Exercise sheet 10 – Deadlock analysis

Goals:

- Deadlocks

Exercise 10.1: Deadlocks 1

The three processes `P1`, `P2`, and `P3` are executing the following code:

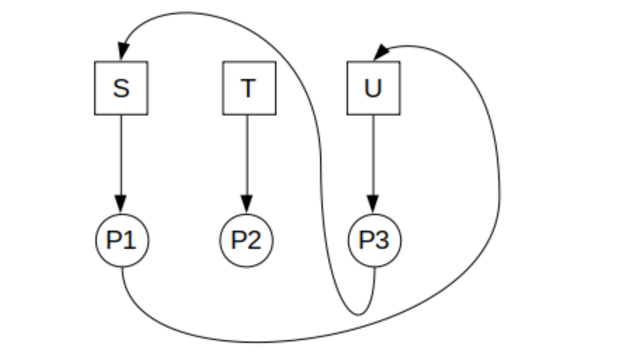
P1	P2	P3

P(S)
P(U) <=
work_with_s_and_u();
V(S)
V(U)
	P(T)
	. <=
	work_with_t();
	V(T)
		P(U)
		P(S) <=
		work_with_s_and_u();
		V(U)
		V(S)

All semaphores start with the value 1; the arrow shows the code which is executed at the moment.

(a) Draw a system resource acquisition graph for this situation!

Proposal for solution:



(b) Show that a deadlock exists.

Proposal for solution: There is a cycle in the system resource acquisition graph! => Deadlock!

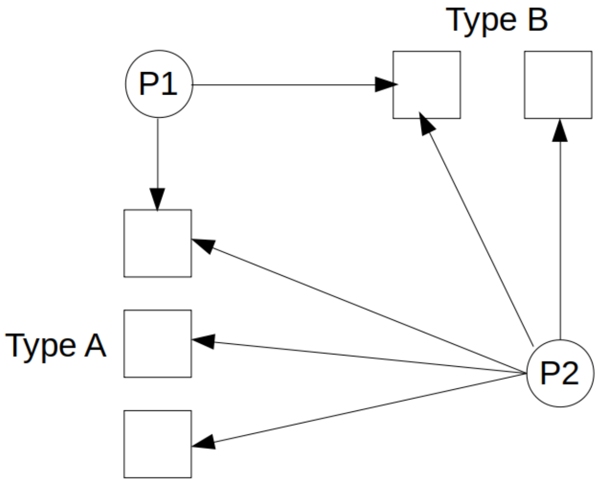
(c) Show two possibilities to avoid the deadlock!

Proposal for solution:

- Try to avoid the circular wait: Switch `P(U)` and `P(S)` in `P1` or `P3`. Please consider also the `V(S)` and `V(U)` operations. Release the semaphores from the inside to the outside. Preferred solution here.
- Try to avoid the non-preemption: If `P1` or `P3` can't acquire `P(U)` or `P(S)` it releases all resources and tries it after some time again. But this can mean that work already done is lost and must be repeated.

Exercise 10.2: Deadlocks 2

Look at this system resource acquisition graph:



(a) Is there a deadlock?

Proposal for solution: No, because there is no cycle in the graph.

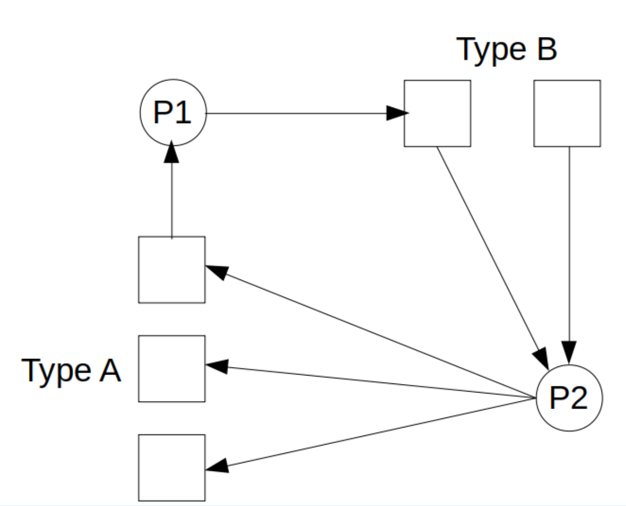
(b) Is the state safe?

Proposal for solution:

- Step 1: Give `P1` the requested resources (1x`A`, 1x`B`).
- Step 2: After `P1` has finished and its resources released, give all resources to `P2`.
- => safe sequence found
- => because there is no deadlock and we have a safe sequence => the state is safe!

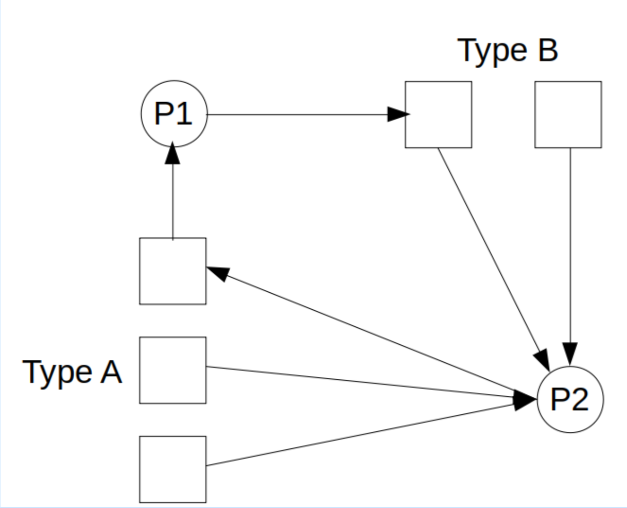
(c) Find a sequence of operations which would cause a deadlock!

Proposal for solution: When `P1` gets one type A resource and `P2` gets the two type B resources, there is a deadlock.



(d) Is it allowed to fulfil the request of `P2` for the two resources of type B?

Proposal for solution: If P2 also gets the three resources of type A, then it can do its job and will end at some time. So the state would be safe. But if P1 would get a resource of type A, then a deadlock would be created.



Exercise 10.3: Deadlocks behaviour

(a) What happens on a deadlock on a desktop system?

Proposal for solution: TODO...

(b) What happens on a deadlock on a server system?

Proposal for solution: TODO...

(c) What happens on a deadlock on a smartphone?

Proposal for solution: TODO...

(d) What happens on a deadlock on a safety critical realtime system (e.g. in a car)?

Proposal for solution: TODO...

Exercise 10.4: Deadlocks analysis on existing C code

(a) Update the `OS_exercises` repository with `git pull`.

Proposal for solution: `git pull`

(b) Change into the `OS_exercises/sheet_10_deadlocks/deadlock_code_analysis` directory.

Proposal for solution:

```
cd sheet_10_deadlocks/deadlock_code_analysis
```

(c) Inspect the `deadlock_analysis.c`.

(d) Build and run the program.

Proposal for solution:

```
make
./deadlock_analysis
```

(e) Does the program work correctly? Is there an error?

Proposal for solution: The program starts, but it seems to block. There might be a deadlock.

(f) Try to analyse the behaviour.

Proposal for solution: The problem is the order of the P/V operations on the semaphores: They are not in the same order.

(g) Fix the bug.

Proposal for solution:

```c
#include <stdio.h>     //printf, perror
#include <stdlib.h>    //EXIT_FAILURE, EXIT_SUCCESS
#include <string.h>    //sprintf
#include <unistd.h>    //open, close, read, write
#include <pthread.h>   //pthread_*
#include <fcntl.h>     //flags: O_CREAT, O_EXCL
#include <semaphore.h> //sem_open, sem_wait, sem_post, sem_close
#include <errno.h>     //errno

#define SEMAPHORE1_NAME "/sem1"          //name of semaphore
#define SEMAPHORE2_NAME "/sem2"          //name of semaphore
sem_t*   semaphore1 = NULL;             //pointer to semaphore
sem_t*   semaphore2 = NULL;             //pointer to semaphore
const int     PERM  = 0600;              //permission to the semaphore (read + write)

void create_semaphore() {
    semaphore1 = sem_open(SEMAPHORE1_NAME, O_CREAT, PERM, 1);
    if(semaphore1 == SEM_FAILED) {
        perror("Error when creating the semaphore ...\n");
        exit(EXIT_FAILURE);
    }
    semaphore2 = sem_open(SEMAPHORE2_NAME, O_CREAT, PERM, 1);
    if(semaphore1 == SEM_FAILED) {
        perror("Error when creating the semaphore ...\n");
        exit(EXIT_FAILURE);
    }
}

void close_semaphore() {
    if(sem_close(semaphore1) == -1) {
        perror("Error can't close semaphore ...\n");
        exit(EXIT_FAILURE);
    }
    if(sem_close(semaphore2) == -1) {
        perror("Error can't close semaphore ...\n");
        exit(EXIT_FAILURE);
    }
}

void delete_semaphore() {
    if(sem_unlink(SEMAPHORE1_NAME) == -1) {
        switch(errno)
        {
        case EACCES:       //fall through
        case ENAMETOOLONG:
            perror("Error can't delete (unlink) semaphore ...\n");
            exit(EXIT_FAILURE);
            break;
        case ENOENT: //semaphore already deleted, no error should be printed!
            break;
        }
    }
    if(sem_unlink(SEMAPHORE2_NAME) == -1) {
        switch(errno)
        {
        case EACCES:       //fall through
        case ENAMETOOLONG:
            perror("Error can't delete (unlink) semaphore ...\n");
            exit(EXIT_FAILURE);
            break;
        case ENOENT: //semaphore already deleted, no error should be printed!
            break;
        }
    }
}

void* worker1() {
    printf("w1 started\n");

    for(int i = 0; i < 5; ++i){
        sem_wait(semaphore1); usleep(1);
        sem_wait(semaphore2);
            printf("w1 in critical area: working...\n");
            sleep(1);
        sem_post(semaphore2);
        sem_post(semaphore1);
    }

    printf("w1 ends\n");
    return NULL;
}

void* worker2() {
    printf("w2 started\n");

    for(int i = 0; i < 5; ++i){
        sem_wait(semaphore1); usleep(1);
        sem_wait(semaphore2);
            printf("w2 in critical area: working...\n");
            sleep(1);
        sem_post(semaphore2);
        sem_post(semaphore1);
    }

    printf("w2 ends\n");
    return NULL;
}

int main(int argc, char** argv){
    //create semaphores
    delete_semaphore();
    create_semaphore();

    //start worker thread
    pthread_t thread_w1;
    pthread_t thread_w2;
    pthread_t thread_w3;

    int thread_create_state = -1;
    thread_create_state = pthread_create(&thread_w1, NULL, &worker1, NULL);
    if(thread_create_state != 0) {
        printf("Failed creating thread\n");
        exit(EXIT_FAILURE);
    }
    thread_create_state = pthread_create(&thread_w2, NULL, &worker2, NULL);
    if(thread_create_state != 0) {
        printf("Failed creating thread\n");
        exit(EXIT_FAILURE);
    }


    //Wait for the termination of all threads
    pthread_join(thread_w1, NULL);
    pthread_join(thread_w2, NULL);

    //close & delete semaphores
    close_semaphore();
    delete_semaphore();

    return EXIT_SUCCESS;
}
```

(h) Build and run the program.

Proposal for solution:

```
make
./deadlock_analysis
```

Now it should work as expected.