CS342 Operating Systems – Spring 2023
Project #3 – Resource Manager Library, Deadlocks

Assigned: April 26, 2023.
Due date: May 12, 2023, 23:59. Document version: 1.1

- This project will be done in groups of two students. You can do it individually as well. The group members can be from different sections.
- Please start as soon as possible. Work incrementally, step by step.
- You will program in C. Programs will be tested in Ubuntu Linux.
- Goals of the project: Exercise with: deadlock detection, deadlock avoidance, locks and conditions variables, multithreaded programs, resource management, developing thread-safe libraries.

In this project you will develop a resource manager library, that will manage a set of resources for a multithreaded program (application). A program that is linked with the library may create multiple threads and each thread may request resources from the library, use them, and release them, as many times as it wishes, during its lifetime.

When a program is started, with a function call to the library, the program will be able to specify how many resource types and how many instances of each type will be managed by the library. Then the library will do resource allocation and management for the threads that are created by the program and that are running concurrently. The resources that the library will manage are not real resources, they are simulated resources. A thread will be blocked if the resources that it requests are not available or if it is not safe to allocate the requested resources. The library will be able to detect deadlocks and avoid deadlocks, when desired.

Your library will **implement** the following **functions**.

int **rm_init (**int `N`, int `M`, int `existing[M]`, int `avoid`**).**

This function will initialize the necessary structures and variables in the library to do resource management. `N` is the number of threads and `M` is the number of resource types. The parameter `existing` is an array of `M` integers indicating the existing resource instances of each type, initially all available. The `avoid` parameter is used to specify if the library will do deadlock avoidance or not. If `avoid` is `1`, then deadlock avoidance will be used while allocating resources. If `avoid` is `0`, then no deadlock avoidance will be applied, and therefore deadlocks may occur. The function will return `0` if initialization is successfully done, and will return `-1` in case there is an error encountered; for example, when the value of `N` or `M` exceeds the maximum number of threads or resources types supported by the library; or if any specified value is negative, etc.

int **rm_thread_started** (int `tid`).

This function will be called by a thread immediately after it starts execution. That means this function will be called at the beginning of a thread start function. With this function, the thread will indicate to the library that it has become alive and its identifier is `tid`. The `tid` parameter value is determined by the main thread (i.e., the application) and passed to the thread, i.e., to the thread start function, by specifying it as the last argument to the `pthread_create()` function. In this way each created thread will have a unique integer id selected by the main thread (application) from the range [`0`, `N-1`], where `N` is the number threads to create.

Note that this is not the thread ID of type `pthread_t` that is assigned internally by the `pthread_create()` function. A thread can learn its internal ID by calling `pthread_self()`. In this way, in the library, you can associate the `tid` value (determined by user application) for a thread with its internal thread ID (determined by `pthread_create()`).

This function should be called by a thread at the beginning of the thread start function, before any other library call, such as `rm_claim()`, or `rm_request()`.

This function can not be called by the main thread. The library will not consider the main thread as a thread that can request resources. Only the threads created by the main thread can request resources. If, for example, 5 threads are created by the main thread, the library will only consider these 5 threads (with tid values 0, 1, 2, 3, 4) to keep information about them, like their requests, allocations, etc.

int **rm_claim** (int `claim[M]`).

If deadlock avoidance is desired (indicated with `rm_init()`), then a thread will use this function to indicate to the library its maximum resource demand. This function should be called just after calling `rm_thread_started()`, before any request is issued. The `claim` array is used to specify how many resource instances of each type a thread may need at most. The library populates a MaxDemand matrix with the claim information it receives from all threads. Deadlock avoidance algorithm, implemented inside the `rm_request()` function, will use the claim information to avoid deadlocks. The function will return `0` if successful. It will return `-1` in case of an error, such as trying to claim more instances than existing.

int **rm_thread_ended**().

This function will be called by a thread just before termination. With this function, the thread indicates to the library that it is terminating. It will return `0` upon success, `-1` upon failure.

int **rm_request (**int `request[M]`**).**

This function is called by a thread to request resources from the library. The function will allocate the requested resources if resources are available. If deadlock avoidance is used, resources may not be allocated even when they are available, because it may not be safe to do so.

The `request` parameter is an integer array of size `M`, and indicates the number of instances of each resource type that the calling thread is requesting.

If the requested resources are available, then they will be allocated and the function will return without getting blocked. If the requested resources are not available, the thread will be blocked inside the function (by use of a condition variable), until the requested resources can be allocated, at which time resources will be allocated and the function will return. The function will return `0` upon success (resources allocated). It will return `-1` if there is an error condition, for example, the number of requested instances for a resources type is greater than the number of existing instances.

If deadlock avoidance is used, resources will be allocated only if it is safe to do so. If the new state would not be safe, the requested resources are not allocated and the calling thread is blocked, even though there are resources available to satisfy the request.

int **rm_release (**int `release[M]`**).**

This function is called by a thread to release resources. The number of instances of each resource type that the thread wants to release is indicated with the array `release` of size `M`. The function will deallocate the indicated resource instances. It will also wake up the blocked threads (that were blocked at `rm_request()` function) so that each thread can check if it can continue now. The `rm_release()` function will return `0` upon success. It will return `-1` in case of an error condition, for example, when trying to release more instances than allocated.

Note that with a release call, it is possible that a thread may not release all its allocated resources, but only some of them. Therefore, in an application, the number of request calls and release calls do not have to be equal.

int **rm_detection().**

This function will check if there are deadlocked processes at the moment. The function will return the number of deadlocked processes, if any. If there is no deadlocked process, `0` will be returned. In case of any error, `-1` will be returned.

void **rm_print_state** (char `headermsg[]`).

When called, this function will print out information about the current state in the library. The information will include the content of the Existing vector, Available vector, Allocation matrix, Request matrix, MaxDemand matrix, and the Need matrix. If deadlock avoidance is not specified, then MaxDemand matrix and Need matrix will have all zeros. An application that is using the library may call this function whenever it wants to learn about the state. This function will be the only function in your library (in the submitted version) that will print out something to the screen. The `headermsg` parameter is a string that will be printed out at the beginning of the state information. An example output is below. You need to produce a very similar output in terms of the format. In the example below, we have two resource types, R0 and R1, and three threads, T0, T1, T2, created by the main thread. The `headermsg` is "`The current state`".

```
##########################
The current state
###########################
Exist:
     R0 R1
     8  5

Available:
     R0 R1
     3  5

Allocation:
     R0 R1
T0:  5  0
T1:  0  0
T2:  0  0

Request:
     R0 R1
T0:  0  0
T1:  2  0
T2:  0  0

MaxDemand:
     R0 R1
T0:  8  4
T1:  8  5
T2:  0  3

Need:
     R0 R1
T0:  3  4
T1:  8  5
T2:  0  3
###########################
```

Note that the library functions described above may be called by multiple threads **simultaneously**. Be careful about race conditions. Your library should be free of race conditions.

The **header** file `rm.h` is given below. It is the **interface** of your library to the applications. `MAXR` is the maximum number of resources types that can be managed by the library. `MAXP` is the maximum number of threads that can be created by an application to make resource requests to the library.

```
#define MAXR 100 // max num of resource types supported
#define MAXP 100 // max num of threads supported

int rm_init(int p_count, int r_count,
            int r_exist[], int avoid);
int rm_thread_started(int tid);
int rm_thread_ended();
int rm_claim (int claim[]); // only for avoidance
int rm_request (int request[]);
int rm_release (int release[]);
int rm_detection();
void rm_print_state (char headermsg[]);
```

Your library functions (i.e., your library) will be **implemented** in a C file called **`rm.c`**. In that file, you will also define the global variables and structures that your library functions will use. You will implement the functions described above. You can implement additional functions if you wish. But these additional functions will not be part of the **interface** of the library. That means their prototypes should not be included in `rm.h` and therefore an application should not directly call these extra functions.

You will use Pthreads mutex and condition variables for your **synchronization** needs. In your library implementation (in `rm.c`), you can use as many mutex and condition variables as you wish. *Hint*: Defining just one mutex variable (lock variable) is enough. defining one condition variable per thread (an array of condition variables) is enough. These synchronization variables need to be defined as global variables.

A set of files (a skeleton) is provided in **github** so that you can start quickly: https://github.com/korpeoglu/cs342spring2023-p3.git. You can clone it. The skeleton of the `rm.c` file is given below. You need to complete it.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include "rm.h"

int DA;  // indicates if deadlocks will be avoided or not
int N;   // number of processes
int M;   // number of resource types
int ExistingRes[MAXR]; // Existing resources vector

//..... other definitions/variables .....
//.....
//.....

int rm_thread_started(int tid)
{
    int ret = 0;
    return (ret);
}


int rm_thread_ended()
{
    int ret = 0;
    return (ret);
}


int rm_claim (int claim[])
{
    int ret = 0;
    return(ret);
}


int  rm_init(int  p_count,  int  r_count,  int  r_exist[],    int
avoid)
{
    int i;
    int ret = 0;

    DA = avoid;
    N = p_count;
    M = r_count;
    // initialize (create) resources
    for (i = 0; i < M; ++i)
        ExistingRes[i] = r_exist[i];
    // resources initialized (created)

    //....
    // ...
    return  (ret);
}


int rm_request (int request[])
{
    int ret = 0;

    return(ret);
}


int rm_release (int release[])
{
    int ret = 0;

    return (ret);
}


int rm_detection()
{
    int ret = 0;

    return (ret);
}


void rm_print_state (char hmsg[])
{
    return;
}
```

A Sample Application

A multi-threaded application, for example `app.c`, that will use your library will first include the header file “`rm.h`” corresponding to your library. An application will be compiled and linked with your library as follows:

```
     gcc -Wall -o app  -L. -lrm  -lpthread app.c
```

Below is a sample application showing the use of the library.

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdarg.h>
#include "rm.h"

#define NUMR 1        // number of resource types
#define NUMP 2        // number of threads

int AVOID = 1;
int exist[1] =  {8};  // resources existing in the system

void pr (int tid, char astr[], int m, int r[])
{
    int i;
    printf ("thread %d, %s, [", tid, astr);
    for (i=0; i<m; ++i) {
        if (i==(m-1))
            printf ("%d", r[i]);
        else
            printf ("%d,", r[i]);
    }
    printf ("]\n");
}


void setarray (int r[MAXR], int m, ...)
{
    va_list valist;
    int i;

    va_start (valist, m);
    for (i = 0; i < m; i++) {
        r[i] = va_arg(valist, int);
    }
    va_end(valist);
    return;
}


void *threadfunc1 (void *a)
{
    int tid;
    int request1[MAXR];
    int request2[MAXR];
    int claim[MAXR];

    tid = *((int*)a);
    rm_thread_started (tid);

    setarray(claim, NUMR, 8);
    rm_claim (claim);

    setarray(request1, NUMR, 5);
    pr (tid, "REQ", NUMR, request1);
    rm_request (request1);

    sleep(4);

    setarray(request2, NUMR, 3);
    pr (tid, "REQ", NUMR, request2);
    rm_request (request2);

    rm_release (request1);
    rm_release (request2);

    rm_thread_ended();
    pthread_exit(NULL);
}


void *threadfunc2 (void *a)
{
    int tid;
    int request1[MAXR];
    int request2[MAXR];
    int claim[MAXR];

    tid = *((int*)a);
    rm_thread_started (tid);

    setarray(claim, NUMR, 8);
    rm_claim (claim);

    setarray(request1, NUMR, 2);
    pr (tid, "REQ", NUMR, request1);
    rm_request (request1);

    sleep(2);

    setarray(request2, NUMR, 4);
    pr (tid, "REQ", NUMR, request2);
    rm_request (request2);

    rm_release (request1);
    rm_release (request2);

    rm_thread_ended ();
    pthread_exit(NULL);
}


int main(int argc, char **argv)
{
    int i;
    int tids[NUMP];
    pthread_t threadArray[NUMP];
    int count;
    int ret;

    if (argc != 2) {
        printf ("usage: ./app avoidflag\n");
        exit (1);
    }

    AVOID = atoi (argv[1]);

    if (AVOID == 1)
        rm_init (NUMP, NUMR, exist, 1);
    else
        rm_init (NUMP, NUMR, exist, 0);

    i = 0;  // we select a tid for the thread
    tids[i] = i;
    pthread_create (&(threadArray[i]), NULL,
                    (void *) threadfunc1, (void *)
                    (void*)&tids[i]);

    i = 1;  // we select a tid for the thread
    tids[i] = i;
    pthread_create (&(threadArray[i]), NULL,
                    (void *) threadfunc2, (void *)
                    (void*)&tids[i]);

    count = 0;
    while ( count < 10) {
        sleep(1);
        rm_print_state("The current state");
        ret = rm_detection();
        if (ret > 0) {
            printf ("deadlock detected, count=%d\n", ret);
            rm_print_state("state after deadlock");
        }
        count++;
    }

    if (ret == 0) {
        for (i = 0; i < NUMP; ++i) {
            pthread_join (threadArray[i], NULL);
            printf ("joined\n");
        }
    }
}
```

Develop a Multi-threaded Application

Develop an application that will create some number of threads that will run concurrently and will request, use, and release resources concurrently. The number of threads should be at least 3, and the number of resource types should be at least 5. Your application will use your library. The library will do the allocation and release of resources. Your application should demonstrate the occurrence of deadlocks, detection of deadlocks, and avoidance of deadlocks. It is up to you how to demonstrate these, i.e., what to put into your application. You will write one application, called `myapp.c` (executable name: `myapp`), to demonstrate deadlocks, deadlock detection, and deadlock avoidance. The application will take one command line parameter, a flag value (`0` or `1`). If flag is `0`, deadlocks will not be avoided and your application will demonstrate the occurrence and detection of deadlocks. If flag is `1`, then deadlock avoidance will be done and demonstrated.

We will also develop our test applications that will be linked with your library. Our test applications may create many threads. Each thread may issue many request/release calls to the library. We will check if deadlocks occur, if they can be detected, and if they can be avoided.

As a result of compilation of the library (rm.c), we will obtain a binary library file, called **librm.a**. Below is a Makefile that is showing how to build the library `librm.a` and how to link an application `app.c` with the library.

```
all: librm.a  app

librm.a:  rm.c
     gcc -Wall -c rm.c
     ar -cvq librm.a rm.o
     ranlib librm.a

app: app.c
     gcc -Wall -o app app.c -L. -lrm -lpthread

clean:
     rm -fr *.o *.a *~ a.out  app rm.o rm.a librm.a
```

Submission

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both students will be written, separated by a dash '-'. In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include `README.txt`, `Makefile`, and program source files. We should be able to compile your **programs** by just typing `make`. No binary files (executable files) will be included in your submission. Then tar and gzip the directory, and submit it to Moodle.

For example, a project group with student IDs 21404312 214052104 will create a directory named "21404312-214052104" and will put their files there. Then, they will tar the directory (package the directory) as follows:

```
tar cvf 21404312-214052104.tar 21404312-214052104
```

Then they will gzip the tar file as follows:

```
gzip 21404312-214052104.tar
```

In this way they will obtain a file called 21404312-214052104.tar.gz. Then they will upload this file into Moodle (one upload per group). For a project done individually, just the ID of the student will be used as file or directory name.

Tips and Clarifications:

- Start early, work incrementally.
- If you wish you can develop your programs in another OS environment (MacOS, Windows, etc.). But finally, you have to make sure that your programs compile and run properly in Linux. Test them as much as possible. We will test your programs in Linux.
- You are suggested to implement and test deadlock detection first. Then deadlock avoidance.
- You will include a `Makefile` with your project submission. Make sure it works in your environment (name your files accordingly). We will just type "`make`" and your programs should compile.
- You can not change the interface file (`rm.h`).
- Assuming no deadlock avoidance, we do the following assumption. When a request is made by a process P, either all the requested resources will be allocated to the process P (if they are available), or none of them will be allocated, and request will be blocked (pending). That means no partial allocation will done for a request. For example, if a process is requesting 2 A, 1 B, 3 C, and we just have have 2 C available, these 2 Cs will not be allocated; request (2,1,3) will be pending (blocked).
- When deadlock avoidance is used and when a process requests resources and if it is not safe to allocate resources (they are available but not safe), then the process is not allocated any resources and is blocked. It will stay blocked until resources are available and safe to allocate.
- If you modify `rm.c` file, make sure it is compiled when you run "`make`". To ensure that, first run "`make clean`", and then "`make`".
- The submitted version of your library (`rm.c`) should not print out anything, except the `rm_print_state()` function (it will print the current state). But, while developing and testing your library, it can print messages. You must delete them before submission.
- Note that avoidance algorithm is not using the Request matrix for safety check. It is using the Need matrix, which means that it is also using the MaxDemand matrix and Allocation matrix.
- Note that if deadlock avoidance is not used, MaxDemand and Need matrices are not used.
- Maximum number of threads supported by the library should be 100.

- Maximum number of resource types supported by the library should be 100.