Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of two processes with busy waiting.

```
// flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn = 0 // or 1

P0:
flag[0] = true;
while (flag[1] == true)
   if (turn != 0) {
      flag[0] = false;
      while (turn != 0) {
         // busy wait
      }
      flag[0] = true;
   }
   // critical section
   turn = 1;
flag[0] = false;
```

```
P1:
flag[1] = true;
while (flag[0] == true)
   if (turn != 1) {
      flag[1] = false;
      while (turn != 1) {
         // busy wait
      }
      flag[1] = true;
   }
   // critical section
   turn = 0;
flag[1] = false;
```
The Idea Behind Dekker

Communication via three variables:
- \( \text{flag}[i] = \text{true} \) process \( P_i \) wants to enter its critical section
- \( \text{turn} = i \) process \( P_i \) has priority when both want to enter

P0:
```java
flag[0] = true;
while (flag[1] == true) {
  if (turn != 0) {
    flag[0] = false;
    while (turn != 0) {
      // busy wait
    }
  }
  flag[0] = true;
}
```
```
// critical section
turn = 1;
flag[0] = false;
```

In process \( P_i \):
- if \( P_{i-1} \) does not want to enter, proceed immediately to the critical section

Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of two processes with busy waiting.
```java
// flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn = 0 // or 1
```

P0:
```java
flag[0] = true;
while (flag[1] == true) {
  if (turn != 0) {
    flag[0] = false;
    while (turn != 0) {
      // busy wait
    }
  }
  flag[0] = true;
}
```
```
// critical section
turn = 1;
flag[0] = false;
```

P1:
```java
flag[1] = true;
while (flag[0] == true) {
  if (turn != 1) {
    flag[1] = false;
    while (turn != 1) {
      // busy wait
    }
  }
  flag[1] = true;
}
```
```
// critical section
turn = 0;
flag[1] = false;
```
A Note on Dekker’s Algorithm

Dekker’s algorithm has the three desirable properties:
- **ensure mutual exclusion**: at most one process executes the critical section
- **deadlock free**: the process will never wait for each other
- **free of starvation**: if a process wants to enter, it eventually will

Applications for Dekker: implement a \((map \circ reduce + map)\) operation concurrently

```plaintext
T acc = init();
for (int i = 0; i < c; i++) {
    <T,U> (acc, tmp) = f(acc, i);
    g(tmp, i);
}
```

- accumulating a value by performing two operations \(i\) and \(g\) in sequence
- the calculation in \(i\) of the \(i\)th iteration depends on iteration \(i - 1\)
- non-trivial program to parallelize
- idea: use two threads, one for \(i\) and one for \(g\)

Concurrent Reduce + Map

Create an \(n\)-place buffer for communication between processes \(P_f\) and \(P_g\).

```plaintext
T acc = init();
Buffer<U> buf = buffer<T>(n); // some locked buffer
```

```plaintext
Pf:
for (int i = 0; i < c; i++) {
    <T,U> (acc, tmp) = f(acc, i);
    buf.put(tmp);
}
```

```plaintext
Pg:
for (int i = 0; i < c; i++) {
    T tmp = buf.get();
    g(tmp, i);
}
```

Dekker’s Algorithm and Weakly-Ordered

Problem: **Dekker’s algorithm requires sequentially consistency.**
Idea: insert memory barriers between all variables common to both threads.

```plaintext
flag[0] = true;
sfence();
while (!fence(), flag[1] == true)
    if (!fence(), turn != 0) {
        flag[0] = false;
sfence();
        while (!fence(), turn != 0) {
            // busy wait
        }
        flag[0] = true;
sfence();
    }

// critical section
turn = 1;
sfence();
flag[0] = false; sfence();
```

Dekker’s Algorithm and Weakly-Ordered

Problem: Dekker’s algorithm requires sequentially consistency.
Idea: insert memory barriers between all variables common to both threads.

- insert a load memory barrier \(\text{l fence}()\) in front of every read from common variables
Dekker’s Algorithm and Weakly-Ordered

Problem: Dekker’s algorithm requires sequentially consistency.
Idea: insert memory barriers between all variables common to both threads.

```c
P0:
flag[0] = true;
sfence();
while (lfence(), flag[1] == true)
  if (lfence(), turn != 0) {
    flag[0] = false;
sfence();
    while (lfence(), turn != 0) {
      // busy wait
    }
    flag[0] = true;
sfence();
  }
  // critical section
  turn = 1;
sfence();
flag[0] = false; sfence();
```

- insert a load memory barrier `lfence()` in front of every read from common variables
- insert a write memory barrier `sfence()` after writing a variable that is read in the other thread
- the `lfence()` of the first iteration of each loop may be combined with the preceding `sfence()` to an `mfence()`

Discussion

Memory barriers reside at the lowest level of synchronization primitives.

Where are they useful?
- when several processes implement an automaton and ...
- synchronization means coordinating transitions of these automata
- when blocking should not de-schedule threads
- often used in operating systems

Why might they not be appropriate?
- difficult to get right, possibly inappropriate except for specific, proven algorithms
- often synchronization with locks is as fast and easier
- too many fences are costly if store/invalidate buffers are bottleneck

Discussion

Memory barriers reside at the lowest level of synchronization primitives. Where are they useful?
- when several processes implement an automaton and ...
- synchronization means coordinating transitions of these automata
- when blocking should not de-schedule threads
- often used in operating systems

Why might they not be appropriate?
- difficult to get right, possibly inappropriate except for specific, proven algorithms
- often synchronization with locks is as fast and easier
- too many fences are costly if store/invalidate buffers are bottleneck

What do compilers do about barriers?
- C/C++: it’s up to the programmer, use `volatile` for all thread-common variables to avoid optimizations which are only correct for sequential programs
- C++11: use of `atomic` variables will insert memory barriers
- Java,Go,..: the runtime system must guarantee this
Summary

Memory consistency models:
- strict consistency
- sequential consistency
- weak consistency
Illustrating consistency:
- happened-before relation
- happened-before process diagrams
Intricacy of cache coherence protocols:
- the effect of store buffers
- the effect of invalidate buffers
- the use of memory barriers
Use of barriers in synchronization algorithms:
- Dekker's algorithm
- stream processing, avoidance of busy waiting
- inserting fences

Future Many-Core Systems: NUMA

Symmetric multi-processing (SMP) has its limits:
- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus

~ use a bus locally, use point-to-point links globally: **NUMA**
- non-uniform memory access partitions the memory amongst CPUs
- a directory states which CPU holds a memory region
- Intel's MESF to reduce communication overhead
- Interprocess communication between Cache-Controllers (**ccNUMA**): onchip on Opteron or in chipset on Itanium

Overhead of NUMA Systems

Communication overhead in a NUMA system.
- Processors in a NUMA system may be fully or partially connected.
- The directory of who stores an address is partitioned amongst processors.
- A cache miss that cannot be satisfied by the local memory at A:
  - A sends a retrieve request to processor B owning the directory
  - B tells the processor C who holds the content
  - C sends data (or status) to A and sends acknowledgment to B
  - B completes transmission by an acknowledgment to A

source: [Intel]
References

Intel.
An introduction to the intel quickpath interconnect.

Leslie Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.

Paul E. McKenny.
Memory Barriers: a Hardware View for Software Hackers.

Scott Owens, Susmit Sarkar, and Peter Sewell.
A better x86 memory model: x86-TSO.

Future Many-Core Systems: NUMA

Symmetric multi-processing (SMP) has its limits:
- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus
  → use a bus locally, use point-to-point links globally: NUMA
- non-uniform memory access partitions the memory amongst CPUs
- a directory states which CPU holds a memory region
- Intel's MESIF to reduce communication overhead
- Interprocess communication between Cache-Controllers (ccNUMA):
onchip on Opteron or in chipset on Itanium

Why Memory Barriers are not Enough

Communication via memory barriers has only specific applications:
- coordinating state transitions between threads
- for systems that require minimal overhead (and no de-scheduling)
Often certain pieces of memory may only be modified by one thread at once.
- can use barriers to implement automata that ensure mutual exclusion
- → generalize the re-occurring concept of enforcing mutual exclusion
Why Memory Barriers are not Enough

Communication via memory barriers has only specific applications:
- coordinating state transitions between threads
- for systems that require minimal overhead (and no de-scheduling)
Often certain pieces of memory may only be modified by one thread at once.
- can use barriers to implement automata that ensure mutual exclusion
- generalize the re-occurring concept of enforcing mutual exclusion
Need a mechanism to update these pieces of memory as a single atomic execution:

\[ A \]

\[ a = 1, b = 1 \]

\[ a \]

\[ b \]

- several values of the objects are used to compute new value
- certain information from the thread flows into this computation
- certain information flows from the computation to the thread

Atomic Executions

A concurrent program consists of several threads that share common resources:
- resources are often pieces of memory, but may be an I/O entity
  - a file can be modified through a shared handle
- for each resource an invariant must be retained
  - a head and tail pointer must define a linked list
- an invariant may span several resources
- during an update, an invariant may be broken
- several resources must be updated together to ensure the invariant
- which particular resources need to be updated may depend on the current program state

Overview

We will address the established ways of managing synchronization.
- present techniques are available on most platforms
- likely to be found in most existing (concurrent) software
- techniques provide solutions to solve common concurrency tasks
- techniques are the source of common concurrency problems

Presented techniques applicable to C, C++ (pthread), Java, C# and other imperative languages.
Overview

We will address the established ways of managing synchronization.
- present techniques are available on most platforms
- likely to be found in most existing (concurrent) software
- techniques provide solutions to solve common concurrency tasks
- techniques are the source of common concurrency problems

Presented techniques applicable to C, C++ (pthread), Java, C# and other imperative languages.

Learning Outcomes

- Principle of Atomic Executions
- Wait-Free Algorithms based on Atomic Operations
- Locks: Mutex, Semaphore, and Monitor
- Deadlocks: Concept and Prevention

Atomic Execution: Varieties

Definition (Atomic Execution)

A computation forms an atomic execution if its effect can only be observed as a single transformation on the memory.

Several classes of atomic executions exist:
- Wait-Free: an atomic execution always succeeds and never blocks
- Lock-Free: an atomic execution may fail but never blocks
- Locked: an atomic execution always succeeds but may block the thread
- Transaction: an atomic execution may fail (and may implement recovery)

These classes differ in
- amount of data they can access during an atomic execution
- expressivity of operations they allow
- granularity of objects in memory they require
Wait-Free Atomic Executions

Wait-Free Updates

Which operations on a CPU are atomic executions? (\(j\) and \(\text{tmp}\) are registers)

Program 1
\[
i++;
\]

Program 2
\[
j = i;
i = i+k;
\]

Program 3
\[
\text{int} \ \text{tmp} = i;
i = j;
j = \text{tmp};
\]

Answer:
- none by default (even without store and invalidate buffers, \textit{why}?)
- but all of them \textit{can} be atomic executions

Wait-Free Updates

Which operations on a CPU are atomic executions? (\(j\) and \(\text{tmp}\) are registers)

Program 1
\[
i++;
\]

Program 2
\[
j = i;
i = i+k;
\]

Program 3
\[
\text{int} \ \text{tmp} = i;
i = j;
j = \text{tmp};
\]

Answer:
- none by default (even without store and invalidate buffers, \textit{why}?)
- but all of them \textit{can} be atomic executions

The programs can be atomic executions:
- \(i\) must be in memory (e.g. declared as volatile)
- Idea: \textit{lock} the cache/bus for an address for the duration of an instruction; on x86:
  - Program 1 can be implemented using a \texttt{lock inc [addr.i]} instruction
  - Program 2 can be implemented using \texttt{mov eax,k; lock xadd [addr.i],eax; mov reg,j,eax}
  - Program 3 can be implemented using \texttt{lock xchg [addr.i],reg.j}
Wait-Free Updates

Which operations on a CPU are atomic executions? (j and tmp are registers)

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
<th>Program 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i++;</td>
<td>j = i;</td>
<td>int tmp = i;</td>
</tr>
<tr>
<td></td>
<td>i = i+k;</td>
<td>i = j;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>j = tmp;</td>
</tr>
</tbody>
</table>

Answer:
- none by default (even without store and invalidate buffers, why?)
- but all of them can be atomic executions

The programs can be atomic executions:
- i must be in memory (e.g. declared as volatile)
- idea: lock the cache/bus for an address for the duration of an instruction; on x86:
  - Program 1 can be implemented using a lock inc [addr.i] instruction
  - Program 2 can be implemented using mov eax,k;
    lock xadd [addr.i],eax; mov reg.j,eax
  - Program 3 can be implemented using lock xchg [addr.i],reg.j

⚠️ Without lock, the load and store generated by i++ may be interleaved with a store from another processor.

Wait-Free Bumper-Pointer Allocation

Garbage collectors often use a **bumper pointer** to allocated memory:

**Bumper Pointer Allocation**

```c
char heap[2^20];
char* firstFree = &heap[0];

char* alloc(int size) {
    char* start = firstFree;
    firstFree = firstFree + size;
    if (start+size>sizeof(heap)) garbage_collect();
    return start;
}
```

- firstFree points to the first unused byte
- each allocation reserves the next size bytes in heap

Without lock, the load and store generated by i++ may be interleaved with a store from another processor.

Marking Statements as Atomic

Rather than writing assembler: use made-up keyword `atomic`:

**Program 1**

```c
atomic {
    i++;
}
```

**Program 2**

```c
atomic {
    j = i;
    i = i+k;
    j = tmp;
}
```
Marking Statements as Atomic

Rather than writing assembler: use made-up keyword `atomic`:

Program 1
```c
atomic {
  i++;}
```

Program 2
```c
atomic {
  j = i;
  i = i+k;
}
```

Program 3
```c
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```

The statements in an `atomic` block execute as `atomic execution`:

```
A
\rightarrow
i
```

- `atomic` only translatable when a corresponding atomic CPU instruction exist
- the notion of requesting `atomic execution` is a general concept

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:
- no control flow possible, no behavioral change depending on data
- often, there are instructions that execute an operation conditionally

Program 4
```c
atomic {
  r = b;
  b = 0;
}
```

Program 5
```c
atomic {
  r = b;
  b = 1;
}
```

Program 6
```c
atomic {
  [k=i];
  if (r) i = j;
}
```

Operations `update` a memory cell and `return` the previous value.
- the first two operations can be seen as setting a flag `b` to `v \in \{0,1\}` if `b`
  - this operation is called `modify-and-test`
- the third case generalizes this to arbitrary values
  - this operation is called `compare-and-swap`

---

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:
- no control flow possible, no behavioral change depending on data
- often, there are instructions that execute an operation conditionally

Operations `update` a memory cell and `return` the previous value.
- the first two operations can be seen as setting a flag `b` to `v \in \{0,1\}` if `b`
  - this operation is called `modify-and-test`
- the third case generalizes this to arbitrary values
  - this operation is called `compare-and-swap`

---

Lock-Free Algorithms

If a `wait-free` implementation is not possible, a `lock-free` implementation might still be viable.
Lock-Free Algorithms

If a \textit{wait-free} implementation is not possible, a \textit{lock-free} implementation might still be viable.

Common usage pattern for \textit{compare and swap}:

1. read the initial value in \( i \) into \( k \) (using memory barriers)
2. calculate a new value \( j = f(k) \)
3. update \( i \) to \( j \) if \( i = k \) still holds
4. go to first step if \( i \neq k \) meanwhile

\textbf{⚠️ note:} \( i = k \) must imply that no thread has updated \( i \)

\textit{general recipe for lock-free algorithms}

- given a \textit{compare-and-swap} operation for \( n \) bytes
- try to group variables for which an invariant must hold into \( n \) bytes
- read these bytes atomically
- calculate a new value
- perform a \textit{compare-and-swap} operation on these \( n \) bytes

\textbf{⚠️ note:} \( i = k \) must imply that no thread has updated \( i \)

\textit{general recipe for lock-free algorithms}

- given a \textit{compare-and-swap} operation for \( n \) bytes
- try to group variables for which an invariant must hold into \( n \) bytes
- read these bytes atomically
- calculate a new value
- perform a \textit{compare-and-swap} operation on these \( n \) bytes

\textbf{⚠️ calculating new value must be \textit{repeatable}}