Computation structures

Support for problem-solving lesson #8

Exercise 1

Give an implementation of the mutual exclusion between 2 processes using only the blocking message queues of size 0 as synchronization mechanism.

Exercise 1

Recall

- A blocking message queue of size 0 means that:
  - **q?x** will block the calling process until another process executes **q!x**;
  - Symmetrically, **q!x** will block the calling process until another process executes **q?x**.

- Blocking message queues of size 0 are an *abstract* concept
  - They are not directly implemented (at least, not using System V).
  - They can, however, be implemented using semaphores.

```
shared semaphore sReceive = 0, sSend = 0;
q!x  → signal(sReceive); wait(sSend);
q?x → signal(sSend);      wait(sReceive);
```

  - Does not consider the message **x**. See slide 248 for a complete example.

Exercise 1

What you should not do:

```
#define wait 0
#define signal 1
shared chan q[0];
```

```
//Process 1                //Process 2
while(true) {              while(true) {
  //Non critical             //Non critical
  q!wait;                    q?wait;
  //Critical                 //Critical
  q!signal;                  q?signal;
}                          }
```

Why isn't this good?

- This is not mutual exclusion. This is *Rendez-vous*. Both processes will wait each other before and after the critical section (and both will thus be able to execute instructions *in* the critical section).
- How would you scale this to a mutual exclusion with $N > 2$ processes?

Exercise 1

Each process should execute the same code (to be scalable)

```
#define wait 0
#define signal 1
shared chan q[0];
```

```
//Process 1
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 2
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

But now, each process is blocked. How can I get out of the deadlock?

Exercise 1

Each process should execute the same code (to be scalable)

```
#define wait 0
#define signal 1
shared chan q[0];
```

```
//Process 1
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 2
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 3
int value = 0;
while(true) {
  if(value == 0) {
     q?wait;
     value = 1;
  } else {
     q?signal;
     value = 0;
  }
}
```

But now, each process is blocked. How can I get out of the deadlock?

By addind an "unlocker" process.

Exercise 1

Let's convince ourselves that this works, by using a possible inter-leaving.

```
#define wait 0
#define signal 1
shared chan q[0];
```

```
//Process 1
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 2
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 3
int value = 0;
while(true) {
  if(value == 0) {
     q?wait;
     value = 1;
  } else {
     q?signal;
     value = 0;
  }
}
```

Process 1 gets the hand, and tries to enter the critical section. It is blocked on the q!wait operation.

Exercise 1

Let's convince ourselves that this works, by using a possible inter-leaving.

```
#define wait 0
#define signal 1
shared chan q[0];
```

```
//Process 1
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 2
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 3
int value = 0;
while(true) {
  if(value == 0) {
     q?wait;
     value = 1;
  } else {
     q?signal;
     value = 0;
  }
}
```

Process 3 gets the hand, and unlocks Process 1 thanks to the q?wait operation. It is also free to continue.

Exercise 1

Let's convince ourselves that this works, by using a possible inter-leaving.

```
#define wait 0
#define signal 1
shared chan q[0];
```

```
//Process 1
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 2
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 3
int value = 0;
while(true) {
  if(value == 0) {
     q?wait;
     value = 1;
  } else {
     q?signal;
     value = 0;
  }
}
```

Process 3 makes another loop, but is blocked on the **q?signal** operation.

Exercise 1

Let's convince ourselves that this works, by using a possible inter-leaving.

```
#define wait 0
#define signal 1
shared chan q[0];
```

```
//Process 1
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

→

```
//Process 2
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 3
int value = 0;
while(true) {
  if(value == 0) {
     q?wait;
     value = 1;
  } else {
     q?signal;
     value = 0;
  }
}
```

Process 2 takes the hand, but is blocked on the q!wait operation.

Exercise 1

Let's convince ourselves that this works, by using a possible inter-leaving.

```
#define wait 0
#define signal 1
shared chan q[0];
```

```
//Process 1
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 2
while(true) {
  //Non critical
  q!wait;
  //Critical
  q!signal;
}
```

```
//Process 3
int value = 0;
while(true) {
  if(value == 0) {
     q?wait;
     value = 1;
  } else {
     q?signal;
     value = 0;
  }
}
```

Process 1 is thus the only one that can proceed, and also the only one that can enter the critical section. It will unlock Process 3 when executing the q!signal operation, and Process 2 will get a chance to enter the critical section.

System V

From theory to practice

- Using system V message queues requires an additional include: <sys/msg.h>
- It also requires a structure to store the messages

```
struct mymsgbuf {
        long mtype;
        char mtext[MAX_SEND_SIZE]; //Be careful about the terminating '0'
};
```

- Creating a message queue: **int msgget (key_t** *key*, **int** *msgflg* **);**
- Posting a message: **int msgsnd (int** *msqid*, **struct msgbuf** **msgp*, **int** *msgsz*, **int** *msgflg* **);**
- Reading a message: **int msgrcv (int** *msqid*, **struct msgbuf** **msgp*, **int** *msgsz*, **long** *mtype*, **int** *msgflg* **);**
- Other operations on queues : **int msgctl (int** *msgqid*, **int** *cmd*, **struct msqid_ds** **buf* **);**

Exercise 2

Consider the following programs:

```
1   #include <stdio.h>
    #include <stdlib.h>
    #include <sys/msg.h>

    #define MSGLEN 128
6   #define KEY 345782

    struct {
          long mtype;
          char buf[MSGLEN];
11  } msg;

    int main() {
        int qid;

16      if((qid =msgget(KEY,IPC_CREAT|0666)) < 0)
            die("could not access the queue");

        if (msgrcv(qid,&msg,MSGLEN,1,0) < 0)
            die("failed to receive");
21      printf("got '%s'\n",msg.buf);

        if(msgctl(qid,IPC_RMID,0) < 0)
           die("warning : trailing queue");

26     return EXIT_SUCCESS;
    }
```

```
    /** → reuse lines 1...11 of receiver **/
2
    int main(int argc, char ** argv) {

        if (argc != 2) {
            fprintf(stderr,
7                       "Usage : %s <message>\n",
                        argv[0]);
            return EXIT_FAILURE;
        }

12      int qid;

        if ((qid=msgget(KEY,IPC_CREAT|0666)) < 0)
            die("could not access the queue");

17      msg.type = 1;
        strncpy(msg.buf, argv[1], MSGLEN);
        if(msgsnd(qid,&msg,MSGLEN,0) < 0)
            die("failed to send");

22      return EXIT_SUCCESS;
    }

    int die(char *msg) {
        perror(msg); exit(EXIT_FAILURE);
27  }
```

Can they be used to implement a *Rendez-vous* between two scripts?

Exercise 2

Can they be used to implement a *Rendez-vous* between two scripts?

```c
1   #include <stdio.h>
    #include <stdlib.h>
    #include <sys/msg.h>

    #define MSGLEN 128
6   #define KEY 345782

    struct {
          long mtype;
          char buf[MSGLEN];
11  } msg;

    int main() {
        int qid;

16      if((qid =msgget(KEY,IPC_CREAT|0666)) < 0)
            die("could not access the queue");

        if (msgrcv(qid,&msg,MSGLEN,1,0) < 0)
            die("failed to receive");
21      printf("got '%s'\n",msg.buf);

        if(msgctl(qid,IPC_RMID,0) < 0)
          die("warning : trailing queue");

26     return EXIT_SUCCESS;
    }
```

```c
    /** → reuse lines 1...11 of receiver **/
2
    int main(int argc, char ** argv) {

        if (argc != 2) {
            fprintf(stderr,
7                       "Usage : %s <message>\n",
                        argv[0]);
            return EXIT_FAILURE;
        }

12      int qid;

        if ((qid=msgget(KEY,IPC_CREAT|0666)) < 0)
            die("could not access the queue");

17      msg.type = 1;
        strncpy(msg.buf, argv[1], MSGLEN);
        if(msgsnd(qid,&msg,MSGLEN,0) < 0)
            die("failed to send");

22      return EXIT_SUCCESS;
    }

    int die(char *msg) {
        perror(msg); exit(EXIT_FAILURE);
27  }
```

Not really.

The receiver will wait until the sender sent something, but the reverse is not true.

Exercise 3

Consider the following programs:

```
 1  #include <stdio.h>
    #include <stdlib.h>
    #include <sys/msg.h>

    #define MSGLEN 128
 6  #define KEY 345782

    struct {
          long mtype;
          char buf[MSGLEN];
11  } msg;

    int main() {
       int qid;

16     if((qid =msgget(KEY,IPC_CREAT|0666)) < 0)
           die("could not access the queue");

       if (msgrcv(qid,&msg,MSGLEN,1,0) < 0)
           die("failed to receive");
21     printf("got '%s'\n",msg.buf);

       if(msgctl(qid,IPC_RMID,0) < 0)
          die("warning : trailing queue");

26    return EXIT_SUCCESS;
    }
```

```
    /** → reuse lines 1...11 of receiver **/
 2
    int main(int argc, char ** argv) {

       if (argc != 2) {
          fprintf(stderr,
 7                   "Usage : %s <message>\n",
                     argv[0]);
          return EXIT_FAILURE;
       }

12     int qid;

       if ((qid=msgget(KEY,IPC_CREAT|0666)) < 0)
          die("could not access the queue");

17     msg.type = 1;
       strncpy(msg.buf, argv[1], MSGLEN);
       if(msgsnd(qid,&msg,MSGLEN,0) < 0)
          die("failed to send");

22     return EXIT_SUCCESS;
    }

    int die(char *msg) {
       perror(msg); exit(EXIT_FAILURE);
27  }
```

Modify the above programs in order to design a reader and a writer that communicate through a message queue:

- The writer sends messages coming from the standard input (stdin) on the queue and ends by sending the "." symbol.
- The reader displays the messages from the queue on the standard output (stdout) and stops when it receives the "." symbol .

Exercise 3

Modify the program (...)

```
1   #include <stdio.h>
    #include <stdlib.h>
    #include <sys/msg.h>

    #define MSGLEN 128
6   #define KEY 345782

    struct {
          long mtype;
          char buf[MSGLEN];
11  } msg;

    int main() {
       int qid;

16     if((qid =msgget(KEY,IPC_CREAT|0666)) < 0)
           die("could not access the queue");

       if (msgrcv(qid,&msg,MSGLEN,1,0) < 0)
           die("failed to receive");
21     printf("got '%s'\n",msg.buf);

       if(msgctl(qid,IPC_RMID,0) < 0)
          die("warning : trailing queue");

26    return EXIT_SUCCESS;
    }
```

```
    /** → reuse lines 1...11 of receiver **/
2
    int main(int argc, char ** argv) {

       if (argc != 2) {
          fprintf(stderr,
7                    "Usage : %s <message>\n",
                     argv[0]);
          return EXIT_FAILURE;
       }

12     int qid;

       if ((qid=msgget(KEY,IPC_CREAT|0666)) < 0)
          die("could not access the queue");

17     msg.type = 1;
       strncpy(msg.buf, argv[1], MSGLEN);
       if(msgsnd(qid,&msg,MSGLEN,0) < 0)
          die("failed to send");

22     return EXIT_SUCCESS;
    }

    int die(char *msg) {
       perror(msg); exit(EXIT_FAILURE);
27  }
```

Original programs

Exercise 3

Modify the program (…)

```
1    #include <stdio.h>
     #include <stdlib.h>
     #include <sys/msg.h>

     #define MSGLEN 128
6    #define KEY 345782

     struct {
          long mtype;
          char buf[MSGLEN];
11   } msg;

     int main() {
        int qid;

16      if((qid =msgget(KEY,IPC_CREAT|0666)) < 0)
            die("could not access the queue");

        do {
           if (msgrcv(qid,&msg,MSGLEN,1,0) < 0)
21             die("failed to receive");
           printf("got '%s'\n",msg.buf);
        } while(strcmp(msg.buf,".") != 0);

        if(msgctl(qid,IPC_RMID,0) < 0)
26         die("warning : trailing queue");

       return EXIT_SUCCESS;
     }
```

```
     /** → reuse lines 1…11 of receiver **/
2
     int main(int argc, char ** argv) {

       /* if (argc != 2) {
           fprintf(stderr,
7                    "Usage : %s <message>\n",
                     argv[0]);
           return EXIT_FAILURE;
        }*/

12      int qid;

        if ((qid=msgget(KEY,IPC_CREAT|0666)) < 0)
           die("could not access the queue");

17      msg.type = 1;
        do {
            fgets(msg.buf,MSGLEN,stdin);
            if(msgsnd(qid,&msg,MSGLEN,0) < 0)
               die("failed to send");
22      } while(strcmp(msg.buf,".") != 0);

        return EXIT_SUCCESS;
     }

27   int die(char *msg) {
        perror(msg); exit(EXIT_FAILURE);
     }
```

Updated programs

Exercise 4

Simulate a message queue using only semaphores and shared memory.

For simplicity, we consider the case of only two processes sending each other integer values as messages.

Exercise 4

- Our implementation has to respect the semantics of the message queue:
  - The queue has a finite size (N).
  - When sending on a full queue, the sender must be blocked.
  - When receiving from an empty queue, the receiver must be blocked.
  - There must effectively be a message passing (the reader must be able to receive and read what the writer sent, in order, without any message loss).

- But the authorized simplifications make the problem easier:
  - Only two processes → no need for message type.
  - Only integers → no need for character string trimming.

Exercise 4

Let's first start without any synchronization.

```
shared int queue[N];
```

```
int in = 0;
int readFromQueue()
{

    int rc = queue[in];
    in = (in+1)%N

    return rc;
}
```

```
int out = 0;
void postToQueue(int val)
{

    queue[out] = val;
    out = (out+1)%N

}
```

Exercise 4

The reader must be blocked if the queue is empty.

```
shared int queue[N];
shared semaphore empty = 0;
```

```
int in = 0;
int readFromQueue()
{
    wait(empty);
    int rc = queue[in];
    in = (in+1)%N

    return rc;
}
```

```
int out = 0;
void postToQueue(int val)
{

    queue[out] = val;
    out = (out+1)%N
    signal(empty);
}
```

Exercise 4

The writer must be blocked if the queue is full.

```
shared int queue[N];
shared semaphore empty = 0;
shared semaphore full = N;
```

```
int in = 0;
int readFromQueue()
{
    wait(empty);
    int rc = queue[in];
    in = (in+1)%N
    signal(full);
    return rc;
}
```

```
int out = 0;
void postToQueue(int val)
{
    wait(full);
    queue[out] = val;
    out = (out+1)%N
    signal(empty);
}
```