Operating System
Blen Daniel Assefa

OS 2021 Problem Sheet #5

Problem 5.1: Bank transfers

```
#include <pthread.h>

typedef struct account {
      unsigned int number;
      unsigned int money;
      pthread_mutex_t lock;
} account_t;

void transfer(unsigned int money, account_t *from, account_t *to)
{
      pthread_mutex_lock(&from->lock);
      pthread_mutex_lock(&to->lock);
      from->money -= money;
      to->money += money;
      pthread_mutex_unlock(&to->lock);
      pthread_mutex_unlock(&from->lock);
}
```

a. For a deadlock situation to occur there needs to be at least two processes running simultaneously and it could arise if and only if all of the following conditions hold:

1. Mutual exclusion
2. Hold and Wait
3. No preemption
4. Circular wait

For the given code:

Mutual Exclusion: Well, if someone is trying to transfer while another person tries to transfer to the person who is transferring, then both of the transfer functions are actually trying to read the same account and manipulate the values of that account. The obvious resource that is going to be used simultaneously is the mutex lock.

Hold and Wait, No preemption and Circular wait: happens if the two accounts are called on from each other, and when the first person to transfer tries to transfer again to the same person.

i.e. if **`account_t *from`** transfers to **`account_t *to`** and if **`account_t *from`** tries transfers to **`account_t *to`** but in the middle, if **`account_t *from`** tries transfers to **`account_t *to`** again

The deadlock situation happens when transfer() is initially called and the **reverse** transfer (between the same users but in the opposite direction) is called but is waiting in line until the first one finishes. Then when the first transfer is done, and when the first **`account_t *to`** mutex is unlocked, the other waiting transfer gets the signal and locks their account (**`account_t`**

`*from)` . Then if there is a situation where possibly, as soon as it the first transfer() is done, there happens to be a second same call, and the **`account_t *from`** is locked again to do the same transfer, then it locks the first account, but it can't actually lock the second account because the second (**`account_t *to`**) is already locked to be used by another transfer call that is waiting for the first account (**`account_t *from`**) to be unlocked. Then there is a deadlock situation.

b.

```
void transfer(unsigned int money, account_t *from, account_t
*to)
{

      pthread_mutex_lock(&from->lock);
      from->money -= money;
      pthread_mutex_unlock(&from->lock);

      pthread_mutex_lock(&to->lock);
      to->money += money;
      pthread_mutex_unlock(&to->lock);
}
```

By making these changes, it guarantees that there won't be a deadlock situation because all the four circumstances won't happen if the account access locks before opening another account.

Problem 5.2: Safe States

$$M = \begin{bmatrix} 2 & 5 & 3 & 3 & 2 \\ 3 & 5 & 8 & 10 & 1 \\ 4 & 12 & 4 & 9 & 2 \\ 6 & 1 & 4 & 5 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \qquad A = \begin{bmatrix} 0 & 5 & 3 & 1 & 1 \\ 0 & 2 & 1 & 1 & 1 \\ 0 & 7 & 1 & 2 & 1 \\ 3 & 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$$

$$N = M - A = \begin{bmatrix} 2 & 0 & 0 & 2 & 1 \\ 3 & 3 & 7 & 9 & 0 \\ 4 & 5 & 3 & 7 & 1 \\ 3 & 0 & 3 & 4 & 5 \\ 0 & 0 & 0 & 2 & 4 \end{bmatrix}$$

$n = 5\ (processes)$
$m = 5\ (resource\ types)$
$t = (6,\ \ 17,\ \ 9,\ \ 10,\ \ 7)$

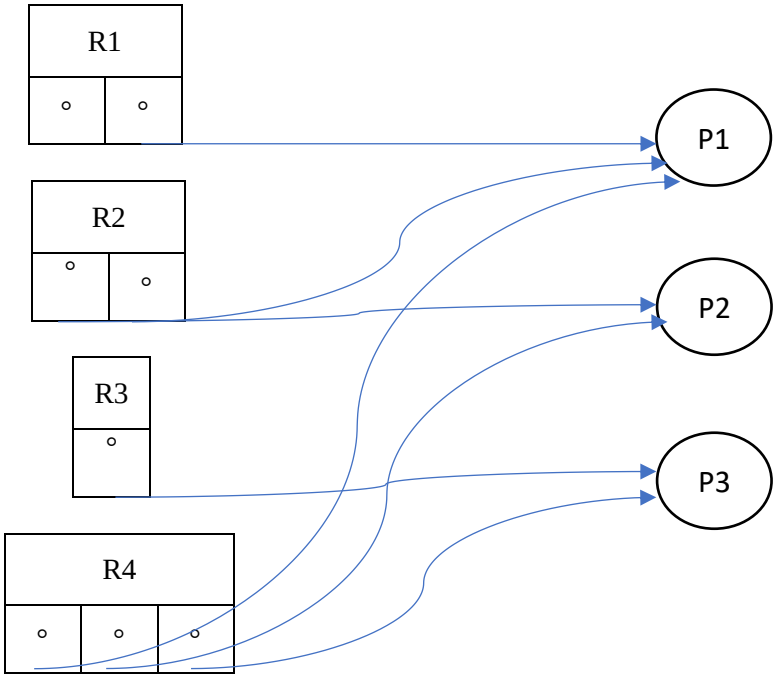
$$The\ current\ set\ of\ available\ resources\ are\colon\ a = (2,\ \ 0,\ \ 0,\ \ 3,\ \ 3)$$

$a = (2, \quad 0, \quad 0, \quad 3, \quad 3)$ $\quad R = \{1\} - termination\ of\ process\ 1$
$a = (2, \quad 5, \quad 3, \quad 4, \quad 4)$ $\quad R = \{5\} - termination\ of\ process\ 5$
$a = (3, \quad 7, \quad 6, \quad 6, \quad 5)$ $\quad R = \{4\} - termination\ of\ process\ 4$
$a = (6, \quad 8, \quad 7, \quad 7, \quad 5)$ $\quad R = \{3\} - termination\ of\ process\ 3$
$a = (6, \quad 15, \quad 8, \quad 9, \quad 6)$ $\quad R = \{2\} - termination\ of\ process\ 2$
$a = (6, \quad 17, \quad 9, \quad 10, \quad 7)$ $\quad R = \{\} - stop$

Since there is a sequence that allows all processes to receive their still needed resources, the new state is safe and the resource request can be granted.

Problem 5.3: deadlock detection

a) The resource allocation graph looks like this:



b)

$$N = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$n = 3\ (processes)$
$m = 4\ (resource\ types)$
$t = (2, \quad 2, \quad 1, \quad 3)$

$$The\ current\ set\ of\ available\ resources\ are: \quad a = (1, \quad 0, \quad 0, \quad 0)$$

$a = (1, \quad 0, \quad 0, \quad 0)$ $\quad R = \{2\} - termination\ of\ process\ 2$
$a = (1, \quad 1, \quad 0, \quad 1)$ $\quad R = \{3\} - termination\ of\ process\ 3$
$a = (1, \quad 1, \quad 1, \quad 2)$ $\quad R = \{1\} - termination\ of\ process\ 1$

$a = (2, \quad 2, \quad 1, \quad 3) \qquad\qquad R = \{\} - stop$

Since there is a sequence that allows all processes to receive their still needed resources, the new state is safe and the resource request can be granted.