Loyola University

Real-Time and Embedded Systems

PRACTICE 3 RESULTS

Authors:
Martyna Baran
Zuzanna Jarlaczynska

December 7, 2023

Problem 1: Explaining the obtained output

The posted code is an example of the use of Round Robin policy in a multi-threaded system. RR involves allocating a certain amount of CPU time to each thread, after which it moves on to the next thread, then returns to the first thread and so on. In the code, tasks A, B and C are run one after the other in a specific order.

In the case of my system, this code initially generated rather unexpected results, as the first two rounds of the loop for tasks A and B managed to finish before task C started. This situation is independent of the task itself and only depends on the operating system. However, by the third loop cycle, we already get the expected result.

At this point my operating system has already run all the functions, and we see that our string is starting to form individual letters or numbers, from each individual task. The system allocates a specific execution time to each thread, after which it switches to the next thread until all functions have completed their execution. Calling the join function at the end causes each thread, including the main thread, to wait for the others to finish before finally terminating. Therefore, because of the previous situation, threads A and B write out their results an extra 2 times because they have to wait for thread C to finish.

I realize that the expected result was most likely to be a string in which the threads write out one of their letters and move on to the next. However, the purpose of this task was to run the program and describe what I got, which is what I am presenting. For this task, we did not use synchronization mechanisms such as semaphores, mutexes or conditional variables to coordinate the threads as needed. Without this, the operating system decides the scheduling, and concurrency can lead to different combinations of thread execution, which was the cause of the initial disturbance that was later stabilized after the second loop.

We have obtained the following output.

```
1a2b3c4d1a2b3c4d51a62b73c84d51a62b73c84d51a62b73c84d51a62b73c84d51a62b73c84d51a62b73c84d51a62b73c84d51a62b73c84d51a62b73c84d567856781526374815263748152a63b74c81d52a63bd
```

Problem 2: FIFO and Round Robin Policy and Semaphores

(1) In the first scenario, we were about to check the execution of the code with different policies attributed to three tasks. In our case, we decided to do it in the following way:

- Task A: Round Robin Policy with priority $= 40$
- Task B: FIFO Policy with priority $= 50$
- Task C: Round Robin Policy with priority $= 40$

We decided to use this kind of division of policies because the second task (B task) prints the letters instead of numbers. As a result, we will be able to see the difference.

We use two types of policies: FIFO and Round Robin. FIFO stands for "First In First Out" and assumes that tasks that are created first are the first to be disposed of. Round Robin policy is an arrangement of choosing all elements in a group equally in some rational order, usually from the top to the bottom of a list and then starting again at the top of the list and so on.

The first thing we can see is that, as expected, the first executed task in task B - the one with the highest priority. Every time it starts before the other two tasks printing one letter. Then the microprocessor allows other A and C tasks with Round Robin Policy to ping the available resources to execute their code. They only have a slight amount of time so they print one number and let the other task work.

In our case the code makes a little mistake at the beginning - only task A and B prints their values. After one loop everything executes properly. The situation is acceptable as we need to remember that outputs depend strongly on the computer and its resources so they can differ. In general, it properly shows how should the code work.

C Code for exercise 2A

```c
#include <stdio.h>
```

```
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <semaphore.h>

void* a(void* ptr) {
 for (int i=0; i< 10; i++){

     printf("1");
     printf("2");
     printf("3");
     printf("4");

 }
 return NULL;
}

void* b(void* ptr) {
 for (int i=0; i< 10; i++){
     printf("a");
     printf("b");
     printf("c");
     printf("d");

 }
 return NULL;
}

void* c(void* ptr) {
for (int i=0; i< 10; i++){
        printf("5");
        printf("6");
        printf("7");
        printf("8");

 }
 return NULL;
}

int main() {
 printf("\n\n");
 pthread_attr_t attr1, attr2, attr3;

 pthread_attr_init(&attr1);
 pthread_attr_init(&attr2);
 pthread_attr_init(&attr3);

 pthread_t t1, t2, t3;
// setting the policy
 struct sched_param param_a, param_b, param_c;
 param_a.sched_priority = 40;
 param_b.sched_priority = 50;
 param_c.sched_priority = 40;
 int rVal1 = pthread_attr_setschedpolicy(&attr1, SCHED_RR);
 pthread_attr_setschedparam(&attr1, &param_a);

 int rVal2 = pthread_attr_setschedpolicy(&attr2, SCHED_FIFO);
 pthread_attr_setschedparam(&attr2, &param_b);

 int rVal3 = pthread_attr_setschedpolicy(&attr3, SCHED_RR);
 pthread_attr_setschedparam(&attr3, &param_c);

  if (rVal1 != 0 || rVal2 !=0 || rVal3 !=0) {
  // Failed to set the desired scheduler policy.
```

```c
            perror("pthread_attr_setschedpolicy(&attr1, SCHED_RR)");
            exit(1);
        }

    // creating tasks all with Round Robin policy
     pthread_create(&t1, &attr1, a, NULL);
     pthread_create(&t2, &attr2, b, NULL);
     pthread_create(&t3, &attr3, c, NULL);

     pthread_join(t1, NULL);
     pthread_join(t2, NULL);
     pthread_join(t3, NULL);

     pthread_exit(0);

     return 0;
    }
```

We have obtained the following output.

```
a1b25c36d47a18b25c36d47a18b25c36d47a18b25c36d47a18b25c36d47a18b25c36d47a18b25c36d47a18b25c36d47a18b25c36d478
```

(2) To solve this problem, we decided to use three binary semaphores and a variable to be modified by the shuffle. Initially, we only initialize one semaphore, thus allowing task A to start. Each task makes the semaphore available to the next task, thus unlocking it. Furthermore, we have introduced a global variable "count" that is incremented by task A and decremented by task B. The print function in task C only fires when the value of the variable manipulated by the previous tasks is equal to 1, but, due to the use of the semaphore, it will never be reached at this point in the program. This keeps all tasks running, but only tasks A and B print values to the screen.

C Code for exercise 2B

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <semaphore.h>

sem_t sem_a, sem_b, sem_c;
int count;
void* a(void* ptr) {
 for (int i=0; i< 10; i++){
    sem_wait(&sem_a);
    printf("1");
    printf("2");
    printf("3");
    printf("4");
    count++;
    sem_post(&sem_b);
 }
 return NULL;
}
void* b(void* ptr) {
 for (int i=0; i< 10; i++){
    sem_wait(&sem_b);
    printf("a");
    printf("b");
    printf("c");
```

```
    printf("d");
    count--;
    sem_post(&sem_c);
 }
 return NULL;
}
void* c(void* ptr) {
for (int i=0; i< 10; i++){
    sem_wait(&sem_c);
    if(count ==1){
        printf("5");
        printf("6");
        printf("7");
        printf("8");
        sem_post(&sem_a);
    }
    else{
        sem_post(&sem_a);
    }
 }
 return NULL;
}
int main() {
 printf("\n\n");
 //initializing attributes
 pthread_attr_t attr1, attr2, attr3;
 pthread_attr_init(&attr1);
 pthread_attr_init(&attr2);
 pthread_attr_init(&attr3);

 //initializing semaphores
 sem_init(&sem_a,0,1);
 sem_init(&sem_b,0,0);
 sem_init(&sem_c,0,0);

 pthread_t t1, t2, t3;
 // setting the policy
 int rVal1 = pthread_attr_setschedpolicy(&attr1, SCHED_RR);

 int rVal2 = pthread_attr_setschedpolicy(&attr2, SCHED_RR);

 int rVal3 = pthread_attr_setschedpolicy(&attr3, SCHED_RR);

 if (rVal1 != 0 || rVal2 !=0 || rVal3 !=0) {
 // Failed to set the desired scheduler policy.
    perror("pthread_attr_setschedpolicy(&attr, SCHED_RR)");
    exit(1);
 }

// creating tasks
 pthread_create(&t1, &attr1, a, NULL);
 pthread_create(&t2, &attr2, b, NULL);
 pthread_create(&t3, &attr3, c, NULL);

 pthread_join(t1, NULL);
 pthread_join(t2, NULL);
 pthread_join(t3, NULL);

 pthread_exit(0);
 sem_destroy(&sem_a);
 sem_destroy(&sem_b);
 sem_destroy(&sem_c);
 return 0;
}
```

We have obtained the following output.

```
1234abcd1234abcd1234abcd1234abcd1234abcd1234abcd1234abcd1234abcd1234abcd1234abcd
```

Problem 3: Reader Writers Problem

(1) We decided to design a code that will face a no-starvation version of the Reader-Writer problem. In general, this problem pertains to any situation where a data structure, database, or file system is read and modified by concurrent threads. While the data structure is being written or modified it is often necessary to bar other threads from reading, in order to prevent a reader from interrupting a modification in progress and reading inconsistent or invalid data. However, in the classical algorithm, we only consider the situation where a reader can always access a resource as soon as another reader is currently using it because multiple simultaneous readers are not a problem. Then if a writer arrives while there are readers in the critical section, it might wait in queue forever while readers come and go. As long as a new reader arrives before the last of the current readers departs, there will always be at least one reader in the room. As I conceived it, I decided to add a 'queue' semaphore which, when a writer who would like to enter the critical section arrives, he will block any further readers from entering before them and queue them up. As soon as the writer leaves the room, all the queued readers will be able to use the resource again.

C Code for exercise 3

```c
#include<semaphore.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

sem_t x,y, queue;
pthread_t tid;
pthread_t writerthreads[3],readerthreads[5];
int readercount = 0;

void *reader(void* param)
{
    sem_wait(&queue);
    sem_wait(&x);
    readercount++;

    if(readercount==1){
        sem_wait(&y);
    }
    sem_post(&x);
    sem_post(&queue);
    printf("There is %d readers inside.\n",readercount);
    sleep(1);
    sem_wait(&x);
    readercount--;
    if(readercount==0)
    {
        sem_post(&y);
    }
    printf("Reader %d is leaving\n",readercount+1);
    sem_post(&x);
    return NULL;
}
void *writer(void* param)
{
```

```
        int arg;
        arg = *(int*) param;
        sem_wait(&queue);
        printf("Writer %d is trying to enter\n", arg);
        sem_wait(&y);
        printf("Writer %d has entered\n", arg);
        sem_post(&y);
        printf("Writer %d is leaving\n", arg);
        sem_post(&queue);
        return NULL;
    }

    int main()
    {
        printf("\n\n");
        int i[5] = {1, 2, 3, 4, 5};
        int j[3] = {1, 2, 3};

        sem_init(&x,0,1);
        sem_init(&y,0,1);
        sem_init(&queue,0,1);

        pthread_attr_t attr1;
        pthread_attr_init(&attr1);
        pthread_attr_setschedpolicy(&attr1, SCHED_RR);

        pthread_create(&readerthreads[0],&attr1,reader,&i[0]);
        pthread_create(&writerthreads[0],&attr1,writer,&j[0]);
        pthread_create(&readerthreads[1],&attr1,reader,&i[1]);
        pthread_create(&readerthreads[2],&attr1,reader,&i[2]);
        pthread_create(&readerthreads[3],&attr1,reader,&i[3]);
        pthread_create(&writerthreads[1],&attr1,writer,&j[1]);
        pthread_create(&writerthreads[2],&attr1,writer,&j[2]);
        pthread_create(&readerthreads[4],&attr1,reader,&i[4]);

        for(int b=0;b<3;b++)
         {
             pthread_join(writerthreads[b],NULL);
         }
         for(int k=0;k<5;k++)
              {
             pthread_join(readerthreads[k],NULL);
              }
      sem_destroy(&x);
      sem_destroy(&y);
      sem_destroy(&queue);

     }
```

We have obtained the following output.

```
There is 1 readers inside.
Writer 1 is trying to enter
Reader 1 is leaving
Writer 1 has entered
Writer 1 is leaving
There is 1 readers inside.
There is 2 readers inside.
There is 3 readers inside.
Writer 2 is trying to enter
Reader 3 is leaving
Reader 2 is leaving
Reader 1 is leaving
Writer 2 has entered
Writer 2 is leaving
Writer 3 is trying to enter
Writer 3 has entered
Writer 3 is leaving
There is 1 readers inside.
Reader 1 is leaving
```