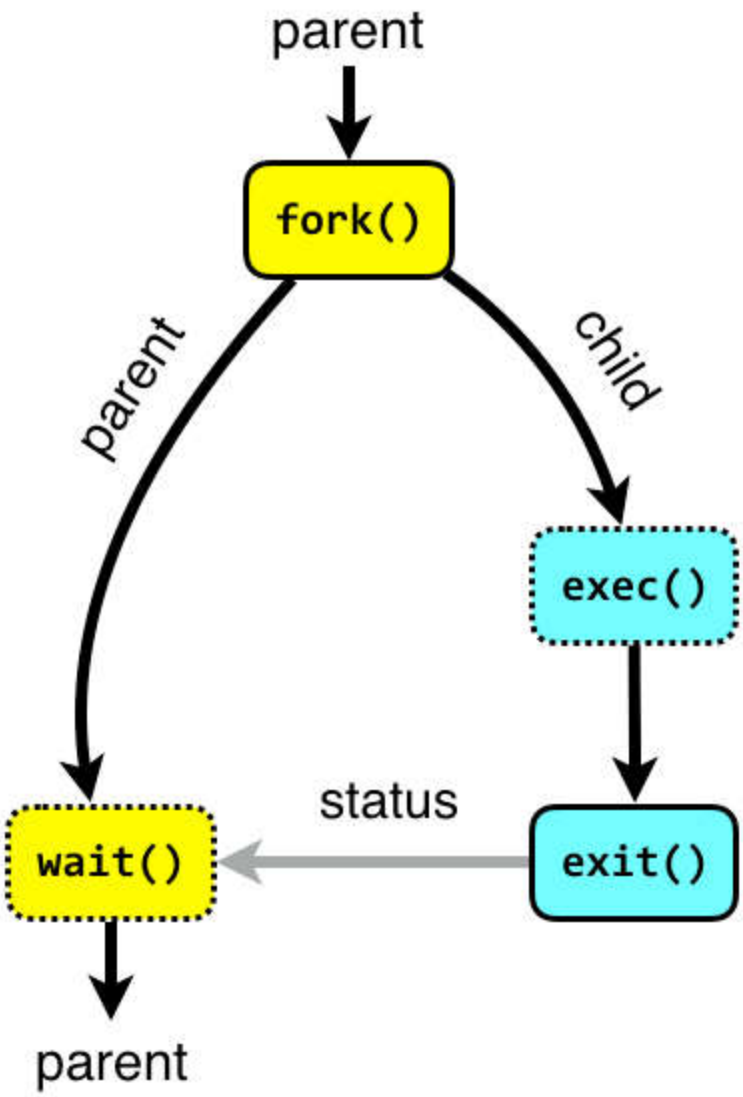
Read: Process Management

One of the philosophies behind Unix is the motto **do one thing and do it well (https://en.wikipedia.org/wiki/Unix_philosophy#Do_One_Thing_and_Do_It_Well)**. In this spirit, basic process management is done with a number of system calls, each with a single (simple) purpose. These system calls can then be combined to implement more complex behaviors.



The following system calls are used for basic process management.

fork

A parent process uses fork to create a new child process. The child process is a copy of the parent. After fork, both parent and child executes the same program but in separate processes.

exec

Replaces the program executed by a process. The child may use exec after a fork to replace the process' memory space with a new program executable making the child execute a different

program than the parent.

exit

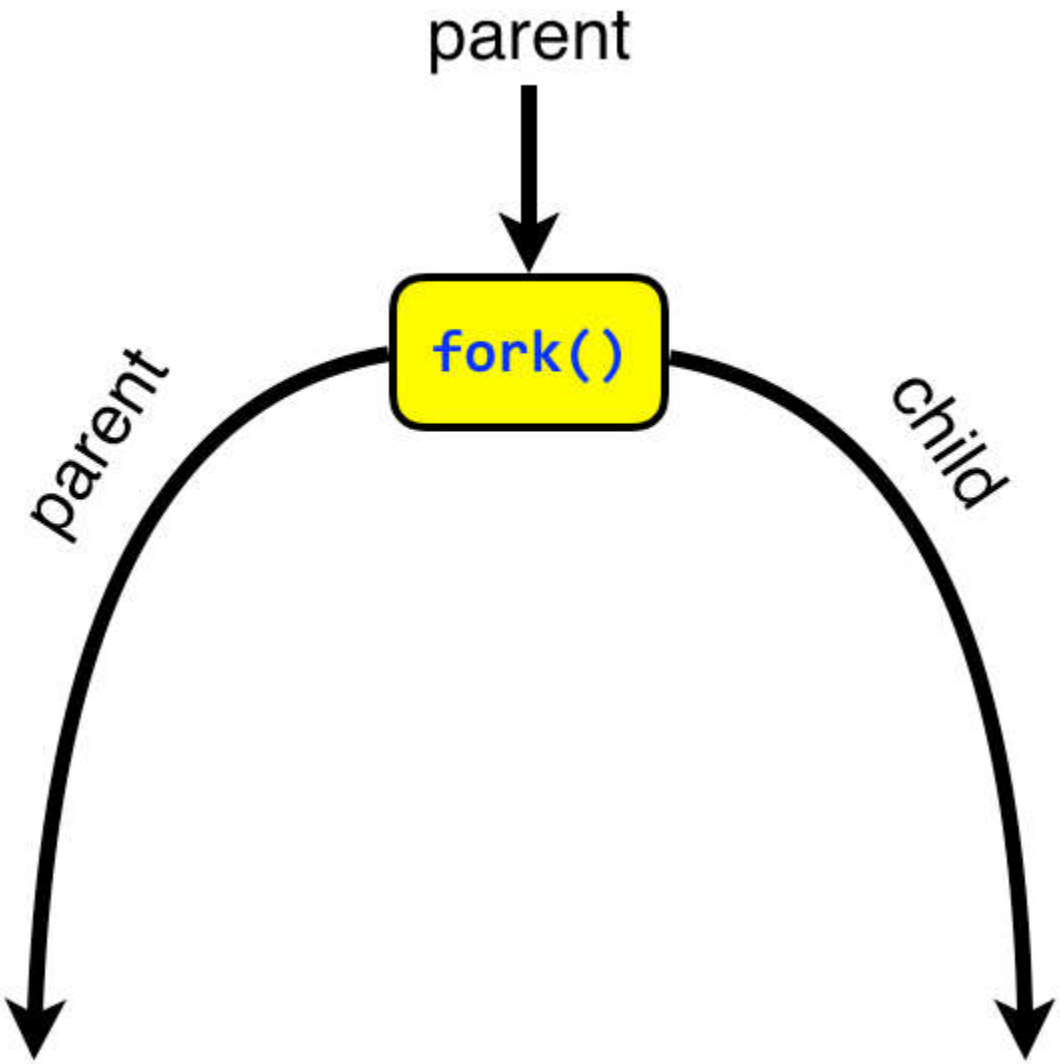
Terminates the process with an exit status.

wait

The parent may use wait to suspend execution until a child terminates. Using wait the parent can obtain the exit status of a terminated child.

Parent and child

The process invoking `fork` is called the **parent**. The new process created as the result of a `fork` is the **child** of the parent.



After a successful fork, the child process is a copy of the parent. The parent and child processes executes the same program but in separate processes.

Fork

The `fork` system call is the primary (and historically, only) method of process creation in Unix-like operating systems.

```
#include <unistd.h>

pid_t fork(void);
```

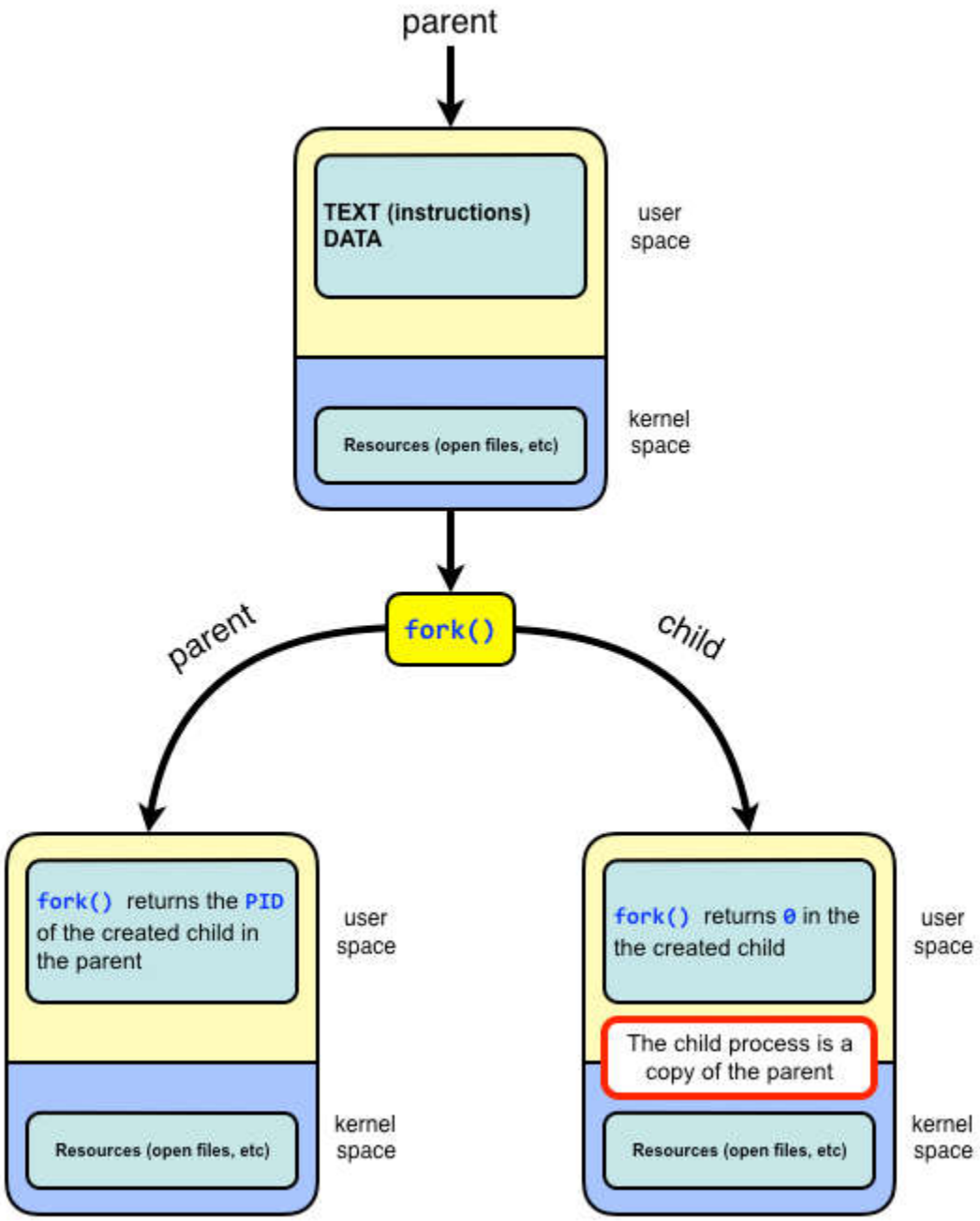
Return value

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

Fork returns twice on success

On success `fork` returns twice: once in the parent and once in the child. After calling `fork`, the program can use the fork return value to tell whether executing in the parent or child.

- If the return value is `0` the program executes in the new child process.
- If the return value is greater than zero, the program executes in the parent process and the return value is the process ID (PID) of the created child process.
- On failure `fork` returns `-1`.



Template program

In the file `module-2/examples/src/fork-template.c` you find a template for a typical program using `fork`.

```
1 #include <stdio.h>  // perror()
2 #include <stdlib.h> // exit(), EXIT_SUCCESS, EXIT_FAILURE
3 #include <unistd.h> // fork()
4
5 int main(void) {
6
7   pid_t pid;
```

```
 8
 9   switch (pid = fork()) {
10
11    case -1:
12       // On error fork() returns -1.
13       perror("fork failed");
14       exit(EXIT_FAILURE);
15
16    case 0:
17       // On success fork() returns 0 in the child.
18
19       // Add code for the child process here.
20
21       exit(EXIT_SUCCESS);
22
23    default:
24       // On success fork() returns the pid of the child to the parent.
25
26       // Add code for the parent process here.
27
28       exit(EXIT_SUCCESS);
29   }
30 }
```

Header files

On lines 1-3 a number of header files are included to get access to a few functions and constants from the **C Standard library** (https://www.tutorialspoint.com/c_standard_library/index.htm) .

pid_t

One line 7 the variable `pid` of type `pid_t` is declared. The `pid_t` data type is the data type used for process IDs.

fork

On line 9 the parent process calls `fork` and stores the return value in the variable `pid`.

switch

On line 9 a **switch-statement** (https://www.tutorialspoint.com/cprogramming/switch_statement_in_c.htm) is used to check the return value of `fork`.

Error (case -1)

On failure `fork` returns `-1` and execution continues in the `case -1` branch of the switch statement (line 11). The operating system was not able to create a new process. The parent uses **perror**

(https://www.tutorialspoint.com/c_standard_library/c_function_perror.htmB) to print an error message (line 13) and then terminates with exit status `EXIT_FAILURE` (line 14).

Child (case 0)

On success `fork` returns `0` in the new child process and execution continues in the `case 0` branch of the switch statement (line 16). Any code to be executed only by the child is placed here (line 19). The child terminates with exit status `EXIT_SUCCESS` (line 21).

Parent (default)

If the value `fork` returned by fork was neither `-1` (error) nor `0` (child), execution continues in the parent process in the `default` branch of the switch statement (line 23). In this case, the value returned by `fork` is the process ID (PID) of the newly created child process.

A first fork example

In the file `module-2/examples/fork.c` you find a program with the following `main` function.

```
int main(void) {
  pid_t pid;

  switch (pid = fork()) {
    case -1:
      // On error fork() returns -1.
      perror("fork failed");
      exit(EXIT_FAILURE);
    case 0:
      // On success fork() returns 0 in the child.
      child();
    default:
      // On success fork() returns the pid of the child to the parent.
      parent(pid);
  }
}
```

The code for the child is in the function `child` and the code for the parent in the function `parent`.

```
void child() {
  printf(" CHILD <%ld> I'm alive! My PID is <%ld> and my parent got PID <%ld>.\n",
         (long) getpid(), (long) getpid(), (long) getppid());
  printf(" CHILD <%ld> Goodbye!\n",
         (long) getpid());
  exit(EXIT_SUCCESS);
}

void parent(pid_t pid) {
```

```
  printf("PARENT <%ld> My PID is <%ld> and I spawned a child with PID <%ld>.\n",
         (long) getpid(), (long) getpid(), (long) pid);
  printf("PARENT <%ld> Goodbye!\n",
         (long) getpid());
  exit(EXIT_SUCCESS);
}
```

Both parent and child prints two messages and then terminates. Navigate to the directory `module-2/examples`. Compile using make.

```
$ make
```

Run the program.

```
$ ./bin/fork
```

You should see output similar to this in the terminal.

```
PARENT <87628> Spawned a child with PID = 87629.
PARENT <87628> Goodbye.
 CHILD <87629> I'm alive and my PPID = 1.
 CHILD <87629> Goodbye.
```

Run the program multiple times and look specifically at the `PPID` value reported by the child. Sometimes the child reports `PPID = 1` but sometimes it is equal to the PID of the parent. Clearly the PID of the parent is not `1`? Why doesn't report the "correct" PPID value all the time?

Orphans

An orphan process is a process whose parent process has terminated, though it remains running itself. Any orphaned process will be immediately adopted by the special init system process with PID 1.

Processes execute concurrently

Both the parent process and the child process competes for the CPU with all other processes in the system. The operating systems decides which process to execute when and for how long. The process in the system execute **concurrently (https://en.wikipedia.org/wiki/Concurrent_computing)**.

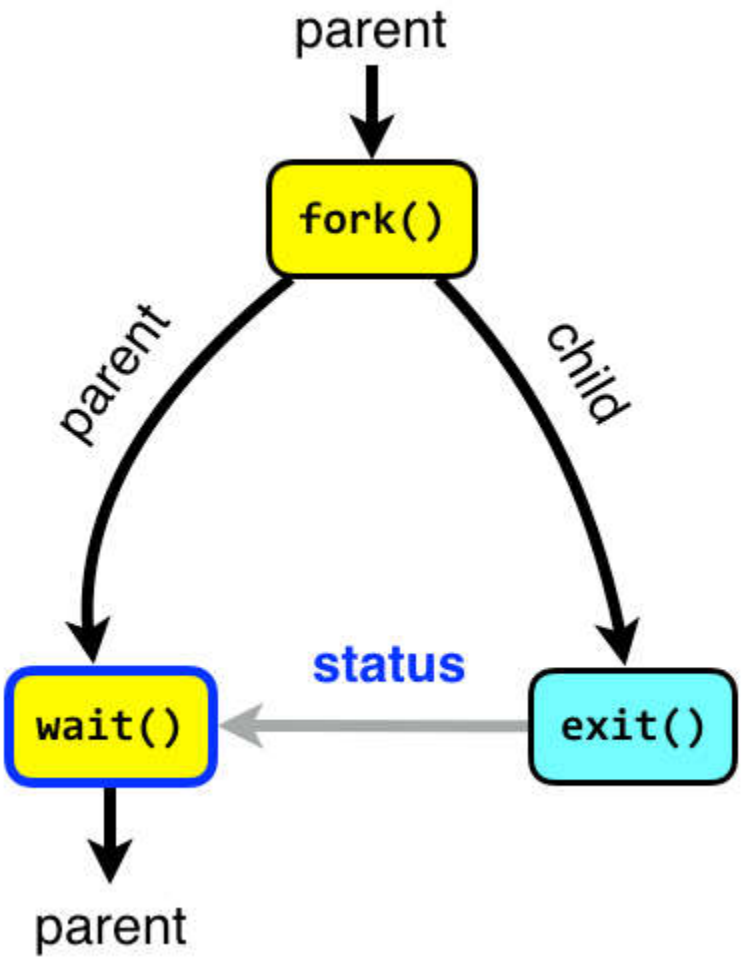
In our example program:

- most often the parent terminates before the child and the child becomes an orphan process adopted by init (PID = 1) and therefore reports PPID = 1
- sometimes the child process terminates before its parent and then the child is able to report

PPID equal to the PID of the parent.

Wait

The **wait** (http://www.tutorialspoint.com/unix_system_calls/wait.htm) system call blocks the caller until one of its child process terminates. If the caller doesn't have any child processes, `wait` returns immediately without blocking the caller. Using `wait` the parent can obtain the exit status of the terminated child.




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

pid_t wait(int *status);
```

status

If `status` is not NULL, `wait` store the **exit status** of the terminated child in the int to which `status` points. This integer can be inspected using the `WIFEXITED` and `WEXITSTATUS` macros.

Return value

On success, `wait` returns the PID of the terminated child. On failure (no child), `wait` returns `-1`.

WIFEXITED

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

WIFEXITED(status);
```

status

The integer status value set by the `wait` system call.

Return value

Returns true if the child terminated normally, that is, by calling `exit` or by returning from `main`.

WEXITSTATUS

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

int WEXITSTATUS(status);
```

status

The integer status value set by the `wait` system call.

Return value

The exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to `exit` or as the argument for a return statement in `main`. This macro should be employed only if WIFEXITED returned true.

Example using wait

In the `module-2/examples/fork_exit_wait.c` example program the parent execute the `parent` function.

```
 1 void parent(pid_t pid) {
 2
 3   printf("PARENT <%ld> Spawned a child with PID = %ld.\n",
 4          (long) getpid(), (long) pid);
 5
 6   wait(NULL);
 7
 8   printf("PARENT <%ld> Child with PID = %ld terminated.\n",
 9          (long) getpid(), (long) pid);
10
11   printf("PARENT <%ld> Goodbye.\n",
12          (long) getpid());
13
14   exit(EXIT_SUCCESS);
```

```
15 }
```

On line 6 the parent calls `wait(NULL)` to wait for the child process to terminate.

Compile and run the program. Now the parent should always wait for the child to terminate before terminating itself. As a consequence the child should:

- newer be adopted by init
- new report PPID = 1
- always report PPID equal to the PID of the parent.

Example using wait to obtain the exit status of the child

In the `module-2/examples/fork_exit_wait_status.c` example program the parent execute the `parent` function.

```
 1 void parent(pid_t pid) {
 2   int status;
 3
 4   printf("PARENT <%ld> Spawned a child with PID = %ld.\n",
 5          (long) getpid(), (long) pid);
 6
 7   wait(&status);
 8
 9   if (WIFEXITED(status)) {
10     printf("PARENT <%ld> Child with PID = %ld and exit status = %d terminated.\n",
11            (long) getpid(), (long) pid, WEXITSTATUS(status));
12   }
13
14   printf("PARENT <%ld> Goodbye.\n",
15          (long) getpid());
16
17   exit(EXIT_SUCCESS);
18 }
```

One line 2 the parent creates the variable `status`. On line 7 the parent calls `wait(&status)` to wait for the child process to terminate. The `&` is the address-of operator and `&status` returns the address of the `status` variable. When the child terminates the exit status of the child will be stored in variable `status`.

Compile using make.

```
$ make
```

Run the program.

```
$ ./bin/fork_exit_wait_status
```

In the output you should be able to see that the parent obtained the exit status of the child.

```
PARENT <99340> Spawned a child with PID = 99341.
 CHILD <99341> I'm alive and my PPID = 99340.
 CHILD <99341> Goodbye, exit with status 42.
PARENT <99340> Child with PID = 99341 and exit status = 42 terminated.
PARENT <99340> Goodbye.
```

Zombies

A terminated process is said to be a zombie or defunct until the parent does `wait` on the child.

- When a process terminates all of the memory and resources associated with it are deallocated so they can be used by other processes.
- However, the exit status is maintained in the PCB until the parent picks up the exit status using `wait` and deletes the PCB.
- A child process always first becomes a zombie.
- In most cases, under normal system operation zombies are immediately waited on by their parent.
- Processes that stay zombies for a long time are generally an error and cause a resource leak.

An example with a zombie process

In the `module-2/examples/fork_zombie.c` example program the child terminates before the parent does `wait` on the child and becomes a zombie process. The parent execute the `parent` function.

```
 1 void parent(pid_t pid) {
 2
 3   printf("PARENT <%ld> Spawned a child with PID = %ld.\n",
 4          (long) getpid(), (long) pid);
 5
 6   printf("PARENT <%ld> Press any key to reap a zombie!\n",
 7          (long) getpid());
 8
 9   getchar();
10
11   pid = wait(NULL);
12
13   printf("PARENT <%ld> Zombie child with PID = %ld",
14          (long) getpid(), (long) pid);
15
16   exit(EXIT_SUCCESS);
17 }
```

On line 9 the parent uses **getchar (https://www.tutorialspoint.com/c_standard_library/c_function_getchar.htm)** to block itself until the user presses a key on the keyboard.

When the child terminates, the exit status of the child is stored in the child process control block (PCB). The operating system deallocates all memory used by the child but the PCB cannot be deallocated until the parent does `wait` on the child.

Compile using make.

```
$ make
```

Run the program.

```
$ ./bin/fork_zombie
```

In the output you should be able to see that the child terminates and that the parent blocks waiting for a keypress.

```
PARENT <4636> Spawned a child with PID = 4637.
PARENT <4636> Press any key to reap a zombie!
 CHILD <4637> I'm alive and my PPID = 4636.
 CHILD <4637> Goodbye.
```

The child process has terminated but the parent has yet not read the exit status of the child using `wait`. The child process has now become a zombie process.

Monitor

Open a second terminal and navigate to the `module-2` directory. The `module-2/tools/monitor` tool can be used to view process status information about process. Use the `--help` flag to see the documentation.

```
$ ./tools/monitor --help
```

This is the built in documentation for the monitor tool.

```
Usage: monitor [-s delay] [-p pid] cmd

A top-like command that only lists USER, PID, STAT and COMM for the
current user and and proceses with a command name with a grep match of cmd.

Options:
 -s delay    Delay in seconds between refresh, default = 1.
 -p pid      Include process with PID pid.
```

The `cmd` argument is the name of the program executed by the processes we want to monitor. Use the monitor tool to view process status information for the parent and child, both executing the `fork_zombie` program.

```
$ ./tools/monitor fork_zombie
```

On Linux you should see something similar to this.

```
Monitoring processes matching: fork_zombie

Press Ctrl-C to exit

USER       PID  PPID S COMMAND
abcd1234  4636  4311 S fork_zombie
abcd1234  4637  4636 Z fork_zombie <defunct>
```

In the `PID` column you see the PID of the listed processes. The first line shows information about the parent and the second line shows information about the child.

The `S` column show the status of the process.

- The parent got status `S` (sleep) meaning the process is waiting for an event to complete. In this case the parent is blocked waiting for the child to terminate.
- The child got status `Z` (zombie) meaning the process terminated but not yet reaped by its parent.

Another name used for a zombie process is **defunct**.

Reap the zombie

From the terminal used to run the `fork_zombie` program, press any key to make the parent do `wait` on the child.

```
PARENT <4636> Spawned a child with PID = 4637.
PARENT <4636> Press any key to reap a zombie!
 CHILD <4637> I'm alive and my PPID = 4636.
 CHILD <4637> Goodbye.

PARENT <4636> Zombie child with PID = 4637 reaped!
PARENT <4636> Press any key to terminate!
```

In the terminal used to run `monitor` the zombie process should have disappear, leaving only the parent process.

```
Monitoring processes matching: fork
```

```
Press Ctrl-C to exit

USER       PID  PPID S COMMAND
abcd1234  4636  4311 S fork_zombie
```

The parent is now blocked, waiting for user input.

Terminate the parent

From the terminal used to run the `fork_zombie` program, press any key to make the parent terminate.

```
PARENT <4636> Spawned a child with PID = 4637.
PARENT <4636> Press any key to reap a zombie!
 CHILD <4637> I'm alive and my PPID = 4636.
 CHILD <4637> Goodbye.

PARENT <4636> Zombie child with PID = 4637 reaped!
PARENT <4636> Press any key to terminate!

PARENT <4636> Goodbye!
```

Execute a new program in the child

If you don't want to execute the same program in both the parent and the child, you will need to use a system call of the `exec` family. The `exec` system calls will replace the currently executing program with a new executable.

In `module-2/examples/src/child.c` you this small program.

```
#include <stdio.h>    // puts(), printf(), perror(), getchar()
#include <stdlib.h>   // exit(), EXIT_SUCCESS, EXIT_FAILURE
#include <unistd.h>   // getpid(), getppid()

int main(void) {

  printf(" CHILD <%ld> I'm alive and my PPID = %ld.\n",
       (long) getpid(), (long) getppid());

  printf(" CHILD <%ld> Press any key to make me terminate!\n",
         (long) getpid());

  getchar();

  printf(" CHILD <%ld> Goodbye!\n",
         (long) getpid());
```

```
  exit(127);
}
```

Compile using make.

```
$ make
```

Run the program.

```
$ ./bin/child
```

First this program simply prints two messages to the terminal and then wait for a key-press.

```
CHILD <33172> I'm alive and my PPID = 81166.
CHILD <33172> Press any key to make me terminate!
```

After you press any key in the terminal the program terminates.

```
CHILD <33172> I'm alive and my PPID = 81166.
CHILD <33172> Press any key to make me terminate!

CHILD <33172> Goodbye!
```

The `module-2/examples/src/fork_exec.c` program uses `execv` to make the child process execute the `module-2/examples/bin/child` executable. After fork the child executes the `child` functions.

```
 1 void child() {
 2   char *const argv[] = {"./bin/child", NULL};
 3
 4   printf(" CHILD <%ld> Press any key to make me call exec!\n",
 5          (long) getpid());
 6
 7   getchar();
 8
 9   execv(argv[0], argv);
10
11   perror("execv");
12   exit(EXIT_FAILURE);
13 }
```

On line 2 the needed argument vector is constructed. On line 7 the child waits for a key-press. After the key-press, on line 9, the child use `execv` to replace the program executed by the child process by the `child` executable. If `execv` is successful control will never be returned and lines 11 and 12 should not be reached.

Compile using make.

```
$ make
```

Run the program.

```
$ ./bin/fork_exec
```

```
PARENT <33422> Spawned a child with PID = 33423.
 CHILD <33423> Press any key to make me call exec!
```

Open a second terminal and use the `ps` command with the `-p` option to see information about the child process.

```
$ ps -p 33206
  PID TTY          TIME CMD
  33423 ttys023    0:00.00 ./bin/fork_exec
```

Note that the child process currently is executing the `.bin/fork_exec` executable.

In the first terminal, press any key.

```
 CHILD <33423> I'm alive and my PPID = 33422.
 CHILD <33423> Press any key to make me terminate!
```

From the other terminal and use the `ps` command with the `-p` option to see information about the child process.

```
$ ps -p 33206
  PID TTY          TIME CMD
  33423 ttys023    0:00.00 ./bin/child
```

Note that the child process now executes the `./bin/child` executable.

In the first terminal, press any key to make the child process terminate. Now the parent performs wait on the child and reports the child exit status.

```
  CHILD <33423> Goodbye!
PARENT <33422> Child with PID = 33423 and exit status = 127 terminated.
PARENT <33422> Goodbye!
```