**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:

- 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
- an invariant may span several resources
- during an update, an invariant may be broken
- 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
- an invariant may span several resources
- during an update, an invariant may be broken
  - 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 would 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 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
- techniques are the source of common concurrency problems

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

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.

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.

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 Atomic Executions**

**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; 
```
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

\[ \text{int tmp} = i ; \\
    i = j ; \\
    j = \text{tmp} ; \]

Answer:
- 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? (j and tmp are registers)

Program 1

\[ i++ ; \]

Program 2

\[ j = i ; \\
    i = i + k ; \]

Program 3

\[ \text{int tmp} = i ; \\
    i = j ; \\
    j = \text{tmp} ; \]

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. declared as volatile)
- Idea: lock the cache/bus for an address for the duration of an instruction on x86:
  - Program 1 can be implemented using a lock inc [addr] instruction
  - Program 2 can be implemented using mov eax, k;
    lock xadd [addr], eax; mov reg, eax
  - Program 3 can be implemented using lock xchg [addr], reg

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

\[ \text{int tmp} = i ; \\
    i = j ; \\
    j = \text{tmp} ; \]

Garbage collectors often use a bumper pointer to allocated memory:

Bumper Pointer Allocation

\[ \text{char} \_\text{heap}[2\to 20] ; \]
\[ \text{char} \* \text{firstFree} = \_\text{heap}[0] ; \]

\[
\text{char} \* \text{alloc} (\text{int size}) \{ \\
    \text{char} \* \text{start} = \text{firstFree} ; \\
    \text{firstFree} = \text{firstFree} + \text{size} ; \\
    \text{if} (\text{start} + \text{size} > \text{sizeof} (\_\text{heap}) ) \text{garbage}\_\text{collect} () ; \\
    \text{return} \ \text{start} ; \\
\}
\]

- 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 xadd [firstFree], eax` instruction
- requires inline assembler

**Marking Statements as Atomic**

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

Program 1
```c
atomic {
    i++;
    j = i;
    i = i+k;
}
```

Program 2
```c
atomic {
    int tmp = i;
    i = j;
    j = tmp;
}
```

Program 3
```c
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
- often, there are instructions 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 ∈ {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
- 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 \textit{wait-free} implementation is not possible, a \textit{lock-free} implementation might still be viable.
Common usage pattern for \textit{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

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

\textit{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
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

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**.

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

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
- ensures **mutual exclusion**, no two threads execute at once
- blocks other threads as soon as one thread executes the critical section
- can be **acquired** and **released**

⚠️ may **deadlock** the program
Semaphores and Mutexes

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);
}
```

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

A counting semaphore can track how many resources are still available.
- a thread acquiring a resource executes `wait()`
- if a resource is still available, `wait()` returns
- once a thread finishes using a resource, it calls `signal()` to `release`

A counting semaphore can track how many resources are still available.
- a thread acquiring a resource executes `wait()`
- if a resource is still available, `wait()` returns
- once a thread finishes using a resource, it calls `signal()` to `release`

Special case: initializing with \( s = 1 \) gives a binary semaphore:
- can be used to block and unblock a thread
- can be used to protect a single resource
- in this case the data structure is also called mutex
Implementation of Semaphores

A *semaphore* does not have to wait busily:

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

Busy waiting is avoided:
- a thread failing to decrease executes *de_schedule()*
- *de_schedule()* enters the operating system and inserts the current thread into a queue of threads that will be woken up when *s* becomes non-zero, usually by *monitoring writes* to *s*
- once a thread calls *signal()* the first thread waiting on *s* is extracted
- the operating system lets the second thread call *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(&s);
        } while (!avail);
    }
```

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

In general, the implementation is more complicated
- *wait()* may busy wait for a few iterations
  - avoids de-scheduling if the lock is released frequently
  - better throughput for semaphores that are held for a short time
- *signal()* might have to inform the OS that *s* has been written

Mutexes

One common use of semaphores is to guarantee mutual exclusion.
- in this case, a binary semaphore is also called a *mutex*
- e.g. add a lock to the double-ended queue data structure

⚠️ decide what needs protection and what not
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:
1. is a re-occurring pattern; should be generalized
2. becomes problematic if recursive calls it blocks
3. if a thread \( t \) waits for a data structure to be filled:
   - \( t \) will call e.g. \( \text{pop()} \) and obtain \(-1\)
   - \( t \) then has to call again, until an element is available
   - \( t \) is busy waiting and produces contention on the lock

Monitor: a mechanism to address these problems:
1. a procedure associated with a monitor acquires a lock on entry and releases it on exit
2. 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:
1. is a re-occurring pattern; should be generalized
2. becomes problematic if recursive calls it blocks
3. if a thread \( t \) waits for a data structure to be filled:
   - \( t \) will call e.g. \( \text{pop()} \) and obtain \(-1\)
   - \( t \) then has to call again, until an element is available
   - \( t \) is busy waiting and produces contention on the lock

Monitor: a mechanism to address these problems:
1. a procedure associated with a monitor acquires a lock on entry and releases it on exit
2. if that lock is already taken, proceed if it is taken by the current thread...
   - 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 `count` and the id of the thread `tid` 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 {
                mine = thread_id()==m->tid;
                if (mine) m->count++;
                // wake up threads
                if (mine) m->tid=0;
            }
            m->tid = thread_id();
        }
    }
    if (!mine) de_schedule(&m->tid);}
```

Condition Variables

✓ Monitors simplify the construction of thread-safe resources.

Still: Efficiency problem when using resource to synchronize:

- if a thread `t` waits for a data structure to be filled:
  - `t` will call e.g. `pop()` and obtain `-1`
  - `t` then has to call again, until an element is available
  - `t` is busy waiting and produces contention on the lock

Idea: create a `condition variable` on which to block while waiting:

```c
struct monitor { int tid; int count; int cond; };
```

Condition Variables

✓ Monitors simplify the construction of thread-safe resources.

Still: Efficiency problem when using resource to synchronize:

- if a thread `t` waits for a data structure to be filled:
  - `t` will call e.g. `pop()` and obtain `-1`
  - `t` then has to call again, until an element is available
  - `t` is busy waiting and produces contention on the lock

Idea: create a `condition variable` on which to block while waiting:

```c
struct monitor { int tid; int count; int cond; };
```

Define these two functions:

1. `wait` for the condition to become true
   - called while being `inside` the monitor
   - temporarily releases the monitor and blocks
   - when `signalled`, re-acquires the monitor and returns

2. `signal` waiting threads that they may be able to proceed
   - one/all waiting threads that called `wait` will be woken up, two possibilities:
     - `signal-and-urgent-wait`: the signalling thread suspends and continues once the `signalled` thread has released the monitor
     - `signal-and-continue`: the signalling thread continues, any `signalled` thread enters when the monitor becomes available