Notation for Atomic Actions

In a shared memory system, we say an operation is atomic if it is done with no interleaving of other operations. For example, the hardware of a computer might be designed such that, the execution of a test-and-set instruction can not be interrupted, nor interleaved with the effects of any other machine instruction. We introduce the following notation for atomic operations:

Unconditional atomic actions `< S >`

The intended meaning is that statement `S` is done atomically. No internal state of `S` is visible to other processes. For example:

```
< x = x + 1 ;  y = y + 1 ; >    // x and y are shared variables
```

Other threads that access the values of `x` and `y` will encounter either 1) both values *before* they are both incremented, or 2) both values *after* they are incremented. No other thread can encounter a post-increment value for `x` and a pre-increment value for `y`.

Conditional atomic actions `< await(B) S >`

The intended meaning is that this thread does not proceed until boolean condition `B` becomes *true*. Condition `B` is guaranteed to be true at the beginning of `S`. In many cases, `B` will remain true for the duration of statement `S`, up until the last instruction in `S`. (See section on flag principles). No internal state of `S` is visible to other threads. For example:

```
bool lock ;
< await( lock == false )  lock = true ;  >
```

Concurrent Execution `process f[ id = 1 to n]`

We use the notation:

```
process f[ id = 1 to n ]
```

to define a set of `n` concurrent threads executing process `f`.

Critical Section

A *critical section* is a region of code where only one thread can be permitted in that region at any point in time. Any method that implements a critical section must support the possibility that the critical section is in a loop. The "critical section problem" is to implement an entry and exit protocol so that the following code runs correctly:

```
process CS [ i = 1 to n ]  //  n concurrent threads

    while (true) {
       entry protocol ;
       critical section ;
       exit protocol ;
       non-critical section ;
    }
```

In addition, a satisfactory solution must ensure the following properties.

- Mutual Exclusion
- No Deadlock or Livelock[1]
- Eventual Entry
- No Unnecessary Delay

Special Instructions: Test-And-Set

A `test-and-set` instruction is defined in C++-like pseudocode as follows:

```
bool test_and_set( bool & lock )
{
   < bool temp = lock ;
     lock = true ;
     return temp ; >
}
```

A Simple "Spin Lock" Implementation of Critical Section:

```
bool lock = false ;  // Global shared variable, initially false.

process CS [ i = 1 to n ]   //  n concurrent threads

    while (true) {
       while ( test_and_set( lock ) ) /* continue */  ;
       critical section ;
       lock = 0 ;
       non-critical section ;
    }
```

Notice that the test-and-set instruction writes to its memory location every time the instruction is performed. This can lead to memory contention, i.e., many threads are all trying to write to one memory location. A somewhat better alternative is called "test, then test-and-set". The idea here is that each thread tests the value of the lock variable (read only) before using the test-and-set instruction. A solution using "test, then test-and-set" is discussed on the next page.

[1]Livelock is similar to deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

In contrast, deadlock would occur when two people meet in a narrow corridor, and each waits (forever) for the other to move out of the way.

Reducing Memory Contention: Test, then Test-And-Set

```
bool lock = false ;    // Global shared variable, initially false.

process CS [ id = 1 to n ] {
   while(true) {
      while ( lock )  /* continue */ ;     // Entry protocol
      while ( test_and_set( lock ) ) {     // Entry protocol
          while ( lock ) /* continue */ ;  // Entry protocol
      }                                    // Entry protocol

      critical-section ;       //   Critical work
      lock = false ;           //   Exit protocol
      non-critical section ;   //   Non-critical work
   }
}
```

The idea of this approach is that a thread only *reads* the value of `lock` until some other process releases the lock. The test-and-set instruction is only used to acquire the lock after the lock has been seen to be available.

We denote the entry and exit protocols by `CSenter` and `CSexit` respectively. We can then use the critical section protocols to implement arbitrary unconditional and conditional atomic actions.

Unconditional atomic actions `< S >`

```
CSenter ;
S  ;
CSexit ;
```

Conditional atomic actions `< await(B) S >`

```
CSenter ;
while (!B) {CSexit ; delay() ; CSenter ; }
S  ;
CSexit ;
```

The idea of the `delay()` is that it reduces memory contention. Observe that the thread executing this code can not proceed until some other thread alters a variable that causes condition `B` to become true. Multiple threads executing this code without the delay would cause memory contention because each thread repeatedly accesses the critical section protocols, and the variables in `B`.

The spin-lock solutions described above do not control the order in which delayed processes enter their critical sections when multiple processes are attempting to do so. Under some scheduling conditions, it could be the case in practice that some threads are inadvertently

given "preferential treatment" and that "non-preferred" threads never enter their critical sections. One solution to this problem is called the "ticket algorithm". The ticket algorithm can be efficiently implemented if the hardware has an atomic `fetch_and_add` instruction.

Special Instructions: Fetch-And-Add

A fetch-and-add instruction is defined in C++-like pseudocode as follows:

```
int fetch_and_add( int & v, int increment ) {
    < int temp = v ;
      v = v + increment ;
      return temp ; >
}
```

Ticket Algorithm

```
int number = 1 ;    // Global shared integer (scalar) variable
int next = 1   ;    // Global shared integer (scalar) variable
int turn[n]    ;    // Global shared integer (array) variable

process CS_using_ticket_algorithm [ id = 1 to n ]
{
    while(true) {
       turn[id] = fetch_and_add( number, 1 ) ;        // Entry protocol
       while ( turn[id] != next )  /* continue */ ;   // Entry protocol

       critical-section ;

       next = next + 1 ;     // Exit protocol

       non-critical section ;

}
```

The ticket algorithm eliminates the memory contention issue associated with the spin-lock solution. Notice that each thread busy-waits on an array entry which is unique for each thread; i.e., the array entries are in different memory locations and could also be arranged to be in different cache lines.

A potential problem with this solution is that the global counters `number` and `next` could overflow. In practice, this is unlikely. Assume we use 64-bit unsigned integers to implement the global counters. If 1,000 threads each attempt to enter their critical sections once per microsecond, simple arithmetic shows that the counters will not overflow for over 584 years.

Special Instructions: Compare-And-Swap

A `compare-and-swap` instruction is defined in C++-like pseudocode as follows:

```
int compare_and_swap( int * Register, int Tvalue, int NewValue )
{
   < temp = *Register ;      /* Register holds a memory address. */
     if ( temp == Tvalue ) *Register = NewValue ;  >
   return temp ;
}
```

The `compare-and-swap` instruction was included in the IBM 370 series computers (circa 1970). The Intel 80486 processor (circa 1989) introduced a compare-and-exchange (CMPXCHG) instruction, similar to the compare-and-swap.

It should be noted that the functionality of an atomic `fetch-and-add` instruction (as used by the Ticket algorithm), can be easily implemented using the `compare-and-swap` instruction.

Software Based Alternatives

Prior to the widespread hardware support for mutual exclusion, the **Bakery Algorithm** was used to ensure mutual exclusion (MUTEX) without the use of any special hardware instructions. The Bakery algorithm would appear to be now of historical interest, since hardware support for mutual exclusion (usually via `compare-and-swap`) has been in all major processor designs (e.g., 80486, SPARC V9, MIPS R1000 and later) for over twenty years.

Barrier Synchronization

A *barrier* is a point in the code where all threads wait until all threads arrive at the barrier. Any method that implements a barrier must support the possibility that the barrier is in a loop. The "barrier problem" is to implement a protocol so that the following code runs correctly:

```
process Worker[ i = 1 to n ]      //  n threads concurrently execute.

    while (true) {
       perform task i ;
       wait for all n tasks to complete their task ;
    }
```

Flag-based Synchronization Principles

1. The process that waits for a synchronization flag should be the one that clears that flag.
2. A flag should not be set until is is known that it is clear.

Barrier Using a Coordinator Process

```
bool arrive[n] = { false, false, ... , false } ;  // Global shared array,
                                                  // initially false.

bool continue[n] = { false, false, ... , false } ;  // Global shared array,
                                                    // initially false.

process Worker[ i = 1 to n ]
{
   while (true) {
      code to implement task i ;
      arrive[i] = true ;  // Process i announces its arrival at the barrier.
      < await ( continue[i] ) ; >
      continue[i] = false ;        // Uses flag syncronization principles.
}

process Coordinator
{
   while (true) {
      for i = 1 to n do {
        < await (arrive[i]) ;  >   // Waits for all to arrive.
        arrive[i] = false ;        // Uses flag syncronization principles.
      }

      for i = 1 to n do {   //  Allow all workers to continue
         continue[i] = true ;
      }
  }
}
```

Tree Structured Barrier

In a tree structured barrier, each thread acts as a node in a (complete) binary tree. Nodes act differently depending on whether they are a leaf node, an interior node, or the root node.

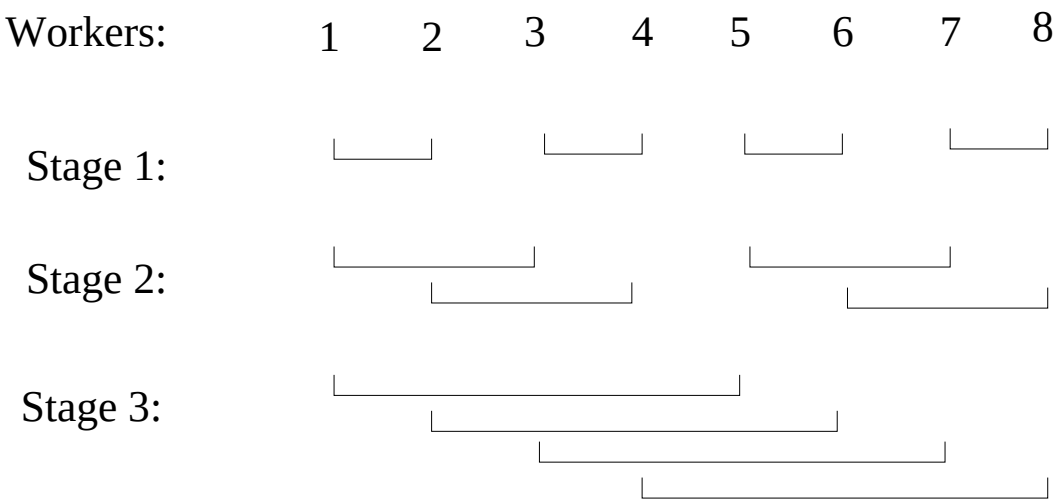
```
bool arrive[n] = { false, false, ... , false } ;  // Global shared array,
                                                  // initially false.

bool continue[n] = { false, false, ... , false } ;  // Global shared array,
                                                    // initially false.

process uses_a_tree_barrier[ i = 1 to n ]
{
   while (true) {
      code to implement task i ;

      // Implemenation of the barrier:
      Leaf node L:
           arrive[L] = true ;
           < await ( continue[L] ) ; >
           continue[L] = false ;

      Interior node I:
           < await ( arrive[Left Child] ) ; >
           arrive[Left Child] = false ;        // Flag sync principles again.

           < await ( arrive[Right Child] ) ; >
           arrive[Right Child] = false ;        // Flag sync principles again.

           arrive[I] = true ;
           < await ( continue[I] ) ; >
           continue[I] = false ;

           continue[Left Child] = true ;   // Allow child nodes to proceed
           continue[Right Child] = true ;

      Root node R:
           < await ( arrive[Left Child] ) ; >
           arrive[Left Child] = false ;        // Flag sync principles again.

           < await ( arrive[Right Child] ) ; >
           arrive[Right Child] = false ;        // Flag sync principles again.

           continue[Left Child] = true ;   // Allow child nodes to proceed
           continue[Right Child] = true ;

   }  // end while ...
}
```

Two Process Symmetric Barrier

```
// Barrier code for worker process W[i]

< await ( arrive[i] == false ) ; >   // Prevent race-around-re-entry
arrive[i] = true ;
< await ( arrive[j] ) ; >
arrive[j] = false ;

// Barrier code for worker process W[j]

< await ( arrive[j] == false ) ; >   // Prevent race-around-re-entry
arrive[j] = true ;
< await ( arrive[i] ) ; >
arrive[i] = false ;
```

N-Process Symmetric Barrier

We might try to use the two process barrier shown above in a sequence of stages as illustrated below:[2]



But, there is a problem with the two-process symmetric barrier when it is used in a butterfly barrier. Consider:

- Process 1 arrives and waits for process 2.
- Process 2 is slow to arrive
- Process 3 arrives and waits for process 4.
- Process 4 arrives. Both process 3 and 4 continue to the next stage

In the second stage, process 3 waits for `arrive[1]` to become true. However, `arrive[1]` is already true because process 1 is still waiting on process 2. Process 3 would then incorrectly proceed.

[2]This arrangement is called a *butterfly* barrier.

N-Process Symmetric Barrier Using Counters

To correct the problem described above, there are two alternatives:

- Use a different array of flags for each stage.
- Use counters and modify the worker process accordingly.

Pseudocode for the modified worker process (using counters) is shown below:

```
int arrive[n] = { 0, 0, ... , 0 } ;  // Global shared array, initially 0.


//  Barrier code for worker process i

for s = 1 to number_of_stages do
{
    arrive[i] =  arrive[i] + 1  ;
    j = compute_neighbor_j_for_stage(i,s) ;
    while ( arrive[j] < arrive[i] ) /* continue to loop */  ;
}
```

Observe this solution eliminates the need for two arrays, and also eliminates the need for clearing the other process' flag. There is also no need for a special test-and-set, or fetch-and-add instruction.

Ella's Coloring