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
\[
\begin{align*}
    a &= 1; \\
    &\text{sfence();} \\
    b &= 1;
\end{align*}
\]

Thread B
\[
\begin{align*}
    \text{while } (b == 0) \{} \\
    &\text{assert(a == 1);} \\
    \text{\}}
\end{align*}
\]

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-intensive computation can fill up store buffers in other CPUs
**Invalidate Queue**

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

```
~ immediately acknowledge an invalidate and apply them later
```

Invalidate Queue

```
~ 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

```
~ 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
```
Happened-Before Model for Invalidate Buffers

Thread A

\[
\begin{align*}
a &= 1; \\
sfence(); \\
b &= 1;
\end{align*}
\]

Thread B

\[
\begin{align*}
\text{while} \ (b == 0) \ {} \\
\text{assert}(a == 1);
\end{align*}
\]

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

Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.
- might read an out-of-date value
- need a way to establish sequential consistency between writes of other processors and local reads

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 `sfence` 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)

---

Happened-Before Model for Read Fences

**Thread A**

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

**Thread B**

```c
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
Modern CPUs use a `weakly-ordered memory model`:
- reads and writes are not synchronized unless requested by the user
- many kinds of memory barriers exist with subtle differences
- most systems provide a barrier that is both, `read` and `write` (e.g. `mfence` on x86)
- 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
- otherwise inline assembler has to be used

Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of two processes with busy waiting.

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

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

P1:
flag[1] = true;
while (flag[0] == true)
    if (turn ! = 0) {
        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

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

In process \( P_i \):

- if \( P_i \) 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

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

In process \( P_i \):

- if \( P_i \) does not want to enter, proceed immediately to the critical section
- **\( \sim \) 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

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

In process \( P_i \):

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

```java
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 $P_i$ does not want to enter, proceed immediately to the critical section
- $\sim$ flag[i] is a lock and may be implemented as such
- if $P_i$, also wants to enter, wait for turn to be set to $i$
- while waiting for turn, reset flag[i] to enable $P_i$ 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
- **deadlock free**: the process will never wait for each other
- **free of starvation**: if a process wants to enter, it eventually will
A Note on Dekker’s Algorithm

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

Applications for Dekker: implement a \((\text{map} \circ \text{fold})\) operation concurrently

```java
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 \((\text{map} \circ \text{fold})\) operation concurrently

```java
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 \(i - 1\)

Concurrent Fold

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

```java
T acc = init();
Buffer<\{U\} buf = buffer\{U\}(n); // some buffer object with lock
```

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

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

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

T acc = init();
Buffer< Acc > 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,U> (acc, tmp) = f(acc, i);
   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.

Memory Consistency: The Dekker Algorithm
**Concurrent Fold**

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

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

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 $f$.

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

**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.

Before:
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;

After:

```c
PO:
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;
```

- 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

Memory Consistency | The Dekker Algorithm
--- | ---
32 / 35 | 32 / 35
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
- difficult to get right, possibly inappropriate except for specific, proven algorithms
- often synchronization with locks is as fast and easier

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/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.

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

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)

Often certain pieces of memory may only be modified by one thread at once.
- can use barriers to implement automata that ensure mutual exclusion
- generalize the re-occurring concept of enforcing mutual exclusion

Why Memory Barriers are not Enough

Communication via memory barriers has only specific applications:
- coordinating state transitions between threads
- for systems that require minimal overhead (and no de-scheduling)

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

Diagram:
- Several values of the objects are used to compute new value
- Certain information from the thread flows into this computation
- Certain information flows from the computation to the thread
Atomic Executions

A concurrent program consists of several threads that share common resources:

- resources are often pieces of memory, but may be an I/O entity
  - a file can be modified through a shared handle
- for each resource an invariant must be retained
  - a head and tail pointer must define a linked list
- 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
- 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 seems to be broken.

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

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

Learning Outcomes

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

**Definition (Atomic Execution)**

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

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

These classes differ in:
- **amount of data** they can access during an atomic execution
- **expressivity** of operations they allow
- **granularity** of objects in memory they require

---

**Wait-Free Updates**

Which operations on a CPU are atomic executions?

**Program 1**

```plaintext
i++;
```

**Program 2**

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

**Program 3**

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

**Answer:**
- Some (e.g., `inc`, `dec`, `cmpxchg`)
- 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?

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:

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
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**

```c
i++;
```

**Program 2**

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

**Program 3**

```c
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:

```c
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:

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

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

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

Thread-safe implementation:
- the **alloc** function can be used from multiple threads when implemented using a lock/swap/firstFree, eax instruction
- requires inline assembler

Marking Statements as Atomic

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

**Program 1**

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

**Program 2**

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

**Program 3**

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

```
A
i
j

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

A
i
j

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

- `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`:

```
A
\--\-
  \--\-
    \--\-
      \--\-
        \--\-
           \--\-

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

- `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
- 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

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**

---

**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 swap:
- read the initial value in $i$ into $k$ (using memory barriers)
- calculate a new value $j = f(k)$
- update $i$ to $j$ if $i = k$ still holds
- 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
~ general recipe for *lock-free* algorithms
   1. given a compare-and-swap operation for \( n \) bytes
   2. try to group variables for which an invariant must hold into \( n \) bytes
   3. read these bytes atomically
   4. calculate a new value
   5. perform a compare-and-swap operation on these \( n \) bytes

Limitations of Wait- and Lock-Free Algorithms

Wait-/Lock-Free algorithms are severely limited in terms of their computation:
1. 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:
1. restricted to the semantics of a *single* atomic operations
2. 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

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

Semaphores and Mutexes

A (counting) \textit{semaphore} is an integer with the following operations:

```c
void wait() {
    bool avail;
    do {
        atomic {
            s = s + 1;
        }
        if (avail) s--;
    } while (!avail);
}
```

\textit{A counting semaphore can track how many resources are still available.}

\textit{mutex}: ensures mutual exclusion using a binary semaphore