Exam – Course 2.37.1 – Semantics, languages and algorithms for multicore programming

March 1 2012

Instructions: only printed documents are authorised. You can admit the result of one question and move on. Leave optional questions until the end. Use one sheet for Exercises 1, 2 and 3, one for Exercise 4, and one for Exercise 5.

Our goal is to study (part of) the memory model of the *Go Programming Language* and to implement some subsets of the runtime. We follow the official documentation, taken from `http://golang.org`. The relevant excerpts are reported below in sans-serif font. Code snippets are in `type-writer font`.

For the sake of this exam, a Go program is a succession of global variables and function declarations. Global variables are declared with the `var` keyword, while functions are declared with the `func` keyword. When a program is executed, global variables are initialised and then the `main` function is invoked. The program on the left below illustrates the syntax, and running it outputs `Hello`. Concurrent tasks (called *goroutines*) are spawned with the `go` keyword followed by a function invocation. The program on the right below can thus nondeterministically output `HelloWorld` or `WorldHello`. Assignment is familiar: `a = b`. If needed, standard constructs as `if .. then .. else` or `for ..` are available.

```
var s string = "Hello"

func hello() {
  print s
}

func main() {
  hello()
}
```

```
func hello() {
  print "Hello"
}
func world() {
  print "World"
 }
func main() {
  go hello()
  go world()
}
```

Exercise 1. Ambiguity in the documentation of the Go Memory Model

The text that follows in sans-serif font is taken literally (modulo minor editing) from the official documentation. Unclear or ambiguous sentences are thus part of the exam questions.

The Go memory model specifies the conditions under which reads of a variable in one goroutine can be guaranteed to observe values produced by writes to the same variable in a different goroutine.

Within a single goroutine, reads and writes must behave as if they executed in the order specified by the program. That is, compilers and processors may reorder the reads and writes executed within a single goroutine only when the reordering does not change the behaviour within that goroutine as defined by the language specification. Because of this reordering, the execution order observed by one goroutine may differ from the order perceived by another. For example, if one goroutine executes `a = 1; b = 2;`, another might observe the updated value of `b` before the updated value of `a`.

To specify the requirements of reads and writes, we define *happens before*, a partial order on the execution of memory operations in a Go program. If event `e1` happens before event `e2`, then we say that `e2` happens after `e1`. Also, if `e1` does not happen before `e2` and does not happen after `e2`, then we say that `e1` and `e2` happen concurrently.

Within a single goroutine, the happens-before order is the order expressed by the program.

Definition 1. (allowed to observe a write).

A read r of a variable v is allowed to observe a write w to v if both of the following hold:

- `w` happens before `r`;
- there is no other write `w'` to `v` that happens after `w` but before `r`.

Definition 2. (guaranteed to observe a write).

To guarantee that a read `r` of a variable `v` observes a particular write `w` to `v`, ensure that `w` is the only write `r` is allowed to observe. That is, `r` is guaranteed to observe `w` if both of the following hold:

- `w` happens before `r`;
- any other write to the shared variable `v` either happens before `w` or after `r`.

Question 1.

In what follows you can assume that the order expressed by the program is total. Are Definition 1. and Definition 2. equivalent within a single goroutine, that is, is it always the case that if a read is *allowed* to observe a read then it is *guaranteed* to observe the read (and vicecersa)? Are them equivalent in general? If yes, justify your answer. If not, sketch a program in which they are different.

Answer: In a single co-routine, since the order expressed by the program is total, happens-before is total and the two definitions coincide. With multiple go-routines it is easy to build a program where happens-before is not total: in this case Definition 2. implies Definition 1., but not the converse.

Reads and writes of values larger than a single machine word behave as multiple machine-word-sized operations in an unspecified order.

The go statement that starts a new goroutine happens before the goroutine's execution begins. The exit of a goroutine is not guaranteed to happen before any event in the program.

Question 2.

Consider the two programs below.

```
var a string
func f() {
  print(a)
}
func main() {
  a = "hello, world"
  go f()
}
```

```
var a int
func f() {
  a = 42
}
func main() {
  go f()
  print(a)
}
```

Enumerate all the possible outputs of these two programs. *Hint*: for the program on the right, be careful about the values that a read is *allowed* to observe according to the definitions above.

Answer: The first always prints `hello world` because the assignement to `a` happens before the print statement and there are no other assignments. In the second program, the read from `a` is not related to the write to `a` by happens-before; according to Definition 1. the read instruction is *not allowed* to see the write of `42` to `a`. According to the official documentation, in the second program, an aggressive compiler might delete the entire `go` statement.

Question 3.

The documentation states that a read r may observe the value written by a write w that happens concurrently with r. In the program below it can happen that g prints 2 and then 0.

```
var a, b int
func f() {
  a = 1
  b = 2
}
func g() {
  print(b)
  print(a)
}
func main() {
  go f()
  g()
}
```

Is this behaviour consistent with the definition of what reads are allowed to observe? Why?

Answer: The update to `b` does not happen before the read from `b`. As such according to Definition 1. the print statement is not allowed to see the update with `2`. As it is, Definition 1. seems to contradict this example.

Exercise 2. Go Channels

In what follows, we will ignore Definition 1. (what reads are *allowed* to observe), while retaining Definition 2. We will also assume that the Go Memory Model is a DRF model: a Go program is undefined if it exhibits a data-race. Data-races are defined as usual: a program has a race if there exists a sequentially consistent execution with two conflicting accesses not related by happens before.

Channel communication is the main method of synchronisation between goroutines. Each send on a particular channel is matched to a corresponding receive from that channel, usually in a different goroutine.

Remark: channels can be *unbuffered*, created with e.g. `make(chan int)` for a channel over which integers can be exchanged, and *buffered*, created with e.g. `make(chan int, 10)` for a channel with buffer size `10`. Send is a statement denoted `c <- v`, where `c` is a channel name and `v` is a value. Receive is an expression denoted `<-c`, that evaluates to a value sent on the channel.

A send on a channel happens before the corresponding receive from that channel completes. A receive from an unbuffered channel happens before the send on that channel completes.

Question 4.

Enumerate all the outputs of the program on the left and of the program on the right below.

```
var c = make(chan int, 10)
var a int
func f() {
  a = 42
  c <- 0
}
func main() {
  go f()
  <-c
  print(a)
}
```

```
var c = make(chan int)
var a int
func f() {
  a = 42
  <-c
}
func main() {
  go f()
  c <- 0
  print(a)
}
```

Answer: Both programs always print `42`.

Question 5.

Is the following a correct implementation of the message passing idiom?

```
var a string
var done bool
func setup() {
  a = "hello, world"
  done = true
}
func main() {
  go setup()
  for !done {}  /* loops if done is false */
  print(a)
}
```

If not propose a variant that always prints `hello, world`.

Answer: As it is, the program is racy. To correctly implement the MP idiom, it is enough to replace the synchronisation by a pair of send/receive over a channel.

Exercise 3. (Soundness of Optimisations of the Go Programming Language)

Question 6.

If `c` is a buffered channel, is it correct to perform the following optimisations? Suppose that `r1` and `r2` are local variables, `x` is a global variable.

1. `<-c; r1 = x  =>  r1 = x; <-c`
2. `<-c; x = r1  =>  x = r1; <-c`
3. `r1 = x; c <- r2  =>  c <- r2; r1 = x`

Answer: To answer this question is it enough to observe that over a buffered channel, send has a release semantics (analogous to `unlock`), and receive an acquire semantics (analogous to `lock`). As a consequence, all the optimisations above are incorrect as they move memory accesses outside a critical section.

Question 7.

(Optional) Same question as above, but supposing that `c` is an unbuffered channel.

Answer: Over an unbuffered channel, receive has acquire semantics, and send has both a acquire and release semantics. All these optimisations are then incorrect by the same argument as above.

Exercise 4. (Unbuffered Channels)

We consider the following pthread implementation of a "channel". First, the definition of a structure and the two usual `alloc` and `free` functions:

```
typedef enum { EMPTY,FULL, } status_t ;

typedef struct {
  volatile int val ;
  volatile status_t flag ;
  pthread_mutex_t *mutex ;
  pthread_cond_t *full,*empty ;
} channel_t ;

channel_t *alloc_channel(void) {
  channel_t *p = malloc_check(sizeof(*p)) ;
  p->flag = EMPTY ;
  p->mutex = alloc_mutex() ;
```

```
  p->full = alloc_cond() ;
  p->empty = alloc_cond() ;
  return p ;
}

void free_channel(channel_t *p) {
  free_mutex(p->mutex) ;
  free_cond(p->full) ;
  free_cond(p->empty) ;
  free(p) ;
}
```

One may notice the `status_t` type, a **enum** type. You may not be familiar with **enum** types: an **enum** type basically is an integer type with a limited set of values with symbolic names — here `EMPTY` and `FULL`, meaning either that the channel holds no value, or that there is a value available.

Then, here are the send and receive functions, `send` and `recv`.

```
void send(channel_t *p, int v) {
  lock_mutex(p->mutex) ;
  while (p->flag != EMPTY) wait_cond(p->empty,p->mutex) ;
  p->val = v ;
  p->flag = FULL ;
  broadcast_cond(p->full) ;
  unlock_mutex(p->mutex) ;
}

int recv(channel_t *p) {
  int r ;
  lock_mutex(p->mutex) ;
  while (p->flag != FULL) wait_cond(p->full,p->mutex) ;
  r = p->val ;
  p->flag = EMPTY ;
  broadcast_cond(p->empty) ;
  unlock_mutex(p->mutex) ;
  return r ;
}
```

In effect, our channel structure is a simplified version of the locked FIFO of course 01 with a queue size of one. The function `send` is the analog of `put` while `recv` is the analog of `get`: a sender first wait for the channel to be empty, stores the sent value, change status to `FULL` and wake receivers up. The code for the receiver is symmetric.

Question 8.

Although buffering is minimal (queue of size one), our channel is *not* unbuffered in the sense of the Go Programming language. Explain why.

Answer: The sender does not wait for a receiver to read the sent value. As a result, the condition "*A receive from an unbuffered channel happens before the send on that channel completes.*" does not hold.

To make our channel unbuffered, we can augment the channel structure with a new state `READ` meaning that some receiver has now read the sent value and by a specific condition variable for the sender to wait for that receiver to operate. More specifically, the structure and the alloc and free functions are now as follows:

```
typedef enum { EMPTY,FULL, READ } status_t ;

typedef struct {
  volatile int val ;
  volatile status_t flag ;
  pthread_mutex_t *mutex ;
  pthread_cond_t *full,*empty,*writer ;
} channel_t ;
```

```
channel_t *alloc_channel(void) {
  channel_t *p = malloc_check(sizeof(*p)) ;
  p->flag = EMPTY ;
  p->mutex = alloc_mutex() ;
  p->full = alloc_cond() ;
  p->empty = alloc_cond() ;
  p->writer = alloc_cond() ;
  return p ;
}

void free_channel(channel_t *p) {
  free_mutex(p->mutex) ;
  free_cond(p->full) ;
  free_cond(p->empty) ;
  free_cond(p->writer) ;
  free(p) ;
}
```

Question 9.

Write `send` and `recv` that perform the unbuffered send and receive operations.

Answer: The code can be written by closely connecting status change and condition variables: there are three condition variables (`empty`, `full` and `writer`) that correspond to the three possible status value (`EMPTY`, `FULL` and `READ`). Threads will suspend on the condition variable to wait for the status to have the associated value, while thread that change the status to a specific value will signal on the associated condition variable.
For instance, the sender will suspend on `writer` until a receiver reads the value and then signals on `writer`. Also observe that the sender now sets status to `EMPTY`, as a result, the sender signals on `empty`, while the receiver performed those operations in the buffered channel.

```
1  void send(channel_t *p, int v) {
2    lock_mutex(p->mutex) ;
3    while (p->flag != EMPTY) wait_cond(p->empty,p->mutex) ;
4    p->val = v ;
5    p->flag = FULL ;
6    broadcast_cond(p->full) ;
7    while (p->flag != READ) wait_cond(p->writer,p->mutex) ;
8    p->flag = EMPTY ;
9    broadcast_cond(p->empty) ;
10   unlock_mutex(p->mutex) ;
11 }
12
13 int recv(channel_t *p) {
14   int r ;
15   lock_mutex(p->mutex) ;
16   while (p->flag != FULL) wait_cond(p->full,p->mutex) ;
17   r = p->val ;
18   p->flag = READ ;
19   broadcast_cond(p->writer) ;
20   unlock_mutex(p->mutex) ;
21   return r ;
22 }
```

A few remarks: The receiver signals on the condition variable `p->writer` using `broadcast_cond` (line 19). As one and only one (sender) thread is waiting on `p->writer`, one could have used `signal_cond` instead.
The call `wait_cond(p->writer,p->mutex)` in `send` (line 7), is inserted in a **while** loop so as to check that the channel status equals `READ` before resuming execution and setting the channel status to `EMPTY`. This loop protects the code against "*spurious wakeups*. Namely, the POSIX specification permits such wakeups, that is some thread suspended on some condition variable can wake up whithout

any other thread having signaled the condition variable. As the course was ambiguous on the point of spurious wakeups, omitting the loop was not considered a programming error, although it is one.

Exercise 5. (Goroutines)

Question 10.

The previous channel implementations operate on plain POSIX threads. In the Go language, channels are used to synchronise coroutines or concurrent tasks called "goroutines".

Describe a scenario where a functionally live program with two goroutines and one channel may deadlock with the previous channel implementation.

Answer: Sending a message blocks until the acknowledgement of the reception of this message. The previous implementation stalls the processor executing the send operation. On a single-processor system, this leaves no chance to the scheduler to activate the receiving goroutine. In fact, any unbuffered communication on a single-processor system leads to a deadlock.

Question 11.

We now assume that the scheduler for goroutines uses a shared work list implemented as a stack. Each POSIX worker thread runs the same scheduler algorithm, concurrently. The scheduler pops goroutines from the stack to be scheduled on a this POSIX thread.

We would like to revisit the implementation of `wait_cond` and `broadcast_cond`, not as blocking synchronisations, but as task pool operations displacing goroutines from the work list to a waiting task pool and back.

Propose a data structure to handle the goroutines waiting for a condition variable to be broadcasted and list the operations on this data structure involved when waiting and signalling a condition variable. Show that it is not necessary to use a concurrent data structure for this purpose.

Answer: There are multiple solutions. The simplest, most efficient one may be to attach a simple list to each condition variable. Concurrent accesses must be protected by a mutex, alleviating the need for a concurrent queue. Insertion and removal may take constant time.
`wait_cond` checkpoints the state of the calling goroutine, pops it from the work list (stack), and inserts it into the condition variable's list of waiting goroutines, it then unlocks the mutex.
`broadcast_cond` removes the goroutine from the condition variable's list and pushes it back to the work list.

Question 12.

We assume that the Go compiler operates like the Cilk one (but using a simpler, shared stack): when encountering a `go` statement, the compiled code pushes the continuation of a goroutine to the stack then calls the goroutine directly.

Show that in the absence of channel synchronisations, the maximal size of the work list is linear in the number of processors and in the height of the tree of goroutine spawns (the linear expansion principle of Cilk).

Answer: A short proof is based on the complexity analysis of greedy schedules by Blumofe and Leiserson. The abovementionned scheduling algorithm is exactly the busy leaves algorithm. This algorithm enforces the busy leaves property, which itself leads to an at most linear expansion.
The full proof can be extracted from Lemmas 1 and 2 and Theorem 2 of the paper.

Question 13.

Sketch a Go program where the maximal size of the task pool is arbitrarily larger than the height of the tree of goroutine spawns, while an alternative compilation and/or scheduling algorithm would executed in bounded memory.

Answer:

```
func chain (n, c) {
  if (n > 0) {
    var next_c = make (chan int)

    go chain (n-1, next_c)
    next_c <- c
  } else
    <- c
}

func main() {
  var c = make (chan int)
  var n = ...

  chain (n, c)

  c <- 42
}
```

A chain of channels is built, each instance of `chain` sending an unbuffered message to the next one, with the last instance of `chain` receiving (and dropping) its incoming message. None of these instances can complete until the send statement `c<-42`. The scheduling algorithm pushes the continuation of the `chain` onto the stack and eagerly executes the recursive call. As a result, the send statement in the `main` is not executed until the recursion ends. The opposite compilation choice of delaying the spawned goroutine and eagerly completing the current one leaves at most one instance of `chain` waiting.