CS342 - Spring 2019
Project #3
Synchronization and Deadlocks

Assigned: **April 2, 2019.**
Due date: **April 21, 2019, 23:55.**

Objectives

- Practice multi-threaded programming.
- Practice synchronization: **mutex** and **condition variables**; Pthreads API.
- Practice deadlock detection and avoidance methods.
- Practice designing and performing experiments.

You can do this project in groups of two students each. You will use C and Linux.

Resource Allocation Library

In this project you will design and develop a resource allocation library (**libralloc.a**) that will simulate the behaviour of a kernel in terms of resource allocation and deadlock handling. Like a kernel, it will allocate resources to multiple processes. It will be able to do deadlock avoidance and deadlock detection as well. Multiple processes requesting resources will be simulated with multiple threads. A multithreaded application using the library may create multiple threads and each thread will be like a process requesting resources whenever needed and releasing when finished. The library will do resource access control and allocation. If resources are not available or it is not safe to allocate, the requesting thread will be blocked by the library.

You library will implement the following functions.

- **int ralloc_init (int N, int M, int exist[M], int handling_method)**. This function will initialize the necessary structures and variables in the library to do resource allocation and access control. N is the number of processes, and M is the number of resources types (type 0, type 1, ..., type M-1). Exist is an array of M integers indicating the existing resource instances in the system for each resource type. For example, Exist[0] is the number of instances in the system for resource type 0. The parameter handling_method is the deadlock handling method to use. It can be one of DEADLOCK_NOTHING (no deadlock handling), DEADLOCK_AVOIDANCE (deadlocks will be avoided), DEADLOCK_DETECTION (deadlocks will be detected). When DEADLOCK_AVOIDANCE is indicated, the library will do deadlock avoidance. So your library should implement a deadlock avoidance algorithm. When DEADLOCK_DETECTION is indicated, the library will do deadlock detection when requested. Hence your library should implement a deadlock detection algorithm as well. The function will return 0 upon success, otherwise it will return -1. This function

will be called by an application before creating any threads. It will return 0 upon success, -1 otherwise.

- **int ralloc_maxdemand (int pid, int r_max[M])**. This function will be called by an application thread when started. It will be used to indicate the maximum resource usage of the calling thread. The function will record the maximum demand information for the calling thread inside the library structures. With this call the library will have information about the maximum possible resource usage of the calling thread. The pid parameter is the integer id of the calling thread. It is a number between 0 and N-1. N is number of threads the application has created (i.e., the number of processes that are simulated requesting resources from a kernel). This function will be called by a thread at the beginning before making any resource requests yet. It will return 0 upon success, -1 otherwise.
- **int ralloc_request (int pid, int demand[M])**. This function is called by a thread to request resources from the library (like requesting resources from kernel). The function will allocate resources if it is OK to do so. The pid parameter is the id of the calling process (thread). The demand parameter is an integer array of size M and indicates the number of resource instances requested by the process for each resource type. If deadlock avoidance is not applied and if resources are available, then requested resources will be allocated and function will return. If resources are not available, the calling thread will be blocked inside the function and therefore the function will not return immediately. Internally in the library, you will use conditions variables to do this. If deadlock avoidance is used, then the calling thread may be blocked even though there are resources available, if allocation would leave the system in an unsafe state. If deadlock avoidance is indicated, the function will execute the deadlock avoidance algorithm. The function will return 0 upon success, -1 otherwise.
- **int ralloc_release (int pid, int demand[M])**. This function is called by a thread to release resources. The id of the thread releasing resources is indicated with the pid parameter. The number of resource instances to release for each resource type is indicated with the demand array. This function will deallocate the indicated resource instances. It will also check if it is now possible to satisfy any pending request. If deadlock avoidance is applied, avoidance algorithm needs to be run while attempting to allocate the released resources to some other process. The function will return 0 upon success, -1 otherwise.
- **int ralloc_detection (int procarray[M])**. This function will check if there are any deadlocked processes at the moment. If there are, the respective entries in the procarray will be set to 1. Otherwise the values are -1. The number of deadlocked processes will be returned as the return value. If there is no deadlock, return value will be 0.
- **int ralloc_detection (int procarray[M])**. This function will do any clean up if necessary. It will return 0 upon success, -1 otherwise.

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

The header file ralloc.h is given below. MAX_RESOURCES_TYPES is the maximum number of resources types that can be simulated. MAX_PROCESSES is the

maximum number of processes that can be simulated.

```
#ifndef RALLOC_H
#define RALLOC_H

#include <pthread.h>

#define MAX_RESOURCE_TYPES 10
#define MAX_PROCESSES 20

#define DEADLOCK_NOTHING   1
#define DEADLOCK_DETECTION 2
#define DEADLOCK_AVOIDANCE 3

int ralloc_init(int p_count, int r_count, int r_exist[], int d_handling);
int ralloc_maxdemand(int pid, int r_max[]);
int ralloc_request (int pid, int demand[]);
int ralloc_release (int pid, int demand[]);
int ralloc_detection(int procarray[]);
int ralloc_end();

#endif /* RALLOC_H */
```

You will implement ralloc.c. In that you can define the variables and structures you wish. You can use as many mutex and condition variables as you want.

Develop a Multi-threaded Application

Develop an application that will create some number of threads simulating concurrently running processes requesting resources from the kernel (your library) from time to time. Your application will use your library. The library will do the allocation and release of resources. The library may block (wait) and wakeup the threads when necessary. Your application should demonstrate the occurance of deadlocks, detection of deadlocks, and avoidance of deadlocks. It is upto you what to put into your application.

We will develop our test applications that we will be linked with your library. Our test applications may create many threads. Each thread may issue many request and release calls to the library. Our applications may expect deadlock avoidance from your library. They may also request deadlock detection to be done. =

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

```
gcc -Wall -o app -L. -lralloc app.c
```

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

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

```
#include "pthread.h"
#include "ralloc.h"

int handling_method;          // deadlock handling method

#define M 3                   // number of resource types
int exist[3] =  {12, 8, 10};  // resources existing in the system

#define N 5                   // number of processes - threads
pthread_t tids[N];            // ids of created threads

void *aprocess (void *p)
{
    int req[3];
    int k;
    int pid;

    pid =  *((int *)p);

    printf ("this is thread %d\n", pid);
    fflush (stdout);

    req[0] = 2;
    req[1] = 2;
    req[2] = 2;
    ralloc_maxdemand(pid, req);

    for (k = 0; k < 10; ++k) {

        req[0] = 1;
        req[1] = 1;
        req[2] = 1;
        ralloc_request (pid, req);

        // do something with the resources

        ralloc_release (pid, req);

        // call request and release as many times as you wish with
        // different parameters
    }
    pthread_exit(NULL);
}


int main(int argc, char **argv)
{
    int dn; // number of deadlocked processes
    int deadlocked[N]; // array indicating deadlocked processes
    int k;
    int i;
```

```
    int pids[N];

    for (k = 0; k < N; ++k)
        deadlocked[k] = -1; // initialize

    handling_method = DEADLOCK_DETECTION;
    ralloc_init (N, M, exist, handling_method);

    printf ("library initialized\n");
    fflush(stdout);

    for (i = 0; i < N; ++i) {
        pids[i] = i;
        pthread_create (&(tids[i]), NULL, (void *) &aprocess,
                        (void *)&(pids[i]));
    }

    printf ("threads created = %d\n", N);
    fflush (stdout);

    while (1) {
        sleep (10); // detection period
        if (handling_method == DEADLOCK_DETECTION) {
            dn = ralloc_detection(deadlocked);
            if (dn > 0) {
                printf ("there are deadlocked processes\n");
            }
        }
        // write code for:
        // if all treads terminated, call ralloc_end and exit
    }
}
```

Experiments and Report

Lets say we are interested in finding the overhead of deadlock avoidance. Design and do experiments to measure and evaluate the overhead of deadlock avoidance compared to not using deadlock avoidance. Write a report describing your experiments and giving your results. Try to interpret and discuss your results. What about the cost of deadlock detection? Measure and report that as well.

Submission

Put your report.pdf file, your program files, a Makefile, and a README.txt file into a directory named with your ID (for a group, a single file will be uploaded using the ID of one of the students). Then tar and gzip the directory. For example a student with ID 21404312 will create a directory named 21404312 and will put the files there. Then he will tar the directory (package the directory) as follows:

```
tar cvf 21404312.tar 21404312
```

Then he will gzip the tar file as follows:

```
gzip 21404312.tar
```
In this way he will obtain a file called 21404312.tar.gz. Then he will upload this file in Moodle.

Late submission will not be accepted (no exception). A late submission will get 0 automatically (you will not be able to argue it). Make sure you make a submission one day before the deadline. You can then overwrite it.

Tips and Clarifications

- You need to learn how to use Pthreads mutex and condition variables. There are links to some resources in the References section of the course webpage. You can find additional resources from Internet.
- A project 3 skeleton of code is posted in github:
  `https://github.com/korpeoglu/cs342spring2019_p3`
  You can clone it and start with it. At least, it will help you with the compilation of the library and programs.
- You will include the 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.
- We may put clarifications to the homepage of the course (near the project assignment).
- You are suggested to test your ralloc library first by test programs including simple cases.
- You can not change the interface (ralloc.h)