Homework 5

Sherry Usman

October 13, 2022

0.1 bank transfers

a. An example of a situation where a deadlock might occur is when a person A is trying to transfer money from account A to account B while in parallel (simultaneously) person B is trying transfer money from account B to account A.
Person A will acquire locks in the order *from* and *to* and will release locks in the order *to* and *from*.
Person B will acquire locks on *to* and *from* and will release them in the order *from* and *to*.
Thus if Person A acquires lock *from* and person B acquires the lock *to* both will be deadlocked as Person A will no longer be able to acquire lock *to* and person B will no longer be able to acquire the lock *from*. Thus each person/thread will be waiting for the other person/thread to execute and release a resource. Thus both persons will be deadlocked.
b. The solution to this problem would be the following:

```
#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)
{
    if (from->number == to->number){
        throw exception "cannot transfer money from and to same account"
    }
    else if (from->number < to->number){
        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);
    }
    else {
        pthread_mutex_lock(&to->lock);
        pthread_mutex_lock(&from->lock);
        from->money -= money;
        to->money += money;
        pthread_mutex_unlock(&from->lock);
        pthread_mutex_unlock(&to->lock);

    }

}
```

If we check the ids of the accounts, we will be able to make sure that no thread deadlocks another.

0.2 safe states

To check whether the current state is a safe state we first need to calculate the resources that are still available.

alloc =

$$\begin{bmatrix} 0 & 5 & 3 & 1 & 1 \\ 0 & 2 & 1 & 1 & 1 \\ 0 & 7 & 1 & 2 & 1 \\ 3 & 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$$

avail = total - colsum(alloc) = $\begin{pmatrix} 6 & 17 & 9 & 10 & 7 \end{pmatrix}$ - $\begin{pmatrix} 4 & 17 & 9 & 7 & 4 \end{pmatrix}$ = $\begin{pmatrix} 2 & 0 & 0 & 3 & 3 \end{pmatrix}$

We now calculate the remaining maximum need N for all processes. $N = M - A$.

$$\text{N=} \begin{bmatrix} 2 & 5 & 3 & 3 & 2 \\ 3 & 5 & 8 & 10 & 1 \\ 4 & 12 & 4 & 9 & 2 \\ 6 & 1 & 4 & 5 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} - \begin{bmatrix} 0 & 5 & 3 & 1 & 1 \\ 0 & 2 & 1 & 1 & 1 \\ 0 & 7 & 1 & 2 & 1 \\ 3 & 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 2 & 1 \\ 3 & 3 & 7 & 9 & 0 \\ 4 & 5 & 3 & 7 & 1 \\ 3 & 0 & 3 & 4 & 5 \\ 0 & 0 & 0 & 2 & 4 \end{bmatrix}$$

Given the available resources, process 1 can get all of its resources.

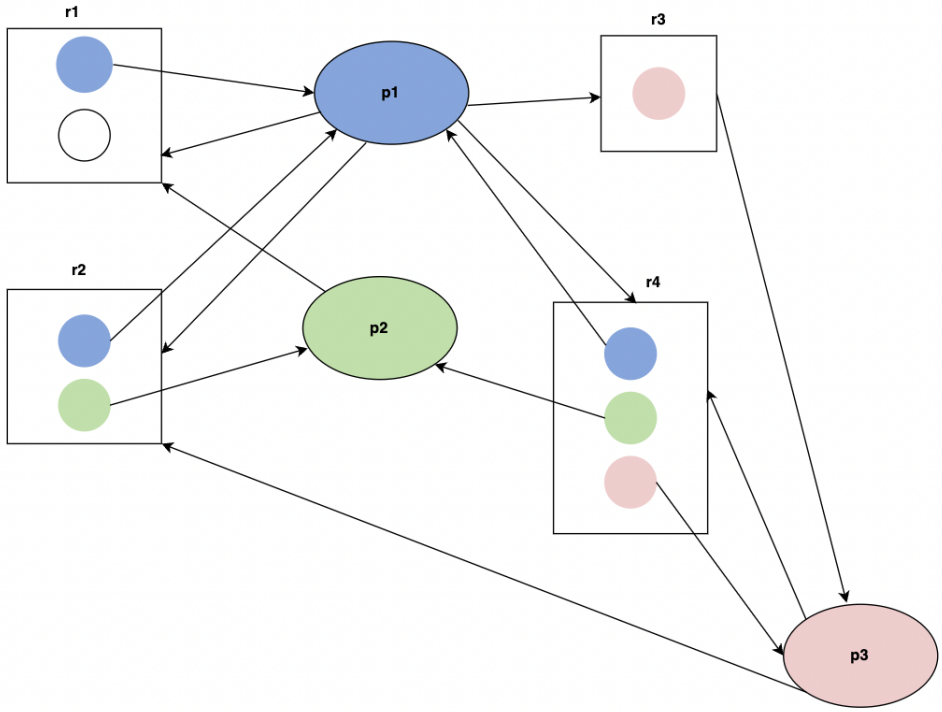
a = $\begin{pmatrix} 2 & 0 & 0 & 3 & 7 \end{pmatrix}$	R=1	termination of process 1
a = $\begin{pmatrix} 2 & 5 & 3 & 4 & 4 \end{pmatrix}$	R=5	termination of process 5
a = $\begin{pmatrix} 3 & 7 & 6 & 6 & 5 \end{pmatrix}$	R=4	termination of process 4
a = $\begin{pmatrix} 6 & 8 & 7 & 7 & 5 \end{pmatrix}$	R=3	termination of process 4
a = $\begin{pmatrix} 6 & 15 & 8 & 9 & 6 \end{pmatrix}$	R=2	termination of process 2
a = $\begin{pmatrix} 6 & 17 & 9 & 10 & 7 \end{pmatrix}$	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.The system is stable since the resources available after all processes are executed is the same as the total number of resources.

0.3 deadlock detection

a. the diagram below shows the RAG (resource allocation graph) for the corresponding A and M matrices.

Figure 1: Raw



b. The following system is under deadlock. P1 is requesting a total of four resources of type r1, r2, r3 and r4. Starting off, p1 is requesting a resource of type r1 and this is not a problem since one instance of r4 is free. Thus, it is allocated to P1.
P1 is also requesting a resource of type r2, r3 and r4. All instances of r2 are already occupied by p1 and p2. Thus p1 is waiting on p2 to release an instance of resource r2. however, p2 is requesting a resource of type r1. IF p1 allows p2 to have an instance of r1 then p2 can finish and release instances of resource type r1, r2 and r4. p1 will then have an instance of r1 as requested.p1 will also be able to have an instance of r4 just released by p2. However, p1 will still need an instance of resource type r3 which is occupied by p3. p3 in turn is requesting a resource of type r2 and r4. Both are occupied by p1.
Thus p3 is waiting for p1 to release instance an instance of r4. However, p1 cannot terminate since it is waiting for p3 to release an instance of r3. This situation is described by the graph below.

Figure 2: RAG if process p2 acquires r1 and terminates.