Exercise sheet 5 – Process/Thread

Goals:

- Process management
- Thread management

Exercise 5.1: Process management

(a) List all running processes.

Proposal for solution: `ps ax` or `ps aux`

(b) What is the meaning of the '`x`' flag?

Proposal for solution: The '`x`' flag shows all daemons. That are the processes which are not attachted to a terminal.

(c) What information do you find for each process?

Proposal for solution: PID: process ID, PPID: parent process ID, PRI: priority, NI: nice-value, SZ: count of 4 KB-pages, RSS: resident set size (memory of process in RAM), VSZ: virtual set size (memory which is allocated to process, this includes memory which is swapped out or is not used yet) (each 1 KB), S=STAT: status of process (S: blocked, R: running)

(d) How many processes are running?

Proposal for solution: Count the number of process from the output of `ps aux`.
Or for those who are familiar with the bash: `echo $[$(ps aux | wc -l) - 1]`

(e) Which processes were created first?

Proposal for solution: The processes with the lowest PIDs were created first.

(f) Why are gaps between the PIDs (process IDs)?

Proposal for solution: These processes have already been terminated.

(g) What is the lowest PID, and what is the meaning of this process?

Proposal for solution: The lowest PID is 1, it's the `systemd` process which starts the user space processes.

(h) What is the meaning of the '`-p`' flag of `pstree`?

Proposal for solution: The '`p`' flag shows the PID of each process.

(i) What are the parent and grand parent processes of `pstree`?

Proposal for solution: `bash` is the parent and `guake` is the grand parent.

Exercise 5.2: Process information

(a) Update the `OS_exercises` repository with `git pull`

```
cd OS_exercises
git pull
cd ~
```

(b) Start the program `OS_exercises/sheet_05_processes/demo_program`.

Proposal for solution: `OS_exercises/sheet_05_processes/demo_program`

(c) Find the process ID (PID) of the running `demo_program`. You may need a separate shell for that.

Proposal for solution: `ps u -C demo_program`
Or for those who are familiar with the bash: `ps ax | grep demo_program` search for `OS_exercises/sheet_05_processes/demo_program`
Read the PID of the program.

(d) How many CPU percentage and memory does the process use?

Proposal for solution: `ps u -C demo_program`
Or for those who are familiar with the bash: `ps aux | grep demo_program` search for `OS_exercises/sheet_05_processes/demo_program`
Read columns CPU (should be about 0.0) and RSS (should be about 207716 KB ≈ 202,85 MB) of the program.

(e) Try to stop the `demo_program`.

Proposal for solution: `kill xxxx` (xxxx is the PID) or CTRL+C in the shell that runs `demo_program`.

Exercise 5.3: Process creation

The file `OS_exercises/sheet_05_processes/process/processCreation.c` provides a skeleton for this exercise.

(a) Create `N` processes.

Proposal for solution:

```
pid_t child_pids[N];

//Create the processes
for(int i = 0; i < N; ++i) {
    child_pids[i] = fork(); //fork the child process

    switch(child_pids[i]) {
        case -1:
            printf ("Error at fork");
            exit(EXIT_FAILURE);
            break;
        case 0:
            //Here: code for child processes

            break;
        // default: not needed
    }
}
```

(b) Each process works something: we simulate that by calling the `work()` function, which sleeps for 20 seconds.

Proposal for solution: Inside the `//Here code for childs`:

```
switch(child_pids[i]) {
    case -1:
        printf ("Error at fork");
        exit(EXIT_FAILURE);
        break;
    case 0:
        //Here: code for child processes
        work();
        break;
    // default: not needed
}
```

(c) Before a process ends, it increases the `counter`.

Proposal for solution:

```
void work() {
    sleep(20); //simulates the "heavy" work!!

    //TODO: Add code for created processes here
    ++counter;

    exit(EXIT_SUCCESS);
}
```

(d) The main (parent) process waits until all its child processes have been finished.

Proposal for solution: In main():

```
//Wait for the termination of all child processes
for(int i = 0; i < N; ++i) {
    waitpid(child_pids[i], NULL, 0);
}
```

(e) After all processes have been finished: it prints the value of the `counter` and exits.

Proposal for solution:

```
//Print counter
printf("All childs have finished, counter: %d \n", counter);
```

(f) Change into the folder `OS_exercises/sheet_05_processes/process` (if you aren't already) and compile the program with `make`

Proposal for solution:

```
cd OS_exercises/sheet_05_processes/process
make
```

(g) Start the program with `./processCreation N` (N stands for the number of processes to create). What is the value of the counter and what have you expected?

Proposal for solution: The value of the counter is 0. This is because variables are not shared between different processes.
The full solution, for reference:

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int counter = 0;

void work() {
    sleep(20); //simulates the "heavy" work!!

    //TODO: Add code for created processes here
    ++counter;

    exit(EXIT_SUCCESS);
}

int main(int argc, char** argv){
    int N = 0; //N contains the number of processes to create

    //determine the number of processes as a program argument
    if(argc == 2){
        N = atoi(argv[1]); //first program argument
    } else {
        printf("Usage: %s N\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    //TODO: Write your code here
    pid_t child_pids[N];

    //Create the processes
    for(int i = 0; i < N; ++i) {
        child_pids[i] = fork(); //fork the child process

        switch(child_pids[i]) {
            case -1:
                printf ("Error at fork");
                exit(EXIT_FAILURE);
                break;
            case 0:
                //Here: code for child processes
                work();
                break;
            // default: not needed
        }
    }

    //Wait for the termination of all child processes
    for(int i = 0; i < N; ++i) {
        waitpid(child_pids[i], NULL, 0);
    }

    //Print counter
    printf("All childs have finished, counter: %d \n", counter);
```

```
56
57      return EXIT_SUCCESS;
58  }
```

Exercise 5.4: Thread creation

The file `OS_exercises/sheet_05_processes/thread/threadCreation.c` provides a skeleton for this exercise.

(a) Create `N` threads. Each thread calls the `work()` function, which simulates working by sleeping for 20 seconds.

Proposal for solution:

```
1  pthread_t thread_ids[N];
2
3  //Create the threads
4  for(int i = 0; i < N; ++i) {
5      int thread_create_state = pthread_create(&thread_ids[i], NULL, &work, NULL);
6      if(thread_create_state != 0) {
7          printf("Failed creating thread\n");
8          exit(EXIT_FAILURE);
9      }
10 }
```

(b) Before a thread ends, it increases the `counter`. Add this to the `work()` function.

Proposal for solution:

```
1  void* work() {
2      sleep(20); //simulates the "heavy" work!!
3
4      //TODO: Add code for created threads here
5      ++counter;
6
7      return NULL;
8  }
```

(c) The main thread waits until all its created threads have been finished.

Proposal for solution:

```
1  //Wait for the termination of all threads
2  for(int i = 0; i < N; ++i) {
3      pthread_join(thread_ids[i], NULL);
4  }
```

(d) After that: it prints the value of the `counter` and exits.

Proposal for solution:

```
1  //Print counter
2  printf("All threads have finished, counter: %d \n", counter);
```

(e) Change into the folder `OS_exercises/sheet_05_processes/thread` (if you aren't already) and compile the program with `make`

Proposal for solution:

```
1  cd OS_exercises/sheet_05_processes/thread
2  make
```

(f) Start the program with `./threadCreation N` (N stands for the number of threads to create). What is the value of the counter and what have you expected?

Proposal for solution: The value of the counter is the number of threads started. This is because variables are shared between different threads.
The full solution, for reference (the program has to be compiled with the `-pthread` commandline parameter):

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

int counter = 0;

void* work() {
    sleep(20); //simulates the "heavy" work!!

    //TODO: Add code for created threads here
    ++counter;

    return NULL;
}

int main(int argc, char** argv){
    int N = 0; //N contains the number of threads to create

    //determine the number of threads as a program argument
    if(argc == 2){
        N = atoi(argv[1]); //first program argument
    } else {
        printf("Usage: %s N \n", argv[0]);
        exit(EXIT_FAILURE);
    }

    //TODO: Add code for the main thread here
    pthread_t thread_ids[N];

    //Create the threads
    for(int i = 0; i < N; ++i) {
        int thread_create_state = pthread_create(&thread_ids[i], NULL, &work, NULL);
        if(thread_create_state != 0) {
            printf("Failed creating thread\n");
            exit(EXIT_FAILURE);
        }
    }

    //Wait for the termination of all threads
    for(int i = 0; i < N; ++i) {
        pthread_join(thread_ids[i], NULL);
    }

    //Print counter
    printf("All threads have finished, counter: %d \n", counter);

    return EXIT_SUCCESS;
}
```

(g) Can you identify some problems that may occur, if the threads access the `counter` variable

in parallel?