Exercises in System Level Programming (SLP) – Summer Term 2024

Exercise 11

Maximilian Ott

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

Chair in Distributed Systems
and Operating Systems

FAU FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Presentation Assignment 6
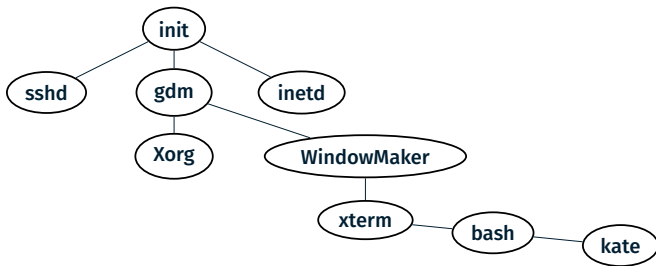

Processes

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)`

1

Process Hierarchy

- 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**
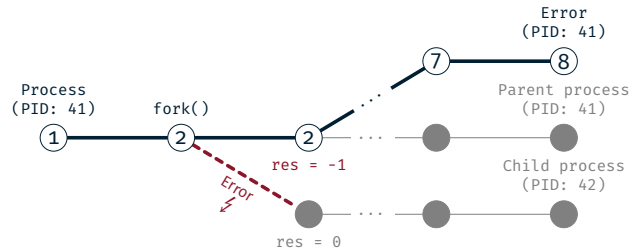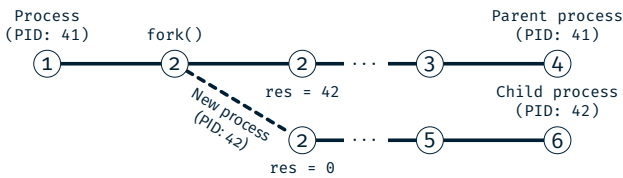
Create Child Processes (1)

```
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

Create Child Processes (2)

```
01  printf("Process (PID: %d)", getpid());
02  pid_t res = fork();
03  if(res > 0) {
04      printf("Parent process (PID: %d)", getpid());
05  } else if(res == 0) {
06      printf("Child process (PID: %d)", getpid());
07  } else {
08      printf("Error (PID: %d)", getpid());
09      // [...] Error handling
10  }
```

Create Child Processes (2)

```
01  printf("Process (PID: %d)", getpid());
02  pid_t res = fork();  // Has an error ⚡
03  if(res > 0) {
04      printf("Parent process (PID: %d)", getpid());
05  } else if(res == 0) {
06      printf("Child process (PID: %d)", getpid());
07  } else {
08      printf("Error (PID: %d)", getpid());
09      // [...] Error handling
10  }
```

Waiting for a Child Process (1)

```
O1   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)

```
O1   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

```
O1   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)

```
O1   int execl(const char *path, const char *arg0, ..., NULL);
O2   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

■ Finding executable programs using the PATH variable

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

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

■ Like `execl(3)`/`execv(3)` and searching in PATH

Examples:

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

```
01  static void die(const char *reason) {
02      perror(reason); exit(EXIT_FAILURE);
03  }
04
05  // [...] Process runs
06  pid_t res = fork();
07  if(res > 0) { // Parent process
08      int status;
09      pid_t term_pid = wait(&status);
10      if(term_pid == -1) { // Error in wait()
11          die("wait");
12      } else {
13          printf("Child %d terminated\n", term_pid);
14      }
15  } else if(res == 0) { // Child process
16      execlp("cp", "cp", "dat", "dat-copy", NULL);
17      // Error in execlp(3)
18      die("execlp");
19  } else {  // Error -- No child process created
20      die("fork");
21  }
```
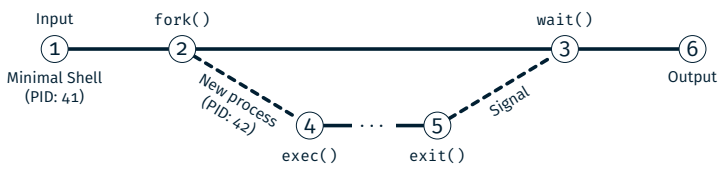
Minimal Shell

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

Reading from Standard Input with fgets

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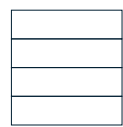
- `fgets(3)` reads one line from the given channel
- `'\n'` is stored as well
- Maximum `size-1` characters + final `'\0'`
- In case of an error or EOF, NULL is returned
⇒ Distinction using `ferror(3)` or `feof(3)`

```
01  char buf[23];
02  while (fgets(buf, 23, stdin) != NULL) {
03      // buf contains line
04  }
05
06  if(ferror(stdin)) {  // Error
07      [...]
08  }
```
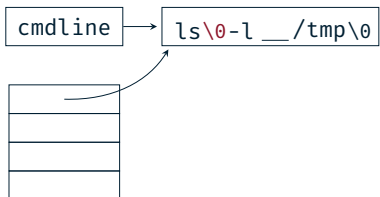
String Manipulation with `strtok(3)`

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

- `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

String Manipulation with `strtok(3)`


```
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, " ");
```

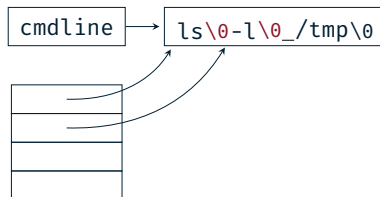## Slide 1 (top-left)

```
cmdline  →  ls\0-l __/tmp\0
```

```
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, " ");
```

Slide 2 (top-right)

```
cmdline  →  ls\0-l\0_/tmp\0
```

```
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, " ");
```

Slide 3 (bottom-left)
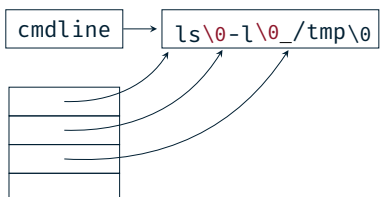
```
cmdline  →  ls\0-l\0_/tmp\0
```

```
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, " ");
```

Slide 4 (bottom-right)

```
cmdline  →  ls\0-l\0_/tmp\0
```

NULL

```
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, " ");
```
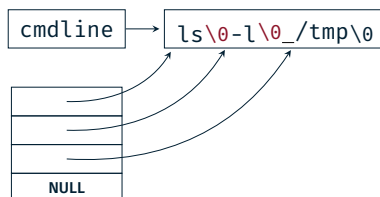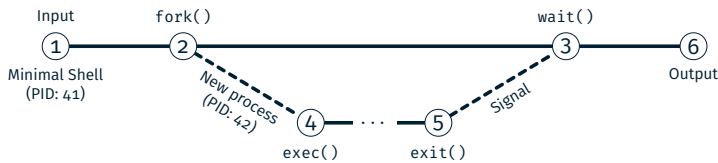
Assignment: `mish`

- Simple shell (**mi**ni **sh**ell) 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 paramter)

```
01  mish> /proj/i4spic/[...]/spic-wait
02  [...]
03  - send 'SIGPIPE' to this process
04    Command: kill -PIPE 3372
05    Expected Output: Signal [3372] = 13
06
07  [...]
08
09  Signal [3372] = 13
10  mish>
```

```
01
02
03
04
05
06
07
08  $> kill -PIPE 3372
09
10
```

- spic-wait (with parameter)

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

Test Programs

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

```
01  mish> /proj/i4spic/[...]/spic-wait
02  [...]
03  - send 'SIGPIPE' to this process
04    Command: kill -PIPE 3372
05    Expected Output: Signal [3372] = 13
06
07  [...]
08
09  Signal [3372] = 13
10  mish>
```

```
01
02
03
04
05
06
07
08  $> kill -PIPE 3372
09
10
```

- spic-wait (with parameter)

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

Test Programs

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

```
01  mish> /proj/i4spic/[...]/spic-wait
02  [...]
03  - send 'SIGPIPE' to this process
04    Command: kill -PIPE 3372
05    Expected Output: Signal [3372] = 13
06
07  [...]
08
09  Signal [3372] = 13
10  mish>
```

```
01
02
03
04
05
06
07
08  $> kill -PIPE 3372
09
10
```

- spic-wait (with parameter)

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

Test Programs

- spic-exit

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

Template

```
01  // DESCRIPTION:
02  //   printStatus() examines the termination of a process and
03  //   prints the source of the exit (signal or exit) and the
04  //   exit code or signal number, respectively.
05  //
06  // PARAMETER:
07  //   pid:    PID of the exited child process
08  //   status: Status bits as retrieved from waitpid(2)
09  //
10  static void printStatus(pid_t pid, int status) {
11      // TODO IMPLEMENT
12  }
```

- /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

Hands-on: run

```
01  ./run <programm> <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