M.Sc. Michael Rethfeldt

1 Message queue within a single process

a. Write a program (only 1 process) that creates a message queue, writes a line of characters to it, reads from the message queue, and finally closes it! Use the functions ftok(), msgget(), msgsnd(), msgrcv() and msgctl()!
b. How can access rights be configured for message queues?

2 Message queues between multiple processes

a. Write a program that creates multiple son processes that communicate with the father process by means of a message queue. In this example, text messages identifying the sons shall be transmitted, e.g., "I am son no. 25 with process ID 0x4501".
   You can start by re-using the source code of task 3 from exercise 2. The number of son processes to be created shall be passed as command line argument to the program. Check if your program always terminates correctly!
b. What is the meaning of parameter ***msgbuf.mtype*** in functions ***msgsnd()/msgrcv()***?
c. **Additional task:**
   Write a program SENDER and a program RECEIVER. Both programs open a message queue. The sender periodically transmits a message containing its PID and current time stamp to the receiver. The time interval shall be configurable via a command line parameter. Check your program by creating multiple sender instances with different transmission intervals!

3 Shared memory and semaphores

a. Explain the following program! What is the shared memory used for? Are there any mistakes in the source code?
b. What functions are used in conjunction with shared memory and semaphores? Are there analogies to the other inter-process synchronization variants used before?

```
////////////////////////////////////////////////////////////////////////////
// Course:                  Real Time Systems
// Lecturer:                Dr.-Ing. Frank Golatowski
// Exercise instructor:     M.Sc. Michael Rethfeldt
// Exercise:                4
// Task:                    3
// Name:                    aufgabe3.c
// Description:             Exchange of data between process using shared memory
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#include <memory.h>
```

```
////////////////////////////////////////////////////////////////////////////
#define DEFAULT_PROCESSCOUNT    3
//#define PERMISSIONS           0666
#define PERMISSIONS             S_IRUSR | S_IWUSR
#define MAXBUFFER               20000
#define LOOPS                   300
#define SEMAPHORE_NUMBER        0
#define LOCK_SEMAPHORE(id)      { if ( semop(id, &(sem_lock[0]), 1 )==-1 )\
                                    printerrorexit("Error locking semaphore!", errno ); }
#define UNLOCK_SEMAPHORE(id)    { if ( semop(id, &(sem_unlock[0]), 1)==-1 )\
                                    printerrorexit("Error unlocking semaphore!", errno ); }
////////////////////////////////////////////////////////////////////////////
typedef struct
{       long    exitcount;
        char    buf[MAXBUFFER];
} shared_mem;
union semun                         // should be defined in sys/sem.h
{   int val;                        // value for SETVAL
    struct semid_ds *buf;           // buffer for IPC_STAT, IPC_SET
    unsigned short *array;          // array for GETALL, SETALL
                                    // Linux specific part:
    struct seminfo *__buf;          // buffer for IPC_INFO
};
static struct sembuf sem_lock[1]=
{   SEMAPHORE_NUMBER, -1,   0       // semaphore 0, operation decrement by 1, flags=0
};
static struct sembuf sem_unlock[1]=
{   SEMAPHORE_NUMBER,  1,   0       // semaphore 0, operation increment by 1, flags=0
};

////////////////////////////////////////////////////////////////////////////
void printerrorexit(char *str, int errornumber)
{   fprintf(stderr, "%s %d=%s\n", str, errornumber, strerror(errornumber));
    exit(errno);
}
////////////////////////////////////////////////////////////////////////////

int main(int argc, char *argv[])
{   key_t       semkey,shmkey;
    int         i, semid, maxprocesses, shmid, status, retvalue;
    shared_mem  *shmem;
    union semun semopts;

    // determine process number
    maxprocesses=DEFAULT_PROCESSCOUNT;
    if ( argc>1 )
    {   if ( sscanf(argv[1], "%d", &i) && i>0 )
            maxprocesses=i;
        else
            printerrorexit("Error getting correct number of processes!", i);
    }
    printf("Number of processes is: %d\n", maxprocesses);

    // create semaphore key
    semkey=ftok( argv[0], 1 );
    if ( semkey==-1 )
        printerrorexit("Error obtaining key_t!", errno);
    printf("SemKey is: 0x%x\n", semkey);
    // create semaphore
    if ((semid = semget(semkey, SEMAPHORE_NUMBER+1, PERMISSIONS | IPC_CREAT)) < 0)
        printerrorexit("Can't create semaphore!", errno);
    semopts.val = 1;
    semctl(semid, SEMAPHORE_NUMBER, SETVAL, semopts);

    // create shared memory key
    shmkey=ftok( argv[0], 2 );
    if ( shmkey==-1 )
        printerrorexit("Error obtaining key_t!", errno);
    printf("ShmKey is: 0x%x\n", shmkey);
    // create shared memory
    if ((shmid = shmget(shmkey, sizeof(shared_mem), PERMISSIONS | IPC_CREAT)) < 0)
        printerrorexit("Can't create shared memory segment!", errno);
    // attach shared memory segment to current process
    if (!(shmem = (shared_mem*) shmat(shmid, 0, 0)))
        printerrorexit("Can't attach shared memory segment!", errno);
    shmem->exitcount=0;
    shmem->buf[0]=0;
```

```
    for ( i=0; i<maxprocesses; i++ )
    {
     switch ( fork() )
        {   case 0:     {   int j;
                            for (j=0; j<LOOPS; j++)
                            {   LOCK_SEMAPHORE(semid);
                                if (strlen(shmem->buf) < MAXBUFFER-100)
                                    snprintf(shmem->buf+strlen(shmem->buf),
                                        MAXBUFFER-strlen(shmem->buf),
                                        "Process: %d, PID: %d, Loop %d.\n", i+1,
                                        getpid(), j);
                                UNLOCK_SEMAPHORE(semid);
                            }
                            LOCK_SEMAPHORE(semid);
                            shmem->exitcount++;
                            UNLOCK_SEMAPHORE(semid);
                            exit(1);
                        }
                        break;
            case -1:    exit(2);
        }
    }
    {   int count;
        while(1)
        {   LOCK_SEMAPHORE(semid);
            if ( shmem->buf[0] != 0 )
                printf("%s\n",shmem->buf);
            shmem->buf[0]=0;
            count=shmem->exitcount;
            UNLOCK_SEMAPHORE(semid);
            if ( count==maxprocesses )
                break;
        }
    }

    // Remove shared memory from system
    retvalue=shmctl(shmid, IPC_RMID, 0);
    if (retvalue)
        printerrorexit("Error removing shared memory!", errno);

    // Remove semaphore from system
    retvalue=semctl(semid, IPC_RMID, 0);
    if (retvalue)
        printerrorexit("Error removing semaphore!", errno);

    return 0;
}
```

Fig. 1: aufgabe3.c

4 Shared memory II

a. What is wrong in the following program?

```
////////////////////////////////////////////////////////////////////////////
// Course:                  Real Time Systems
// Lecturer:                Dr.-Ing. Frank Golatowski
// Exercise instructor:     M.Sc. Michael Rethfeldt
// Exercise:                4
// Task:                    4
// Name:                    aufgabe4.c
// Description:             Exchange of data between process
////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#include <memory.h>
```

```
////////////////////////////////////////////////////////////////////////////
#define DEFAULT_PROCESSCOUNT    3
//#define PERMISSIONS           0666
#define PERMISSIONS             S_IRUSR | S_IWUSR
#define MAXBUFFER               20000
#define LOOPS                   300
#define SEMAPHORE_NUMBER        0
#define LOCK_SEMAPHORE(id)      { if ( semop(id, &(sem_lock[0]), 1 )==-1 )\
                                    printerrorexit("Error locking semaphore!", errno ); }
#define UNLOCK_SEMAPHORE(id)    { if ( semop(id, &(sem_unlock[0]), 1)==-1 )\
                                    printerrorexit("Error unlocking semaphore!", errno ); }

////////////////////////////////////////////////////////////////////////////
typedef struct
{       long    exitcount;
        char    buf[MAXBUFFER];
} shared_mem;
union semun                         // should be defined in sys/sem.h
{   int val;                        // value for SETVAL
    struct semid_ds *buf;           // buffer for IPC_STAT, IPC_SET
    unsigned short *array;          // array for GETALL, SETALL
                                    // Linux specific part:
    struct seminfo *__buf;          // buffer for IPC_INFO
};
static struct sembuf sem_lock[1]=
{   SEMAPHORE_NUMBER, -1,   0       // semaphore 0, operation decrement by 1, flags=0
};
static struct sembuf sem_unlock[1]=
{   SEMAPHORE_NUMBER,  1,   0       // semaphore 0, operation increment by 1, flags=0
};

////////////////////////////////////////////////////////////////////////////
void printerrorexit(char *str, int errornumber)
{   fprintf(stderr, "%s %d=%s\n", str, errornumber, strerror(errornumber));
    exit(errno);
}

////////////////////////////////////////////////////////////////////////////
int main(int argc, char *argv[])
{   key_t       semkey,shmkey;
    int         i, semid, maxprocesses, shmid, retvalue, status;
    shared_mem  *shmem;
    union semun semopts;

    // determine process number
    maxprocesses=DEFAULT_PROCESSCOUNT;
    if ( argc>1 )
    {   if ( sscanf(argv[1], "%d", &i) && i>0 )
            maxprocesses=i;
        else
            printerrorexit("Error getting correct number of processes!", i);
    }
    printf("Number of processes is: %d\n", maxprocesses);

    // create semaphore key
    semkey=ftok( argv[0], 1 );
    if ( semkey==-1 )
        printerrorexit("Error obtaining key_t!", errno );
    printf("SemKey is: 0x%x\n", semkey);

    // create semaphore
    if ((semid = semget(semkey, SEMAPHORE_NUMBER+1, PERMISSIONS | IPC_CREAT)) < 0)
        printerrorexit("Can't create semaphore!", errno);
    semopts.val = 1;
    semctl( semid, SEMAPHORE_NUMBER, SETVAL, semopts);

    shmem=(shared_mem*) malloc(sizeof(shared_mem));
    if (!shmem)
        printerrorexit("Error allocating memory!", errno);

    shmem->exitcount=0;
    shmem->buf[0]=0;

    for ( i=0; i<maxprocesses; i++ )
    {   switch ( fork() )
        {   case 0:     {   int j;
                            for (j=0; j<LOOPS; j++)
                            {   LOCK_SEMAPHORE(semid);
```

```
                                if (strlen(shmem->buf) < MAXBUFFER-100)
                                    snprintf(shmem->buf+strlen(shmem->buf),
                                        MAXBUFFER-strlen(shmem->buf),
                                        "Process: %d, PID: %d, Loop %d.\n", i,
                                        getpid(), j);
                                UNLOCK_SEMAPHORE(semid);
                            }
                            LOCK_SEMAPHORE(semid);
                            shmem->exitcount++;
                            UNLOCK_SEMAPHORE(semid);
                            exit(1);
                        }
                        break;
                case -1:    exit(1);
            }
        }
        {   int count;
            while(1)
            {   LOCK_SEMAPHORE(semid);
                if ( shmem->buf[0] != 0 )
                    printf(shmem->buf);
                shmem->buf[0]=0;
                count=shmem->exitcount;
                UNLOCK_SEMAPHORE(semid);
                if ( count==maxprocesses )
                    break;
            }
        }

        free(shmem);

        // Remove semaphore from system
        retvalue=semctl(semid, IPC_RMID, 0);
        if ( retvalue )
            printerrorexit("Error removing semaphore!", errno);

        return 0;
}
```

Fig. 2: aufgabe4.c

5 Exchange of data between different programs

a. Implement a consumer/producer scenario with shared memory: Study the following source code of the consumer. Write the corresponding program of the producer!

b. Start both programs in two independent terminals and exchange data!

```
////////////////////////////////////////////////////////////////////////////
// Course:                  Real Time Systems
// Lecturer:                Dr.-Ing. Frank Golatowski
// Exercise instructor:     M.Sc. Michael Rethfeldt
// Exercise:                4
// Task:                    5
// Name:                    aufgabe5_consumer.c
// Description:             Exchange of data between indep. proc. using shared memory
////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#include <memory.h>

////////////////////////////////////////////////////////////////////////////
//#define PERMISSIONS           0666
#define PERMISSIONS             S_IRUSR | S_IWUSR
#define MAXBUFFER               20000
#define SEMAPHORE_NUMBER        0
```

```
#define LOCK_SEMAPHORE(id)      { if ( semop(id, &(sem_lock[0]), 1 )==-1 )\
                                    printerrorexit("Error locking semaphore!", errno ); }
#define UNLOCK_SEMAPHORE(id)    { if ( semop(id, &(sem_unlock[0]), 1)==-1 )\
                                    printerrorexit("Error unlocking semaphore!", errno ); }
////////////////////////////////////////////////////////////////////////////
typedef struct
{       long    exitcount;
        char    buf[MAXBUFFER];
} shared_mem;

union semun                         // should be defined in sys/sem.h
{   int val;                        // value for SETVAL
    struct semid_ds *buf;           // buffer for IPC_STAT, IPC_SET
    unsigned short *array;          // array for GETALL, SETALL
                                    // Linux specific part:
    struct seminfo *__buf;          // buffer for IPC_INFO
};

static struct sembuf sem_lock[1]=
{   SEMAPHORE_NUMBER, -1,   0       // semaphore 0, operation decrement by 1, flags=0
};
static struct sembuf sem_unlock[1]=
{   SEMAPHORE_NUMBER,  1,   0       // semaphore 0, operation increment by 1, flags=0
};

////////////////////////////////////////////////////////////////////////////
void printerrorexit(char *str, int errornumber)
{   fprintf(stderr, "%s %d=%s\n", str, errornumber, strerror(errornumber));
    exit(errno);
}

////////////////////////////////////////////////////////////////////////////
// argv[1] = name of semaphore/shared memory pair
int main(int argc, char *argv[])
{   key_t       semkey, shmkey;
    int         i, semid, exitconsumer=0, shmid;
    shared_mem  *shmem;
    union semun semopts;

    // create semaphore key
    semkey=ftok( argv[1], 1 );
    if ( semkey==-1 )
        printerrorexit("Error obtaining semaphore key_t!", errno);
    printf("SemKey is: 0x%x\n", semkey);

    // create semaphore
    if ((semid = semget(semkey, SEMAPHORE_NUMBER+1, PERMISSIONS | IPC_CREAT)) < 0)
        printerrorexit("Can't create semaphore!", errno);
    semopts.val = 1;
    semctl(semid, SEMAPHORE_NUMBER, SETVAL, semopts);

    // create shared memory key
    shmkey=ftok( argv[1], 2 );
    if ( shmkey==-1 )
        printerrorexit("Error obtaining shared memory key_t!", errno);
    printf("ShmKey is: 0x%x\n", shmkey);

    // create shared memory
    if ((shmid = shmget(shmkey, sizeof(shared_mem), PERMISSIONS | IPC_CREAT)) < 0)
        printerrorexit("Can't create shared memory segment!", errno);

    // attach shared memory segment to current process
    if (!(shmem = (shared_mem*) shmat(shmid, 0, 0)))
        printerrorexit("Can't attach shared memory segment!", errno);

    shmem->exitcount=0;
    shmem->buf[0]=0;

    // Read data from shared memory
    while(!exitconsumer)
    {   LOCK_SEMAPHORE(semid);
        if ( shmem->buf[0] != 0 )
        {   char *ptr=shmem->buf;
            while(*ptr)
            {   if ( *ptr==EOF )
                    exitconsumer=1;
                ptr++;
            }
```

```
            printf("%s", shmem->buf);
        }
        shmem->buf[0]=0;
        UNLOCK_SEMAPHORE(semid);
    }

    // Remove shared memory from system
    if (shmctl(shmid, IPC_RMID, 0))
        printerrorexit("Error removing shared memory!", errno);

    // Remove semaphore from system
    if (semctl(semid, IPC_RMID, 0))
        printerrorexit("Error removing semaphore!", errno);

    return 0;
}
```

Fig. 3: aufgabe5_consumer.c