Lecture 7: 4 October 1992

Lecturer: Bill Weihl *Scribes: Wei-Hsin Lee And Dawson Engler*[1]

1 Handouts

- Handout 19: Birrell's paper — "An Introduction to Programming with Threads"
- Handout 20: Examples of Concurrency
- Handout 21: Implementing Atomic Actions
- Handout 22: Non-Atomic Semantics of Spec
- Handout 23: Problem Set #4

2 Today's Topic: Concurrency

Today's topic is an introduction to concurrency. Concurrent programs can get very hard to understand very quickly even if they are small. Just as for sequential programs we need a way to:

- specify concurrent programs
- implement concurrent programs
- and prove them correct

We will talk about these tasks over the next three lectures. We will also cover various pragmatic issues that effect performance. We start with a simple example that will illustrate the types of problems that concurrency can cause. Examples are from Handout 20.

3 Incrementing a Register

We begin with the first example in Handout 20: Incrementing a Register. Given a register that has a state s that is an integer, and two operations:

- read, that returns the current state and
- write, that sets the current state

[1] These notes were prepared by Manish Tuteja last year, based on Rainer Gawlick's notes from the previous year.

The operation we want to perform is an increment. In a sequential system we could perform the increment in three steps:

- read the current value into a temporary
- increment that value
- write the updated value back into the register

This is shown on p. 1 of the handout. Now consider what happens if we have multiple processes accessing this register via the increment procedure. If two processes are executing this procedure concurrently, both could execute the read operation, each could then update the temporary, then both could write back the updated version. The register would only be incremented once instead of twice. This is probably not the intended behavior.

4 Atomicity

To analyze this properly, we need to know the low-level atomic operations. For some operations Spec defines the level of atomicity, for example, the assignment is atomic once the RHS has been computed. We can infer from this that the Read and Write procedures appear to act atomically by examining their bodies. It is possible, however, to implement read and write as non-atomic operations, for example, by viewing the register as an array of bits and operating on the bits one at a time. If operations of two processes can be interleaved at the level of the individual bits, the result could be practically anything.

Therefore we need to know what the atomic operations are; what are those operations that can be performed without any interleaving or interference by any other processes. Without knowing this we can't understand a concurrent program. The way to think about a concurrent program is that the processes each take steps and that they are running asynchronously. The execution of the program is then as follows: at each time step we pick a process and that process does the next atomic step according to its own PC. We then pick the next process to run the next step more or less randomly. The result is interleaving at the level of the atomic operations.

A major issue in concurrent programming is how to make complex operations (like increment) appear atomic. In Spec we have atomicity brackets to define larger atomic operations from smaller ones. (Spec uses double brackets. Often in the literature we will see single brackets.) Code inside atomicity brackets executes atomically.

In the example in Handout 20, assume the read and write are atomic. If the register started with value k and n processors are concurrently executing the increment, the possible result values include any integer from $k + 1$ (if all processors read the current value before any of them update it) to $k + n$ (if the processors perform the increments in sequence) and anything in between. Recall if the reads and writes are not atomic then the possible results might be quite different.

To bring the point home, suppose the increment is adding money to a bank account. If you and your spouse both increment your account you'd like both operations to have an impact on your balance.

This problem appears at all levels of concurrent systems, register operations (as shown here) and at higher levels such as operating systems and data bases, *etc*.

The simplest way to make complex operations atomic is to enclose them in atomicity brackets. This feature is not something that most languages provide (although there are some proposed languages that do). One way to implement the atomicity bracket is to not allow any other process to execute their next step while one process is executing code within atomicity brackets. On multiprocessors, this wastes cycles on the idle processors and can have serious impact on performance. Even on uniprocessors we wouldn't want to implement atomicity brackets this way. Suppose that the process performs I/O, the process would have to wait until the I/O is done before execution proceeds. This is quite inefficient. Suppose that we experience a page fault during a memory read, we would like the processor to switch to another task while the page is read from disk. Forcing the processor to wait until all operations within atomicity brackets complete is wasteful.

So our basic problem is how to make complicated computations appear atomic. This doesn't mean that they execute without interleaving. Atomicity is not an issue of physically serializing things. It is more an issue of what users can see.

5 Locks

Typically, interference results when shared state is manipulated by more than one thread concurrently. If threads do not manipulate the same state (*e.g.*, are only using locals), mutual exclusion is unnecessary. Code that is supposed to run without interference from other processes is called a *critical section*. This term arises because the state of a system may become damaged if critical section code is interrupted or interleaved with code from another process. We saw an example of this in the previous section with the increment procedure. One safe way to implement a critical section is to insure that no two processes are executing the critical section code concurrently. Exclusive use of the critical section is usually achieved with locks. The approach to locks presented on p. 2 of Handout 20 is used in many systems today. Mutex is a standard term in the literature; it stands for mutual exclusion. The state of the mutex is `nil` if no process is holding a lock or it is the thread name (process id) of the process that is holding the lock.

[Aside: There is a confusion between the term process (this has connotation of separate address space) and the term thread (which shares the same address space with other threads). For this class the differences are not critical.]

This spec also provides for releasing and acquiring locks. Release atomically sees if the process holding the lock is the one asking to release. If the caller to release is the current holder, it releases the mutex. If some other process attempts to release the lock, the result is `HAVOC`.

Acquire is more subtle. Atomically if the current state of the mutex is nil it is set to the name of the current process. This must be done atomically—if it is not done atomically race conditions could occur in which two processes both find it `nil` and both acquire it.

An implementation of Mutex is given on p. 3 of Handout 20 under the heading "Spin Locks". Consider the module `BrokenMutexImpl`. The implementation of `PROC Release` is straightforward. Note that, unlike the `Mutex` Spec, this implementation does not store the id of the process that is currently holding the lock. The implementation of `PROC Acquire` is slightly incorrect. `PROC Acquire` has two atomic statements. One checks to see if m = held, the other sets m := held. If two processes both execute the first statement and find $m \neq$ held and then both execute the second statement and set

$m :=$ held, both processes will believe that they have acquired the lock. This situation clearly violates the `Mutex` spec. The race condition illustrated by the example we just described is eliminated in the module `MutexImpl`, which is given on p. 4 of the handout.

5.1 To Spin or To Block

[2]

Additional notes: a process wishing to acquire the lock will continue to read the state of the mutex until the lock is successfully acquired. Locks with this type of behavior are called *spin locks*.

On a uniprocessor, a process that spins while waiting for a mutex to become available wastes cycles. On a multiprocessor however, if there are many processes contending for a mutex and if the mutex is held for long periods of time, then spinning is also not be desirable as the process may need to wait a long time before the mutex is obtained. Hence, it is often preferable to *block* rather than spin. Blocking is used to stop an otherwise spinning process from executing until there is a chance that the condition it is waiting for becomes true. This prevents the spinning process from stealing processor time while it waits.

Blocking, however, is an expensive process in most systems as it requires among other things, entering the kernel, scheduler interaction, and queue manipulation. If t_b is the amount of time needed for a process to block, a process is generally allowed to spin if the expected mutex wait time is shorter than t_b. However, in practice, the expected wait time is often unknown. A hybrid scheme allows a process to spin a duration of t_b before blocking. This 2-phase scheme is no more than a constant factor worse than the best possible scheme, and so, is a competitive algorithm.

6 Semantics of Spec

Recall that atomic statements were modeled as relations on states. Non-atomic statements are more complex. We model them as a set of traces where each trace consists of a sequence of state transitions. Which of the traces a non-atomic statement will follow depends on the state of the system at the time that each step in the statement is executed. Since steps from other processes can alter the state of the system and can be interleaved between the steps of the statement, it cannot be known ahead of time which steps a non-atomic statement will execute. In contrast, atomic statements do not allow any interleaving of steps from other processes. Hence, the trace that an atomic statement would follow is well defined when the statement begins its execution. This fact allows us to model atomic statements as relations on states.

We will now explore the differences between the semantics of atomic and non-atomic statements with some examples.

If we have an atomic statement that can fail (the body of the Acquire can fail if value of mutex is non-nil) then the semantics is as if it were implemented with backtracking in order to find a successful path through the statement that succeeds. In other words, inside atomicity brackets, we use all the angels and backtracking mechanisms necessary in order to make the right non-deterministic choices so

[2]This subsection is drawn from Fall 1992 Lecture 8 notes written by Tony Eng.

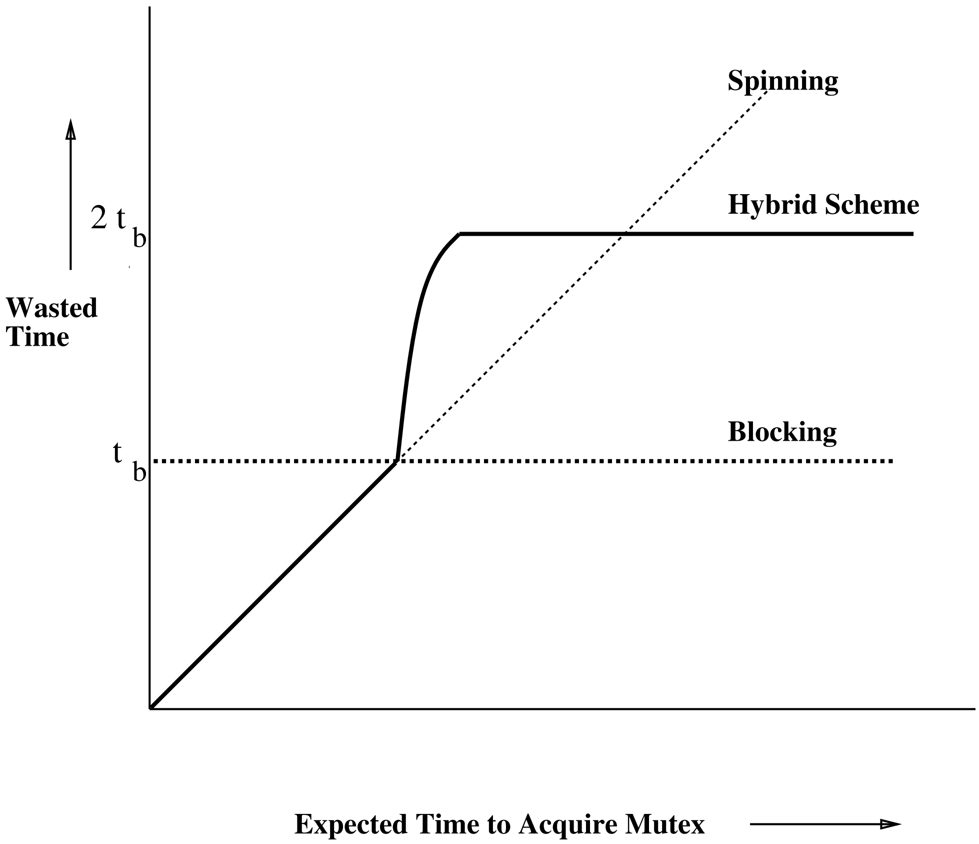

Figure 1: Performance of Hybrid Scheme Compared to Spinning and Blocking.

that the statement will succeed. For example, in the atomic command from page 2 of Handout 20, << m = nil => m := SELF >>, when m # nil there is no outcome, and the PC is reset to the beginning of the atomicity body.

Outside of atomicity brackets, we can choose the next single atomic step based on success or failure of that single step only. That is, for

```
<s1>;<s2>;<s3> []  <s4>;<s5>;<s6> or
<s1>;<s2>;<s3> [*]  <s4>;<s5>;<s6>
```

we can choose an option based on the success of the first atomic operation of that option (s1 or s4) only. If we choose the option (<s1>;<s2>;<s3>) over the option (<s4>;<s5>;<s6>) then <s1> must have succeeded. However, if <s2> fails no backtracking takes place. The program counter remains at <s2> until (based on updates by other processes) <s2> succeeds.

Two aspects make these semantics rational:

- We can't backtrack over a completed atomic operation (*e.g.*, <s1>) since other interleaved processes may have performed subsequent steps based on the completion of that atomic operation.

- We can't use an oracle or an angel to ensure that all the statements will succeed for an option composed of several atomic operations since after one of the atomic operations completes, other processes could perform actions that affect the guards of the remaining operations. In this example, we can't guarantee the success of <s2> and <s3> since after completion of <s1> another process could update values used in computing the guard for s2.

Consider the following additional examples:

- ```
  <<
  G1 => S1
  []
  G2 => S2
  >>
  if G1 => S1 fails then try G2 => S2.
  ```

- ```
  <<S1>>; <<P => S3>>
  []
  <<S2>>
  ```

  Choice is made on the success or failure of S1 and S2. If S1 succeeds and we choose the first of the two options P can still fail. Since this choice is outside atomicity brackets we cannot backtrack. We simply wait at <<P => S3>> until it succeeds.

- If the previous example were changed slightly to

  ```
  <<S1>>; P => S3
  []
  <<S2>>
  ```

then if we choose the first option, we execute `S1` atomically and then try `P`. If `P` fails we wait, as before, until `P` succeeds. However, if `P` succeeds, with the atomicity brackets removed, it is possible that other processes intervene between the success of P and the execution of `S3`.

- ```
  <<
  S; P => S3
  []
  S2
  >>
  ```

  Here we cannot pick the first choice if `P` fails.

- ```
  SKIP; false => HAVOC
  []
  SKIP
  ```

  Here we could choose the first option since SKIP succeeds but we will wait forever at the predicate `false`.

- ```
  <<S1>>; <<P>> => <<S3>>
  [*]
  <<S2>>
  ```

  Choice examines only `S1` and `S2` for success. If both fail, it is the choice that is waiting. That is, when either succeed, processing can continue.

7 Spec's handling of Concurrency

There are only a few fundamental additions to Spec which allow it to handle concurrency. These are summarized below:

1. **Semantics** of non-atomic statements are different than those for atomic statements. Non-atomic statements often represent a sequence of transitions rather than a single transition and therefore have the potential to get stuck in the middle of a statement.

2. **Atomicity brackets** are used for specifying long constructs that are intended to be done atomically.

3. **Primitive atomic operations** must be defined to allow us to reason about atomicity. The following are primitive atomic operations:

   - Expression evaluation
   - Functions
   - Guards
  ```

- Choosing x *and* first transition (underlined) in the following statement:
  `VAR x | P(x) => S`

  Assignment is *not* an atomic operation: it occurs in two steps. The right hand side is first executed then the left hand side is executed. The actual assignment after the rhs and lhs have been executed is atomic.

4. `Thread P(i : INT) = p => s` Thread `P` gets created for every possible value of the argument `i` at the same time.

   Besides using mutex, we can also use condition variables which associate the predicate the thread depends on for progress with a condition variable to allow the possibility of more efficient concurrency (*i.e.*, by blocking the thread until some thread signals that the predicate is satisfied). Condition variables were initially implemented in Monitors with three operations : Wait, Signal, and Broadcast.