Operating Systems 2016/17
Assignment 8

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

Submission Deadline: Monday, December 19th, 2016 – 23:59

A new assignment will be published every week, right after the last one was due. It must be completed before its submission deadline.

The assignments must be filled out online in ILIAS. Handwritten solutions are no longer accepted. You will find the online version for each assignment in your tutorial's directory. **P-Questions** are programming assignments. Download the provided template from ILIAS. Do not fiddle with the compiler flags. Submission instructions can be found on the first assignment.

In this assignment you will dive into the basics of kernel synchronization and take a look at deadlocks.

T-Question 8.1: Kernel Synchronization

a. On a multi-processor system, when must local interrupts be disabled for kernel spinlocks? **1 T-pt**

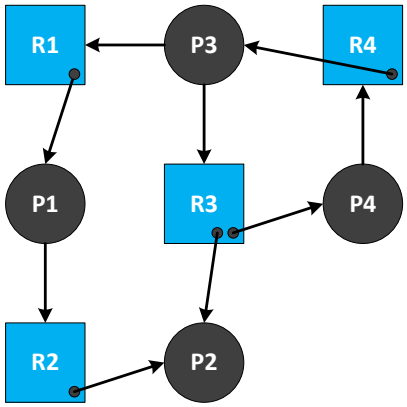
b. Why is disabling interrupts a privileged instruction? **1 T-pt**

T-Question 8.2: Deadlocks

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

b. How can periodic process snapshots be used to recover from deadlocks? What is a major disadvantage of this method? **2 T-pt**

T-Question 8.3: Resource Allocation Graph



a. Describe the situation depicted in the resource allocation graph. **2 T-pt**

b. Has a deadlock occurred in the above situation? Why, or why not? **1 T-pt**

c. What changes if $P1$ also requests $R3$? **1 T-pt**

P-Question 8.1: Multi Mutex

Download the template **p1** for this assignment from ILIAS. You may only modify and upload the file `multi_mutex.c`.

To prevent deadlocks one of the 4 necessary deadlock conditions must be broken. Two methods to achieve this are:

(a) Acquire multiple locks 'atomically', that is acquire all or none.

(b) Assign each lock a fixed number and acquire the locks only in a specific order.

In this question you will write a mutex wrapper that locks multiple pthread mutexes using the above methods.

a. Write a function that unlocks all pthread mutexes in the supplied `mutexv` array. The number of mutexes is given in `mutexc`. The function should return 0 on success, -1 otherwise. **1 P-pt**

```
int multi_mutex_unlock(pthread_mutex_t **mutexv, int mutexc);
```

b. Write a function that uses approach (a) to avoid deadlocks and tries to lock all of the supplied mutexes, or none, if one of the mutexes cannot be acquired. That is, on failure, the function should release all previously acquired mutexes. The function should return 0 on success, -1 otherwise. **2 P-pt**

```
int multi_mutex_trylock(pthread_mutex_t **mutexv, int mutexc);
```

c. Write a function that uses approach (b) to avoid deadlocks. Acquire the mutexes in ascending order based on their addresses in memory. The function should return 0 on success, -1 otherwise. **2 P-pt**

```
int multi_mutex_lock(pthread_mutex_t **mutexv, int mutexc);
```

P-Question 8.2: Interrupt-based Synchronization

Download the template **p2** for this assignment from ILIAS. You may only modify and upload the file `priority_interrupts.c`.

On a single-processor system, synchronization in the kernel may be accomplished by disabling all interrupts. An optimization of this approach is to disable only a subset of interrupts by assigning each interrupt a priority level $p \in [1, \#Interrupts]$ and obeying a set of rules:

(a) The interrupt handler H_p for priority p may access data shared with the interrupt handlers $[H_1, H_{p-1}]$.
(b) Executing H_p will disable the interrupt for level p.
(c) Disabling the interrupt for level p will also disable the interrupts for the levels $[1, p-1]$.
(d) The *interrupt request level (IRQL)* denotes the highest disabled interrupt priority level. Higher levels are activated!
(e) Kernel code that accesses data shared with H_p must raise the IRQL to p if the current level is lower.
(f) When no interrupts are disabled the IRQL is 0.

With this technique high priority interrupts (e.g., reception of a network packet) can still interrupt the kernel if it is in a critical section for a lower level (e.g., timer interrupt).

In this question you will write an emulation of this locking scheme in user-space with the help of POSIX signals. A signal may asynchronously interrupt a thread in user-space and lead it to execute a user-defined signal handler. Read `man 7 signal` to get familiar with POSIX signals. See `man pkill` on how to send a signal to a process from the terminal.

We define 2 priority levels ($p \in [1, 2]$) and use the signals `SIGUSR1` and `SIGUSR2` to represent the corresponding interrupts.

a. Write a function that executes a user-supplied function with the signals `SIGUSR1` and `SIGUSR2` enabled and initialized to run the given handler functions. Your function should fulfill the following requirements: **3 P-pt**

- Uses `sigaction()` to setup and enable the signal handlers for `SIGUSR1` and `SIGUSR2`.
- Configures `SIGUSR2` to also disable `SIGUSR1` on execution (rule (b) and (c)).
- Runs the user-supplied function.
- Restores the original or default handlers for `SIGUSR1` and `SIGUSR2` on exit.
- Returns 0 on success, -1 otherwise.

```
int run_with_signals(void (*runnable)(void),
                     void (*usr1)(void),
                     void (*usr2)(void));
```

b. Write the functions that set the IRQL to a certain level according to the above rules with the help of `sigprocmask()`. The functions should return 0 on success, -1 otherwise. **3 P-pt**

```
int set_irql_0/1/2(void);
```

Total: 10 T-pt 11 P-pt