Exercises

Instructions

There are two exercises.

Most of exercise 1 answer is a program written by completing the templates from the companion archive [http://cambium.inria.fr/~maranget/MPRI/EXO21.tgz](http://cambium.inria.fr/~maranget/MPRI/EXO21.tgz).

Answers should be submitted by email to Luc.Maranget@inria.fr before Monday February 14, noon. Solution to exercise 1 should compile with provided Makefile — make all for 1.1 and make c11 for 1.3.

1 Semaphores

A semaphore is an old fashioned synchronisation primitives that generalises the mutex: the semaphore is given a capacity and at most capacity threads can be in critical section simultaneously. Hence, a mutex is a semaphore with capacity 1.

For historical reasons semaphore lock is called “wait” and semaphore unlock is called “post”.

**Important:** Code template for this exercise is available in directory semaphore from the companion archive.

1.1 Coding a semaphore

Given a semaphore $s$ initialised to capacity $c$, critical sections are defined from a call to `wait_semaphore(s)` (analog of `lock_mutex`) to `post_semaphore(s)` (analog of `unlock_mutex`). The semaphore uses an internal counter `nfree` to count the number of threads allowed to enter critical section. The counter is initialised to $c$ at semaphore creation time, then:

- `wait_semaphore(s)` checks that `nfree` is non-null and decrements it. If `nfree` is null, the thread suspends.
- `post_semaphore(s)` increments `nfree` and release waiting threads.

One may write a semaphore with a mutex (to protect the modifications of `nfree`) and a condition variable (to wait on). Complete the following code:

```c
/* Signature of mutex and condition variable primitives */

pthread_mutex_t *alloc_mutex(void);
void free_mutex(pthread_mutex_t *p);
void lock_mutex(pthread_mutex_t *p);
void unlock_mutex(pthread_mutex_t *p);

pthread_cond_t *alloc_cond(void);
void free_cond(pthread_cond_t *p);
void wait_cond(pthread_cond_t *c, pthread_mutex_t *m);
void signal_cond(pthread_cond_t *c);
void broadcast_cond(pthread_cond_t *c);
```
/* Semaphore structure */
typedef struct {
    volatile int nfree ;
    pthread_mutex_t *mutex ;
    pthread_cond_t *cond ;
} semaphore_t ;

semaphore_t *alloc_semaphore(int capacity) {
    ...
}

void free_semaphore(semaphore_t *p) {
    ...
}

void wait_semaphore(semaphore_t *p) {
    ...
}

void post_semaphore(semaphore_t *p) {
    ...
}

1.2 Semaphore usage

We consider nprocs threads running function T1 below, with argument described by ctx_t below:

typedef struct {
    int size ;
    pthread_barrier_t *b ;
    semaphore_t *sem ;
} common_t ;

typedef struct {
    int id ;
    common_t *common ;
} ctx_t ;

void *T1(void *p) {
    ctx_t *p = _p ;
    common_t *q = p->common ;
    for (int k = q->size-1 ; k >= 0 ; k--) {
        wait_semaphore(q->sem) ;
        printf("+") ;
        printf("-") ;
        post_semaphore(q->sem) ;
        wait_barrier(q->b) ;
        if (p->id == 0) printf("\n") ;
        wait_barrier(q->b) ;
    }
    return NULL ;
}

With a semaphore of capacity 2, q->size = 1 and nprocs == 4. Classify the following outputs as legal or illegal, giving a short explanation in each case:

1. +++-++-
2. ++++-++-
3. -+-+-+-+
4. +-+-+-++
5. ++++++++
1.3 C11 coding

Write the same program using C11 standard primitives. To that aim, you may need:


- A C11 compiler and standard library. On Linux, if your distribution defaults are not sufficient (as it is the case on Ubuntu 18.04 LTS for instance), you can install the musl-tools package and use the musl-gcc compiler.

The companion archive contains a template `sem11.c`, with missing parts shhighlighted by TODO comments.

2 Sequentially consistent or not?

The following small programs are written in pseudo-C. Following our usual conventions x and y are shared memory locations, while r0 and r1 are registers. Moreover, *x = 1 is a store; while r0 = *x is a load. Shared locations and registers hold zero as initial value. By definition, a behaviour is a choice of final values for some observed locations. That is, shared locations x and r0 for Test 1 and Test 2; registers r0 and r1 for Test 3 and Test 4.

We consider valid behaviours, i.e. behaviours that result from executions such that each load of a memory cell reads a value written by a store to the same memory cell or the initial value zero. List all valid behaviours of the four tests, identifying sequentially consistent (SC) behaviours.