Operating System Project 5

Banker's Algorithm

Overview

You will implement Banker's Algorithm by writing a **c program** with three main topics: multithreading, preventing race conditions, and deadlock avoidance.

Multithreading

A multithread example for creating 3 threads.

If you need to pass customer number to threadRunner, you may let some argument be the fourth parameter of pthread_create;

```c
#include <pthread.h>

void *threadRunner(void *param)
{
    /*
    do what you should do in current thread
    */
    pthread_exit(0); // terminate current thread
}

int main()
{
    pthread_t tid[3];
    int i;
    for(i = 0; i < 3; i++) {
        pthread_create(&tid[i], NULL, threadRunner, NULL);
    }

    for(i = 0; i < 3; i++) {
        pthread_join(tid[i], NULL);
    }
}
```

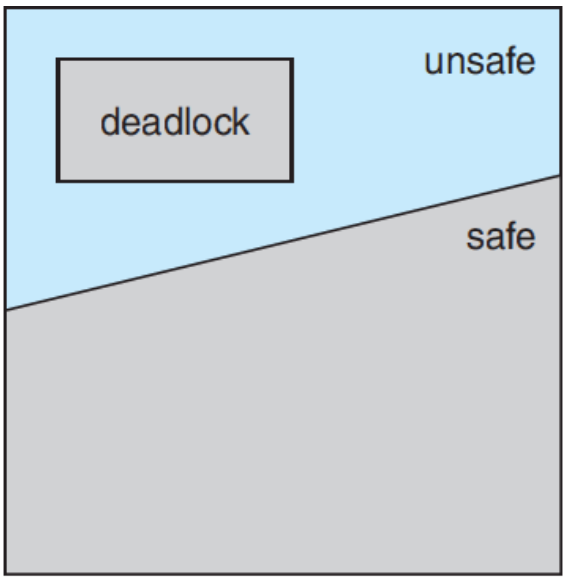
Preventing race conditions

Use pthread's mutex to access shared data.

```c
#include <pthread.h>

pthread_mutex_t mutex;

void *threadRunner(void *param)
{
    pthread_mutex_lock(&mutex);
    /*
    access shared data
    */
    pthread_mutex_unlock(&mutex);
    pthread_exit(0); // terminate current thread
}

int main()
{
    pthread_t tid;
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&tid[i], NULL, threadRunner, NULL);
    pthread_join(tid[i], NULL);
}
```

Deadlock avoidance

Find a sequence of processes <P1, P2, …, Pn> that lets the system be in **safe state** if they are allocated resources in this order. This sequence is called safe sequence.



State spaces

Project details

The Banker

In this project, the banker will consider requests from 5 customers for 3 resources types. The banker will keep track of the resources using the following data structures:

```
#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 3

// the available amount of each resource
int available[NUMBER_OF_RESOURCES];

// the maximum demand of each customer
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

// the amount currently allocated to each customer
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

// the remaining need of each customer
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

// the finishing count of the system,
// if a customer acquire all needs, finishCount++
int finishCount = 0;
```

The Customers

Create 5 customer threads that request and release resources from the bank.
The customer will **continually loop, requesting and then releasing random numbers of resources** (you should use **srand(unsigned int seed)** with **(unsigned)time(NULL)** as seed, and a combination with **rand()** to create a random number in **[0, maximum[i][j]]**, where i is customer number and j is resource number). **After one run of requesting or releasing**, you should call **sched_yield()** to let current thread give up the CPU to let other threads have more chances to get the mutex lock.

You should use the following two functions to request and release resources, and each call of them should be protected by a mutex:

int requestResources(int customerNum, int request[])

```
/*
return code
3 succeeds & all customers finish,
2 succeeds & current customer finishes,
1 succeeds,
0 no request,
-1 fails since request exceeds need
-2 fails since request exceeds available,
-3 fails since since the state is unsafe
*/
int requestResources(int customerNum, int request[])
{
    // check request exceed need

    // check request exceed available

    // pretend to allocate resources

    // check finishAll, finish, no request with precedence finishAll > finish > no request
    // you should release resources you are allocated here if finishAll or finish occur

    // check safety, if not safe, restore the resources which is allocated when pretending to allocate resources
}
```

int releaseResources(int customerNum, int release[])

```
/*
return code
1 succeeds
0 no release,
-1 fails since release exceeds allocation
*/
int releaseResources(int customerNum, int release[])
{
    //check release exceed allocation

    //check no release

    //release
}
```

You should do some things after getting the return code

Code of request: 3

1. Print the information like

```
Request 2 0 0
Request Code 3: customer 3's request succeeds & all customers finish
```

2. Print the state of system like

```
current state

available
resource     10   5    7

             maximum      allocation   need
customer 0   7    4    0    0    0    0    0    0    0
customer 1   4    2    3    0    0    0    0    0    0
customer 2   8    1    7    0    0    0    0    0    0
customer 3   4    0    0    0    0    0    0    0    0
customer 4   4    5    2    0    0    0    0    0    0
```

3. Unlock the mutex and then exit the current thread

Code of request: 2

1. Print the information like

```
Request 1 0 1
Request Code 2: customer 4's request succeeds & finishes
```

2. Print the state of system like

```
current state

available
resource     9    4    5

             maximum      allocation   need
customer 0   6    3    1    0    0    0    6    3    1
customer 1   5    5    2    1    0    0    4    5    2
customer 2   8    1    4    0    1    0    8    0    4
customer 3   0    2    2    0    0    2    0    2    0
customer 4   2    1    2    0    0    0    0    0    0
```

3. Unlock the mutex and then exit the current thread

Code of request: 1

1. Print the information like

```
Request 0 2 1
Request Code 1: customer 3's request succeeds
```

2. Print the state of system like

```
current state

available
resource     4    0    6

             maximum       allocation   need
customer 0   7    3    3   6    1    0   1    2    3
customer 1   7    3    0   0    0    0   0    0    0
customer 2   5    3    1   0    0    0   0    0    0
customer 3   0    4    3   0    2    1   0    2    2
customer 4   0    2    6   0    2    0   0    0    6
```

3. Unlock the mutex

Code of request: 0

1. Print the information like

```
Request 0 0 0
Request Code 0: customer 2 doesn't request
```

2. Print the state of system like

```
current state

available
resource     7    4    7

             maximum       allocation   need
customer 0   2    3    3   0    0    0   2    3    3
customer 1   8    1    0   2    0    0   6    1    0
customer 2   1    1    1   1    1    0   0    0    1
customer 3   0    4    2   0    0    0   0    4    2
customer 4   0    5    7   0    0    0   0    5    7
```

3. Unlock the mutex

Code of request: -1

1. Print the information like

```
Request 6 1 0
Request Code -1: customer 4's request fails since request exceeds need
```

2. Print the state of system like

```
current state

available
resource     7    3    2

                maximum       allocation   need
customer 0   1    5    6    0    0    0    0    0    0
customer 1   2    1    2    0    0    0    0    0    0
customer 2   5    5    3    0    0    0    0    0    0
customer 3   4    3    4    0    0    0    0    0    0
customer 4   8    2    7    3    2    5    5    0    2
```

3. Unlock the mutex

Code of request: -2

1. Print the information like

```
Request 0 5 0
Request Code -2: customer 2's request fails since request exceeds available
```

2. Print the state of system like

```
current state

available
resource     6    4    3

                maximum       allocation   need
customer 0   10   0    0    2    0    0    8    0    0
customer 1   0    3    5    0    0    0    0    0    0
customer 2   0    5    5    0    0    2    0    5    3
customer 3   5    1    0    0    0    0    0    0    0
customer 4   8    3    3    2    1    2    6    2    1
```

3. Unlock the mutex

Code of request: -3

1. Print the information like

```
Request 8 0 3
Request Code -3: customer 1's request fails since the state is unsafe
```

2. Print the state of system like

```
current state

available
resource     8    4    3

             maximum       allocation   need
customer 0   5    4    5   0    0    0   0    0    0
customer 1   9    0    7   0    0    1   9    0    6
customer 2   6    2    3   0    0    0   0    0    0
customer 3   10   1    3   2    1    3   8    0    0
customer 4   9    0    5   0    0    0   0    0    0
```

3. Unlock the mutex

Code of release: 1

1. Print the information like

```
Release 0 0 1
Release Code 1: customer 3's release succeeds
```

2. Print the information like

```
current state

available
resource     9    4    7

             maximum       allocation   need
customer 0   7    3    4   0    0    0   7    3    4
customer 1   7    1    6   0    0    0   7    1    6
customer 2   4    0    0   0    0    0   4    0    0
customer 3   1    1    5   1    1    0   0    0    5
customer 4   0    2    0   0    0    0   0    0    0
```

3. Unlock the mutex

Code of release: 0

1. Print the information like

```
Release 0 0 0
Release Code 0: customer 2 doesn't release any resource
```

2. Print the state of system like

```
current state

available
resource     4    4    5

                maximum       allocation   need
customer 0   4    0    4     0    0    0    0    0    0
customer 1   5    1    3     0    0    0    0    0    0
customer 2   5    5    2     5    1    2    0    4    0
customer 3   1    1    0     1    0    0    0    1    0
customer 4   7    2    0     0    0    0    0    0    0
```

3. Unlock the mutex

Code of release: -1

1. Print the information like

```
Release 5 5 7
Release Code -1: customer 1's release fails since release exceeds allocation
```

2. Print the state of system like

```
current state

available
resource     10   2    5

                maximum       allocation   need
customer 0   3    1    2     0    0    0    0    0    0
customer 1   3    5    3     0    3    2    3    2    1
customer 2   4    0    0     0    0    0    0    0    0
customer 3   1    1    5     0    0    0    0    0    0
customer 4   0    2    0     0    0    0    0    0    0
```

3. Unlock the mutex

Algorithm

Define

Let X and Y be arrays of length n.

X <= Y if and only if X[i] <= Y[i] for all i = 0, 1, …, n-1.

X < Y(Y exceeds X) if X <= Y and X isn't equal to Y

Let m is the number of resources types, n is the number of customers

Safety algorithm (Find out whether or not the system is in a safe state)

1. Let **work** and **finish** be arrays of length m and n, respectively. Initialize **work** = **available** and **finish[i] = false** for i = 0, 1, …, n-1.
2. Find an index i such that both **finish[i] = false** and **need**$_i$ <= **work**. If no such i exists, go to step 4.
3. **work = work + allocation**$_i$, **finish[i] = true**, go to step 2.
4. If **finish[i] == true** for all i, then the system is in a safe state.

Resource Request Algorithm

Let **request**$_i$ be the request array for thread Ti (or for customer i), If **request**$_i$**[j] = k**, then thread Ti wants k instances of resource type R_j.

1. If **request**$_i$ **<= need**$_i$, go to step 2. Otherwise, return corresponding request code.
2. If **request**$_i$ **<= available**, go to step 3. Otherwise, return corresponding request code.
3. Have the system pretend to have allocated the requested resources to thread Ti by modifying the state as follows:

   available = available – request$_i$

   allocation$_i$ **= allocation**$_i$ **+ request**$_i$

   need$_i$ **= need**$_i$ **- request**$_i$

   go to step 4

4. If Ti doesn't need any resources after step3, release the allocated resources to simulate the customer i has finished its maximum demand; then, if all customers finish, return corresponding request code; otherwise, return corresponding request code since the current customer finishes.

   Otherwise, if no request(it means that request$_i$[j] are all 0s for j = 0, 1, …, m), return corresponding request code; otherwise Ti has request, go to step 5.

5. If the system is in safe state, return corresponding request code. Otherwise, restore the original state and then return corresponding request code.

Implementation

Compile

```
gcc banker.c -o banker -lpthread
```

Execute

You should invoke your program by passing the number of resources of each type on the command line.

In this project, you should pass three positive numbers since we have three types of resources. The **available** array will be initialized to these values, the **maximum** array must be initialized to a random number in the range [0, available] according to the corresponding resource, and the **allocation** array must be initialized to 0s (the **need** array is always maximum - allocation). We may not just try **10 5 7**, other test numbers like 15 17 12 may be tested but not limited; however, we don't try number bigger than 20 because it will take too much time.

```
./banker 10 5 7
```

Print Example(this is a part of the whole print out result)

```
Request 1 1 0
Request Code -1: customer 4's request fails since request exceeds need

current state

available
resource     7   3   2

             maximum      allocation  need
customer 0   1   5   6    0   0   0   0   0   0
customer 1   2   1   2    0   0   0   0   0   0
customer 2   5   5   3    0   0   0   0   0   0
customer 3   4   3   4    0   0   0   0   0   0
customer 4   8   2   7    3   2   5   5   0   2

Release 0 0 1
Release Code 1: customer 4's release succeeds

current state

available
resource     7   3   3

             maximum      allocation  need
customer 0   1   5   6    0   0   0   0   0   0
customer 1   2   1   2    0   0   0   0   0   0
customer 2   5   5   3    0   0   0   0   0   0
customer 3   4   3   4    0   0   0   0   0   0
customer 4   8   2   7    3   2   4   5   0   3
```

Requirement

- ✓ (**5%**)Initialize global data correctly; using **srand, rand** as described in this project
- ✓ (**5%**)Use **pthread** to create threads and **mutex** to lock/unlock correctly
- ✓ (**10%**)Use **sched_yield** to yield the CPU
- ✓ (**20%**)Print out correctly, that is, the state should be correct according to request/release and the print format should be appropriate
- ✓ (**35%**)Function **requestResources** is correct; feed parameters correctly when called; safety check would occupy **10%**
- ✓ (**25%**)Function **releaseResources** is correct; feed parameters correctly when called