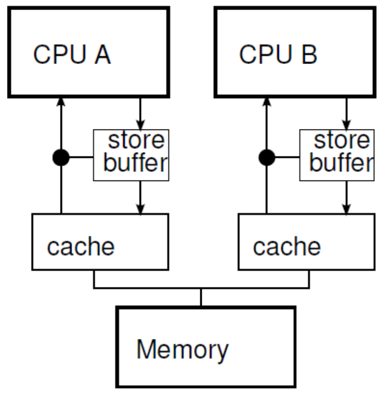
Script **generated by TTT**

Title:	Simon: Programmiersprachen (09.11.2012)
Date:	Fri Nov 09 11:05:51 CET 2012
Duration:	86:04 min
Pages:	111

Store Buffers

Goal: continue execution after write operation



- put each write into a *store buffer* and trigger reception of cache line
- once a cache line has arrived, apply relevant writes
  - store buffer is a *set*
- ⚠ sequential consistency per CPU is violated unless
  - each read checks store buffer before cache
  - on hit, return the value that is waiting to be written
  - a write to the same location is combined with an existing write

What about sequential consistency for the whole system?

Happened-Before Model for Store Buffers

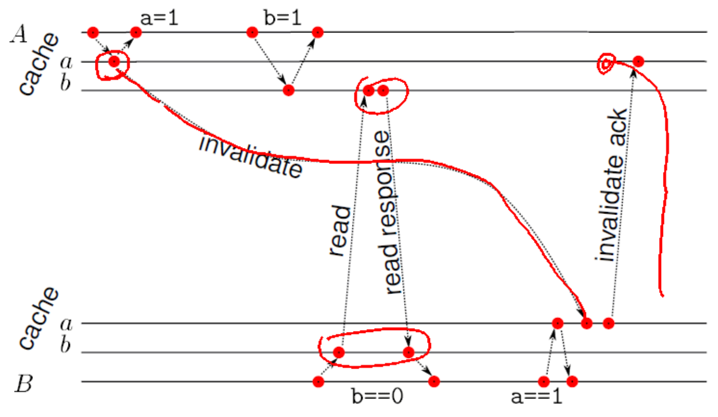
Thread A

```
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 Synchronisation: 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

Happened-Before Model for Write Fences

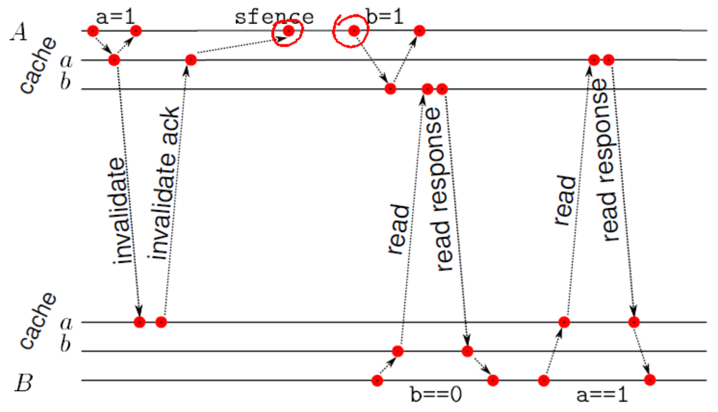
Thread A

```
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

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

Invalidate Queue

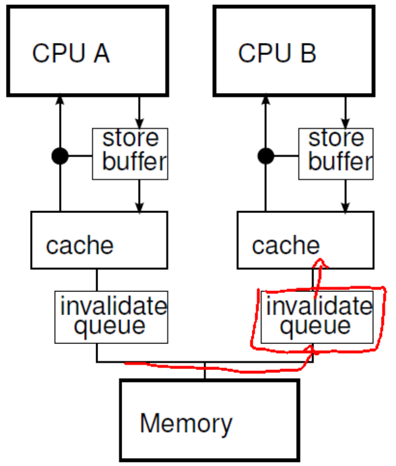
Invalidation of cache lines is costly:

- 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

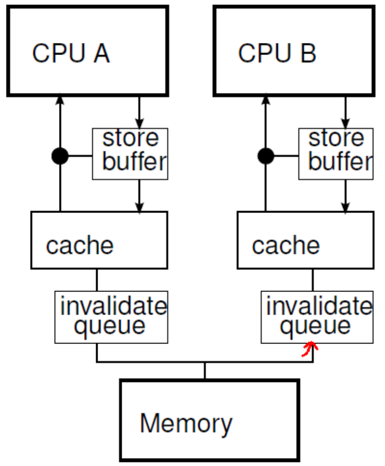


⇝ immediately acknowledge an invalidation and apply them later

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



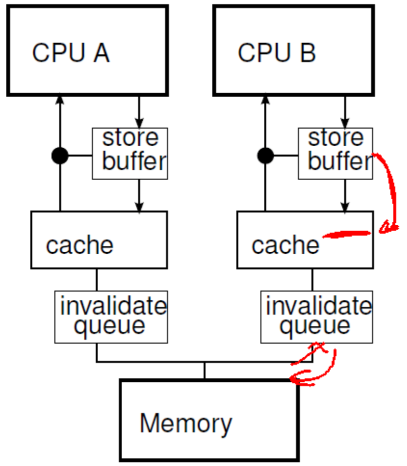
⇝ immediately acknowledge an invalidation and apply them later

- put each invalidate message into an *invalidate queue*

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



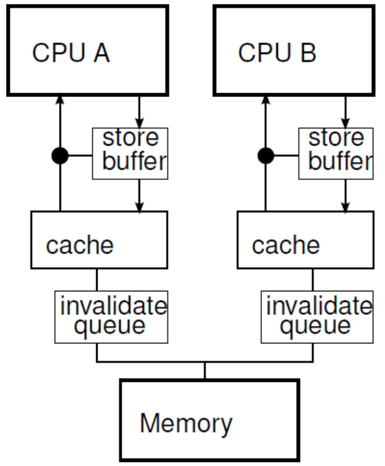
⇝ immediately acknowledge an invalidation and apply them later

- 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
- ⚠ read and writes do *not* consult the invalidate queue

Happened-Before Model for Invalidate Buffers

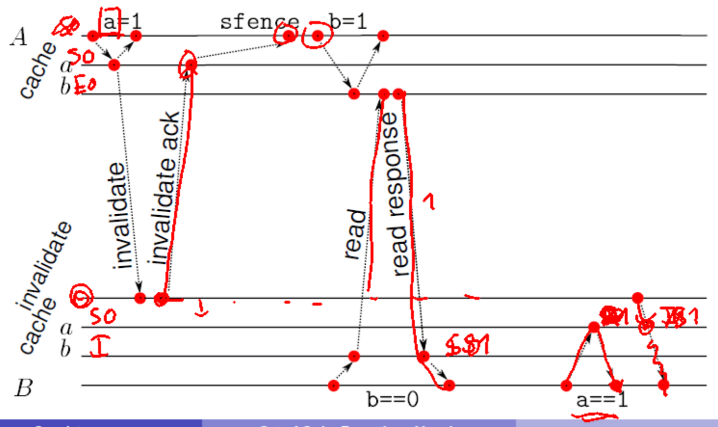
Thread A

```
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
- 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 & with barrier after every write 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
- a read barrier *before* each read gives sequentially consistent read behavior (and is as slow as a system without invalidate queue)

⇝ match each write barrier in one process with a read barrier in another process

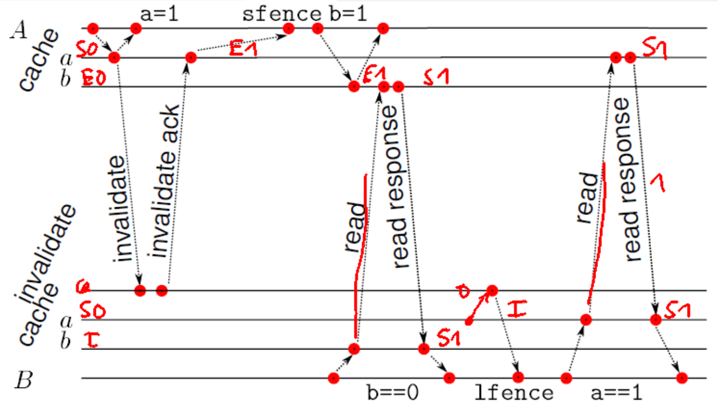
Happened-Before Model for Read Fences

Thread A

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

Thread B

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



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

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

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 lib_ atomic ops

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

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 $(map \circ fold)$ 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 $(map \circ fold)$ 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

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 $(map \circ fold)$ 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 ith iteration depends on iteration $i-1$

Concurrent Fold

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

```
T acc = init();
Buffer<U> buf = buffer<~~T~~u>(n); // some buffer object with lock
```

```
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();
  ~~acc =~~ g(tmp, i);
}
```

Concurrent Fold

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

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

```
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();
  acc = g(tmp, i);
}
```

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

Concurrent Fold

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

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

```
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();
  acc = 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 Fold

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

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

```
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();
  acc = 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

Concurrent Fold

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

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

```
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();
  acc = 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 Fold

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

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

```
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();
  acc = 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 $fold \circ fold$

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

Generalization to $fold \circ fold$

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

⇝ stream processing

- 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 $fold \circ fold$

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

⇝ stream processing

- 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

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

- insert a read memory barrier `lfence()` in front of every ~~write~~ read to 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;
     wfrence();
     while (lfence(), turn != 0) {
       // busy wait
     }
     flag[0] = true;
     sfence();
  }
// critical section
turn    = 1;
sfence();
flag[0] = false;
```

- insert a read memory barrier `lfence()` in front of every ~~write~~ to 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;
     wfrence();
     while (lfence(), turn != 0) {
       // busy wait
     }
     flag[0] = true;
     sfence();
  }
// critical section
turn    = 1;
sfence();
flag[0] = false;
```

- insert a read memory barrier `lfence()` in front of every write to 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

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

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
- might be less efficient than locks if store/invalidate buffers are full

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

Discussion

Memory barriers lie at the lowest level of synchronization primitives.
Where are they useful?

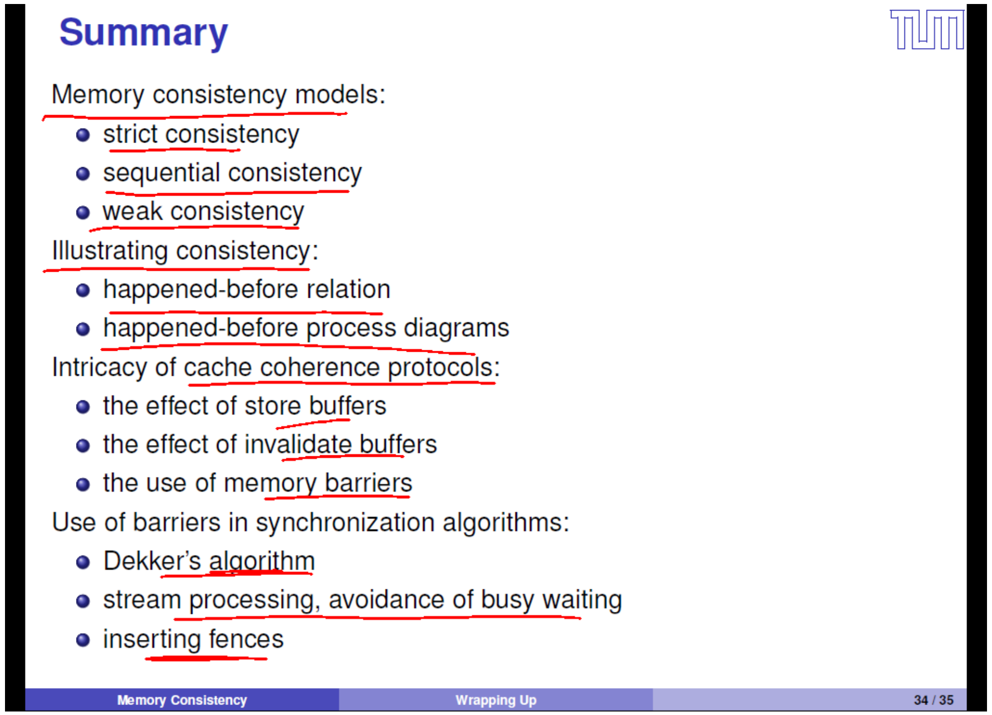
- when several processes implement an automaton and . . .
- synchronization means coordinating transitions of these automata
- when blocking should not de-schedule threads
- often used in operating systems

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
- might be less efficient than locks if store/invalidate buffers are full

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
- C++11: use of *atomic* variables will insert memory barriers
- Java,Go,...: there is little hope of enough control



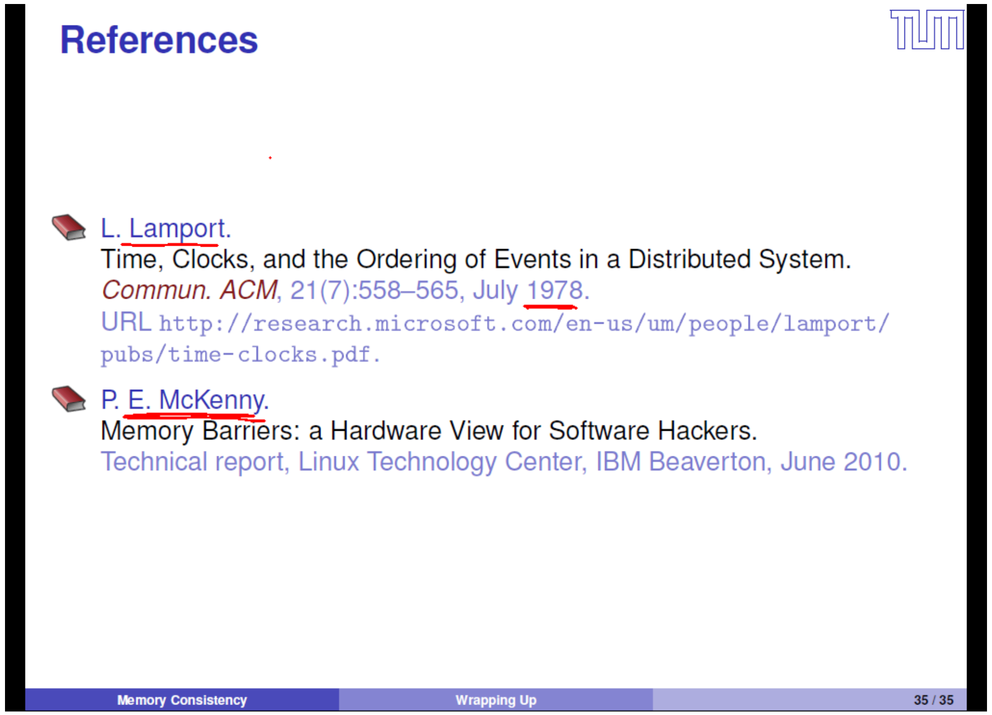
Summary

Memory consistency models:
- strict consistency
- sequential consistency
- weak consistency

Illustrating consistency:
- happened-before relation
- happened-before process diagrams

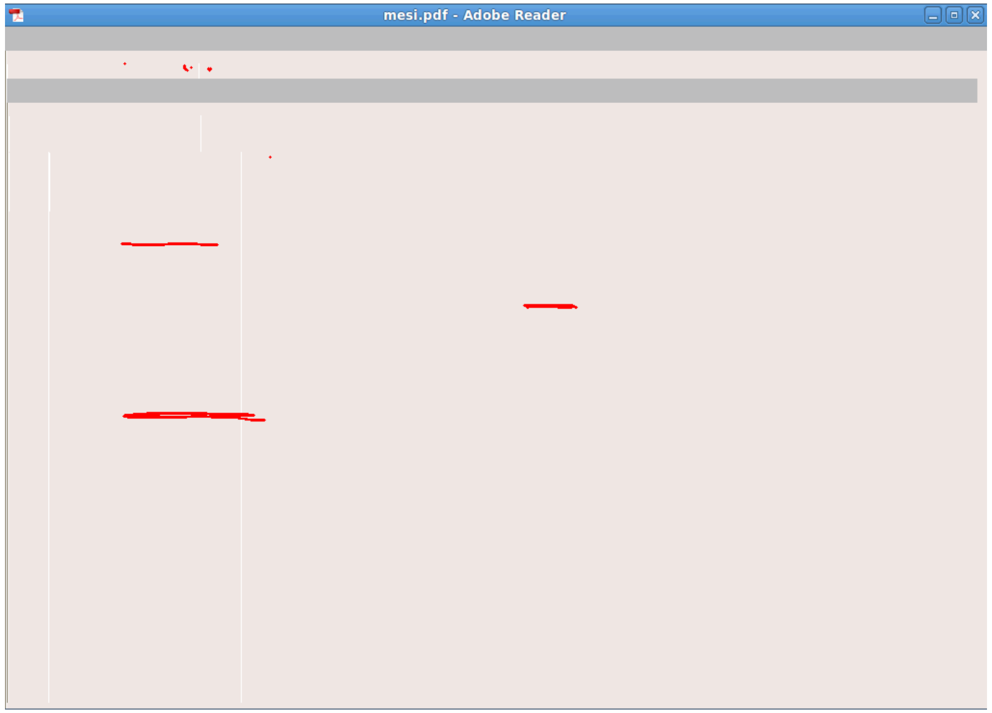
Intricacy of cache coherence protocols:
- the effect of store buffers
- the effect of invalidate buffers
- the use of memory barriers

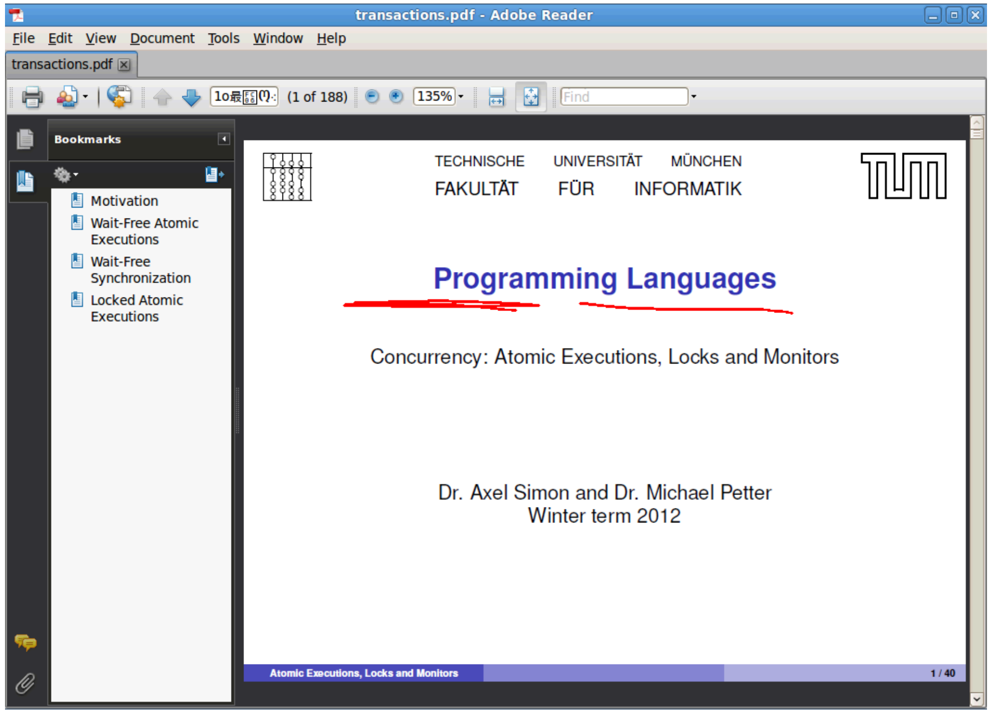
Use of barriers in synchronization algorithms:
- Dekker's algorithm
- stream processing, avoidance of busy waiting
- inserting fences



References

- L. Lamport.
  Time, Clocks, and the Ordering of Events in a Distributed System.
  Commun. ACM, 21(7):558–565, July 1978.
  URL `http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf`.
- P. E. McKenny.
  Memory Barriers: a Hardware View for Software Hackers.
  Technical report, Linux Technology Center, IBM Beaverton, June 2010.




transactions.pdf - Adobe Reader
File Edit View Document Tools Window Help
transactions.pdf
(1 of 188)
135%
Find
Bookmarks

- Motivation
- Wait-Free Atomic Executions
- Wait-Free Synchronization
- Locked Atomic Executions

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Atomic Executions, Locks and Monitors

Dr. Axel Simon and Dr. Michael Petter
Winter term 2012

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)

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*
- $\leadsto$ generalize the re-occurring concept of enforcing mutual exclusion

Why Memory Barriers are not Enough

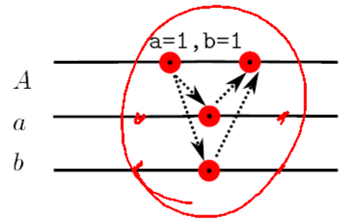
Communication via memory barriers has only specific applications:

- 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*
- $\leadsto$ generalize the re-occurring concept of enforcing mutual exclusion

Need a mechanism to update these pieces of memory as a single *atomic execution*:



- several values of the objects are used to compute new value
- certain information form 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

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

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

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
- during an update, an invariant may be *broken*
- an invariant may span *several* resources

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
- during an update, an invariant may be *broken*
- an invariant may span *several* resources
- $\leadsto$ several resources must be updated together to ensure the invariant
- which particular resources need to be updated may depend on the current program state

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
- during an update, an invariant may be *broken*
- an invariant may span *several* resources
- $\rightsquigarrow$ several resources must be updated together to ensure the invariant
- which particular resources need to be updated may depend on the current program state

Ideally, we want to mark a sequence of operations that update shared resources for *atomic execution* [2]. This would ensure that the invariant never seem to be broken.

Overview

We will address the *established* ways of managing synchronization.

- present techniques are available on most platforms

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

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

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

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

1. Principle of Atomic Executions
2. Wait-Free Algorithms based on Atomic Operations
3. Locks: Mutex, Semaphore, and Monitor
4. 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.

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)

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 Updates

Which operations on a CPU are atomic executions?

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

Wait-Free Updates

Which operations on a CPU are atomic executions?

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

mov rax, [&i]

Answer: add rax, [illegible] mov [&i], rax   inc [&i]

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

Wait-Free Updates

Which operations on a CPU are atomic executions?

Program 1
```
i++;
```

Program 2
```
j = i;
i = i+k;
```

Program 3
```
int tmp = i;
i = j;
j = tmp;
```

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:

Wait-Free Updates

Which operations on a CPU are atomic executions?

Program 1
```
i++;
```

Program 2
```
j = i;
i = i+k;
```

Program 3
```
int tmp = i;
i = j;
j = tmp;
```

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. declare as volatile)
- most CPUs can *lock* the cache for the duration of an instruction; on x86:

Wait-Free Updates

Which operations on a CPU are atomic executions?

Program 1
```
i++;
```

Program 2
```
j = i;
i = i+k;
```

Program 3
```
int tmp = i;
i = j;
j = tmp;
```

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. declare as volatile)
- most CPUs can *lock* the cache for the duration of an instruction; on x86:
- Program 1 can be implemented using a `lock inc [addr_i]` instruction

Wait-Free Updates

Which operations on a CPU are atomic executions?

Program 1
```
i++;
```

Program 2
```
j = i;
i = i+k;
```

Program 3
```
int tmp = i;
i = j;
j = tmp;
```

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. declare as volatile)
- most CPUs can *lock* the cache 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 [addr_j],eax`

Wait-Free Updates

Which operations on a CPU are atomic executions?

Program 1
```
i++;
```

Program 2
```
j = i;
i = i+k;
```

Program 3
```
int tmp = i;
i = j;
j = tmp;
```

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. declare as volatile)
- most CPUs can *lock* the cache 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 [addr_j],eax`
- Program 3 can be implemented using ~~`lock`~~ `xchg [addr_i],[addr_j]`

Wait-Free Updates

Which operations on a CPU are atomic executions?

Program 1
```
i++;
```

Program 2
```
j = i;
i = i+k;
```

Program 3
```
int tmp = i;
i = j;
j = tmp;
```

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. declare as volatile)
- most CPUs can *lock* the cache 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 [addr_j],eax`
- Program 3 can be implemented using `lock xchg [addr_i],[addr_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
```
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`

Wait-Free Bumper-Pointer Allocation

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

Bumper Pointer Allocation
```
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`

Wait-Free Bumper-Pointer Allocation

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

Bumper Pointer Allocation

```
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`

Thread-safe implementation:

- the `alloc` function can be used from multiple threads when implemented using a `lock xadd [_firstFree],eax` instruction
- ⇝ requires inline assembler

Marking Statements as Atomic

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

Program 1

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

Program 2

```
atomic {
  j = i;
  i = i+k;
}
```

Program 3

```
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```

Marking Statements as Atomic

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

Program 1

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

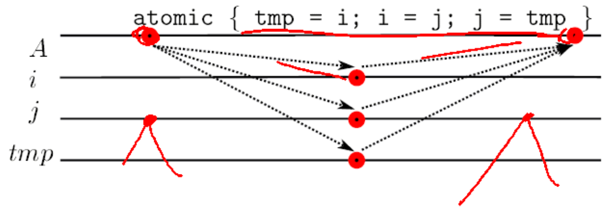
Program 2

```
atomic {
  j = i;
  i = i+k;
}
```

Program 3

```
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```

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



Marking Statements as Atomic

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

Program 1

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

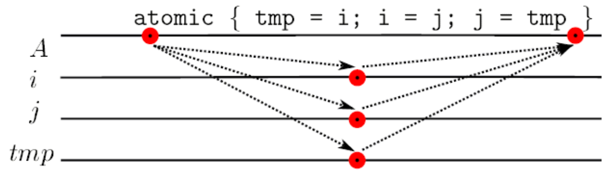
Program 2

```
atomic {
  j = i;
  i = i+k;
}
```

Program 3

```
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```

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



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

Marking Statements as Atomic

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

Program 1

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

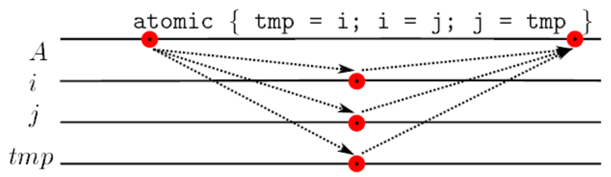
Program 2

```
atomic {
  j = i;
  i = i+k;
}
```

Program 3

```
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```

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



- `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

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:

- no control flow possible, no behavioral change depending on data
- instructions often exist that execute an operation conditionally

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:

- no control flow possible, no behavioral change depending on data
- instructions often exist that execute an operation conditionally

Program 4

```
atomic {
  r = b;
  b = 0;
}
```

Program 5

```
atomic {
  r = b;
  b = 1;
}
```

Program 6

```
atomic {
  r = (k==i);
  if (r) i = j;
}
```

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:

- no control flow possible, no behavioral change depending on data
- instructions often exist that execute an operation conditionally

Program 4 6

```
atomic {
  r = b;
  b = 0;
}
```

Program 5

```
atomic {
  r = b;
  b = 1;
}
```

Program 6 0

```
atomic {        k=1
  r = (k==i);   j=0
  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` not already contains v
  - 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
- instructions often exist that execute an operation conditionally

Program 4

```
atomic {
  r = b;
  b = 0;
}
```

Program 5

```
atomic {
  r = b;
  b = 1;
}
```

Program 6

```
atomic {
  r = (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` not already contains v
  - this operation is called *modify-and-test*
- the third case generalizes this to arbitrary values
  - this operation is called *compare-and-swap*

$\rightsquigarrow$ use as building blocks for algorithms that can *fail*

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.

Common usage pattern for *compare and swap*: calculate $i = f(i);$

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

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.
Common usage pattern for *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

$\rightsquigarrow$ general recipe for *lock-free* algorithms

- given a 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 compare-and-swap operation on these n bytes

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.
Common usage pattern for *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

$\rightsquigarrow$ general recipe for *lock-free* algorithms

- given a 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 compare-and-swap operation on these n bytes

$\rightsquigarrow$ calculating new value must be *repeatable*

Limitations of Wait- and Lock-Free Algorithms

Wait-/Lock-Free algorithms are severely limited in terms of their computation:

- restricted to the semantics of a *single* atomic operations

Limitations of Wait- and Lock-Free Algorithms

Wait-/Lock-Free algorithms are severely limited in terms of their computation:

- restricted to the semantics of a *single* atomic operations
- set of atomic operations is architecture specific, but often includes

Limitations of Wait- and Lock-Free Algorithms

Wait-/Lock-Free algorithms are severely limited in terms of their computation:

- restricted to the semantics of a *single* atomic operations
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand

Limitations of Wait- and Lock-Free Algorithms

Wait-/Lock-Free algorithms are severely limited in terms of their computation:

- restricted to the semantics of a *single* atomic operations
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand

⇝ only very simple algorithms can be implemented, for instance

binary semaphores : a flag that can be acquired (set) if free (unset) and released

Limitations of Wait- and Lock-Free Algorithms

Wait-/Lock-Free algorithms are severely limited in terms of their computation:

- restricted to the semantics of a *single* atomic operations
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand

⇝ only very simple algorithms can be implemented, for instance

binary semaphores : a flag that can be acquired (set) if free (unset) and released

counting semaphores : an integer that can be decreased if non-zero and increased

mutex : ensures mutual exclusion using a binary semaphore

Semaphores and Mutexes

A (counting) *semaphore* is an integer `s` with the following operations:

```
void signal() {
  atomic { s = s + 1; }
}
```

```
void wait() {
  bool avail;
  do {
    atomic {
      avail = s>0;
      if (avail) s--;
    }
  } while (!avail);
}
```

A counting semaphore can track how many resources are still available.