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

Exercise 11

Maximilian Ott

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg
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)`
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 \( \rightarrow \) process hierarchy

- **kate** is a child of **bash**, **bash** is a child of **xterm**
Create Child Processes (1)

Create 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)

```c
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
}
```

```
Process (PID: 41)  fork()  
1  2  ...  3  4
    
Parent process (PID: 41)  

New process (PID: 42)
2  2  ...  5  6

Child process (PID: 42)
```
Create Child Processes (2)

```c
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 }
```

Diagram:

1. Process (PID: 41)
2. `fork()`
3. `res = -1`
   - Error
4. `res = 0`
5. Parent process (PID: 41)
6. Child process (PID: 42)
7. Error (PID: 41)
8. Process (PID: 41)
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)

```c
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)`
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

```c
void exit(int status);
```
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

<table>
<thead>
<tr>
<th>Line</th>
<th>Command</th>
<th>Output / Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td><code>$ cp dat dat-copy</code></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td><code>$ ls</code></td>
<td></td>
</tr>
<tr>
<td>03</td>
<td><code>dat dat-copy</code></td>
<td># no file 'cp'</td>
</tr>
<tr>
<td>04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05</td>
<td><code>$ echo $PATH</code></td>
<td># PATH contains</td>
</tr>
<tr>
<td>06</td>
<td><code>/usr/local/bin:/usr/bin:/bin</code></td>
<td># - /usr/local/bin/</td>
</tr>
<tr>
<td>07</td>
<td></td>
<td># - /usr/bin/</td>
</tr>
<tr>
<td>08</td>
<td></td>
<td># - /bin/</td>
</tr>
<tr>
<td>09</td>
<td><code>$ which cp</code></td>
<td># 'cp' is in /bin/</td>
</tr>
<tr>
<td>10</td>
<td><code>/bin/cp</code></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td><code>$ ls /bin/</code></td>
<td># /bin/ contains many</td>
</tr>
<tr>
<td>13</td>
<td><code>[...]</code></td>
<td># more common programs</td>
</tr>
<tr>
<td>14</td>
<td><code>rm</code></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td><code>cp</code></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td><code>ls</code></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td><code>[...]</code></td>
<td></td>
</tr>
</tbody>
</table>
Execute Programs (2)

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);
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
    execvp("cp", "cp", "dat", "dat-copy", NULL);
    // Error in execvp(3)
    die("execvp");
} 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
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)

```c
char buf[23];
while (fgets(buf, 23, stdin) != NULL) {
    // buf contains line
}
if(ferror(stdin)) { // Error
    [...]
}
```
String Manipulation with `strtok(3)`

```c
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)`

```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, " ");
```
```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, " ");
```
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
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
Assignment: mish - Part a)

- 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)

```
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 Signal [3372] = 13
10 mish>
```

- 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 parameter)

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 Signal [3372] = 13
09 mish>
10 $> kill -PIPE 3372

- 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 parameter)

```
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 Signal [3372] = 13
08 mish>
```

- 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>
// 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
}
Hands-on: run

Screencast: https://www.video.uni-erlangen.de/clip/id/19832
**Hands-on: run**

```
./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