1 Mutexes

a. Inform yourself about the mutex IPC mechanism in POSIX and especially about the functions ***pthread_mutexattr_init()***, ***pthread_mutex_init()***, ***pthread_mutex_lock()***, ***pthread_mutex_unlock()*** and ***pthread_mutex_destroy()***.

b. Explain the following program and clarify these aspects:
Where is the critical section?
How is safe access to the critical section controlled?
Analyze the run-time behavior!
How is the buffer state controlled?
On what condition does the program terminate?

```
//////////////////////////////////////////////////////////////////////////////
// Course:                    Real Time Systems
// Lecturer:                  Dr.-Ing. Frank Golatowski
// Exercise instructor:       M.Sc. Michael Rethfeldt
// Exercise:                  6
// Task:                      1
// Name:                      aufgabe1.c
// Description:               ?
// Compiler call:             cc -o aufgabe1 aufgabe1.c -lpthread
//////////////////////////////////////////////////////////////////////////////
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

//////////////////////////////////////////////////////////////////////////////
int     make_new_item(void);
void    consume_item(char a, int count);
void*   reader_function(void*);
void    writer_function(void);

//////////////////////////////////////////////////////////////////////////////
char                buffer;
int                 buffer_has_item = 0;
pthread_mutex_t     mutex;

//////////////////////////////////////////////////////////////////////////////
int main(int argc, char *argv[])
{   pthread_mutexattr_t mutex_attr;
    pthread_attr_t      attr;
    pthread_t           reader;

    if ( pthread_mutexattr_init(&mutex_attr) == 0 )
    {   if ( pthread_mutex_init(&mutex, &mutex_attr) == 0 )
        {   if ( pthread_attr_init( &attr ) == 0)
            {   if ( pthread_create( &reader, &attr, reader_function,0 ) ==0 )
                {   writer_function();
                    pthread_join(reader,0);
                }
            }
            pthread_mutex_destroy( &mutex);
        }
    }
    printf("Main thread finished!\n");
    return 0;
}

//////////////////////////////////////////////////////////////////////////////
void writer_function(void)              // Father
{   char    a=0;
    int     count=0;
```

```c
    while( a != 'q' )                   // Abort ?
    {
        pthread_mutex_lock( &mutex );

        if ( buffer_has_item == 0 )
        {
            a = make_new_item();
            buffer=a;
            buffer_has_item = 1;
            printf("Thread 1: buffer: '%c', loop count=%d\n", a, count);
            count=0;
        }
        else
            count++;

        pthread_mutex_unlock( &mutex );
        //pthread_yield();
    }
}

//////////////////////////////////////////////////////////////////////////////
void* reader_function(void *ptr)        // Son
{   char     a=0;
    int      count=0;

    while( a != 'q' )                   // Abort ?
    {   pthread_mutex_lock( &mutex );
        if ( buffer_has_item == 1)
        {   a=buffer;
            if ( a != 'q' )
                consume_item(a,count);
            buffer_has_item = 0;
            count=0;
        }
        else
            count++;

        pthread_mutex_unlock( &mutex );
        //pthread_yield();

    }
    printf("Reader thread finished\n");
    return 0;
}

//////////////////////////////////////////////////////////////////////////////
int make_new_item()
{   static int  buffer='a';
    return buffer++;
}

//////////////////////////////////////////////////////////////////////////////
void consume_item(char a, int count)
{   printf("Thread 2: buffer read with '%c', loop count=%d\n", a, count);
}
```

Fig. 1: aufgabe1.c

2 Condition variables

The unnecessary polling of adequate buffer states in task 1 can be avoided by means of condition variables. Condition variables are used to grant access to critical sections only under certain conditions. Note that operations on condition variables within critical sections are performed via corresponding functions of the operating system.

Extend the previous program by using condition variables. Consider the following steps:

a. Determine the constraints for accessing the critical sections in task 1.
b. Inform yourself about the required system functions ***pthread_cond_init()***, ***pthread_cond_wait()***, ***pthread_cond_signal()*** and ***pthread_cond_destroy()***.

Note: ***pthread_cond_wait()*** must be called before ***signal()***! Otherwise the signal is lost!

3 Binary semaphores

The POSIX standard also offers semaphores for inter-thread synchronization. Usage of POSIX semaphores is very simple in comparison to the older UNIX API. Only four functions are relevant: ***sem_init()***, ***sem_wait()***, ***sem_post()*** and ***sem_destroy()***.

a. Modify the source code of task 1 in such a way that POSIX semaphores are used!

4 Counting semaphores

a. Explain the following program:

```
//////////////////////////////////////////////////////////////////////////////
// Veranstaltung:   Echtzeitbetriebssysteme
// Dozent:          Dr.-Ing. Frank Golatowski
// Uebungsleiter:   M.Sc. Michael Rethfeldt
// Uebung:          6
// Aufgabe:         4
// Name:            aufgabe4.c
// Beschreibung:    ?
// Compiler-Aufruf: cc -o aufgabe4 aufgabe4.c -lpthread
//////////////////////////////////////////////////////////////////////////////
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <semaphore.h>
#include <limits.h>
#include <sched.h>
//////////////////////////////////////////////////////////////////////////////
#define MAX_THREADS         50
#define MAX_REGION_ENTRIES  10

void* thread_function(void *ptr);
void enter_region(size_t id);
void leave_region(size_t id);

int             cancel_threads=0;
sem_t           region_semaphore;
pthread_mutex_t region_mutex=PTHREAD_MUTEX_INITIALIZER;
size_t          regionentries[MAX_REGION_ENTRIES];

//////////////////////////////////////////////////////////////////////////////
int main(int argc, char *argv[])
{   pthread_attr_t      attr;
    pthread_t           threads[MAX_THREADS];
    size_t              i, counter=0;

    // 0=local process, MAX_REGION_ENTRIES=initial state
    if ( sem_init( &region_semaphore, 0, MAX_REGION_ENTRIES ) == 0 )
    {   if ( pthread_attr_init( &attr ) ==0 )
        {   for (i=0;i<MAX_THREADS;i++)
            {
                if( pthread_create(&(threads[counter]),&attr,thread_function,(void*)i+1) ==0 )
                    counter++;
            }
            scanf("%d", &cancel_threads);  // set cancel condition
            for (i=0;i<counter;i++)
            {
                pthread_join( threads[i], 0);
            }
        }
        sem_destroy( &region_semaphore);
    }
    printf("Main thread finished!\n");
}

//////////////////////////////////////////////////////////////////////////////
void* thread_function(void *ptr)
{
    size_t threadid=(size_t)ptr;
    time_t  t;

    while( ! cancel_threads )           // check cancel condition
    {
        enter_region(threadid);
        t=time(0);                      // short wait (passive)
        usleep((localtime(&t)->tm_sec+localtime(&t)->tm_min)*1000 );
        leave_region(threadid);
        sched_yield();                  // yield CPU
    }
    printf("Thread %zu finished!\n", threadid );
    return 0;
}
```

```
////////////////////////////////////////////////////////////////////////////////
void print_region_entries()
{
    int i;
    printf("Threads in region: ");
    for (i=0; i<MAX_REGION_ENTRIES; i++)
        printf("%zu ", regionentries[i] );
    printf("\n");

    for(i=0;i<1000000;i++)              // short wait (active)
        i;
}


////////////////////////////////////////////////////////////////////////////////
void enter_region(size_t id)
{
    int i;
    sem_wait( &region_semaphore );
    pthread_mutex_lock(&region_mutex);
    for (i=0;i<MAX_REGION_ENTRIES;i++)
    {
        if ( regionentries[i]==0 )
        {   regionentries[i]=id;
            break;
        }
    }
    print_region_entries();
    pthread_mutex_unlock(&region_mutex);
}


////////////////////////////////////////////////////////////////////////////////
void leave_region(size_t id)
{
    int i;

    pthread_mutex_lock(&region_mutex);
    for (i=0;i<MAX_REGION_ENTRIES;i++)
    {   if ( regionentries[i]==id )
        {   regionentries[i]=0;
            break;
        }
    }
    print_region_entries();
    pthread_mutex_unlock(&region_mutex);
    sem_post( &region_semaphore );
}
```

Fig. 2: aufgabe4.c