Why Memory Barriers are not Enough

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

Often certain pieces of memory may only be modified by one thread at once.
- can use barriers to implement automata that ensure mutual exclusion
- generalize the re-occurring concept of enforcing mutual exclusion
Why Memory Barriers are not Enough

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

\[ \text{Atomic Executions} \]

- 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

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
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 [Harris et al.(2010)](Harris, Larus, and Rajwar). 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
- 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.

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)
Atomic Execution: Varieties

**Definition (Atomic Execution)**
A computation forms an atomic execution if its effects 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? (\(j\) and \(tmp\) are registers)

**Program 1**
\[ i++ \]
**Program 2**
\[ j = i; \]
\[ i = i + k; \]
**Program 3**
\[ \text{int } tmp = i; \]
\[ i = j; \]
\[ j = tmp; \]

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

---

Wait-Free Updates

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

**Program 1**
\[ i++ \]
**Program 2**
\[ j = i; \]
\[ i = i + k; \]
**Program 3**
\[ \text{int } tmp = i; \]
\[ i = j; \]
\[ j = tmp; \]

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

The programs can be atomic executions:
- \textit{i} must be in memory (e.g. declare as volatile)
Wait-Free Updates

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

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 reg.j,eax

Wait-Free Updates

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

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 xchg [addr.i],reg.j
  - Program 3 can be implemented using lock xchg [addr.i],reg.j
Wait-Free Updates

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

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 reg.j,eax
  - Program 3 can be implemented using lock xchg [addr.i],reg.j

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

Wait-Free Bumper-Pointer Allocation

Garbage collectors often use a bumper pointer to allocated memory:

Bumper Pointer Allocation

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

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

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

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

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

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

Program 6
```plaintext
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

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:
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

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

---

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

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

---

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

---

calculating new value must be repeatable
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:

- 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

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

~~ 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

~~ 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 operation
- 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
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 operation
- 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
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 operation
- 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
  monitor: ensures mutual exclusion using a binary semaphore, allows other threads to block until the next release of the resource

We will collectively refer to these data structures as locks.

Locks

A lock is a data structure that

- protects a critical section: a piece of code that may produce incorrect results when executed concurrently from several threads
- it ensures mutual exclusion: no two threads execute at once
- block other threads as soon as one thread executes the critical section
- can be acquired and released
- may deadlock the program
A (counting) semaphore is an integer $s$ with the following operations:

```c
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.
- a thread requiring a resource executes `wait()`
- if a resource is still available, `wait()` returns
- once a thread finishes using a resource, it calls `signal()`
- (choosing which available resource to use requires more synchr.)

Special case: initializing with $s = 1$ gives a binary semaphore:
Semaphores and Mutexes

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

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

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

- A thread requiring a resource executes `wait()`
- If a resource is still available, `wait()` returns
- Once a thread finishes using a resource, it calls `signal()`
- (Choosing which available resource to use requires more synchr.)

Special case: initializing with \( s = 1 \) gives a binary semaphore:

- Can be used to block and unblock a thread

### Implementation of Semaphores

A semaphore does not have to busy wait:

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

Busy waiting is avoided by placing waiting threads into queue:

```c
if (!avail) do schedule(&s);
while (!avail);
```
Implementation of Semaphores

A *semaphore* does not have to busy wait:

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

Busy waiting is avoided by placing waiting threads into queue:

- a thread failing to decrease *s* executes `de_schedule()`
- `de_schedule()` enters the operating system and adds the waiting thread into a queue of threads waiting for a write to memory address &s
- once a thread calls `signal()`, the first thread *i* waiting on &s is extracted

---

Implementation of Semaphores

A *semaphore* does not have to busy wait:

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

Busy waiting is avoided by placing waiting threads into queue:

- a thread failing to decrease *s* executes `de_schedule()`
- `de_schedule()` enters the operating system and adds the waiting thread into a queue of threads waiting for a write to memory address &s
- once a thread calls `signal()`, the first thread *i* waiting on &s is extracted
- the operating system lets *i* return from its call to `de_schedule()`
Practical Implementation of Semaphores

Certain optimisations are possible:

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

In general, the implementation is more complicated

- `wait()` may busy wait for a few iterations
- `signal()` might have to inform the OS that `s` has been written

Practical Implementation of Semaphores

Certain optimisations are possible:

```c
void signal() {
    atomic { s = s + 1; }
    if (avail) s--;
}
```
Making a Queue Thread-Safe

Consider a double ended queue:

```
double-ended queue:
left sentinel
```

```
void PushLeft(DQueue* q, int val) {
    QNode* qn = malloc(sizeof(QNode));
    qn->val = val;
    // prepend node qn
    QNode* leftSentinel = q->left;
    QNode* oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode -> left = qn;
}
```

Mutexes

One common use of semaphores is to guarantee mutual exclusion.
- in this case, a binary semaphore is also called a mutex

```
Mutexes
in this case, a binary semaphore is also called a mutex
```

```
void PushLeft(DQueue* q, int val) {
    QNode* qn = malloc(sizeof(QNode));
    qn->val = val;
    // wait to enter the critical section
    QNode* leftSentinel = q->left;
    QNode* oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode -> left = qn;
    signal(q->s); // signal that we’re done
}
```

Implementing the Removal

By using the same lock q->s, we can write a thread-safe PopRight:

```
double-ended queue: removal
```

```
int PopRight(DQueue* q) {
    QNode* oldRightNode;
    QNode* leftSentinel = q->left;
    QNode* rightSentinel = q->right;
    wait(q->s); // wait to enter the critical section
    oldRightNode = rightSentinel->left;
    if (oldRightNode==leftSentinel) { signal(q->s); return -1; }
    QNode* newRightNode = oldRightNode->left;
    newRightNode->right = rightSentinel;
    rightSentinel->left = newRightNode;
    signal(q->s); // signal that we’re done
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```
Implementing the Removal

By using the same lock $q \rightarrow s$, we can write a thread-safe PopRight:

```c
int PopRight(DQueue* q) {
    QNode* oldRightNode;
    QNode* leftSentinel = q->left;
    QNode* rightSentinel = q->right;
    wait(q->s); // wait to enter the critical section
    oldRightNode = rightSentinel->left;
    if (oldRightNode==leftSentinel) { signal(q->s); return -1; }
    QNode* newRightNode = oldRightNode->left;
    newRightNode->right = rightSentinel;
    rightSentinel->left = newRightNode;
    signal(q->s); // signal that we're done
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```

- error case complicates code $\leftrightarrow$ semaphores are easy to get wrong
- abstract common concept: take lock on entry, release on exit

Mutexes

One common use of semaphores is to guarantee mutual exclusion.
- in this case, a binary semaphore is also called a **mutex**
- add a lock to the double-ended queue data structure
- decide what needs protection and what not

```c
void PushLeft(DQueue* q, int val) {
    QNode *qn = (QNode*) malloc(sizeof(QNode));
    qn->val = val;
    wait(q->s); // wait to enter the critical section
    QNode* leftSentinel = q->left;
    QNode* oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
    signal(q->s); // signal that we're done
}
```

Monitors: An Automatic, Re-entrant Mutex

Often, a data structure can be made thread-safe by
- acquiring a lock upon entering a function of the data structure
- releasing the lock upon exit from this function

Locking each procedure body that accesses a data structure:
Monitors: An Automatic, Re-entrant Mutex

Often, a data structure can be made thread-safe by
- acquiring a lock upon entering a function of the data structure
- releasing the lock upon exit from this function

Locking each procedure body that accesses a data structure:
- is a re-occurring pattern, should be generalized
- becomes problematic in recursive calls: it blocks
- if a thread \( t \) waits for a data structure to be filled:
  - \( t \) will call e.g. \texttt{PopRight} and obtain \texttt{-1}
  - \( t \) then has to call again, until an element is available

---

Monitors: An Automatic, Re-entrant Mutex

Often, a data structure can be made thread-safe by
- acquiring a lock upon entering a function of the data structure
- releasing the lock upon exit from this function

Locking each procedure body that accesses a data structure:
- is a re-occurring pattern, should be generalized
- becomes problematic in recursive calls: it blocks
- if a thread \( t \) waits for a data structure to be filled:
  - \( t \) will call e.g. \texttt{PopRight} and obtain \texttt{-1}
  - \( t \) then has to call again, until an element is available
  - \( t \) is busy waiting and produces contention on the lock
**Monitors: An Automatic, Re-entrant Mutex**

Often, a data structure can be made thread-safe by

- acquiring a lock upon entering a function of the data structure
- releasing the lock upon exit from this function

Locking each procedure body that accesses a data structure:

- is a re-occurring pattern, should be generalized
- becomes problematic in recursive calls: it blocks
- if a thread \( t \) waits for a data structure to be filled:
  - \( t \) will call e.g. `PopRight` and obtain \(-1\)
  - \( t \) then has to call again, until an element is available
  - \( \triangleright \) \( t \) is busy waiting and produces contention on the lock

**Monitor:** a mechanism to address these problems:

- a procedure associated with a monitor acquires a lock on entry and releases it on exit
- if that lock is already taken, proceed if it is taken by the current thread

**Monitors: An Automatic, Re-entrant Mutex**

Often, a data structure can be made thread-safe by

- acquiring a lock upon entering a function of the data structure
- releasing the lock upon exit from this function

Locking each procedure body that accesses a data structure:

- is a re-occurring pattern, should be generalized
- becomes problematic in recursive calls: it blocks
- if a thread \( t \) waits for a data structure to be filled:
  - \( t \) will call e.g. `PopRight` and obtain \(-1\)
  - \( t \) then has to call again, until an element is available
  - \( \triangleright \) \( t \) is busy waiting and produces contention on the lock

**Monitor:** a mechanism to address these problems:

- a procedure associated with a monitor acquires a lock on entry and releases it on exit
- if that lock is already taken, proceed if it is taken by the current thread
  
  \( \sim \) need a way to release the lock after the return of the last recursive call
Implementation of a Basic Monitor

A monitor contains a mutex and the thread currently occupying it:

```c
typedef struct monitor mon_t;
struct monitor { int tid; int count; };
void monitor_init(mon_t * m) { memset(m, 0, sizeof(mon_t)); }
```

Define `monitor_enter` and `monitor_leave`:

- ensure mutual exclusion of accesses to `mon_t`
- track how many times we called a monitored procedure recursively

```c
void monitor_enter(mon_t *m) { atomic {
    bool mine = false;
    while (!mine) {
        atomic {
            m->count--;
            if (m->count==0) {
                m->tid=0;
                mine = thread_id()==m->tid; // wake up threads
            }
            if (mine) m->count++; else m->tid=0;
        }
        m->tid = thread_id();
    }
}
if (!mine) de_schedule(km->tid);
```

Rewriting the Queue using Monitors

Instead of the mutex, we can now use monitors to protect the queue:

```c
double-ended queue: monitor version
void PushLeft(DQueue* q, int val) {
    monitor_enter(q->m);
    ... 
    monitor_leave(q->m);
}
void ForAll(DQueue* q, void* data, void (*callback)(void*, int)){
    monitor_enter(q->m);
    for (QNode* qn = q->left->right; qn!=q->right; qn=qn->right) {
        (*callback)(data, qn->val);
    }
    monitor_leave(q->m);
}
```

Recursive calls possible: