Out-of-Order Execution

performance problem: writes always stall

Thread A

\[ a = 1; \quad // \quad A.1 \]
\[ b = 1; \quad // \quad A.2 \]
\[ assert(a == 1); \quad // \quad B.1 \]

Thread B

\[ while (b == 0) \{ \} \]
\[ a = 1 \]

Memory Consistency

Disproving Sequential Consistency

Given a result of a program with \( n \) threads on a SC system,

- with operations \( p_1, p_1', \ldots \) and \( \bar{p}_2, \bar{p}_2', \ldots \) and \( \ldots p_0, p_0' \)
- there exists a total order \( \exists C. C(p_i) < C(p_j) \) for all \( i, j, k, \ldots \) where \( j = l \) implies \( i < k \),
- such that this execution has the same result.

Idea for showing that a system is not sequentially consistent:

- pick a result obtained from a program run on a SC system
- pick an execution and a total ordering of all operations
- add extra processes to model other system components
- the original order becomes a partial order
- show that total orderings \( C' \) exist for which the result differ
Weakening the Model

There is no observable change if calculations on different memory locations can happen in parallel.
- Idea: model each memory location as a different process.

MESH Example: Happened Before Model

Idea: each cache line one process, A caches b=0 as E, B caches a=0 as E

Observations:
- Each memory access must complete before executing next instruction.
- Second execution of test b==0 stays within cache - no traffic.

Out-of-Order Execution

Performance problem: writes always stall.

Thread A

```
A = 1; // A.1
B = 1; // A.2
```

Thread B

```
while (b == 0) {}; // B.1
assert(a == 1); // B.2
```

~ CPU A should continue executing after a = 1.
Store Buffers

Goal: continue execution after cache-miss write operation

- put each write into a store buffer and trigger fetching of cache line
- once a cache line has arrived, apply relevant writes
  - today, a store buffer is always a queue [OSS09]

H2: \( a = 2 \)
H1: \( b = 1 \)
\( \implies a = 0 \)
\( \land (b \geq a) \)
**Store Buffers**

*Goal: continue execution after cache-miss write operation*

- put each write into a *store buffer* and trigger fetching of cache line
- once a cache line has arrived, apply relevant writes
  - today, a store buffer is always a *queue* [OSS09]
  - two writes to the same location are not merged
-⚠️ sequential consistency per CPU is violated unless

---

**Happened-Before Model for Store Buffers**

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I

**Thread A**

\[
a = 1; \\
b = 1;
\]

**Thread B**

\[
while (b == 0) {} \\
an (== 1)
\]
Explicit Synchronization: Write Barrier

Overtaking of messages is desirable and should not be prohibited in general.
- store buffers render programs incorrect that assume sequential consistency between different CPUs
- whenever two stores in one CPU must appear in sequence at a different CPU, an explicit write barrier has to be inserted
- Intel x86 CPUs provide the sfence instruction
- a write barrier marks all current store operations in the store buffer

Explicit Synchronization: Write Barrier

Overtaking of messages is desirable and should not be prohibited in general.
- store buffers render programs incorrect that assume sequential consistency between different CPUs
- whenever two stores in one CPU must appear in sequence at a different CPU, an explicit write barrier has to be inserted
- Intel x86 CPUs provide the sfence instruction
- a write barrier marks all current store operations in the store buffer
- the next store operation is only executed when all marked stores in the buffer have completed assignment
Explicit Synchronization: Write Barrier

Overtaking of messages is desirable and should not be prohibited in general.
- store buffers render programs incorrect that assume sequential consistency between different CPUs
- whenever two stores in one CPU must appear in sequence at a different CPU, an explicit write barrier has to be inserted
- Intel x86 CPUs provide the sfence instruction
- a write barrier marks all current store operations in the store buffer
- the next store operation is only executed when all marked stores in the buffer have completed
- a write barrier after each write gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)

Invalidate Queue

Invalidation of cache lines is costly:
- all CPUs in the system need to send an acknowledge
Invalidation Queue

Invalidation of cache lines is costly:
- all CPUs in the system need to send an acknowledge
- invalidating a cache line competes with CPU accesses
- a cache-intensive computation can fill up store buffers in other CPUs

--- immediately acknowledge an invalidation and apply them later
- put each invalidate message into an invalidate queue
- if a MESI message needs to be sent regarding a cache line in the invalidate queue then wait until the line is invalidated
- local read and writes do not consult the invalidate queue
Happened-Before Model for Invalidate Buffers

Thread A
a = 1;
sfence();
b = 1;

Thread B
while (b == 0) {}
assert (a == 1);

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I

Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.
- might read an out-of-date value
- need a way to establish sequential consistency between writes of other processors and local reads
- insert an explicit read barrier before the read access
Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.
- might read an out-of-date value
- need a way to establish sequential consistency between writes of other processors and local reads
- insert an explicit read barrier before the read access
- Intel x86 CPUs provide the lfence instruction

Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.
- might read an out-of-date value
- need a way to establish sequential consistency between writes of other processors and local reads
- insert an explicit read barrier before the read access
- Intel x86 CPUs provide the lfence instruction
- a read barrier marks all entries in the invalidate queue

Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.
- might read an out-of-date value
- need a way to establish sequential consistency between writes of other processors and local reads
- insert an explicit read barrier before the read access
- Intel x86 CPUs provide the lfence instruction
- a read barrier marks all entries in the invalidate queue
- the next read operation is only executed once all marked invalidations have completed
- a read barrier before each read gives sequentially consistent read behavior (and is as slow as a system without invalidate queue)
Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.

- might read an out-of-date value
- need a way to establish sequential consistency between writes of other processors and local reads
- insert an explicit read barrier before the read access
- Intel x86 CPUs provide the lfence instruction
- a read barrier marks all entries in the invalidate queue
- the next read operation is only executed once all marked invalidations have completed

Happened-Before Model for Read Fences

Thread A

```
a = 1;
sfence();
b = 1;
```

Thread B

```
while (b == 0) {};
lfence();
assert(a == 1);
```
Happened-Before Model for Read Fences

**Thread A**
- `a = 1;
- `sfence();
- `b = 1;`

**Thread B**
- `while (b == 0) { };
- `lfence();
- `assert(a == 1);`

![Diagram showing memory consistency and cache behavior]

Summary: Weakly-Ordered Memory Models

Modern CPUs use a *weakly-ordered memory model*:
- Reads and writes are not synchronized unless requested by the user.
- Many kinds of memory barriers exist with subtle differences.

Summary: Weakly-Ordered Memory Models

Modern CPUs use a *weakly-ordered memory model*:
- Reads and writes are not synchronized unless requested by the user.
- Many kinds of memory barriers exist with subtle differences.
- Most systems provide a barrier that is both read and write (e.g., `mfence` on x86).
Modern CPUs use a weakly-ordered memory model:

- reads and writes are not synchronized unless requested by the user
- many kinds of memory barriers exist with subtle differences
- most systems provide on barrier that is both, read and write (e.g. mfence on x86)
- ahead-of-time imperative languages can use memory barriers, but compiler optimizations may render programs incorrect
- use the volatile keyword in C/C++

In the latest C++ standard, an access to a volatile variable will automatically insert a memory barrier.

Modern CPUs use a weakly-ordered memory model:

- reads and writes are not synchronized unless requested by the user
- many kinds of memory barriers exist with subtle differences
- most systems provide on barrier that is both, read and write (e.g. mfence on x86)
- ahead-of-time imperative languages can use memory barriers, but compiler optimizations may render programs incorrect
- use the volatile keyword in C/C++

- in the latest C++ standard, an access to a volatile variable will automatically insert a memory barrier
- otherwise, inline assembler has to be used
- memory barriers are the "lowest-level" of synchronization
Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of two processes with busy waiting.

```c
// 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;
```

```c
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:

- `flag[i] = true` process `P_i` wants to enter its critical section
- `turn = i` process `P_i` has priority when both want to enter

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

In process `P_i`:

- if `P_{i-1}` does not want to enter, proceed immediately to the critical section
- `flag[i]` is a lock and may be implemented as such

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

```c
In process `P_i`:

- if `P_{i-1}` does not want to enter, proceed immediately to the critical section
- `flag[i]` is a lock and may be implemented as such
- if `P_{i-1}` also wants to enter, wait for `turn` to be set to 1

```c
flag[0] = true;
```

flag[1] = true;
```
The Idea Behind Dekker

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

```
0:
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
- \( \rightarrow \) flag[i] is a lock and may be implemented as such
- if \( P_{i-1} \) also wants to enter, wait for turn to be set to i
- while waiting for turn, reset flag[i] to enable \( P_{i-1} \) to progress
- algorithm only works for two processes

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
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 \((\text{map} \circ \text{reduce})\) operation concurrently

\[
\text{T acc = init();}
\text{for (int i = 0; i < c; i++)}
\text{\{}
\text{<T,U> (acc,tmp) = f(acc,i);}
\text{g(tmp, i);}
\text{\}}
\]

- accumulating a value by performing two operations \(f\) and \(g\) in sequence
- the calculation in \(f\) of the \(i\)th iteration depends on iteration \(i - 1\)
- non-trivial program to parallelize
Concurrent Reduce

Create an $n$-place buffer for communication between processes $P_f$ and $P_g$.

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

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

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

If $f$ and $g$ are similarly expensive, the parallel version might run twice as fast.

But busy waiting is bad!
  - the cores might be idle anyway: no harm done (but: energy efficiency?)

Concurrent Reduce

Create an $n$-place buffer for communication between processes $P_f$ and $P_g$.

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

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

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

If $f$ and $g$ are similarly expensive, the parallel version might run twice as fast.

But busy waiting is bad!
  - the cores might be idle anyway: no harm done (but: energy efficiency?)
  - $f$ can generate more elements while busy waiting
  - $g$ might remove items in advance, thereby keeping busy if $f$ is slow
Concurrent Reduce

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

\[
\begin{align*}
T & \text{ acc = init(); } \\
\text{Buffer}<U> & \text{ buf = buffer}<T>(n); // some locked buffer }
\end{align*}
\]

\[
\begin{align*}
P_f: & \text{ for (int i = 0; i<n; i++)} \\
& \text{ (T,U) (acc,tmp) = f(acc,i);} \\
& \text{ buf.put(tmp); }
\end{align*}
\]

\[
\begin{align*}
P_g: & \text{ for (int i = 0; i<n; i++)} \\
& \text{ T tmp = buf.get();} \\
& \text{ g(tmp, i); }
\end{align*}
\]

If \( f \) and \( g \) are similarly expensive, the parallel version might run twice as fast.

But busy waiting is bad!

- the cores might be idle anyway: no harm done (but: energy efficiency?)
- \( f \) can generate more elements while busy waiting
- \( g \) might remove items in advance, thereby keeping busy if \( f \) is slow
- \textit{ideal scenario}: keep busy during busy waiting

Generalization to Stream Processing

Observation: \( g \) might also manipulate a state, just like \( f \).

\textit{\textbar bar} computation reduces/maps a function on a sequence of items

- general setup in signal/data processing
- data is manipulated in several stages
- each stage has an internal state
- an item completed in one stage is passed on to the next stage

Use of Dekker’s algorithm:

- could be used to pass information between stages
- but: fairness of algorithm is superfluous
  - producer does not need access if buffer is full
Generalization to Stream Processing

Observation: \( g \) might also manipulate a state, just like \( f \).

- computation reduces/maps a function on a sequence of items
  - general setup in signal/data processing
  - data is manipulated in several stages
  - each stage has an internal state
  - an item completed in one stage is passed on to the next stage

Use of Dekker's algorithm:
  - could be used to pass information between stages
  - but: fairness of algorithm is superfluous
    - producer does not need access if buffer is full
    - consumer does not need access if buffer is empty

Dekker's Algorithm and Weakly-Ordered

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

PO:
flag[0] = true;
sfence();
while (lffence(), flag[1] == true)
  if (lffence(), turn != 0) {
    flag[0] = false;
sfence();
    while (lffence(), 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.

PO:
flag[0] = true;
sfence();
while (lffence(), flag[1] == true)
  if (lffence(), turn != 0) {
    flag[0] = false;
sfence();
    while (lffence(), 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.

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 lie 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

Why might they not be appropriate?
- difficult to get right, possibly inappropriate except for specific, proven algorithms

Discussion
Memory barriers lie 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 lie 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 optimization that are only correct for sequential programs

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

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
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 acknowledge to \( B \)
- \( B \) completes transmission by an acknowledge to \( A \)

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.