Name ______________________ Matriculation no. ______________ Tutorial no. ________

Operating Systems 2013/14
Assignment 3

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marius Hillenbrand
Dipl.-Inform. Marc Rittinghaus

Submission Deadline: Tuesday, December 17th, 2013 – 9:30 a.m.

In this assignment you will study scheduling, communication, and synchronization of processes. In the practical part, you will implement a simple user-level scheduler. All the organizational words of the first assignment are still valid and have to be followed.

Please print out the pages containing **T-Questions** and answer them on your printout. Clearly mark every page with your name, matriculation number and **tutorial number**. Simply put it in the mailbox in the basement of building 50.34 (Info-Neubau).

P-Questions are programming assignments. Download the provided tarball from the VAB and make sure to use the included templates and Makefiles. Do not fiddle with the compiler flags. Submission instructions can be found in the first assignment sheet.

Any assignment handed in after its deadline will be ignored!

T-Question 3.1: Scheduling Algorithms

a. Depict the Gantt chart for scheduling the following processes with Round-Robin scheduling and a timeslice length of 2 time units. New processes are added to the tail of the ready queue. Check the lecture slides for how to draw a Gantt chart for scheduling. **2 T-pt**

Process	Burst length	Arrival time
P_1	5	0
P_2	2	1
P_3	3	2

b. Calculate the average waiting time in the example given in 3.1 a. **1 T-pt**

Name ____________ Matriculation no. ____________ Tutorial no. ____________

c. Calculate the average turnaround-time in the example given in 3.1 a. **1 T-pt**

d. Which of the following scheduling algorithms are vulnerable to process starvation, which are not? (correctly marked: 0.5P, not marked: 0P, incorrectly marked: -0.5P) **3 T-pt**

vulnerable	not vulnerable	
☐	☐	Round-Robin with priorities
☐	☐	FCFS
☐	☐	Lottery Scheduling
☐	☐	Preemptive SJF
☐	☐	MLFB
☐	☐	Linux CFS

T-Question 3.2: Interprocess Communication

a. What are the two fundamental models of interprocess communication? **1 T-pt**

T-Question 3.3: Synchronization

a. What are the three requirements for a valid solution of the critical-section problem? **1 T-pt**

b. What is the difference between a binary semaphore and a counting semaphore? **1 T-pt**

Name ______________________ Matriculation no. ______________ Tutorial no. ________

c. A spinlock might be implemented using the following code:

```
while( TestAndSet( &lock ) ) ;
```

Why can this code exhibit poor performance on modern multiprocessor systems? **1 T-pt**

d. Which of the following statements are correct, which are incorrect? (correctly marked: 0.5P, not marked: 0P, incorrectly marked: -0.5P) **3 T-pt**

correct	incorrect	
☐	☐	Spinlocks can be implemented entirely in userspace.
☐	☐	Disabling interrupts is a valid solution to the critical section problem on uniprocessor systems.
☐	☐	The `compareAndSwap` instruction is useful on uniprocessor systems.
☐	☐	The `Swap` instruction atomically swaps two registers.
☐	☐	Disabling interrupts is a valid solution to the critical section problem on SMP systems.

T-Question 3.4: Deadlock Basics

a. Enumerate and explain the 4 necessary conditions for a deadlock. **2 T-pt**

P-Question 3.1: Scheduling Data Structures

a. Implement a priority-based run queue that could be used by an operating system to keep track of all runnable processes. Each process is identified by a unique id (`id` ≥ 0) and has a priority (where a higher number means a higher priority).

   The algorithmic complexity of each function should be $O(n)$ at most, while `pop` whould work in $O(1)$ because of it is called for each scheduling decision. The following functions need to be supported: **4 P-pt**

```
void insert ( int id, int priority );
int pop ();            // get id of element with highest priority,
                       // and remove it from the list
                       // (return -1 if there are no more elements)
void popId ( int id ); // remove element with given id, if present
```

Note: You only have to implement the data structure and the functions mentioned above! You **don't** have to implement code to fill your data structure with processes (though you might want to do so for testing) or to select/dispatch/schedule the processes.

P-Question 3.2: Signal Handlers

a. Read `man 7 signal` and `man 2 sigaction`. Add a `SIGINT`-handler to the program listed below. The handler should write "Received Signal" on the screen. In addition, when the handler receives three signals within three seconds it should write "Shutting down" and then terminate the program with the exit status 1. **3 P-pt**

```
int main()
{
        printf( "Hello World!" );
        for( ;; )
        {
                /* infinite loop */
        }
        return 0;
}
```

P-Question 3.3: The clone System Call & POSIX Semaphores

a. Recall the code in Tutorial-Assignment 7.1 (full code in asst3-clone/ folder in tarball). OpenMP was used to create multiple threads in the example `main-openmp.c`. Write an alternative which creates the threads using the `clone` system call. Place it into `main-clone.c`. Carefully read the manual page (`man 2 clone`) before you start writing code; think about the flags you need to pass to the `clone` call to have the threads share the same address space and to be waitable by the `waitpid` syscall. Don't modify the signature of the `increment`-function! **2 P-pt**

b. Use POSIX semaphores to solve the critical section problem in `tally.c`. If you failed to complete the a. part of this assignment, use the OpenMP version for testing (`./tally-openmp`). Read `man 3 sem_init`, `man 3 sem_post`, `man 3 sem_wait` and `man 3 sem_destroy`. Keep the critical section as short as possible and clean up the semaphores at the end! **2 P-pt**

P-Question 3.4: Round-Robin Scheduling

In this assignment, we offer you a simple implementation of many-to-one (user-level) threads. `ult.c` contains code for thread management (such as context switching and random blocking/deblocking of threads). `scheduler.c` is meant to contain the implementation of the actual scheduling policy.
To get familiar with the provided code, answer yourself the following questions:

- What thread states are supported by our implementation?
- What function is executed by all threads (except for the idle thread)?
- What is the thread id of the idle thread?
- What is the probability that a thread will block during one time slice?
- What does the thread management code do to emulate a timer interrupt?
- What is the frequency of the emulated timer interrupt?
- Which function can be considered to be the "interrupt-handler" for the emulated timer interrupt?
- Given a pointer `char *p` to a thread's context (in the `threads`-array), write C code that derives the thread's id.

a. The current scheduler only switches between thread 0 and thread 1. Modify it so that it selects the next thread to be executed among **all** runnable threads according to a **Round-Robin scheduling policy**. Do not reorder the `threads`-array! **5 P-pt**

Total:
16 T-pt
16 P-pt