	stop process
SIGCONT	continue stopped process	continue/ignore
SIGKILL	termination	terminate

A complete list is in `signal`(7).

Note The default disposition of SIGKILL and SIGSTOP cannot be changed — they are not handled by the receiving program, but by the OS.

Sending signals

```c
#include <signal.h>

int kill(pid_t pid, int sig);
int raise(int sig);
```

```c
#include <unistd.h>

unsigned int alarm(unsigned int n);
```

- `kill`(2) sends signal `sig` to the process identified by `pid`.
  - `pid` may be zero, or even negative, to send to a group of processes.
- `raise`(3) sends signal `sig` to the calling process.
- Both return `0` on success, or `-1` on error, setting `errno`.
- `alarm`(2) schedules `SIGALRM` to be sent to the caller after `n` seconds.
  - Any previous alarm is **canceled**. If `n` is `0`, no alarm is set.
  - Returns number of **remaining seconds** before delivery of a previously set alarm, or `0` if none was set.

From the shell

`kill` [`-signal`] `pid`... Send signal, or `SIGTERM` to the processes identified by pids.

- This is a shell builtin.
- `kill`(1) describes the program `/bin/kill` instead.

Install a signal handler

```c
#define _POSIX_C_SOURCE 199309L
#include <signal.h>

int sigaction(int sig, const struct sigaction *act,
              struct sigaction *old);

struct sigaction {
    void (*sa_handler)(int);
    /* incomplete, we will extend this on the following slides */
};
```

- `sigaction`(2) sets/queries the action to be taken for signal `sig`.
- If not `NULL`, `act` describes the new action.
- The old setting is returned via `old`, if that is not `NULL`.
- Struct member `sa_handler` points to the function to be called, or is
  `SIG_DFL` to restore the default action, or
  `SIG_IGN` to ignore the signal.
- Returns `0` on success, or `-1` on error, setting `errno`.

Question What is the type of the handler?

First simple example

```
1  void handler(int s)  /* this is called with the signal number as argument */
2  {
3      printf("caught signal %d\n", s);
4      sleep(1);
5  }
6
7  int main(void)
8  {
9      struct sigaction sa;
10
11     memset(&sa, 0, sizeof(sa));     /* clear memory, cf. memset(3) */
12     sa.sa_handler = handler;
13
14     sigaction(SIGINT, &sa, NULL);   /* install handler for SIGINT */
15     sigaction(SIGUSR1, &sa, NULL);  /* same for SIGUSR1 */
16
17     printf("pid is %d\n", getpid());
18     while (1)
19         pause();  /* sleep until signal arrives, cf. pause(2) */
20
21     return 0;
22 }
```

Running the example

- Use two terminals, one to run the program, and one to send signals to it.

```
$ pk-cc -o signal0 signal0.c
$ ./signal0
pid is 10226
^Ccaught signal 2      # caused by pressing C-c
caught signal 2        # other terminal, line 1
caught signal 10       # ... line 2
Terminated             # ... line 3
$
```

```
$ kill -INT 10226
$ kill -USR1 10226
$ kill -TERM 10226
```

- Pressing C-c causes the terminal to send `SIGINT` to the process.
- The program does not handle `SIGTERM`, so the default action is taken.

Notes

- Signals `SIGKILL`, and `SIGSTOP` **cannot be caught**, blocked, or ignored.
- Again, there is more than one **interface** to signals. We will not discuss the older `signal`(2), due to several flaws.