Exam – Course 2.37.1 – Programming shared memory multicore machines
March 3, 2021

Important: Use separate sheets for Part I and Part II.

Part I

This part of the exam includes two independent exercises. Both focus on deterministic parallel programming in an ML-style functional language. We assume that this language has been extended with the functions from the Seq module whose interface is provided in figure 1.

Parallel Radix Sort

The goal is to implement a parallel radix sort on integers. Such a procedure sorts a sequence of sequences according to the lexicographic ordering. As we shall see, it can be implemented in a more efficient way than a general comparison sort. Moreover, in the case of a sequence of machine integers (seen as lists of bits), the lexicographic ordering coincides with the usual ordering, and thus a radix sort can replace slower general-purpose sorts (such as merge sort, quicksort, etc.).

We assume in this exercise that all the numbers we deal with are nonnegative and representable using $W$ bits, for some fixed $W$. We designate by $a_i$ the $i$th bit of $a$, with $i \in [0, W]$. Furthermore, we assume that $a_0$ is the least significant bit of $a$ and $a_{W-1}$ its most significant bit. Given two numbers $x$ and $y$, we write that $x$ is lexicographically smaller than $y$, denoted $x \leq_{\text{lex}} y$, iff there exists $m \in [0, W]$ such that $x_m \leq y_m$ and for all $i \in [m+1, W]$ we have $x_i = y_i$.

For each question below, you will provide ML code using the functions from module Seq from figure 1. You should specify the work and span complexity of your code, which will depend on your use of parallel sequences. All your programs should satisfy internal determinism and be as asymptotically efficient as possible. You are strongly encouraged to reuse code from one question to the next.

Question 1. Write a function extract_bit : int seq -> int -> bool seq which, applied to a sequence of numbers $s$ of size $n$ and an integer $k$, returns a boolean sequence $s'$ of size $n$ such that $s'_i$ corresponds to the $k$th bit of $s_i$. For example, extract_bit [5; 7; 3; 1; 4; 2; 7; 2] must evaluate to [true; true; true; true; false; false; true; false].

Answer:

let extract_bit s k = Seq.map (fun x -> (x lsr k) land 1 = 1) s

Question 2. Write a function init : int -> (int -> 'a) -> 'a seq such that init n f returns the sequence of length $n$ whose element at index $i$ is $f i$. For example, init 4 (fun x -> 4 * x) must evaluate to [0; 4; 8; 12].

Answer:

let init n f = Seq.map f (Seq.psum (Seq.make 1 n))

Question 3. Write a function permute : 'a seq -> int seq -> 'a seq which, applied to a sequence $s$ of size $n$ and a sequence of numbers $p$ representing a permutation of $[0, n]$, returns the sequence $s'$ whose $i$th element is $s_j$ where $j$ is the unique index such that $p_j = i$. For example, permute [5; 7; 3; 1; 4; 2; 7; 2] [3; 4; 5; 6; 0; 1; 7; 2] must evaluate to [4; 2; 2; 5; 7; 3; 1; 7].
module Seq : sig

(** [length s] returns the length of [s]. Runs in constant time. *)
val length : 'a seq -> int

(** [make x n] is the sequence of length [n] whose elements are all [x]. *)
val make : 'a -> int -> 'a seq

(** [read s i] returns the [i]th element of [s]. Runs in constant time. *)
val read : 'a seq -> int -> 'a

(** [write s i x] mutates [s] to set its [i]th element to [x]. Runs in constant time. *)
val write : 'a seq -> int -> 'a -> unit

(** [map f s] applies in parallel [f] to all the elements of [s]. Runs in constant span and linear work. *)
val map : ('a -> 'b) -> 'a seq -> 'b seq

(** [psum s] returns the sequence [s'] whose element at index [i] is the sum of the first [i] elements of [s], e.g., its first element is always [0]. Runs in constant span and linear work. *)
val psum : int seq -> int seq

(** Similar to [map]. All sequences should be of the same length. *)
val map2 : ('a -> 'b -> 'c) -> 'a seq -> 'b seq -> 'c seq

(** Similar to [map2]. All sequences should be of the same length. *)
val map3 : ('a -> 'b -> 'c -> 'd) -> 'a seq -> 'b seq -> 'c seq -> 'd seq
end

Figure 1: A parallel-sequence module for ML
Answer:

```ocaml
let permute s p =  
  let len = Seq.length s in  
  assert (len > 0 && Seq.length p = len);  
  let s' = Seq.make (Seq.read s 0) len in  
  ignore (init (fun i -> write s' (read p i) (read s i)) len);  
  s'
```

**Question 4.** Write a function \( \text{rev} : \text{seq} \rightarrow \text{seq} \) which reverses a sequence. For example, \( \text{rev} \, [0; 2; 1; 3] \) must evaluate to \( 3; 1; 2; 0 \).

Answer:

```ocaml
let rev s =  
  let n = Seq.length s in  
  init (fun i -> Seq.read s (n - i - 1)) n
```

**Question 5.** Write a function \( \text{splitp} : \text{bool seq} \rightarrow \text{int seq} \) which, applied to a boolean sequence \( f \) of length \( n \), returns a permutation (in the sense of \( \text{permute} \)) of \([0, n]\) which sends the indices \( i \) such that \( f_i \) is false to the left of those for which \( f_i \) is true. The initial ordering within these two classes of elements must be preserved. For example, \( \text{splitp} \, [true; true; true; true; false; false; true; false] \) must evaluate to \( [3; 4; 5; 6; 0; 1; 7; 2] \).

Answer:

```ocaml
let splitp flags =  
  let n = Seq.length b in  
  let enumerate flags = Seq.(psum (map int_of_bool f)) in  
  let back_enumerate flags = rev (enumerate (rev flags)) in  
  let i_lo = enumerate (Seq.map not flags) in  
  let i_hi = Seq.map (fun x -> n - x - 1) (back_enumerate flags) in  
  Seq.map3 (fun c lo hi -> if c then hi else lo)
```

**Question 6.** Write a function \( \text{radix_sort} : \text{int seq} \rightarrow \text{int seq} \) which sorts a sequence of numbers. For example, \( \text{radix_sort} \, [5; 7; 3; 1; 4; 2; 7; 2] \) must evaluate to \( [1; 2; 2; 3; 4; 5; 7; 7] \).

Answer:

```ocaml
let radix_sort s =  
  let rec loop s i =  
    if i >= W  
    then s  
    else loop (permute s (splitp (extract_bits s i))) (i + 1)  
  in  
  loop s 0
```

**Cost semantics**

In this exercise, we study the semantics of a language equipped with primitives constituting idealized versions of the \texttt{Seq} module. The goal is to define a cost semantics, following the principles seen during the lectures.
Syntax. The language obeys the following grammar.

\[ M, N, P ::= \]
\[ | \ c \quad \text{constant} \]
\[ | \ \text{op}(M_1, \ldots, M_n) \quad \text{operator} \]
\[ | \ x \quad \text{variable} \]
\[ | \ \text{fun} \ x. M \quad \text{anonymous function} \]
\[ | \ MN \quad \text{application (sequential!)} \]
\[ | \ \text{length}(M) \quad \text{Seq.length } M \]
\[ | \ M[N] \quad \text{Seq.read } M \ N \]
\[ | \ M[N] \leftarrow P \quad \text{Seq.write } M \ N \ P \]
\[ | \ M^N \quad \text{Seq.make } M \ N \]
\[ | \ [M \text{ for } x \text{ in } N] \quad \text{Seq.map } (\text{fun } x \rightarrow M) \ N \]

\[ c ::= \]
\[ | \ () \quad \text{unit} \]
\[ | \ \text{true} | \text{false} \quad \text{boolean} \]
\[ | \ i \quad \text{integer} (i \in \mathbb{Z}/\mathbb{Z}_W) \]

Operators. We willingly omit the exact list of arithmetical and logical operators. It may include any operator that can be implemented within constant time over booleans or fixed-size integers: addition, subtraction, bit shifts, comparisons, boolean negation, etc.

**Question 7.** Define an inductive evaluation judgment \( M \Downarrow V; w; s \) specifying that the closed term \( M \) evaluates to the value \( V \) with work \( w \) and span \( s \), where \( w \) et \( s \) are natural numbers. You will in particular define the syntactic category of values. Work and span must match those specified in figure 1. *Do not handle the construction \( M[N] \leftarrow P \).*

**Answer:** Take as values \( V ::= c | \text{fun} \ x. M | [V_1, \ldots, V] \).

\[
\begin{align*}
& c \Downarrow c; 0; 0 \quad \cdots \quad M \Downarrow V; w; d & N \Downarrow n; w'; d' \\
& M^N \Downarrow V_{1\leq i\leq n}; w + w' + 1; d + d' + 1 \\
& M[N] \Downarrow [V_1, \ldots, V_n]; w; d & (M[V_i/x] \Downarrow V_i'; w_i; d_i)_{1\leq i\leq n} \\
& [M \text{ for } x \text{ in } N] \Downarrow [V'_1, \ldots, V'_n]; w + 1 + \sum_{1\leq i\leq n} w_i; d + 1 + \max_{1\leq i\leq n} d_i
\end{align*}
\]

**Question 8.** What problem arises when trying to add the \( M[N] \leftarrow P \) construct to your cost semantics? Can you sketch a solution? Your answer should not be longer than one paragraph.

**Answer:** The problem is that writes mandate the use of some kind of shared memory (heap) in the semantics, and it is not clear how to thread this heap when specifying parallel constructs. One possible solution is to define a partial “separation” operator on heaps and use it to give split and then recombine the heap in \([M \text{ for } x \text{ in } N]\) and \(M^N\).
Part II

**Note:** This part is made of two independent exercises.

**Sequentially consistent or not?**

The following small programs are written in pseudo-code. Following our usual conventions, \( x \) and \( y \) are shared memory locations, while \( r0 \) and \( r1 \) are registers. Moreover, \( x \leftarrow 1 \) is a store; while \( r0 \leftarrow x \) 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 \( y \) for Test 1 and Test 2; registers \( r0 \) and \( r1 \) for Test 3 and Test 4.

Valid behaviours result (1) from a choice of matching each read event with a write event or initial value and (2) from a choice for ordering writes by the program to the same location. That is, the read-from and coherence relations (\( rf \) and \( co \)) are selected (cf. lesson 03, slides 12 and 13). Also observe that, for a given location \( x \), the initialisation of \( x \) to zero is represented as an initial write event to \( x \) that \( co \)-precedes all the writes to \( x \) performed by the program.

**Question 9.** List all valid behaviours of the programs of Figure 2. No justification is required.

**Answer:**

<table>
<thead>
<tr>
<th>Test 1</th>
<th>Test 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T0 )</td>
<td>( T0 )</td>
</tr>
<tr>
<td>( T1 )</td>
<td>( T1 )</td>
</tr>
<tr>
<td>( x \leftarrow 1 )</td>
<td>( x \leftarrow 1 )</td>
</tr>
<tr>
<td>( y \leftarrow 2 )</td>
<td>( x \leftarrow 2 )</td>
</tr>
<tr>
<td>( y \leftarrow 1 )</td>
<td>( y \leftarrow 1 )</td>
</tr>
<tr>
<td>( \text{Observe } x,y )</td>
<td>( \text{Observe: } x,y )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test 3</th>
<th>Test 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T0 )</td>
<td>( T0 )</td>
</tr>
<tr>
<td>( T1 )</td>
<td>( T1 )</td>
</tr>
<tr>
<td>( x \leftarrow 1 )</td>
<td>( x \leftarrow 1 )</td>
</tr>
<tr>
<td>( r0 \leftarrow y )</td>
<td>( r0 \leftarrow x )</td>
</tr>
<tr>
<td>( y \leftarrow 1 )</td>
<td>( y \leftarrow 1 )</td>
</tr>
<tr>
<td>( r1 \leftarrow x )</td>
<td>( r1 \leftarrow y )</td>
</tr>
<tr>
<td>( \text{Observe } r0,r1 )</td>
<td>( \text{Observe } r0,r1 )</td>
</tr>
</tbody>
</table>

For the first two tests, it suffices to select the write performed by the program that comes last in the coherence order, for locations \( x \) and \( y \). For the last two tests, it suffices to match each read events with a write event to its location, including initial write events.

**Question 10.** Give all the non-sequentially consistent behaviours of programs. That is, give the *valid* behaviours that *cannot* occur as as result of the programs executing on a sequentially consistent machine. No justification is required.
Answer:

Test 1 \{x=1, y=1\}
Test 2
Test 3 \{r0=1, r1=0\}
Test 4

And that’s all as no justification is required. Nevertheless, if you need some, here are a few.

Test 2 and Test 4 have no non-SC behaviours, one may list all choices of \textit{rf} and \textit{co} and see that there is no cycle of \textit{po}\textit{rf}\textit{co}\textit{fr}. For instance, here are the four diagrams that correspond to the four valid behaviours of Test 2. One notices the absence of cycle.

One can also find four interleavings of events that yield all valid behaviours. Here they are for Test 4:

\[
\begin{align*}
\text{Wx1, Wy1, Rx1, Ry1} & \rightarrow r0=1, r1=1 \\
\text{Rx0, Wx1, Wy1, Ry1} & \rightarrow r0=0, r1=1 \\
\text{Wx1, Rx1, Ry0, Wy1} & \rightarrow r0=1, r1=0 \\
\text{Rx0, Ry0, Wx1, Wy1} & \rightarrow r0=0, r1=0
\end{align*}
\]

Test 1 has one non-SC behaviour \{x=1, y=1\}, all other behaviours being SC. Following the interleaving semantics one may argue that the scheduling order of events must start either by \textit{x} \leftarrow 1 or by \textit{y} \leftarrow 1. As a result \textit{x} and \textit{y} cannot both be equal to 1, as the other write to either \textit{x} or \textit{y} (of value 2) will be scheduled afterwards and will override the value 1.

The behaviour \{y=1, x=0\} of Test 3 is non-SC, as can be demonstrated by the following cyclic diagram:

Ticket Locks

Note: Look at the appendix for description of the library primitives you may need. From our point of view, question 14 is difficult. You may consider answering at the end.

Ticket locks are mutual exclusion devices inspired from the ticket queue management system found at administration offices, train station, etc. There is some type of dispenser from which customers pull sequentially numbered tickets upon arrival — see right of figure 3. Once they have their ticket, customers look at some dynamic sign that displays the ticket number (customer) being served — see left of figure 3. Each time the next ticket number (customer) is ready to be served, the “Now Serving” sign is incremented and the number called out. Ticket management systems provide a fairness guarantee: customers being served by increasing ticket numbers, customers are served by following the order of ticket distribution.
Back to programming, ticket locks are intended to provide the same functionality and interface as POSIX locks (type `pthread_mutex_t`) with a better fairness.

Here is tentative C-code for ticket locks. The type of ticket locks is a two field record, `next_ticket` being the number of the ticket available at the machine, and `now_serving` being the number of the ticket being served.

```c
typedef struct {
    volatile unsigned next_ticket;
    volatile unsigned now_serving;
} ticket_lock_t;

ticket_lock_t *alloc_ticket_lock(void) {
    ticket_lock_t *p = malloc_check(sizeof(*p));
    p->now_serving = p->next_ticket = 0;
    return p;
}
```

As shown by the creation function `alloc_ticket_lock`, the initial value of both fields is zero.

Critical sections are delimited by calls to the `lock_ticket` and `unlock_ticket` functions. The `lock_ticket` function is called by threads that attempt to enter a critical section.

```c
void lock_ticket(ticket_lock_t *p) {
    unsigned my_ticket = p->next_ticket; // Get ticket
    p->next_ticket = my_ticket+1; // Increment ticket
    // Wait as long my ticket is not the one being served.
    while (my_ticket != p->now_serving);
}
```

When a thread arrives, it obtains and then increments the queue ticket (field `p->next_ticket`). It then compares its ticket value (local variable `my_ticket`) with the value being served (field `p->now_serving`). If they are the same, the thread is permitted to enter the critical section and the `lock_ticket (...)` call returns. If they are not the same, then another thread must already be in the critical section and this thread busy-waits.

The `unlock_ticket` function is called by threads leaving a critical section. Its code is rather simple:

```c
void unlock_ticket(ticket_lock_t *p) { p->now_serving++; }
```

The thread leaving the critical section controlled by the lock increments the number of the ticket being served. This permits the next waiting thread, the one with the next sequential ticket number, to enter the critical section. In the following, you can neglect effects due to machine integers being of fixed size.

We now consider the test program of figure 4: `tck` is a shared pointer to a ticket lock and `x` is a shared variable of type `int`.

**Question 11.** Some attempts to run the program of figure 4 apparently last forever. Give an explanatory scenario.

**Answer:**
```c
volatile int x = 0;
ticket_lock_t *tck;

void *T(void *p) {
    ticket_lock(tck); x++; ticket_unlock(tck);
    return NULL;
}

int main(int argc, char **argv) {
    tck = alloc_ticket_lock();
    pthread_t th1,th2;
    create_thread(&th1,T,NULL); create_thread(&th2,T,NULL);
    join_thread(&th1); join_thread(&th2);
    printf("x=%i\n",x);
}
```

Figure 4: A simple program for testing tickets locks.

1. The threads attempt to enter critical section at the same time. They both obtain the ticket number 0.
2. Thread number 1 stalls, while thread number 2 passes the loop, enters critical section, calls `ticket_unlock` and terminates. The value of `tck->now_serving` now is 1.
3. Thread number 1 restarts and is stuck in the busy-wait loop, as its local `my_ticket` is 0.

**Question 12.** Correct `lock_ticket`, so that the test program always prints 2 on a SC machine.

**Answer:** One uses a gcc builtin to perform atomic increment of the `next_ticket`. Then, up to wraparound\(^1\), each thread that attempt to enter the critical section gets a fresh ticket value.

```c
void lock_ticket(ticket_lock_t *p) {
    // Get and increment ticket atomically
    unsigned my_ticket = __sync_fetch_and_add(&p->next_ticket,1);

    // Wait as long my ticket is not the one being served.
    while (my_ticket != p->now_serving);
}
```

**Question 13.** Argue that no similar correction of `unlock_ticket` is called for.

**Answer:** The `unlock_ticket` function need not use an atomic primitive to increment the `now_serving` field, because the thread that calls the function is in critical section. Furthermore the critical section ends when the `now_serving` field is written. As a result, no other thread will ever write to the `now_serving` field before it has read its incremented value, or a even larger one.

**Question 14.** Difficult question. We now run the corrected program on a ARM machine, whose memory model is weaker than SC. The program sometimes print 1. Describe a non-SC execution that can explain this outcome.

**Answer:** We give a possible scenario, where both threads read 0 from `x`. By “scenario” we mean a choice of value read. Then the final value of `x` necessarily is 1.

\(^1\)More precisely, two threads can get the same ticket, when the number of threads attempting to enter the critical section exceeds the capacity of field `next_ticket`, e.g. \(2^{32}\) when `unsigned` are 32 bits integers. An unlikely event.
Now consider the second thread, it gets the second ticket (number 1), enters critical section (that is reads 1 from the now_serving field (read $R_2$) and reads the initial value 0 from x (read event $R_1$). We thus have a non-SC cycle:

$$W_1 \xrightarrow{po} W_2 \xrightarrow{rf} R_2 \xrightarrow{po} R_1 \xrightarrow{fr} W_1$$

(This cycle is the only one of the above scenario). Moreover, there is no fence between $W_1$ and $W_2$, on the one hand and a control dependency from $R_2$ to $R_1$ on the other hand. This MP+po+ctrl idiom is allowed on ARM. Notice that the absence of a fence between the two writes suffices to allow the test. Observe that this faulty scenario stems from the lack of import (at the end of lock_ticket) and export (at the beginning of unlock_ticket) barriers.

**Question 15.** Correct lock_ticket and unlock_ticket, so that the test program will always print 2 on any shared memory machine that features a SC-restoring strong fence instruction, available as the call __sync(). The focus will be on correctness, not on efficiency.

**Answer:** Following lesson 02, it suffices to be sure that a strong fence is executed between any two accesses to different shared locations. Being cautious implies inserting __sync() everywhere.

```c
void lock_ticket(ticket_lock_t *p) {
    // Get and increment ticket atomically
    __sync();
    unsigned my_ticket = __sync_fetch_and_add(&p->next_ticket,1);
    __sync();
    // Wait as long my ticket is not the one being served.
    while (my_ticket != p->now_serving);
    __sync();
}

void unlock_ticket(ticket_lock_t *p) {
    __sync();
    p->now_serving++;
    __sync();
}
```

In practice one can insert less strong fences:

```c
void lock_ticket(ticket_lock_t *p) {
    // Get and increment ticket atomically
    // NB: __sync_fetch_and_add executes an initial fence
    unsigned my_ticket = __sync_fetch_and_add(&p->next_ticket,1);
    // NB: __sync_fetch_and_add executes a final fence.
    // Anyway this __sync can be supressed
    // without hindering mutual exclusion, because reading stall value
    // means not entering the critical section.
    while (my_ticket != p->now_serving);
    __sync(); // import barrier, can be lighter depending on arch
}

void unlock_ticket(ticket_lock_t *p) {
    __sync(); // export barrier, can be lighter depending on arch
    p->now_serving++;
    // __sync(); useless, if all primitives start with __sync
}
```

**Question 16.** So as to avoid any weak-memory model issue and high processor usage induced by systematic busy wait, we now consider building ticket locks on top of POSIX threads. Write such an implementation by using POSIX mutexes and condition variables. The focus will be on correctness, not on efficiency.

**Answer:**
typedef struct {
    pthread_mutex_t *mtx;
    pthread_cond_t *cond;
    volatile unsigned now_serving; // Display above counter
    volatile unsigned next_ticket; // Ticket machine showing.
} ticket_lock_t;

ticket_lock_t *alloc_ticket_lock(void) {
    ticket_lock_t *p = malloc_check(sizeof(*p));
    p->mtx = alloc_mutex();
    p->cond = alloc_cond();
    p->now_serving = p->next_ticket = 0;
    return p;
}

void lock_ticket(ticket_lock_t *p) {
    lock_mutex(p->mtx);
    unsigned my_ticket = p->next_ticket++;
    while (my_ticket != p->now_serving) wait_cond(p->cond, p->mtx);
    unlock_mutex(p->mtx);
}

void unlock_ticket(ticket_lock_t *p) {
    lock_mutex(p->mtx);
    p->now_serving++;
    broadcast_cond(p->cond);
    unlock_mutex(p->mtx);
}
Appendix: functions you may need

Those are some the functions that we used during lessons. Most of those are simple wappers that call more basic POSIX primitives. In particular, the wappers perform error checking, crashing the program at first error.

“Atomic” builtins

The “functions” below provide access to machine-level primitives.

```c
/* Strong fence (lesson 02, slide 39)*/
void __sync(void);

/* Read-modify-write operations */
typ __sync_atomic_fetch_and_op(typ *ptr, typ value);
typ __sync_atomic_op_and_fetch(typ *ptr, typ value);
```

Read-modify-writes operations are given above as schemes, where `typ` is a C scalar type (`char`, `int` etc.) and `op` is the name of a basic operation (`add`, `or`, etc.). Those primitives perform the operation `*ptr = *ptr op value` atomically. The “fetch_and_op” builtins return the value of `*ptr` before the operation is performed; while the “op_and_fetch” builtins return the result of the operation. See also lesson 02, slide 4. You can assume that those builtins start and end by executing a strong fence.

Threads (lesson 01, slide 13)

```c
/* Asynchronously execute f(x) on a fresh thread.
   Thread identity is stored in th. */
void create_thread(pthread_t *th, void *(*f)(void *), void *x);

/* Wait for the thread whose identity is stored in th to terminate */
void *join_thread(pthread_t *th);
```

Mutexes (lesson 01, slide 33)

```c
/* Allocate and perform any POSIX-level initialisation */
pthread_mutex_t *alloc_mutex(void);

/* Lock and unlock */
void lock_mutex(pthread_mutex_t *p);
void unlock_mutex(pthread_mutex_t *p);
```

Condition variables (lesson 01, slide 54)

```c
/* Allocate and perform any POSIX-level initialisation */
pthread_cond_t *alloc_cond(void);

/* Suspend (wait) on c, unlocking mutex m */
void wait_cond(pthread_cond_t *c, pthread_mutex_t *m);

/* Awake threads suspended on c */
void signal_cond(pthread_cond_t *c); // One thread
void broadcast_cond(pthread_cond_t *c);// All threads
```