6.895 Theory of Parallel Systems	Lecture 5
Consistency of the Memory Sub-System	
Lecturer: Bradley C. Kuszmaul	*Scribe: Alexandru Caracas, Rui Fan*

Lecture Summary

1. *Mutual Exclusion*
   Introduction to the problems of shared-memory parallel systems and definition of mutual exclusion.

2. *Mutual Exclusion Algorithms*
   Description of the mutual exclusion algorithms of Dekker and Dijkstra.

3. *Memory Consistency*
   Definition of memory consistency in terms of atomic operations.

4. *Memory Subsystem of Hardware Architectures*
   Description of various memory operation primitives such as Load-Link-Store-Conditional and Transactional Memory.

1 Mutual Exclusion

The underlying problem of shared memory parallel systems is that we have more than one thread and shared memory locations, accessed by all concurrent executing threads. The question is how do the threads interact?

Example 1 One example Cilk code which contains a simple determinacy race, already discussed in the previous lectures, is the following:

```
cilk int foo (void) { x++; }
cilk int main (int argc, char *argv[]) {
   spawn foo();
   spawn foo();
   sync;
}
```

Once we detect the determinacy race, e.g. by using the Nondeterminator, our goal is to eliminate it. In order to avoid the determinacy race we have to make sure that only one thread at a time can perform the operation `x++`. As a start we could consider the following idea. Simply prevent context switching, for example by turning off the interrupts as in the following MIPS assembler code:

```
DI            ; turn off interrupts
LW R1, (0)R2
ADD R1,R1,1
SW R1, (0)R2
EI
```

However, there are a number of downfalls with using interrupts for mutual exclusion:

- It only works on a uniprocessor machine. On a multiprocessor machine the other processors are not affected by the `DI` instruction.

- It only works in kernel (not user) mode. On most architectures there is no user-level call that can disable interrupts.
- And it also prevents other threads from doing non-critical work, which is extremely dangerous, since the operating system time slice interrupt will never take place, and the operating system would fail gaining control.

Definition 1 *In a more general conceptual setting, mutual exclusion can be described in the following algorithmic setup:*

```
while (1){
   trying();
   critical_section()
   releasing();
   noncrit();
}
```

In an infinite loop each thread tries to execute `critical_section()`, that is the sequence of operations which need to be executed atomically. In order to be allowed to execute the critical section the thread first needs to get permission. In the above example the task of allocating the right to enter the critical section is given by the part `trying()`. Moreover the thread needs to be cooperative and release hold of its right to enter the critical section in `releasing()`. Consequently it performs the rest of its non-critical task `noncrit()`.

Now the problem is to write the functions `trying()` and `releasing()` so that at most one thread is in `critical_section` at a time. One extremely trivial solution would be:

Example 2 `trying(void) { halt(); }`

The trivial solution is not good since it simply stops the currently executing thread. This illustrates a problem with our specification of mutual exclusion. In addition to the algorithmic description of mutual exclusion from Definition 1 we now introduce some additional definitions.

Definition 2 *Mutual exclusion requires that there is* **Progress**. *If no thread is executing* `critical_section` *and there are some threads which are trying to enter* `critical_section` *(in our example by executing* `trying()`*) then some thread will do* `critical_section`.

Apart from the definition of mutual exclusion with guaranteed progress there are other aspects which are desirable, however not strictly required for *Mutual Exclusion*, such as:

Definition 3 **No Starvation**. *Every executing thread gets a fair chance of executing in the* `critical_section`.

Example 3 An alternate idea for the mutual exclusion problem in the case of two threads would be:

```
int turn=1;
int run(int i) {
   // i is the thread number for the currently executing thread,
   // 0 for one thread and 1 for the other
   while (turn!=i);
   crit();
   turn=1-i;  // it is the other thread's turn
   noncrit();
}
```

However, the problem with the previous approach is that it does not guarantee progress. If, for example, `noncrit()` is an infinite loop in thread 1, then thread 0 would get stuck.

2 Mutual Exclusion Algorithms

This section presents two algorithms that solve the mutual exclusion problem. The first one is due to Dekker and it solves the mutual exclusion problem for two threads. The second is Dijkstra's algorithm for mutual exclusion, which is a generalization applicable to any number of threads.

2.1 Dekker's Algorithm

A solution which works for two threads and uses turns was described by Dekker. The following C code presents his algorithm:

```
int turn;
int wants[2];

// i is the current thread, j=1-i is the other thread
while(1) {                 // trying
   wants[i] = TRUE;
   while (wants[j]) {
      if (turn==j) {
         wants[i] = FALSE;
         while (turn==j) ; // empty loop
         wants[i] = TRUE;
      }
  }
  critical_section();
  turn=j;                  // release
  wants[i] = FALSE;
  noncrit();
}
```

As a visualization of Dekker's algorithm consider the following railroad from Figure 1. Each thread has its own railroad system which intersects with the railroad of the other thread only on the `critical_section`. There are a number of switches which the threads activate as they traverse the railroad. Whenever a thread encounters a semaphore it can change its course according to the indication of the semaphore. The threads "move" in the direction shown by the arrows.

As an illustration, assume that thread 0 starts at the top of its railway system and wants to do `critical_section`. It will first set `turn=1`, and then `wants[0]=TRUE`. Next it will encounter a semaphore which tells it whether thread 1 wants to enter the critical section. Suppose that thread 1 also wants to enter, then thread 0 will switch tracks, and "move" downwards. On this new track it will set `wants[0]=FALSE` and go and check the next semaphore. Assume that now `turn=0`, then thread 0 will remain on this track, and set `wants[0]=TRUE`. Also assume that now thread 1 does not want to enter the `critical_section`. Then thread 0 can go and execute `critical_section`.

2.2 Dijkstra's Algorithm

Dijkstra devised a mutual exclusion algorithm for any number of threads. The following are the conditions for Dijkstra's mutual exclusion problem:

- Only 1 thread may execute `critical_section()` at a time.
- The threads execute in an infinite loop trying to enter `critical_section`.
- *Atomic read* and *atomic write* are available, but no other instructions (no locking instructions).

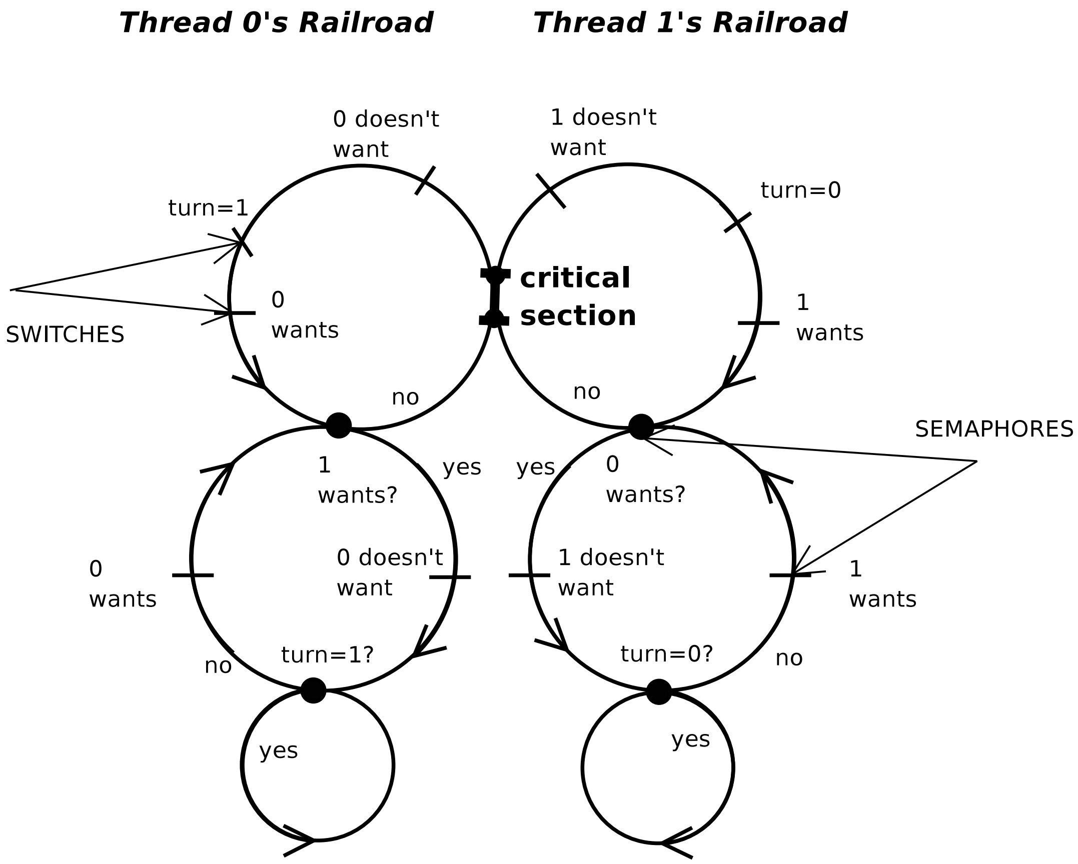

Figure 1: Railroad visualization of Dekker's algorithm for mutual exclusion. The threads "move" in the direction shown by the arrows.

- Progress condition is satisfied.
- No assumptions about relative speeds of threads.

The following is the C code for Dijkstra's Algorithm:

```
char b[N], c[N];  // b[i] and c[i] are only be written by thread i, initialize true
int  k;           // 0 <= k < N

while (1) {
   // i is my own thread number
   b[i] = FALSE;
L: if (k!=i) {
      c[i] = TRUE;
      if (b[k]) k = i;
      goto L;
   }
    else {
      c[i] = FALSE;
      for (j=0; j<N; j++) {
         if (j!=i && !c[j]) {
            goto L;
         }
      }
   }
   crit();
   c[i] = TRUE;
   b[i] = TRUE;
   noncrit();
}
```

3 Proof of Dijkstra's Algorithm

We first show that Dijkstra's algorithm satisfies mutual exclusion. Suppose for contradiction that there are two processes i and j in the critical section at the same time. Then $c[i] = c[j] =$ FALSE. Suppose, without loss of generality, that i was the first to set $c[i] =$ FALSE. Then, j must see $c[i] =$ FALSE in the for loop after assigning $c[j]$, hence j goes to L. Then, j does not enter the critical section, which is a contradiction. Thus, there is at most one process in the critical section at once.

To show that progress is always made, suppose several processes try to enter the critical section. Then each such process i sets $b[i] =$ FALSE. We claim that at least one trying processes advances into the else clause. Consider all the trying processes for which the test "if ($b[k]$)" succeeds. There is at least one such process. Consider the last process i among this set to set $k \leftarrow i$. Then i will enter the else clause.

Suppose only one process i enters the else clause. Then i finds $c[j] =$ TRUE for all $j \neq i$, and enters the critical section. Next, suppose a set P of more than one process enters the else clause. They may all find $c[j] =$ FALSE for some other $j \in P$, and goto L. But now, for every process $i \in P$, $b[i] =$ FALSE, and so i cannot reset $k \leftarrow i$. On the other hand, for some $i \in P$, we have $k = i$. Then i will be the only process to go into the else clause[1], where it finds $c[j] =$ TRUE for all other j, and so can enter the critical section. Thus, Dijkstra's algorithm ensures progress.

[1] Actually, another (slow) process j can set $k \leftarrow j$ and enter the else clause with i. But then we can repeat the previous argument. Each time we repeat the previous argument, we eliminate at least one trying process from going into the else clause, while ensuring there is still at least one process which can go into the else clause. Since the number of trying processes is finite, we are eventually left with exactly one process which can go into the else clause.

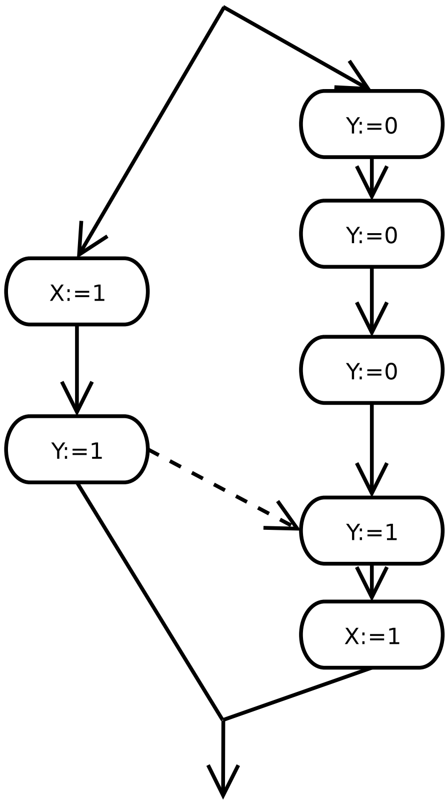

Figure 2: A DAG for the multithreaded program in Example 4. The solid arrows represent dependencies induced by the program order. The dashed arrow represents an implicit dependency of the execution.

4 Memory Consistency

Notice that Dijkstra's algorithm depends on a property of the memory system, which is that all the processes read the last value written to a memory location. This is called *atomic read* and *atomic write*. Here is an example to illustrate the idea:

Example 4

```
  main {
    X = 0; Y = 0;
    spawn thread0();
    spawn thread1();
    sync }

 thread0() {              thread1() {
    X = 1;                   while (Y == 0);
    Y = 1 }                   print(X) }
```

If the reads and writes are atomic, then the value that is printed for X must be 1. This is because to exit the while loop in thread 1, Y must be set to 1. But if writes are atomic, this means that X is already set to 1. We can draw each computation as a DAG, where the arrows point from earlier to later events. For example, one DAG for an execution of the above example is shown in the Figure 2.

As a side note, we define an *observer function* as a mapping that gives, at each point in the execution, for each memory location, which write operation wrote the value that a process sees if it reads that location. A *consistency model* is a set of allowable observer functions. An execution is consistent with the consistency model if it can be observed by an observer function of the model. A hardware implementation of memory is consistent with a consistency model if all its executions are consistent with the model. We now define a consistency model for memory called *sequential consistency*.

Definition 4 *Let G be a DAG of an execution α of a multithreaded program. Then α is* sequentially consistent *if and only if there is a topological sort S of G such that for each read, the most recent write to that memory location according to S wrote the value that the read received.*

Note that there can be more than one topological sort of G demonstrating consistency. All that Definition 4 requires is the existence of some sort consistent with the execution. Also, note that for sequential consistency, the allowable observer functions are the ones for which a topological sort exists on the DAG such that the value for a location seen is the one most recently written to that location.

5 Memory Subsystem of Hardware Architectures

The MIPS multiprocessors guarantee sequential consistency. However, the x86 multiprocessors do not. The reason is that CPU's pipeline instructions and can execute them out of order. Therefore, even though a read may occur later than a write in code, the read can actually be executed before the write. It is complicated to do out of order execution and guarantee strong consistency.

Instead of sequential consistency, Intel offers a *barrier* operation. Specifically, the instruction MFENCE forces all reads and writes before an MFENCE to finish before any reads and writes after the MFENCE can start. However, the Intel MFENCE instruction is very slow. On a 2.5Ghz Pentium 4, one MFENCE instruction takes 120ns. Interestingly, the barrier instruction on a 1.5Ghz AMD64 Opteron takes only 8ns.

5.1 Additional Memory Primitives

As we saw, mutual exclusion is very complicated to do using only atomic read and write memory. Thus, most machines provide additional memory instructions to make mutual exclusion easier. One common instruction is *test-and-set*. The following pseudocode demonstrates what test-and-set does:

Example 5
```
test_and_set(int *l) {
  atomically {
    int old_value = *l;
    *l = 1;
    return old_value;
  }
}
```

As the example shows, test-and-set atomically reads the old value from a memory location and writes a 1 to that location. To demonstrate its usefulness, consider the following simple algorithm for mutual exclusion using test-and-set.

Example 6
```
while (test_and_set(&lock));
critical_section();
lock = 0;
```

The first process which gets a 0 from test-and-set can enter the critical section. All the other processes have to wait till that process is done and resets lock.

Another popular memory instruction is *load-linked store-conditional (LLSC)*. This is provided, for example, on SGI. LLSC is illustrated in the following code:

Example 7

```
again:
  LL R1, R2       // R1 = *R2
  compute
  SC R1, R2       // *R2 = R1 but only if *R2 has not changed since LL
  bz again:       // the zero flag is set so we can do it again if it failed
```

As the example shows, LLSC lets a process read a memory location, do some computation, then write a value back to that memory location. But the write only succeeds if the location didn't change since the read. If the location did change, LLSC sets the zero flag to let the process know to try again. LLSC lets us increment a variable without a lock, as shown in the following example. Note that this code performs the increment directly, without attempting to get mutual exclusion on the variable. Also note that the algorithm may not make progress, if for example, two processes continually perform LLSC alternately.

Example 8

```
again:
  LL R1, R2
  ADD R1, R1, 1
  SC R1, R2
  bz again
```

To end this lecture, we mention a few other mutual-exclusion type memory operations offered by many architectures. The most important among these is *compare-and-swap (CAS)*. The operation CAS(x, y, v) does the following. If $x = y$, then the operation sets $x \leftarrow v$. Otherwise, it returns an error code saying the CAS failed. Some architectures also offer *double compare-and-swap (DCAS)*. Similar to CAS, DCAS$(x_1, x_2, y_1, y_2, v_1, v_2)$ compares x_1 to y_1, and x_2 to y_2. If both sets of comparisons return true, then DCAS sets $x_1 \leftarrow v_1$, and $x_2 \leftarrow v_2$. Otherwise, it returns an error code saying the DCAS failed.

Another approach to ensuring atomicity is to use *transactional memory*. This technique performs a sequence of memory operations atomically, all as part of a transaction. This allows a simple and uniform way of implementing many complex concurrent data structures.