Lab 5: Synchronizing Threads

Master M1 MOSIG – Université Grenoble Alpes (Ensimag & UFR IM2AG)

2026

In this lab, we are going to study some synchronization problems.

1 Instructions

This lab is not graded. The problems are to be solved on paper.

2 Stack with semaphores

A stack is a data structure used to store a set of elements, which has two principal operations: `push()`, which adds an element to the set, and `pop()`, which removes the most recently added element that was not yet removed.

Figure 1 describes the implementation of a stack based on a fixed size array using semaphores. This implementation is not correct as it might lead to a deadlock.

Question 2.1: *Describe an execution scenario that would lead to a deadlock.*

Question 2.2: *Modify the code of Figure 1 to have a correct implementation of the stack.*

3 Reader-Writer problem

The reader-writer problem is a synchronization problem where some reader threads might read shared data and some writer threads might modify this shared data. If a writer wants to modify data, it must first get exclusive access to the data. On the other hand, multiple readers should be able to read at the same time.

Figure 2 describes a solution to the reader-writer problem. Before starting reading the data, a reader has to call the `startRead()` function. It must then call `endRead()` when it finishes. Similarly, a writer has to call `startWrite()/endWrite()` when it starts/finishes writing respectively.

```
1   #define N 10
2
3   void *buffer[N];
4   int nb_item;
5   sem_t S1;
6   sem_t S2;
7   sem_t S3;
8
9   void init_stack(void)
10  {
11    sem_init(&S1, 0, 1);
12    sem_init(&S2, 0, N);
13    sem_init(&S3, 0, 0);
14    nb_item=0;
15  }
16
17  void push(void *item)
18  {
19    sem_wait(&S1);
20    sem_wait(&S2);
21    buffer[nb_item]= item;
22    nb_item++;
23    sem_post(&S3);
24    sem_post(&S1);
25  }
26
27  void* pop(void)
28  {
29    void *item;
30
31    sem_wait(&S1);
32    sem_wait(&S3);
33    item = buffer[nb_item -1];
34    nb_item--;
35    sem_post(&S2);
36    sem_post(&S1);
37
38    return item;
39  }
```

Figure 1: The stack

A reader-writer synchronization algorithm can give priority to readers (when readers and writers are contending to access the shared data, readers get access) or to writers (when readers and writers are contending to access the shared data, a writer gets access).

Question 3.3: *Does the solution described in Figure 2 give priority to readers or to writers? Explain.*

```
1   /* global variables */
2   int readerCount = 0:
3   int writer = 0;
4   mutex_t mutex = MUTEX_INITIALIZER;
5   cond_t cond = COND_INITIALIZER;
6
7   void startRead(void) {
8     mutex_lock(&mutex);
9     readerCount = readerCount + 1;
10    while(writer == 1) { cond_wait(&cond, &mutex); }
11    mutex_unlock(&mutex);
12  }
13
14  void endRead(void) {
15    mutex_lock(&mutex);
16    readerCount = readerCount - 1;
17    cond_broadcast(&cond);
18    mutex_unlock(&mutex);
19  }
20
21  void startWrite(void) {
22    mutex_lock(&mutex);
23    while((readerCount > 0) || (writer == 1)) { cond_wait(&cond, &mutex);}
24    writer = 1;
25    mutex_unlock(&mutex);
26  }
27
28  void endWrite(void) {
29    mutex_lock(&mutex);
30    writer = 0;
31    cond_broadcast(&cond);
32    mutex_unlock(&mutex);
33  }
```

Figure 2: Reader-Writer synchronization

Question 3.4: *Let us consider the case where there are* multiple readers and a single writer.[1] *Answer questions A to D. Each suggested modification is considered separately. We will assume that we do not have to deal with* spurious *wakeups.*

(A) Can we replace `while` *line 10 with* `if`*?*

(B) Can we replace `while` *line 23 with* `if`*?*

(C) Can we replace `cond_broadcast()` *line 17 with* `cond_signal()`*?*

(D) Can we replace `cond_broadcast()` *line 31 with* `cond_signal()`*?*

For each question, determine whether the algorithm remains correct and explain.

[1]Note that the writer may perform one or several consecutive write requests. Similarly, each reader may perform one or several consecutive read requests.

Question 3.5: *What is the main problem that you can foresee with the code provided in Figure 2.*

Question 3.6: *Propose a new version of the code that gives priority to the other camp (if the algorithm from Figure 2 gives priority to readers, write one that gives priority writers – or the other way around). In this first step, use a single condition variable.*

Question 3.7: *Propose a new version of your code that uses multiple condition variables to avoid waking up more threads than strictly necessary on calls to* `endRead()` *and* `endWrite()`*.*

Comment: Note that in practice, the `Pthread` library provides an implementation of a read-write lock that can be used for Reader-Writer synchronization. See the man pages for the following functions:

- `pthread_rwlock_init()`
- `pthread_rwlock_rdlock()`
- `pthread_rwlock_wrlock()`
- `pthread_rwlock_unlock()`
- `pthread_rwlock_destroy()`

4 About Peterson's algorithm

Algorithm 1 is a modified version of Peterson's algorithm that sets the `not_turn` variable before the boolean `wants`.

Algorithm 1 Peterson's algorithm for thread T_i

Global Variables:

```
1:  bool wants[2] = {false, false};
2:  int not_turn; /* can be 0 or 1 */

3:  enter_CS()
4:     not_turn = i;
5:     wants[i] = true;
6:     while wants[1-i] == true and not_turn == i do
7:        /* do nothing */
8:     end while

9:  leave_CS()
10:    wants[i] = false;
```

Question 4.8: *Discuss about the correctness of this new version of the algorithm.*