Exercise 11

Maximilian Ott
Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Presentation Assignment 6
Processes

- Processes are an execution environment for programs
  - Have a process ID (PID, positive integer)
  - Execute a program
- Each process is assigned resources they need
  - Memory
  - Address space
  - Opened files
  - ...
- Visualization of processes: `ps(1)`, `pstree(1)`, `htop(1)`
Between all processes, a parent-child relation exists
- The first process is started by the system kernel (e.g. `init`)
- A tree of processes is created -> process hierarchy

```
- kate is a child of bash, bash is a child of xterm
```

Create Child Processes (1)
```
01 pid_t fork(void);
```
- Creates a new child process
- Exact copy of the parent process:
  - Data and stack segment (copy)
  - Text segment (shared use)
  - File descriptors (open files)
  - **Exception**: Process ID
- Parent and child process both return from the call to `fork(2)`
- Difference is the returned value of `fork(2)`
  - Parent: PID of the child
  - Child: 0
  - Error: -1
```
printf("Process (PID: %d)", getpid());

pid_t res = fork();

if(res > 0) {
    printf("Parent process (PID: %d)", getpid());
} else if(res == 0) {
    printf("Child process (PID: %d)", getpid());
} else {
    printf("Error (PID: %d)", getpid());
    // [...] Error handling
}
```
Waiting for a Child Process (1)

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

- `wait(2)` is blocking until an arbitrary child process terminates
- Returns
  - > 0 Process ID of the child process
  - -1 Error
- `status` contains the reason for the termination:
  - `WIFEXITED(status)` exit(3) or return from `main()`
  - `WIFSIGNALED(status)` Process terminated by signal
  - `WEXITSTATUS(status)` Exit status
  - `WTERMSIG(status)` Signal number
- Further macros: see documentation `wait(2)`

Waiting for a Child Process (2)

```
01 pid_t waitpid(pid_t pid, int *status, int options);
```

- `waitpid(2)` is blocking until a certain child process terminates
  - `pid` > 0 Child process with process ID `pid`
  - `pid` = -1 Arbitrary child process
  - ...
- Options:
  - `WNOHANG` Returns immediately if no child terminated
    (not blocking)
  - ...
- Returns
  - > 0 Process ID of the child process
  - 0 No process has terminated (when using `WNOHANG`)
  - -1 Error – details see `waitpid(2)`
Terminate Processes

01 void exit(int status);

- Terminates the currently running process with the given exit status
- Frees all resources that were used by the process
  - Memory
  - File descriptors
  - Process management data
  - ...
- Process enters a so called zombie state
  - Makes it possible for the parent to react to its termination
  - Zombie processes still use some resources
    ⇒ Parent has to keep up with its zombies
- If the parent terminates before its child:
  ⇒ Passed on to the init process and cleared by it

Execute Programs (1)

01 int execl(const char *path, const char *arg0, ..., NULL);
02 int execv(const char *path, char *const argv[]);

- Replaces the currently running program within the process
  - Is replaced: text-, data- and stack segment
  - Remains: file descriptors, working directory, ...
- Calling parameters for exec(3)
  - Path of the new program
  - Arguments for the main() function
- Static number of arguments: execl(3)
- Dynamic number of arguments: execv(3)
- Last argument: NULL pointer
- exec(3) only returns in case of an error
PATH Environment Variable

- Finding executable programs using the PATH variable

```
01 $> cp dat dat-copy
02 $> ls
dat dat-copy    # no file 'cp'
03
04 $> echo $PATH    # PATH contains
05 /usr/local/bin:/usr/bin:/bin  # - /usr/local/bin/
06    # - /usr/bin/
07    # - /bin/
08
09 $> which cp
10 /bin/cp    # 'cp' is in /bin/
11
12 $> ls /bin/
13 [...]    # /bin/ contains many
14 rm
15 cp
16 ls
17 [...]  
```

Execute Programs (2)

- Like execl(3)/execv(3) and searching in PATH

Examples:

```
01 int execlp(const char *file, const char *arg0, ..., NULL);
02 int execvp(const char *file, char *const argv[]);
```

```
01 // absolute path and static list of arguments
execl("/bin/cp", "/bin/cp", "x.txt", "y.txt", NULL);
02
03 // Searching in PATH and static list of arguments
execlp("cp", "cp", "x.txt", "y.txt", NULL);
04
05 // Searching in PATH and dynamic list of arguments
char *args[] = { "cp", "dat", ..., "copy/", NULL };
06 execvp(args[0], args);
```
Example: fork(2), exec(3) and wait(2)

```c
static void die(const char *reason) {
    perror(reason); exit(EXIT_FAILURE);
}

// [...] Process runs
pid_t res = fork();
if(res > 0) { // Parent process
    int status;
    pid_t term_pid = wait(&status);
    if(term_pid == -1) { // Error in wait()
        die("wait");
    } else {
        printf("Child %d terminated\n", term_pid);
    }
} else if(res == 0) { // Child process
    execlp("cp", "cp", "dat", "dat-copy", NULL);
    // Error in execlp(3)
    die("execlp");
} else { // Error -- No child process created
    die("fork");
}
```

Minimal Shell
Functionality of a Minimal Shell

1. Wait for the user input
2. Create a new process
3. Parent: wait for termination of the child
4. Child: start program
5. Child: program terminates
6. Parent: Output the exit status of the child

```
    Minimal Shell
        (PID: 41)
        ↘
        fork() ↙
        ␣ ↘
        exec() ↙
        ␣ ↘
        exit() ↙
        ␣ ↘
        wait() ↙
        ␣ ↘
        Output

New process
        (PID: 42)
```

Reading from Standard Input with fgets

```
char *fgets(char *s, int size, FILE *stream);

- fgets(3) reads one line from the given channel
- '\n' is stored as well
- Maximum size-1 characters + final '\0'
- In case of an error or EOF, NULL is returned
  ⇒ Distinction using ferror(3) or feof(3)

char buf[23];
while (fgets(buf, 23, stdin) != NULL) {
    // buf contains line
}

if(ferror(stdin)) { // Error
    [...]
}
String Manipulation with `strtok(3)`

- `strtok(3)` breaks the string into tokens
- Tokens are separated by delimiters
- Each call returns a pointer to the next token
- `delim`: string that contains all delimiters (e.g. " \t\n")
- `str`:  
  - **first call** pointer to the string
  - **all following calls** `NULL`
- Consecutive delimiters are skipped
- Delimiters after a token are replaced by '\0'
- At the end of the string: `strtok(3)` returns `NULL`

```
cmdline ls -l __/tmp\0
```

```
01 char *strtok(char *str, const char *delim);

01 char cmdline[] = "ls -l /tmp";
02 char *a[4];
03 a[0] = strtok(cmdline, " ");
04 a[1] = strtok(NULL, " ");
05 a[2] = strtok(NULL, " ");
06 a[3] = strtok(NULL, " ");
```
String Manipulation with `strtok(3)`

```c
char cmdline[] = "ls -l /tmp";
char *a[4];
a[0] = strtok(cmdline, " ");
a[1] = strtok(NULL, " ");
a[2] = strtok(NULL, " ");
a[3] = strtok(NULL, " ");
```
String Manipulation with `strtok(3)`

```c
char cmdline[] = "ls -l /tmp";
char *a[4];
a[0] = strtok(cmdline, " ");
a[1] = strtok(NULL, " ");
a[2] = strtok(NULL, " ");
a[3] = strtok(NULL, " ");
```
Assignment: mish

- Part a)

Simple shell (mini shell) for executing commands

Typical procedure:
- Output a prompt
- Wait for user input
- Tokenize the input
  - Command name
  - Arguments
- Create a new process
  - parent: waits for termination of the child
  - child: executes the command
- Output the exit status
Repetition: basic cycle of a minimal shell

1. Waiting for a input from the user
2. Creating a new process
3. Parent: Waiting foe the termination of the child
4. Child: Starting the program
5. Child: Program terminates
6. Parent: Outputting the status of the child

Examples:

01 # Regular termination with Exit (Exitstatus = 0)
02 mish> ls -l
03 ...
04 Exit status [2110] = 0
05
06 # Invalid/empty input
07 mish>
08 mish> foo
09 foo: No such file or directory
10 Exit status [7342] = 1
11
12 # Termination by signal (here SIGINT = 2)
13 mish> sleep 10
14 Signal [1302] = 2
Excursion: fflush(3)

- Prompt does not print a '\n'
- Standard library buffers stdout line by line
⇒ The line buffer has to be flushed with fflush(3) after an output

Test Programs

- Test programs: /proj/i4spic/<idm>/pub/aufgabe8/
- spic-wait (without parameter)

```
mish> /proj/i4spic/[...]/spic-wait
[...]
- send 'SIGPIPE' to this process
  Command: kill -PIPE 3372
  Expected Output: Signal [3372] = 13
[...]
Signal [3372] = 13
mish>
```

- spic-wait (with parameter)

```
mish> /proj/i4spic/<idm>/pub/aufgabe8/spic-wait 15
Sending signal 15 (Terminated) to myself (PID: 4239)
Signal [4239] = 15
mish>
```
Test Programs

- Test programs: /proj/i4spic/<idm>/pub/aufgabe8/
- spic-wait (without parameter)

```
mish> /proj/i4spic/.../spic-wait
  - send 'SIGPIPE' to this process
  Command: kill -PIPE 3372
  Expected Output: Signal [3372] = 13

$> kill -PIPE 3372
```

- spic-wait (with parameter)

```
mish> /proj/i4spic/<idm>/pub/aufgabe8/spic-wait 15
  Sending signal 15 (Terminated) to myself (PID: 4239)
  Signal [4239] = 15
```

Test Programs

- Test programs: /proj/i4spic/<idm>/pub/aufgabe8/
- spic-wait (without parameter)

```
mish> /proj/i4spic/.../spic-wait
  - send 'SIGPIPE' to this process
  Command: kill -PIPE 3372
  Expected Output: Signal [3372] = 13

$> kill -PIPE 3372
```

- spic-wait (with parameter)

```
mish> /proj/i4spic/<idm>/pub/aufgabe8/spic-wait 15
  Sending signal 15 (Terminated) to myself (PID: 4239)
  Signal [4239] = 15
```
### spic-exit

```
mish> /proj/i4spic/<idm>/pub/aufgabe8/spic-exit 12
Exiting with status 12
Exit status [6272] = 12
mish>
```

### Template

```
// DESCRIPTION:
// printStatus() examines the termination of a process and
// prints the source of the exit (signal or exit) and the
// exit code or signal number, respectively.

// PARAMETER:
// pid: PID of the exited child process
// status: Status bits as retrieved from waitpid(2)

static void printStatus(pid_t pid, int status) {
    // TODO IMPLEMENT
}
```

- `/proj/i4spic/<idm>/pub/aufgabe8/mish_vorlage.c`
- The template does **not** contain:
  - all functions, description of functionality, variables, etc.
- Template does not replace making your own considerations about the structure
- During development, it could be useful to omit the flag `Werror` in the makefile
Hands-on: run

Screencast: https://www.video.uni-erlangen.de/clip/id/19832

```
01 ./run <program> <param0> [params...]
```

- run receives a program name and a list of parameters
  - Creates a new process for each parameter
  - Executes the given program and passes a parameter to it
  - Waits for the termination and continues with the handling of the next parameter

- Example call: 
  `./run echo Car House Cat`

- Generated program calls:
  - `echo Car`
  - `echo House`
  - `echo Cat`

- (System-)Calls: `fork(2), exec(3), wait(2)`

- Keep in mind the error handling