Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 20 October 4, 1993

# Examples of Concurrency

This handout contains a number of example specifications and programs that will be used in the lectures on concurrency.

## 1 Incrementing a Register

The first example involves incrementing a register that has Read and Write operations. If we ignore concurrency, we might write the specification of the register as follows:

```
MODULE Register =

VAR s: Int      % state of register

PROC Read () -> Int = RET s

PROC Write (i: Int) = s := i

END Register
```

To increment the register, we could use the following procedure:

```
PROC Increment () =
   VAR t: Int |
   t := REGISTER.Read();
   t := t + 1;
   REGISTER.Write(t)
```

If multiple threads can use the register concurrently, we need to know what the atomic operations are. Let's assume that Read and Write are each atomic, giving the following revised specification of the register:

```
MODULE Register =

VAR s: Int      % state of register

APROC Read () -> Int = << RET s >>

APROC Write (i: Int) = << s := i >>

END Register
```

Suppose the register starts with value $k$ and $n$ threads execute Increment in parallel. What are the possible final states?

Suppose that we weaken our atomicity assumptions: assume that the value of a register is represented as a sequence of bits, and that the only atomic operations are reading and writing individual bits. Now what are the possible final states if $n$ threads execute Increment in parallel?

# 2 Mutexes

Here is a specification of a simple Mutex module, which can be used to ensure mutually exclusive execution of critical sections. The state of a mutex is `nil` if the mutex is available, and otherwise is the thread that holds the mutex.

```
MODULE Mutex =

TYPE M = (Thread + Null)

VAR m := nil     % either nil or the thread holding the mutex

% The pseudo-variable SELF is defined to be the current thread

APROC Acquire () = << m = nil  =>  m := SELF >>

APROC Release () =
    <<  m # SELF  => HAVOC
      [*] m := nil >>

END Mutex
```

Note that a real system would provide a module that allows a program to create new mutexes dynamically, for example by allocating them on a heap. Spec does not provide pointers or heap allocation, so we cannot model this kind of program directly. However, it is easy to change the specification to use `SEQ M` as the rep, and to have an operation `Create` that extends the sequence with a new mutex and returns its index, which is then passed as an argument to `Acquire` and `Release`. The same comment applies to later examples in this handout.

Here's a simple use of mutexes to make the Increment procedure atomic:

```
PROC Increment () =
   Mutex.Acquire();
   VAR t: Int |
   t := REGISTER.Read();
   t := t + 1;
   REGISTER.Write(t);
   Mutex.Release()
```

This prevents concurrent calls of Increment from interfering with each other. If there are other accesses to the register, they must also use the mutex to avoid interfering with threads executing Increment.

# 3 Spin Locks

A simple way to implement a mutex is to use a "spin lock." The name is derived from the behavior of a thread waiting to acquire the lock — it "spins," repeatedly attempting to acquire the lock until it is finally successful.

Here's an incorrect implementation:

```
MODULE BrokenMutexImpl =

TYPE M = ENUM[available, held]

VAR m := available

PROC Acquire () =
   DO << m=held => SKIP >> OD; << m:=held >>

PROC Release () = << m := available >>

END BrokenMutexImpl
```

Why is it wrong?

Here's a correct implementation. It uses a more complex atomic statement in the Acquire procedure. This statement records the initial value of the lock, and then sets it to held. The next time around the loop, the initial value is tested; if it was available, then this thread was successful in atomically changing the state of the lock from available to held. Otherwise some other thread must hold the lock, so we "spin", repeatedly trying to acquire it until we succeed. The atomic statement in Acquire corresponds to the atomic "test-and-set" instruction used in many real machines to implement locks.

```
MODULE MutexImpl =

TYPE M = ENUM[available, held]

VAR m := available

PROC Acquire () =
   VAR b: Bool := held |
   DO b=held => << b:=m; m:=held >> OD

PROC Release () = << m := available >>

END MutexImpl
```

# 4 Condition Variables

Here is a specification for condition variables:

```
MODULE Condition =

TYPE C = SET Thread

VAR c := {}

PROC Wait () =
    Mutex.m # SELF  => HAVOC
   [*] << c := c ++ SELF; Mutex.m := nil >>;
       << (Mutex.m = nil) /\ ~(SELF IN c)
              => Mutex.m := SELF >>

PROC Signal () =
   % can wake up more than one; usually exactly one
   << VAR c' | (c' <= c) /\ (c' # {}) => c := c - c'
     [*] skip >>

PROC Broadcast () = << c := {} >>

END Condition
```

# 5 Readers/Writers

Here is a specification of a module that provides locks with two modes: read and write. Several threads can hold a lock in read mode, but only one thread can hold a lock in write mode, and no thread can hold a lock in read mode if some thread holds it in write mode. In other words, read locks can be shared, but write locks are exclusive.

```
MODULE ReadersWriters =

TYPE ST = SET Thread
     RW = [r: ST, w: ST]

VAR rw := RW{r:={}, w:={}}

APROC StartRead () =
   << rw.w={} => rw.r := rw.r ++ SELF >>

APROC EndRead () =
   << ~(SELF IN rw.r) => HAVOC
     [*] rw.r := rw.r - {SELF} >>

APROC StartWrite () =
   << rw.w={} /\ rw.r={} => rw.w := {SELF} >>

APROC EndWrite () =
   << ~(rw.w = {SELF}) => HAVOC
     [*] rw.w := {} >>

END ReadersWriters
```

Here is an implementation using mutexes and condition variables:

```
MODULE ReadersWritersImpl =

VAR rw: Int := 0  % positive value gives number of readers,
                  % zero means not in use, and -1 means
                  % there is one writer.

PROC StartRead () =
   Mutex.Acquire();
   DO rw < 0 => Condition.Wait() OD;
   rw := rw + 1;
   Mutex.Release()
```

```
PROC EndRead () =
   Mutex.Acquire();
   rw := rw - 1;
   BEGIN rw = 0 => Condition.Signal() [*] SKIP END;
   Mutex.Release()

PROC StartWrite () =
   Mutex.Acquire();
   DO rw # 0 => Condition.Wait() OD;
   rw := -1;
   Mutex.Release()

PROC EndWrite () =
   Mutex.Acquire();
   rw := 0;
   Condition.Broadcast();
   Mutex.Release()

END ReadersWritersImpl
```

# 6 Implementation of Mutexes and Condition Variables

This section presents an implementation of mutexes and condition variables based on the one in the Topaz operating system from DEC SRC. The implementation of mutexes has a "fast path" that stays out of the kernel when an attempt to acquire a mutex finds it available and when a call to release a mutex finds no other thread waiting for it.

Notes on the implementation: SpinLock and ReleaseSpinLock acquire and release a global lock used in the kernel to protect thread queues. Scheduling a thread moves it from the kernel list of blocked threads to the ready list, from which it can be assigned to a processor to run. Descheduling a thread moves it to the kernel list of blocked threads (and also releases the global lock in the kernel).

```
MODULE MutexImpl =

VAR lock_bit: Bool := false
    queue: SEQ Thread := {}

% External procedures:

PROC Acquire (m) =
  DO VAR b: Bool | b := TestAndSet();
   ~b => RET
```

```
   [*] KernelAcquire()
  OD

PROC Release () =
  lock_bit := false;
  queue.isEmpty => RET
  [*] KernelRelease()

% Internal procedures:

% Sets lock_bit to true; returns old value.
PROC TestAndSet () -> Bool =
  << VAR b: Bool := lock_bit |
     lock_bit := true;
     RET b >>

% KernelAcquire and KernelRelease run in the kernel.
PROC KernelAcquire () =
  SpinLock();
  queue := queue ++ SELF;
  BEGIN
    lock_bit = false => queue := queue.reml % remove SELF and return
                                            % to Acquire to try again
     [*] Deschedule(SELF);   % also releases spin lock
         RET
  END;
  ReleaseSpinLock()

PROC KernelRelease () =
  SpinLock();
  VAR t: Thread := queue.head |
  queue := queue.tail;
  Schedule(t);   % The newly scheduled thread will compete
                 % with others to acquire the mutex.
  ReleaseSpinLock()

END MutexImpl
```

```
MODULE ConditionImpl =

VAR eventCount: Int := 0
    queue: SEQ Thread := {}

PROC Wait () =
  VAR i: Int |
  i := eventCount; % save value of eventCount
  Mutex.Release();
  KernelBlock(i);
  Mutex.Acquire()

% Internal procedure; runs in kernel
PROC KernelBlock (i: Int) =
  SpinLock();
  BEGIN % if eventCount has changed, must have been a Signal, so return
    i = eventCount => queue := queue ++ SELF; Deschedule(SELF); RET
   [*] SKIP
  END;
  ReleaseSpinLock()

% Signal and Broadcast always run in the kernel
PROC Signal () =
  SpinLock();
  eventCount := eventCount + 1;
  BEGIN ~queue.isEmpty => VAR t: Thread := queue.head |
                    queue := queue.tail;
                    Schedule(t)
   [*] SKIP
  END;
  ReleaseSpinLock()

PROC Broadcast () =
  SpinLock();
  eventCount := eventCount + 1;
  DO ~queue.isEmpty => VAR t: Thread := queue.head |
                    queue := queue.tail;
                    Schedule(t)
  OD;
  ReleaseSpinLock()

END ConditionImpl
```

# 7 A FIFO Buffer

In this section we give a specification and implementation of a simple buffer that could be used as a communication channel between two threads.

```
% Specification
MODULE Buffer[T] =

VAR b: SEQ T := {}

PROC Produce (t) = << b := b++t >>

PROC Consume () -> T =
   << b.size > 0 => VAR t := b.head | b := b.tail; RET t >>

END Buffer

% Implementation
MODULE BufferImpl[T] =

% Abstraction function:
%
%      b = items (not quite: depends on process state)

VAR items: SEQ T := {}

PROC Produce (t) =
   Mutex.Acquire();
   items := items++t;
   Condition.Signal();
   Mutex.Release()

PROC Consume () -> T =
   Mutex.Acquire();
   DO items.size = 0 => Condition.Wait() OD;
   VAR t := items.head |
   items := items.tail;
   Mutex.Release();
   RET t

END BufferImpl
```