Script generated by TTT

Title: Petter: Programmiersprachen (15.10.2014)

Date: Wed Oct 15 14:15:57 CEST 2014

Duration: 90:27 min

Pages: 85

Out-of-Order Execution

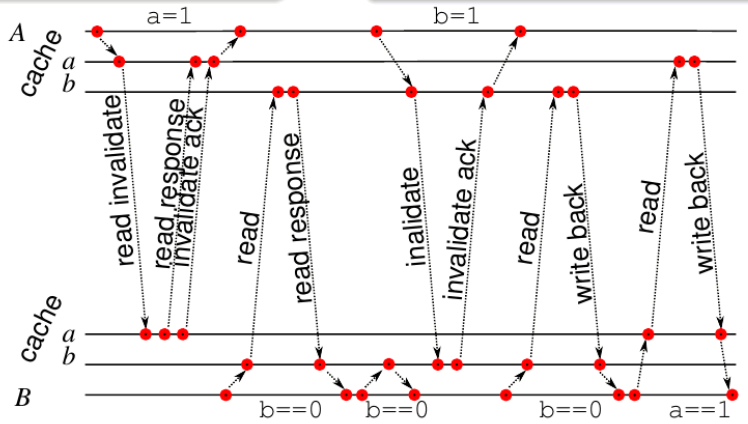
performance problem: writes always stall

Thread A

```
a = 1;      // A.1
b = 1;      // A.2
```

Thread B

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



Summary

Sequential consistency:

- a characterization of well-behaved programs
- a model for different speed of execution
- for fixed paths through the threads *and* a total order between accesses to the same variable: executions can be illustrated by happened-before diagram with one process per variable
- MESI cache coherence protocol ensures SC for processors with caches

Disproving Sequential Consistency

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

1. with operations $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$
2. there exists a total order $\exists C \,.\, C(p_i^j) < C(p_k^l)$ for all $i, j, k, l \ldots$ where $j = l$ implies $i < k$,
3. such that this execution has the same result.

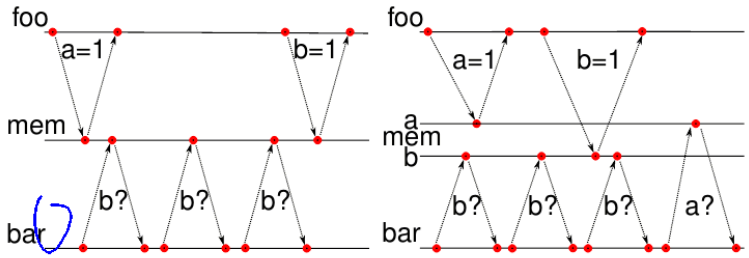
Idea for showing that a system is *not* sequentially consistent:

- pick a result obtained from a program run on a SC system
- pick an execution 1 and a total ordering of all operations 2
- add extra processes to model other system components
- the original order 2 becomes a partial order $\rightarrow$
- show that total orderings C' exist for $\rightarrow$ for which the result differ

Weakening the Model

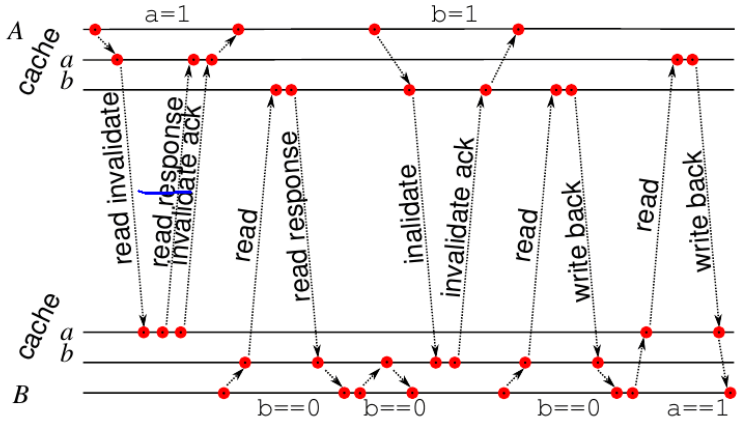
There is no observable change if calculations on different memory locations can happen in parallel.

- idea: model each memory location as a different process



MESI 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
  ⇝ add edge
- second execution of test `b==0` stays within cache ⇝ no traffic

Out-of-Order Execution

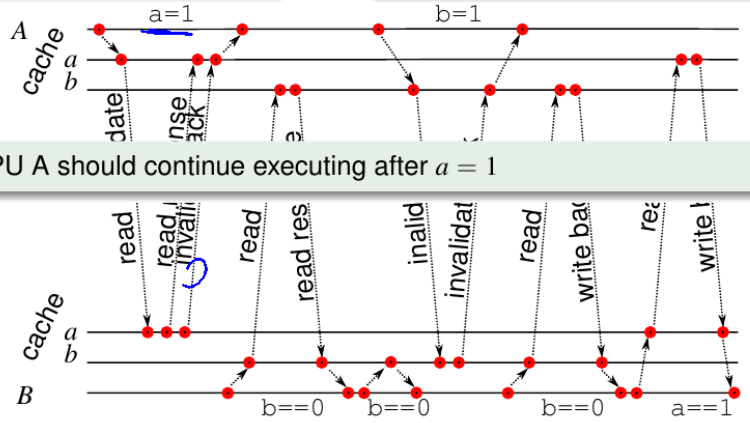
performance problem: writes always stall

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 $a = 1$

Out-of-Order Execution

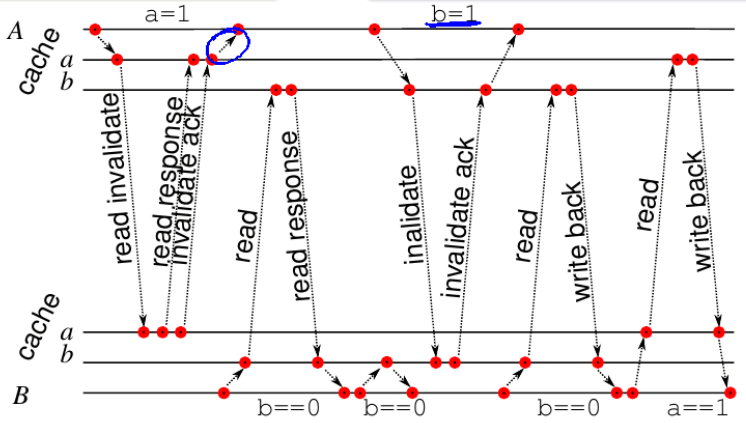
performance problem: writes always stall

Thread A

```
a = 1;      // A.1
b = 1;      // A.2
```

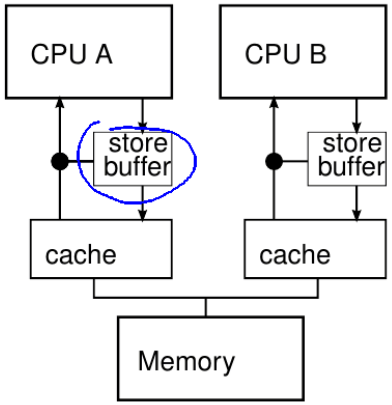
Thread B

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



Store Buffers

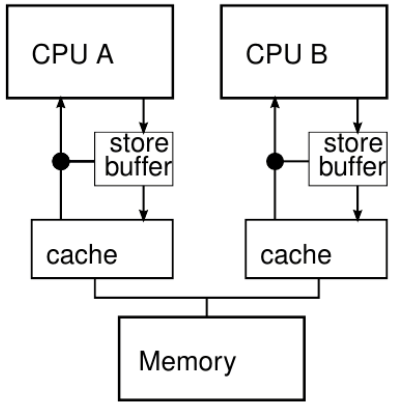
Goal: continue execution after *cache-miss* write operation



- put each write into a *store buffer* and trigger fetching of cache line

Store Buffers

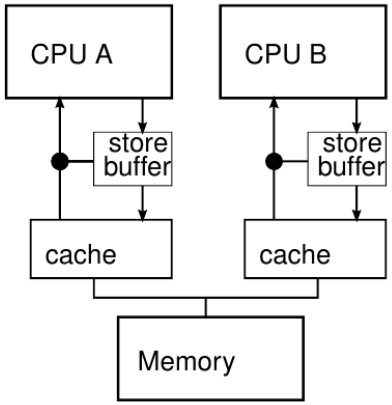
Goal: continue execution after *cache-miss* write operation



- put each write into a *store buffer* and trigger fetching of cache line
- once a cache line has arrived, apply relevant writes

Store Buffers

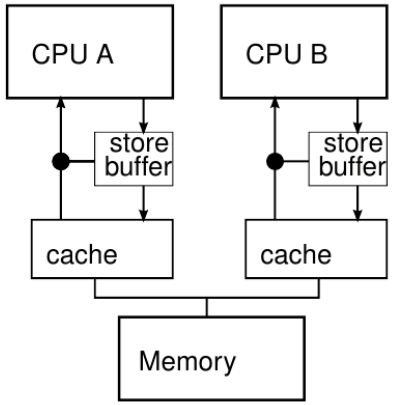
Goal: continue execution after *cache-miss* write operation



- 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]

Store Buffers

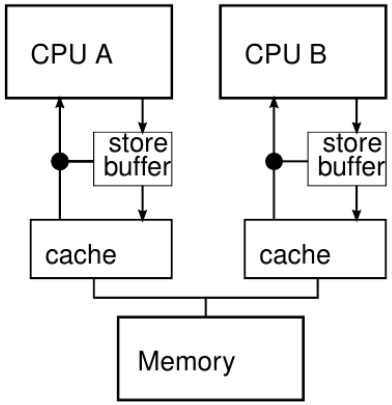
Goal: continue execution after *cache-miss* write operation



- 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

Store Buffers

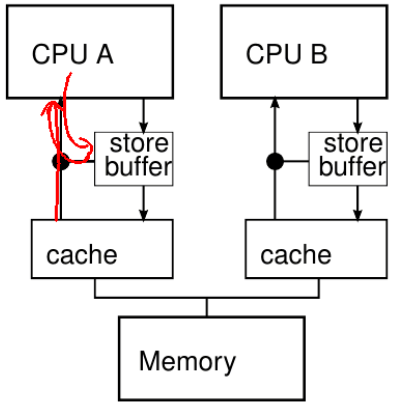
Goal: continue execution after *cache-miss* write operation



- 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

Store Buffers

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

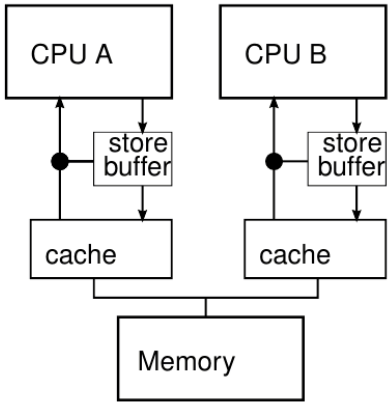


- put each write into a *store buffer* and trigger fetching of cache line
- once a cache line has arrived, apply relevant writes
  - today, a store buffer is always a *queue* [OSS09]
  - two writes to the same location are not merged
- ⚠ sequential consistency per CPU is violated unless
  - each read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

a=1 → SB → CACHE
assert(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]
  - two writes to the same location are not merged
- ⚠ sequential consistency per CPU is violated unless
  - each read checks store buffer before cache
  - on hit, return the youngest value that is waiting to be written

Happened-Before Model for Store Buffers

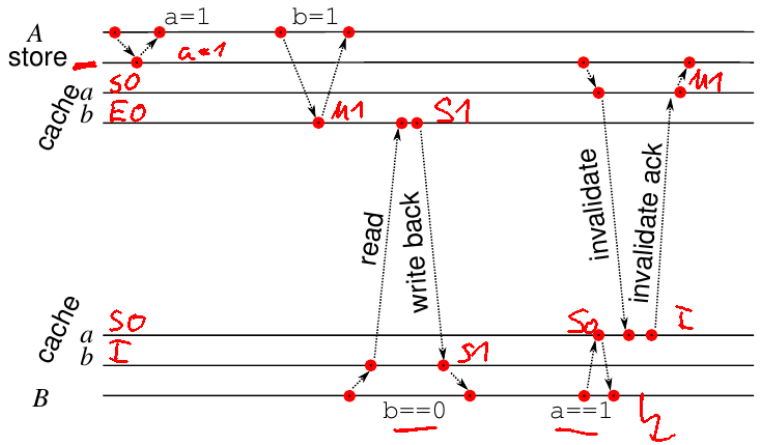
Thread A

```
a = 1;
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: 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

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

Memory Consistency | Out-of-Order Execution of Stores | 31 / 48

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

Memory Consistency | Out-of-Order Execution of Stores | 31 / 48

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

Memory Consistency | Out-of-Order Execution of Stores | 31 / 48

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)

Happened-Before Model for Write Fences

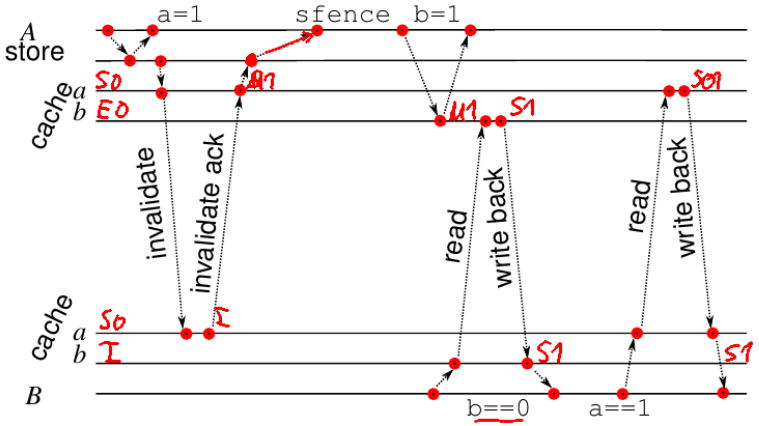
Thread A

```
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



Invalidate Queue

Invalidation of cache lines is costly:

- all CPUs in the system need to send an acknowledge

Happened-Before Model for Write Fences

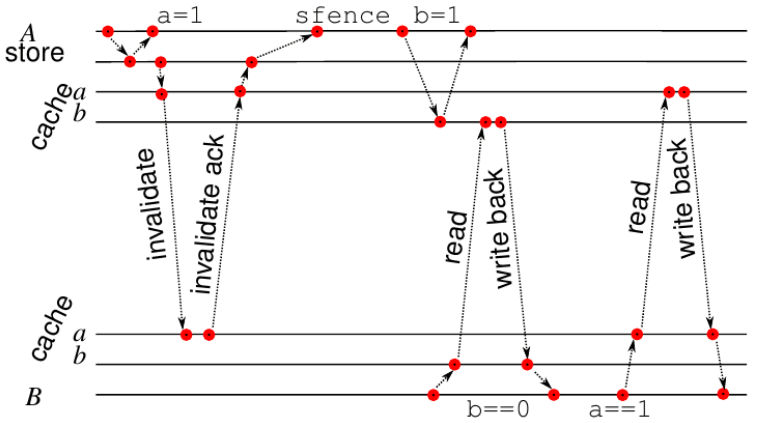
Thread A

```
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



Invalidate Queue

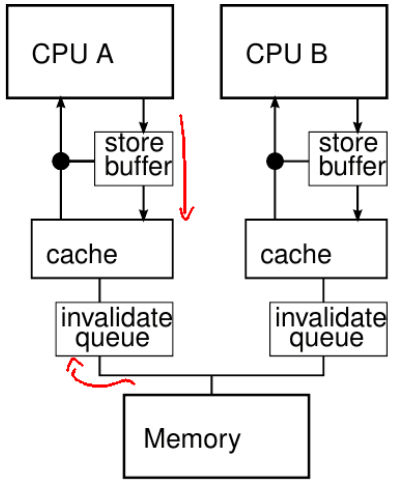
Invalidation of cache lines is costly:

- all CPUs in the system need to send an acknowledge
- invalidating a cache line competes with CPU accesses
- a cache-intense computation can fill up store buffers in other CPUs

Invalidate 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-intense computation can fill up store buffers in other CPUs

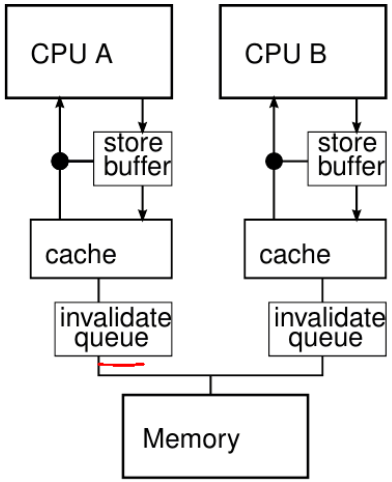


⇝ immediately acknowledge an invalidation and apply them later

Invalidate 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-intense computation can fill up store buffers in other CPUs



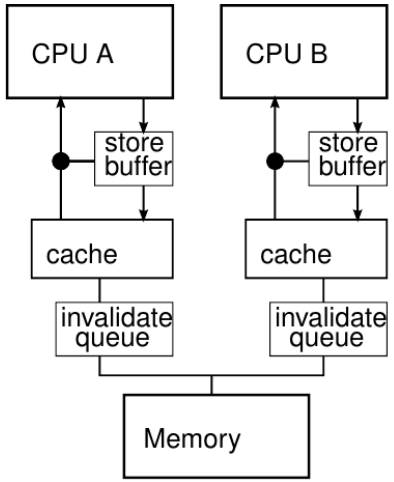
⇝ immediately acknowledge an invalidation and apply them later

- 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

Invalidate 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-intense 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

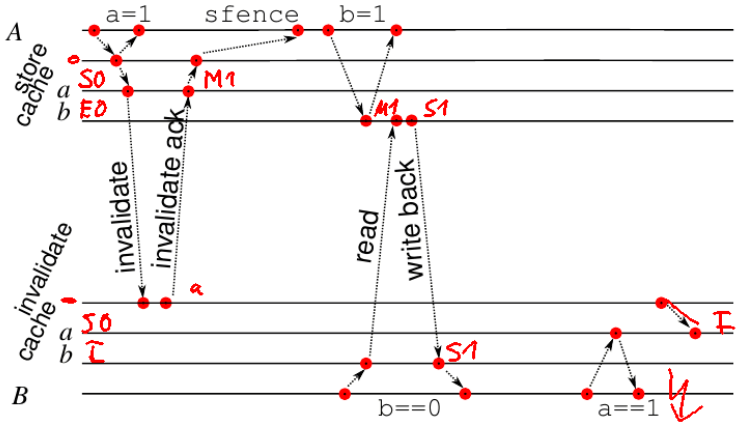
Thread A

```
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

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

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

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

Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.

- might read an out-of-date value

Memory Consistency | Out-of-Order Execution of Loads | 35 / 48

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)

$\rightsquigarrow$ match each write barrier in one process with a read barrier in another process

Memory Consistency | Out-of-Order Execution of Loads | 35 / 48

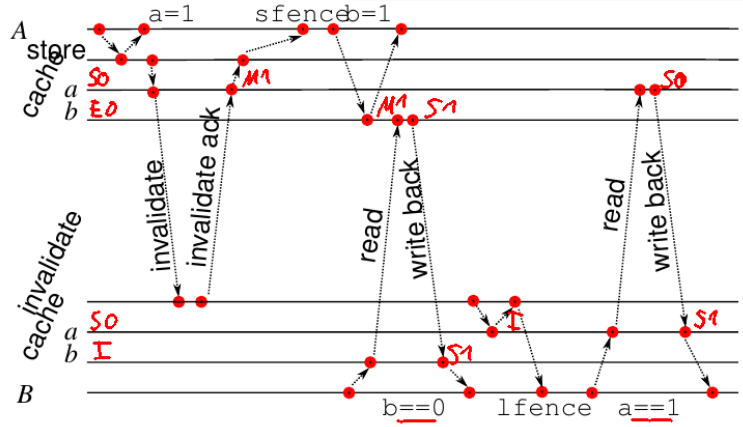
Happened-Before Model for Read Fences

Thread A

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

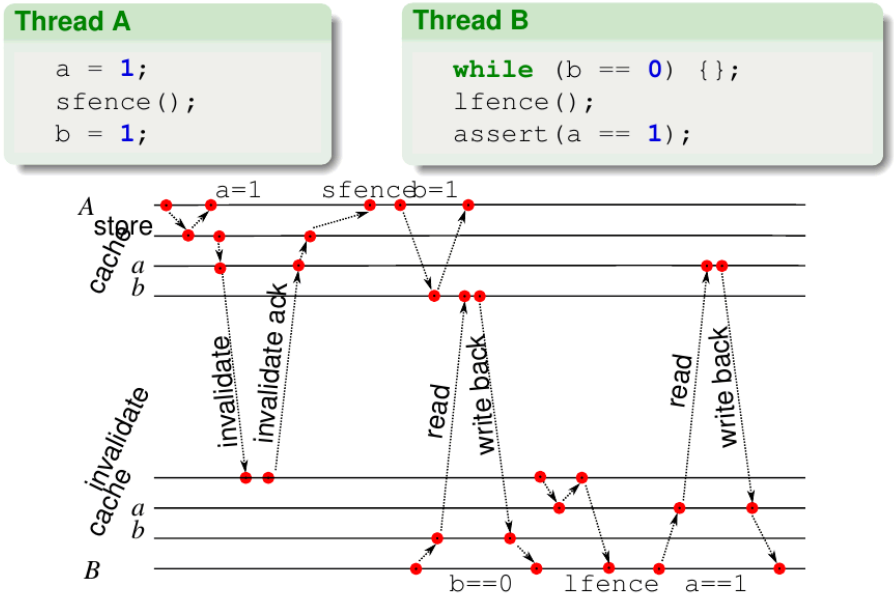
Thread B

```
while (b == 0) {};
lfence();
assert(a == 1);
```



Memory Consistency | Out-of-Order Execution of Loads | 36 / 48

Happened-Before Model for Read Fences



Summary: Weakly-Ordered Memory Models

Modern CPUs use a *weakly-ordered memory model*:

- reads and writes are not synchronized unless requested by the user

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 on barrier that is both, read and write (e.g. `mfence` on x86)

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 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

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 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++

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 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

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 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.

```
//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:

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

```
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;
```

In process P_i:

- if P_{1-i} does not want to enter, proceed immediately to the critical section

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

```
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;
```

In process P_i:

- if P_{1-i} does not want to enter, proceed immediately to the critical section

- $\rightsquigarrow$ `flag[i]` is a *lock* and may be implemented as such

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

```
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;
```

In process P_i:

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

- if P_{1-i} also wants to enter, wait for `turn` to be set to `i`

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

```
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;
```

In process P_i:

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

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

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 (*map* ∘ *reduce*) operation concurrently

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


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* ∘ *reduce*) operation concurrently

```
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 `f` and `g` in sequence
- the calculation in `f` of the *i*th iteration depends on iteration $i-1$

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* ∘ *reduce*) operation concurrently

```
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 `f` and `g` in sequence
- the calculation in `f` of the *i*th iteration depends on iteration $i-1$
- non-trivial program to parallelize

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* ∘ *reduce*) operation concurrently

```
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 `f` and `g` in sequence
- the calculation in `f` of the *i*th iteration depends on iteration $i-1$
- non-trivial program to parallelize
- idea: use two threads, one for `f` and one for `g`

Concurrent Reduce

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

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

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

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


Concurrent Reduce

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

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

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

```
Pg:
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?)

Memory Consistency The Dekker Algorithm 41 / 48

Concurrent Reduce

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

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

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

```
Pg:
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

Memory Consistency The Dekker Algorithm 41 / 48

Concurrent Reduce

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

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

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

```
Pg:
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

Memory Consistency The Dekker Algorithm 41 / 48

Concurrent Reduce

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

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

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

```
Pg:
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
- *ideal scenario*: keep busy during busy waiting

Generalization to Stream Processing

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

Generalization to Stream Processing

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

$\rightsquigarrow$ 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

Generalization to Stream Processing

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

$\rightsquigarrow$ 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.

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

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

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()`

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()`

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 . . .

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

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

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

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:

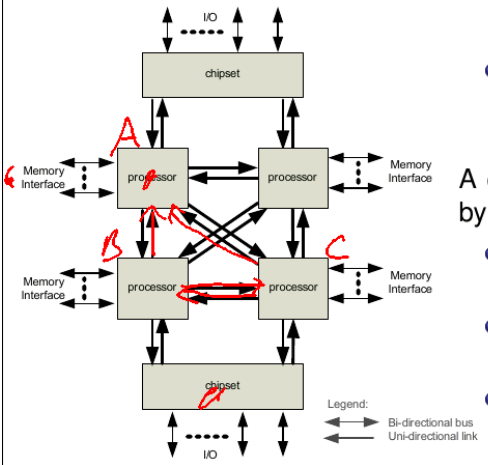
- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus

$\rightsquigarrow$ 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.



source: [Int09]

- 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.
  Technical Report 320412, 2009.
- Leslie Lamport.
  Time, Clocks, and the Ordering of Events in a Distributed System.
  Commun. ACM, 21(7):558–565, July 1978.
- Paul E. McKenney.
  Memory Barriers: a Hardware View for Software Hackers.
  Technical report, Linux Technology Center, IBM Beaverton, June 2010.
- Scott Owens, Susmit Sarkar, and Peter Sewell.
  A better x86 memory model: x86-TSO.
  Technical Report UCAM-CL-TR-745, University of Cambridge, Computer Laboratory, March 2009.