Computation structures

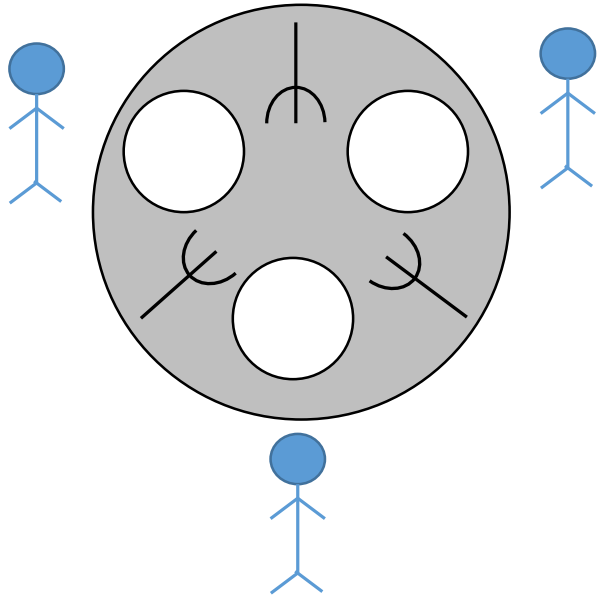
Support for problem-solving lesson #8

Exercise 1

Reminder

- Message queues are communication mechanisms that allow the passing of messages between processes
- Two operations can be performed on message queues : read (q?) and write (q!).
- q!x will send the message *x* into the queue *q* when there's room left.
- q?<x> will:
  - Read the message *x* (and remove it from the queue) until it can be found;
  - Read the oldest message (and remove it from the queue) if *x* is a variable.
- q!x and q?<x> are (generally) blocking and atomic operations.

Exercise 1



The dining philosopher's problem

Three philosophers are seated at a round table. A plate is placed in front of them and there are three forks on the table (one to the left and one to the right of each philosopher).

A philosopher can think (without any constraint) or eat (and requires two forks to do so). After eating, a philosopher will put back his forks on the table so that other philosophers (or possibly, the same one) can eat.

Use the C language to implement the code of the philosophers processes using only message queues to protect the resources (the forks).

Exercise 1

Things to pay attention to:

- A philosopher can only eat if he has two forks in hand.
- We must ensure that no deadlock (nor livelock) will ever happen.
  - Each philosopher first takes the fork to his left, then the fork to his right (whenever available).
  - Worst case : Each philosopher takes the fork to his left, nobody can eat and there's no available forks left (deadlock).
  - Each philosopher first takes the fork to his left (blocking mode), then tries to get the fork to his right (non-blocking mode) and, if that's not possible, it drops the left fork and tries again after some time.
  - Worst case : Each philosopher takes the fork to his left, notices that the right fork is not available, then drops the left fork and tries again indefinetely without succeeding (live lock).

Exercise 1

- Understanding live locks

Img src : www.aussiebrokers.com.au, thumb10.shutterstock.com, webitou.com

Exercise 1

- Understanding live locks

Img src : www.aussiebrokers.com.au, thumb10.shutterstock.com, webitou.com

Exercise 1

How to get out of a live-lock?

- Let the processes decide for themselves who should get the resource.
  - Talk to the lady in the supermarket and decide together who should go first.
  - (Network) Try again after a random period of time.

    (livelock still possible in theory, but works in practice).

- Decide, in advance, who should get the resource first in case of "conflict".
  - e.g. "Elders always go first".
  - In our case : Assign a number to each philosopher.

Exercise 1

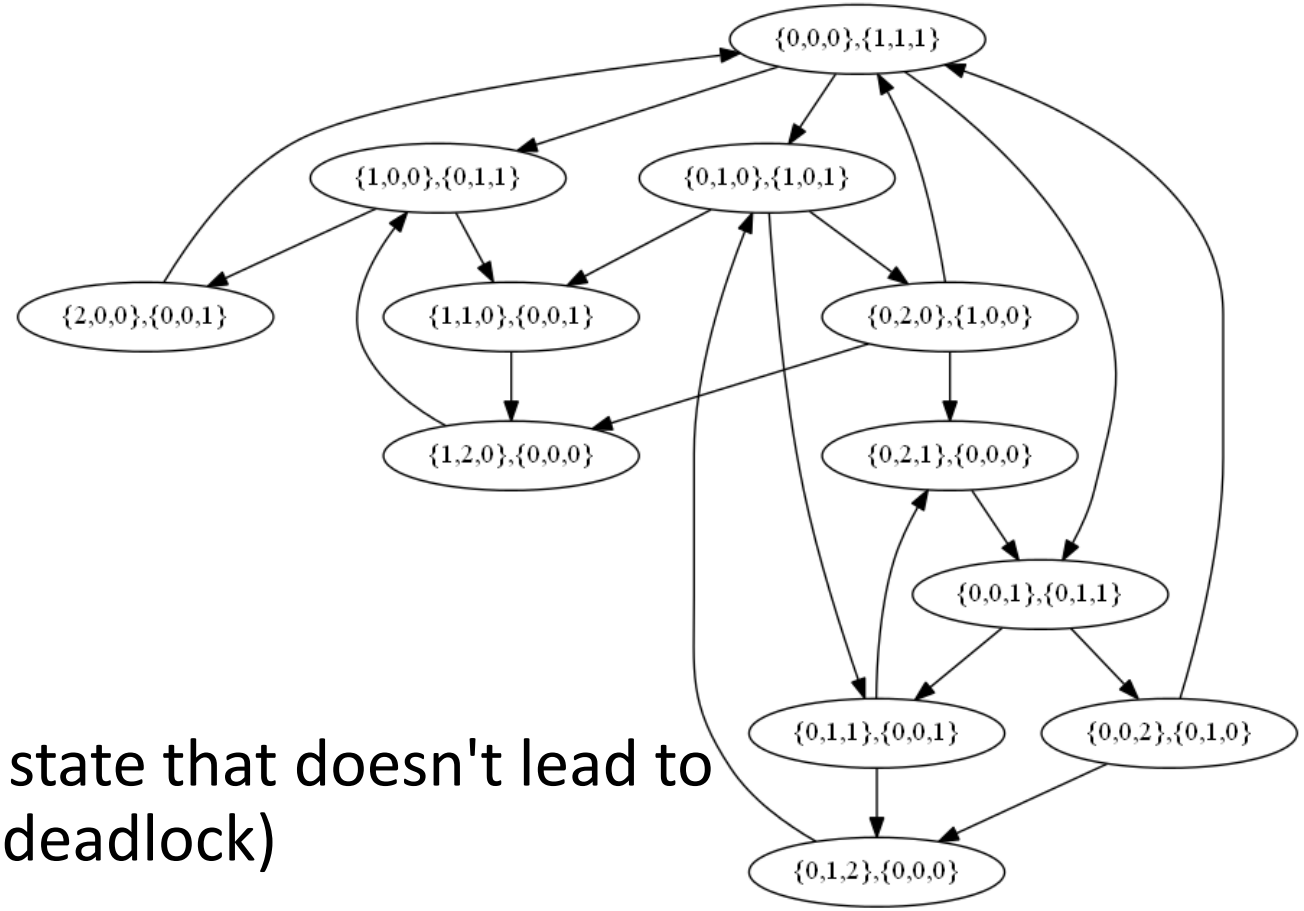
- What if I want to use IPCs in blocking mode only?
- Make the order for the processes implicit by imposing an order on the resources (semaphores) themselves.
- Always take the resource with the lowest number first.

Exercise 1

Things to pay attention to:

- A philosopher can only eat if he has two forks in hand.
- We must ensure that no deadlock (nor livelock) will ever happen.
  - Each philosopher first takes the fork to his left, then the fork to his right (whenever available).
  - Worst case : Each philosopher takes the fork to his left, nobody can eat and there's no available forks left (deadlock).
  - Each philosopher first takes the fork to his left (blocking mode), then tries to get the fork to his right (non-blocking mode) and, if that's not possible, it drops the left fork and tries again after some time.
  - Worst case : Each philosopher takes the fork to his left, notices that the right fork is not available, then drops the left fork and tries again indefinetely without succeeding (live lock).
- Solution : One of the philosophers takes the right fork and then the left fork.

Exercise 1



- You can't find any state that doesn't lead to another state (no deadlock)
- You can't find any loop where no philosophers eat (no livelock)
- In general, order your mutex resources and use that order in **wait()** operations

Exercise 1

Now that we solved the (dead or live) lock problem, the rest is straightforward.

```
shared chan q[3] = {1,2,3};
```

```
//The first two philosophers
int id; //equals 1 or 2
while(true) {
        q?id;
        q?(id+1);
        //Critical section
        q!id;
        q!(id+1);
        //Non critical section
}
```

```
//The last philosopher
while(true) {
        q?1;
        q?3;
        //Critical section
        q!3;
        q!1;
        //Non critical section
}
```

Exercise 1

To stop the program in a clean way, we can use a semaphore and an integer in shared memory.

```
shared semaphore NbStop = 0;
shared int finished = 0;
shared chan q[3] = {1,2,3};
```

```
//The first two philosophers
int id; //equals 1 or 2
while(finished == 0) {
        q?id;
        q?(id+1);
        //Critical section
        q!id;
        q!(id+1);
        //Non critical section
}
signal(NbStop);
```

```
//The control process
getchar();
finished = 1;
for(int i=0, i<3; i++) {
        wait(NbStop);
}
//Delete NbStop;
//Delete finished;
//Delete q;
```

```
//The last philosopher
while(finished == 0) {
        q?1;
        q?3;
        //Critical section
        q!3;
        q!1;
        //Non critical section
}
signal(NbStop);
```

Exercise 1 (solution)

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/msg.h>
#include <string.h>

#define MAX_SEND_SIZE 2

union semun {
      int val;                    /* value for SETVAL */
      struct semid_ds *buf;       /* buffer for IPC_STAT, IPC_SET */
      unsigned short int *array;  /* array for GETALL, SETALL */
      struct seminfo *__buf;      /* buffer for IPC_INFO */
};

struct mymsgbuf {
      long mtype;
      char mtext[MAX_SEND_SIZE];
};

void send_message(int qid, struct mymsgbuf *qbuf, long type, char
*text)
{
      /* Send a message to the queue */
      qbuf->mtype = type;
      strcpy(qbuf->mtext, text);

      if((msgsnd(qid, (struct msgbuf *)qbuf,   strlen(qbuf->mtext)+1, 0))
==-1) {
            perror("msgsnd");
            exit(1);
      }
}
```

```c
void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
      /* Read a message from the queue */
      qbuf->mtype = type;
      msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
}

void locksem(int sid, int member)
{
      struct sembuf sem_lock={ 0, -1, 0};
      if( member<0 || member>0) {
            fprintf(stderr, "semaphore member %d out of range\n", member);
            return;
      }
      sem_lock.sem_num = member;
      if((semop(sid, &sem_lock, 1)) == -1) {
            fprintf(stderr, "Wait failed\n");
            exit(1);
      }
}

void unlocksem(int sid, int member)
{
      struct sembuf sem_unlock={ member, 1, 0};
      int semval;
      if( member<0 || member>0) {
            fprintf(stderr, "semaphore member %d out of range\n", member);
            return;
      }
      sem_unlock.sem_num = member;
      /* Attempt to unlock the semaphore set */
      if((semop(sid, &sem_unlock, 1)) == -1) {
            fprintf(stderr, "Signal failed\n");
            exit(1);
      }
}
```

Exercise 1 (solution; cont'd)

```
writeshm(int* segptr, int index, int value)
{
	segptr[index] = value;
	printf("(Controler) Wrote %d\n", value);
	fflush(stdout);
}

int readshm(int* segptr, int id, int index)
{
	if(segptr[index] > 0)
		printf("(Philosopher %d) Read %d\n", (id+1),
segptr[index]);
	return segptr[index];
}

remove_sem(int semid)
{
	semctl(semid, 0, IPC_RMID, 0);
	printf("Semaphore set marked for deletion\n");
}

remove_shm(int shmid)
{
	shmctl(shmid, IPC_RMID, 0);
	printf("Shared memory segment marked for deletion\n");
}

void remove_queue(int qid)
{
	/* Remove the queue */
	msgctl(qid, IPC_RMID, 0);
	printf("Message queue marked for deletion\n");
}
```

```
philosopher(int msgqueue_id, int phil_id, int* segptr, int semid)
{
	struct mymsgbuf qbuf;

	while(readshm(segptr,phil_id,0) == 0) //While not stopped
	{

		read_message(msgqueue_id, &qbuf, (phil_id+1));
		printf("(Philosopher %d) Taking left fork\n",(phil_id+1));
		fflush(stdout);

		read_message(msgqueue_id, &qbuf, (phil_id+2));
		printf("(Philosopher %d) Taking right fork\n",(phil_id+1));

		printf("(Philosopher %d) Eating\n",(phil_id+1));
		fflush(stdout);

		send_message(msgqueue_id, (struct mymsgbuf *)&qbuf, (phil_id+1), "1");
		printf("(Philosopher %d) Dropping left fork\n",(phil_id+1));
		fflush(stdout);

		send_message(msgqueue_id, (struct mymsgbuf *)&qbuf, (phil_id+2), "1");
		printf("(Philosopher %d) Dropping right fork\n",(phil_id+1));
		printf("(Philosopher %d) Thinking\n",(phil_id+1));
		fflush(stdout);
	}

	unlocksem(semid,0);
}
```

Exercise 1 (solution; cont'd)

```
last_philosopher(int msgqueue_id, int*segptr, int semid)
{
        struct mymsgbuf qbuf;

        while(readshm(segptr,2,0) == 0) //While not stopped
        {

                read_message(msgqueue_id, &qbuf, 1);
                printf("(Philosopher %d) Taking right fork\n",3);
                fflush(stdout);

                read_message(msgqueue_id, &qbuf, 3);
                printf("(Philosopher %d) Taking left fork\n",3);

                printf("(Philosopher %d) Eating\n",3);
                fflush(stdout);

                send_message(msgqueue_id, (struct mymsgbuf *)&qbuf, 3, "3");
                printf("(Philosopher %d) Dropping left fork\n",3);
                fflush(stdout);

                send_message(msgqueue_id, (struct mymsgbuf *)&qbuf, 1, "1");
                printf("(Philosopher %d) Dropping right fork\n",3);
                printf("(Philosopher %d) Thinking\n",3);
                fflush(stdout);
        }

        unlocksem(semid,0);
}
```

```
controler(int msgqueue_id, int*segptr, int semid, int shmid)
{
        getchar();
        writeshm(segptr,0,1);
        locksem(semid,0);
        locksem(semid,0);
        locksem(semid,0);
        remove_queue(msgqueue_id);
        remove_shm(shmid);
        remove_sem(semid);
}

int main(int argc, char *argv[])
{
    key_t key_q, key_mem, key_sem;
    int   msgqueue_id;
    struct mymsgbuf qbuf;
    int id, cntr;
    pid_t pid;
    int   shmid, semid;
    int *segptr;
    union semun semopts;


    /* Create unique key via call to ftok() */
    key_q = ftok(".", 'q');
    key_mem = ftok(".", 'm');
    key_sem = ftok(".", 's');

    /* Open the queue - create if necessary */
    if((msgqueue_id = msgget(key_q, IPC_CREAT|0660)) == -1) {
        perror("msgget");
        exit(1);
    }
```

Exercise 1 (solution; cont'd)

```
//Filling the message queue
send_message(msgqueue_id, (struct mymsgbuf *)&qbuf, 1, "1");
send_message(msgqueue_id, (struct mymsgbuf *)&qbuf, 2, "2");
send_message(msgqueue_id, (struct mymsgbuf *)&qbuf, 3, "3");

/* Open the shared memory segment - create if necessary */
if((shmid = shmget(key_mem, sizeof(int), IPC_CREAT|IPC_EXCL|0666)) == -1)
{
      printf("Shared memory segment exists - opening as client\n");

      /* Segment probably already exists - try as a client */
      if((shmid = shmget(key_mem, sizeof(int), 0)) == -1)
      {
            perror("shmget");
            exit(1);
      }
}
else
{
      printf("Creating new shared memory segment\n");
}

/* Attach (map) the shared memory segment into the current process */
if((segptr = (int *)shmat(shmid, 0, 0)) == (int *)-1)
{
      perror("shmat");
      exit(1);
}

writeshm(segptr,0,0);

//Creating the semaphore array
printf("Attempting to create new semaphore set with 1 member\n");
if((semid = semget(key_sem, 1, IPC_CREAT|IPC_EXCL|0666)) == -1)   {
      fprintf(stderr, "Semaphore set already exists!\n");
      exit(1);
}

semopts.val = 0;
semctl(semid, 0, SETVAL, semopts);

//Creating the philosopher processes
id = 0;
for(cntr = 0; cntr < 3; cntr++)
{
    pid = fork();
    if(pid < 0) {
            perror("Process creation failed");
            exit(1);
    }
    if(pid == 0)  {
            //This is a son
            if(cntr < 2)
                  philosopher(msgqueue_id,id, segptr, semid);
            else
                  last_philosopher(msgqueue_id, segptr, semid);
            cntr = 3;
    }
    else  {
            //This is the father
            id++;
    }
}

    if(pid != 0)
            controler(msgqueue_id, segptr, semid, shmid);

    return(0);
}
```

Exercise 1 (execution)

```
ms805:~/cpp/test$ ./R5_ex3
Creating new shared memory segment
(Controler) Wrote 0
Attempting to create new semaphore set with 1 members
(Philosopher 1) Taking left fork
(Philosopher 1) Taking right fork
(Philosopher 1) Eating
(Philosopher 1) Dropping left fork
(Philosopher 1) Dropping right fork
(Philosopher 1) Thinking
(Philosopher 1) Taking left fork
(Philosopher 1) Taking right fork
(Philosopher 1) Eating
(Philosopher 1) Dropping left fork
(Philosopher 1) Dropping right fork
(Philosopher 1) Thinking
(Philosopher 2) Taking left fork
(Philosopher 2) Taking right fork
(Philosopher 2) Eating
(Philosopher 2) Dropping left fork
(Philosopher 2) Dropping right fork
(Philosopher 2) Thinking
(Philosopher 3) Taking right fork
(Philosopher 2) Taking left fork
(Philosopher 2) Taking right fork
(Philosopher 2) Eating
(Philosopher 2) Dropping left fork
(Philosopher 2) Dropping right fork
(Philosopher 3) Taking left fork
(Philosopher 3) Eating
(Philosopher 2) Thinking
(Philosopher 2) Taking left fork
(Philosopher 3) Dropping left fork
(Philosopher 3) Dropping right fork
(Philosopher 2) Taking right fork
(Philosopher 1) Taking left fork
(Philosopher 2) Eating
(Philosopher 3) Thinking
(Philosopher 2) Dropping left fork
(Philosopher 1) Taking right fork
(Philosopher 1) Eating
(Philosopher 1) Dropping left fork
```

```
(Philosopher 1) Thinking
(Philosopher 3) Taking right fork
(Philosopher 2) Dropping right fork
(Philosopher 3) Taking left fork
(Philosopher 3) Eating
(Philosopher 2) Thinking
(Philosopher 3) Dropping left fork
(Philosopher 2) Taking left fork
(Philosopher 2) Taking right fork
(Philosopher 3) Dropping right fork
(Philosopher 1) Taking left fork
(Philosopher 2) Eating
(Philosopher 3) Thinking
(Philosopher 2) Dropping left fork
(Philosopher 1) Taking right fork
(Philosopher 1) Eating
(Philosopher 1) Dropping left fork
(Philosopher 1) Dropping right fork
(Philosopher 1) Thinking

(Philosopher 3) Taking right fork
(Philosopher 2) Dropping right fork
(Controler) Wrote 1
(Philosopher 3) Taking left fork
(Philosopher 3) Eating
(Philosopher 2) Thinking
(Philosopher 2) Read 1
(Philosopher 3) Dropping left fork
(Philosopher 3) Dropping right fork
(Philosopher 1) Taking left fork
(Philosopher 1) Taking right fork
(Philosopher 1) Eating
(Philosopher 1) Dropping left fork
(Philosopher 1) Dropping right fork
(Philosopher 1) Thinking
(Philosopher 1) Read 1
(Philosopher 3) Thinking
(Philosopher 3) Read 1
Message queue marked for deletion
Shared memory segment marked for deletion
Semaphore set marked for deletion
ms805:~/cpp/test$
```