SE205 - TD2 — Énoncé

General Instructions

- You can download all source files from:
  `https://se205.wp.mines-telecom.fr/TD2/`

Compiling using C11

Although already several years old (and pretty much known years before officially standardized), C11 is not supported by all compilers available. In particular, compilers dating from before 2012 often only support small subsets. Compilers that are supposed to support C11 entirely are GCC 3.8 (or above) and Clang 3.3 (or above). Visual Studio 2013 and the Intel compiler (version 14.0) only provide partial support.

To compile programs you can then can use the following command:

```
gcc -pedantic -Werror -Wall -std=c11 -pthread -O3 -g -o <you-program> <your-c11-code>
```

Threads in C11

The C11 standard defines a set of functions to operate on threads (`threads.h`). However, many C libraries do not support it, e.g., glibC installed on most Linux machines does not (yet) support it. We will thus resort to the `pthreads` library. Note that we will not cover all functions, only those needed for the exercises of today (use `man pthreads` for more information).

```
// Thread identifiers (may vary depending on operating system).
typedef unsigned long int pthread_t;


// Create a new thread, starting with execution of START-ROUTINE getting passed
// ARG. Creation attributed come from ATTR.  The new thread identifier is stored
// in *NEWTHREAD.
//
// Simply use NULL for ARG and ATTR for the exercises.
extern int pthread_create(pthread_t *newthread,
                          const pthread_attr_t *attr,
                          void *(*start_routine) (void *),
                          void *arg);

// Make calling thread wait for termination of the thread TH.  The exit status
// of the thread is stored in *THREAD_RETURN, if THREAD_RETURN is not NULL.
//
// Simply use NULL for THREAD_RETURN for the exercises.
extern int pthread_join(pthread_t th, void **thread_return);

/* Obtain the identifier of the current thread.  */
extern pthread_t pthread_self(void);
```

1 Simple Threads (15 minutes)

Aims: *Create and operate on threads.*

- Given the example code in the lecture, write a C11 program that creates two threads, each printing a simple message – like `"Hello World!"` (using `printf`).
- Compile and run your program.
- What is `pthread_join()` actually doing here?

  Hint: Try to move code lines around, e.g., move the call to `pthread_join()` for the first thread before the creation of the second thread.

  It causes the current thread to wait for the other thread to terminate, i.e., it is a means to synchronize threads.
- Print the threads' identifiers.

  Hint: Convert the thread identifier to an `unsigned int`, then simply use `printf`.

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

// Code for thread A.
void *threadA(void *parameter)
{
  printf("Hello␣from␣Thread␣A␣with␣ID␣%u\n", (unsigned int)pthread_self());
  return NULL;
}

// Code for thread B.
void *threadB(void *parameter)
{
  printf("Hello␣from␣Thread␣B␣with␣ID␣%u\n", (unsigned int)pthread_self());
  return NULL;
}

// Create two threads, printing a simple message and their respective IDs.
// Always returns 0.
int main()
{
  // create the two threads.
  pthread_t a,b;
  pthread_create(&a, NULL, &threadA, NULL);
  pthread_create(&b, NULL, &threadB, NULL);

  // wait for both threads to terminate:
  // otherwise an error occurs: Threads a and b are (potentially) destroyed
  // before their respective threads terminate.
  pthread_join(a, NULL);
  pthread_join(b, NULL);

  return 0;
}
```

Atomics in C11

C11 provides type wrappers (`_Atomic` and `_atomic`) and functions (prefixed by `atomic_`), which provides mechanisms to precisely specify the meaning of memory accesses to shared variables:

```
// Load a value from an atomic object OBJ.
C atomic_load(const volatile A* obj);

// Load a value from an atomic object OBJ, using the provided memory order
// ORDER.
C atomic_load_explicit(const volatile A* obj, memory_order order);


// Swaps the value of an object OBJ with an other value DESIRED.
C atomic_exchange(volatile A* obj, C desired);

// Swaps the value of an object OBJ with an other value DESIRED, using the
// provided memory order ORDER.
C atomic_exchange_explicit(volatile A* obj, C desired, memory_order order);


// Perform an atomic compare and exchange operation: (1) the value of OBJ is
// compared with the value EXPECTED; (2a) if they match, the value of DESIRED
// is copied to OBJ; (2b) if they do not match, the current value of OBJ is
// copied to EXPECTED.
//
// In case (2a) the operation is said to be successful, while in the other case
// the operation is said to have failed.
//
// The return value of the function indicates whether the operation failed or
// succeeded.
_Bool atomic_compare_exchange_strong(volatile A* obj, C* expected, C desired);

// Same as before, except that the provided  memory orders are used in case of
// a success (SUCC) or failure (FAIL).
_Bool atomic_compare_exchange_strong_explicit(volatile A* obj,
                                              C* expected, C desired,
                                              memory_order succ,
                                              memory_order fail);


// Atomically increment the value of OBJ by adding ARG to it.
C atomic_fetch_add(volatile A* obj, M arg);

// Atomically increment the value of OBJ by adding ARG to it, using the provided
// memory order ORDER.
C atomic_fetch_add_explicit(volatile A* obj, M arg, memory_order order);

// Similar functions for other operations (subtraction, logical or, logical xor,
// logical and) exist).


// Establish an ordering between non-atomic accesses in a program, according to
// the memory order ORDER.
void atomic_thread_fence(memory_order order);
```

You can find more information on the C11 atomics library, including examples, on the follow website: `http://en.cppreference.com/w/c/atomic`.

2 Array with two Producers (20 minutes)

Aims: *Reason about races conditions and thread interleavings.*

- Compile and run the program from below.
- Is the program race-free? Explain your answer in detail.

  Hint: The `counter` variable plays a key role for this question.

  Yes. Accesses to the atomic counter variable are race-free by definition. Due to the atomicity of the counter operation, every iteration in the while loop of the threads will have a unique index. All accesses to the shared array are thus disjoint and cannot cause any race.
- Which kind of initerleavings are possible for this program? Which interleavings can you observe? What might be happening here?

  Hint: Vary the `SIZE` of the shared array.

  Theoretically all kinds of interleavings are possible: all entries could belong to only one thread, or any number x of entries could belong to one thread and the remaining `SIZE` - X elements to the other under any possible permutation.
  Initially only one thread runs (while the second is being created). Thus, for small `SIZE`, only this single thread fills the array entirely before the other thread even started. This changes only at a size of several hundred entries.

```
#include <stdatomic.h>
#include <stdio.h>
#include <pthread.h>

#define SIZE 10

// A shared array of unsigned integers (non-atomic, no memory model specified)
static unsigned int x[SIZE] = {0,};

// A shared counter (atomic, with memory order sequential consistency)
static atomic_uint counter = 0;

// The same code for all threads.
void *thread(void *parameter)
{
  while(counter < SIZE)
  {
    x[atomic_fetch_add(&counter, 1)] = (unsigned int)pthread_self();
  }
  return NULL;
}

// Create and start two threads and see how their executions interleave.
// Always returns 0.
int main()
{
  // create the two threads
  pthread_t a,b;
  pthread_create(&a, NULL, &thread, NULL);
```

```
  pthread_create(&b, NULL, &thread, NULL);

  // wait for both threads to finish
  pthread_join(a, NULL);
  pthread_join(b, NULL);

  // print the contents of the shared array
  for(unsigned int i = 0; i < SIZE; i++)
  {
    printf("x[%u] = %x\n", i, x[i]);
  }

  return 0;
}
```

3 Total Store Order (20 minutes)

Aims: *Understand the importance of memory models and their impact.*

- Compile and run the program from below.
- Will this program ever terminate? What do you observe when you run the program (maybe try running it several times)?

  Hint: You saw this example in the lecture. Could this be an artifact of the memory model of the computer architecture of your PC (x86)?

  This depends on the memory model. Under the sequentially consistent model, this program does not terminate. However, since the variables are not atomic, sequentially consistent is not guaranteed. What we observe here is the TSO-model of the x86 architecture (PC) – most likely caused by a store buffer.
- Call `atomic_thread_fence()` after the assignments to `x` and `y` for both threads. What happens now?

  Hint: Read up what a memory fence or memory barrier is (`http://en.cppreference.com/w/c/atomic`)

  The fence ensures that the ordering of the memory accesses executed by each thread is respected. The program now does not terminate anymore.
- Is the program race-free? Explain your answer in detail.

  No. But, well that was not the purpose of this exercise.
- What does this example demonstrate?

  That programmers have to be aware of memory models, otherwise they might produce wrong code and introduce bugs that are extremely difficult to reproduce.

```
#include <stdatomic.h>
#include <stdio.h>
#include <pthread.h>
```

```
// Two shared variables (neither atomic, nor with a well specified memory order)
static int x = 0;
static int y = 0;

// A shared counter (atomic, with memory order sequential consistency)
static atomic_int counter = 0;

// First thread, accessing shared variables
void *threadA(void *parameter)
{
  x = 1;
  if (y == 0)
    counter++;

  return NULL;
}

// Second thread, accessing shared variables
void *threadB(void *parameter)
{
  y = 1;
  if (x == 0)
    counter++;

  return NULL;
}

// Create and start two threads over and over again and check the outcome.
// Always returns 0.
int main()
{
  unsigned int i = 0;
  while(++i)
  {
    // create the two threads
    pthread_t a,b;
    pthread_create(&a, NULL, &threadA, NULL);
    pthread_create(&b, NULL, &threadB, NULL);

    // wait for both threads to finish
    pthread_join(a, NULL);
    pthread_join(b, NULL);

    // Can this happen? Should this happen?
    if (counter == 2)
    {
      printf("He!␣(%d)\n", i);
      return 0;
    }

    // reset shared variables before restarting the experiment.
    x = 0;
    y = 0;
    counter = 0;
  }

  return 0;
}
```

4 Implementing a Mutex (40 minutes)

Aims: *Use atomic variables to synchronize threads.*

Someone implemented a simple linked list and tested the function to insert an element at the head of the list. The tests showed that the code (see below) is not working as expected when several threads operate on the same list at the same time.

There are various problems in the code. First of all, variable `x`, holding the list, is not atomic. This means that the program contains races, e.g., due to accesses to the variable. Even if `x` were atomic, the program would not work as expected. The problem is that the condition to test whether $SIZE$ elements have been inserted and the actual insertion are not atomic.

Clearly, a mechanism is needed to make the following operations atomic:

- All accesses to `x`.
- The evaluation of the test condition (including the call to `my_list_size()`).
- The actual insertion (including the call to `my_list_insert()`).

A mutex allows us to do exactly this. More precisely, it allows us to control whether a thread can execute a piece of code. In order to provide this functionality a mutex offers two functions: (1) `lock()`, which allows a thread to acquire the lock, and (2) `unlock()`, which releases the lock again. When properly used the mutex guarantees that at most one thread can acquire the mutex at any moment. When a thread calls `lock()` the function only returns when the mutex was successfully acquired. If the mutex was acquired by another thread before, `lock()` waits as long as needed until it can acquire the mutex, which of course means that the other thread has to call `unlock()` at some point.

One way to implement a mutex is to use two counters (e.g., as a record): `ticket` and `turn`. The `lock()` function atomically increments `ticket`, while retrieving and storing the previous value in a local variable. It then waits until the `turn` counter reaches the value of the local variable. The `unlock()` function increments the `turn` counter.

- Implement a simple mutex as a record holding the two counters `ticket` and `turn`. Then implement the two functions `my_mutex_lock()` and `my_mutex_unlock()` using the C11 atomic primitives.

  Hint: Use the `atomic_fetch_add()` function.

- Uses your mutex to correctly synchronize the two threads in the program from below. Compile and run the program. Verify that your mutex actually works.

- Which guarantees have to be provided by the memory model with regard to the two counters in order for the mutex algorithm to be correct?

  The increment operation on `ticket` has to be atomic (or course). This implies that the counter values become visible in a specific ordering. The increment of `turn` in theory is not required to be atomic (since at most one thread may acquire the mutex). The memory model has to ensure, however, that the increments of the counter become visible to all threads in a coherent and consistent manner – a memory fence would suffice here for instance.

- Try to change your implementation such that the `turn` counter is *not* atomic, but only a normal variable. Is this safe?

  Hint: How many threads will execute the increment of `turn`?

- Change your code to use `atomic_compare_exchange_strong()` instead of `atomic_fetch_add()`.

  Hint: Store the current counter value in a local variable, then pass the variable's original value and its value incremented by `1` to `atomic_compare_exchange_strong()`. If that fails (return value) start another attempt to increment.

- What is the problem with this implementation of a mutex?

  Hint: Look at the system utilization when you run your program, e.g., by invoking the command `htop` on the command line.

  The threads always execute some operation (this is called busy waiting). If the number of threads is large, this may quickly overload the system as most threads are just spinning in order to wait instead of doing useful computations. This also puts a high load on the memory (and the bus leading there).

```
#include <stdatomic.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// A simple mutex implementation using two counters ticket and turn.
//
// When a thread locks the mutex the ticket counter is incremented atomically.
// and the thread waits until the turn variable reaches the previous value of
// the ticket counter.
//
// The unlock operation only increments the turn counter. This is not needed to
// be atomically (since only one thread my acquire the lock at any moment). In
// order to ensure a coherent and consistent view of all threads on the
// counter's value a memory fence is introduced.
typedef struct
{
  // An atomic counter. We are using unsigned int because its behavior is well
  // defined on an overflow.
  atomic_uint ticket;

  // A non-atomic counter. We are using unsigned int because its behavior is
  // well defined on an overflow.
  volatile unsigned int turn;
} my_mutex_t;


// initialize the counters of a mutex.
void init_my_mutex(my_mutex_t *m)
{
  m->ticket = 0;
  m->turn = 0;
}

// Perform an "atomic" increment of ticket using compare-and-swap instruction
// and return the previous value of ticket.
// this is equivalent to ticket.fetch_add(1);
```

```
static unsigned int my_inc(my_mutex_t *m)
{
  unsigned int tmp = atomic_load(&m->ticket);
  while(!atomic_compare_exchange_strong(&m->ticket, &tmp, tmp + 1))
    ; // do nothing

  return tmp;
}

// Acquire the mutex, i.e., atomically increment ticket and wait for turn to
// reach the previous value of ticket.
void my_mutex_lock(my_mutex_t *m)
{
  // atomically increment the ticket counter, i.e., get a unique number.
  unsigned int tmp = my_inc(m);
  while(tmp != m->turn)
    ; // do nothing
}

void my_mutex_unlock(my_mutex_t *m)
{
  m->turn++;
  atomic_thread_fence(memory_order_seq_cst);
}

//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************

static const unsigned int SIZE = 2000;

typedef struct my_list_entry_t
{
  unsigned int value;
  struct my_list_entry_t *next;
} my_list_entry_t;

// A shared list of thread IDs.
my_list_entry_t *x = NULL;

// An atomic counter variable used to track the number of list insertions
// executed.
atomic_uint insertion_counter = 0;

// insert a new value at the head of the linked list.
my_list_entry_t *my_list_insert(my_list_entry_t *head, unsigned int value)
{
  // allocate new list entry
  my_list_entry_t *new_entry;
  new_entry = (my_list_entry_t *)malloc(sizeof(my_list_entry_t));

  // initialize new list entry
  new_entry->next = head;
  new_entry->value = value;

  // track number of insertions
  atomic_fetch_add(&insertion_counter, 1);

  // return new entry as new first element of the list
```

```
  return new_entry;
}

// count the number of elements in the linked list.
unsigned int my_list_size(my_list_entry_t *head)
{
  unsigned int counter = 0;
  while(head != NULL)
  {
    counter++;
    head = head->next;
  }

  return counter;
}

// A simple mutex ensuring mutual exclusive access to list x.
my_mutex_t m;

// The same code for all threads.
void *thread(void *parameter)
{
  unsigned int done = 0;
  while(!done)
  {
    my_mutex_lock(&m);
    if (my_list_size(x) < SIZE)
    {
      x = my_list_insert(x, (unsigned int)pthread_self());
    }
    else
    {
      done = 1;
    }
    my_mutex_unlock(&m);
  }

  // should never be reached
  return NULL;
}

// Create and start two threads and observe the interleavings, synchronizing
// accesses to a shared list using a home-brew mutex implementation.
// Always returns 0.
int main()
{
  // create the two threads
  pthread_t a,b;
  pthread_create(&a, NULL, &thread, NULL);
  pthread_create(&b, NULL, &thread, NULL);

  // wait for both threads to finish
  pthread_join(a, NULL);
  pthread_join(b, NULL);

  // print the contents of the shared lis
  unsigned int i = 0;
  while(x)
  {
    printf("x[%u] = %x\n", i++, x->value);
```

```
    x=x->next;
  }

  printf("insertions: %u (%u)\n", insertion_counter, SIZE);

  return 0;
}
```