University of Stuttgart
Institute of Parallel and Distributed Systems
Distributed Systems Department

Winter Term 2017/2018

Operating Systems

Assignment Sheet 2

Available:	Monday, November 6, 2017, 08:00 am
Due Date:	Monday, November 20, 2017, 10:00 am
Discussion:	Friday, November 24, 2017, 11:30-13:00, Room V38.02

General remarks: The intent of the exercises is to help you get more familiar with fundamental concepts of modern operating systems as presented in the lecture.

Work on the problems with your group and hand in a single solution which will be graded for all members of the group. We will review your solution and hand out the graded version in the exercise sessions. Everybody has to present at least once in the sessions. If you hand in a solution but are not able to present it, no points will be given *for the whole exercise*! If we detect any plagiarism in your solution, *the whole group* will not get points *for the whole sheet.*

Please read the *SubmissionGuidlines.pdf* and *InformationOnExercises.pdf* in ILIAS in the Course Material folder. Especially, **be concise** and avoid long answers.

For additional reference, consult the text books cited in the lecture and explore the Web. You can find additional information on lectures and exercises at the lecture homepage. If you have any questions, you may contact us on the e-learning system ILIAS.

Questions

1. (9 points) Processes and Threads
   (a) Consider the Five State Process Model as presented in the lecture. Explain its limitations. (1 points)
   (b) The Seven State Process Model adds two more states, i.e., Blocked/Suspend and Ready/suspend, to improve the Five State Model. In this model, can a process in the "Ready/Suspend" state directly transit to "Running" state? Briefly justify your answer. (2 points)
   (c) Compare and contrast a user-level thread with a kernel-level thread on at least two criteria. (2 points)
   (d) Consider a situation where a CPU-bound task, called task A, needs to be implemented as a single process. To this end, Task A can have two possible implementations, i.e, either a single-thread process OR a multi-threaded process. Assume that Task A is faster when multi-threaded? (4 points)
      1. On a uni-processor system single threaded.
      2. On a multi-processor system while using user-level threads.
      3. On a multi-processor system while using kernel-level threads.

      Briefly state the benefits when Task A is executed in each of the above systems in terms of execution time.

2. (5 points) Multiprogramming

   Suppose that we have a multi-programmed computer in which each job has identical characteristics. In one computation period, T, for a job, the initial one quarter is spent in processor activity and the subsequent three quarters of time in I/O. Each job runs for a total of N periods. Assume that a simple round-robin scheduling is used (the operating system switches to the next process after $T/4$), and that I/O operations can overlap with processor operation. Further assume that an arbitrary number of I/O operations can be performed in parallel. We define the following quantities:

- Turnaround time (TAT) = actual time to complete the job
- Throughput = average number of jobs completed per time period T
- Processor utilization = percentage of time that the processor is active (not waiting).

Provide a graphical presentation of two entire cycles $2T$ for one, five, and eight simultaneous jobs. Based on your graph and additional analysis if necessary, compute the above defined quantities in each of the three cases.

NOTE: In case of TAT, you are only required to provide it for the first job.

3. (11 points) Implementation Task

In the following implementation task, you are required to extend the shell that you developed in Exercise Sheet 1 with more advanced commands and features. Each team is advised to read, discuss, and consequently, divide the task among the team-members for its efficient completion.

As in Exercise Sheet 1, you are advised *against* the use of the `system()` function in your solution, so avoid its use as much as possible.

(a) **ls** command: Implement the list directory command, `ls <directory>`, which lists the contents of the directory specified by `<directory>`. You may use any format to display the directory contents, and you don't have to implement additional `ls` arguments.

(b) **cd** command: Implement the change directory command, cd `<directory>`, which changes to the directory specified by `<directory>`. If no directory is specified, change to your own home directory.

(c) **Environment:** Implement commands for listing, defining and un-defining environment variables. Use `environ` to list all the environment strings currently defined. Use `setenv <envar> <value>` to set the environment variable `<envar>` to `<value>`. If `setenv` is used with `<envar>` only, set the value for that environment variable to the empty string. Use `unsetenv <envar>` to undefine environment variable `<envar>`. Use appropriate output if a variable is already defined (in the case of `setenv`) or undefined (in the case of `unsetenv`). At startup, the shell environment should contain `shell=<pathname>/gbsh` where `<pathname>/gbsh` is the full path for your shell executable (not a hardwired path back to your home directory, but the one from which it was executed).

(d) All other command line input is interpreted as program invocation which should be done in 2 steps. In the first step, the shell forking (**fork** command) is used to create child a process. Next, the child process executes the program using one of the commands from the **exec** family of commands.

   i. Hand over function calls like `top`, `ps` and `man csh` to see what's happening.

   ii. Modify the program to avoid the creation of zombie processes.

(e) Now add support for I/O redirection on either or `both` stdin and/or `stdout`. Use `>` for output redirection, and `<` for input redirection. Consider the following command lines:

```
<cmd> <arg1> <arg2>   >  <outputfile>
<cmd> <arg1> <arg2>   <  <inputfile>   >  <outputfile>
```

The first line will execute command `<cmd>` with two arguments and print the output to file `<outputfile>`. In the second line, the contents of file `<inputfile>` will be input to `<cmd>` and the result of the command will again be output to file `<outputfile>`.

Also, `stdout` redirection should be possible for the internal commands `pwd`, `ls`, `environ`. For output redirection (`>`) the `<outputfile>` is created if it does not exist and truncated if it does.

You can check if input and output redirection work together by executing `wc -m <test.txt > test.cnt`, which should write the number of characters in `test.txt` (a text file created by you, that contains some text) into the file `test.cnt`.

Hint: To add I/O redirection, modify the child process created by **fork** by adding some code to open the input and output files specified on the command line. This should be done using the **open** system call. Next, use the **dup2** system call to replace the standard input or standard output streams with the appropriate file that was just opened. Finally, call **exec** to run the program.