Marking Statements as Atomic

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

### Program 1
```plaintext
atomic {
    i++; 
}
```

### Program 2
```plaintext
atomic {
    j = i;
    i = i+k;
    j = tmp;
}
```

### Program 3
```plaintext
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 exists
- 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

Operations \textit{update} a memory cell and \textit{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 \textit{modify-and-test}
- the third case generalizes this to arbitrary values
  - this operation is called \textit{compare-and-swap}
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 $i$

---

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
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

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
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
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

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
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);
}
```

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

Semaphores and Mutexes

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

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

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

- if a resource is still available, `wait()` returns
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);
}
```

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 {
            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 \( t \) waiting on \( &s \) is extracted.
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
- 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(&s);
    } while (!avail);
}
```

In general, the implementation is more complicated
- `wait()` may busy wait for a few iterations
  - saves de-scheduling if the lock is released frequently

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

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

In general, the implementation is more complicated
- `wait()` may busy wait for a few iterations
  - saves 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
  - using a semaphore with a single thread reduces to `if (s--) s++;`
- using semaphores in sequential code has no or little penalty
- program with concurrency in mind?
Making a Queue Thread-Safe

Consider a double ended queue:

```
    double-ended queue
    10   20   90
   left sentinel   right sentinel
```

double-ended queue: adding an element

```c
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
- Add a lock to the double-ended queue data structure

Implementing the Removal

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

double-ended queue: removal

```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;
}
```
Implementing the Removal

By using the same lock *q*→*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* newNode = oldRightNode->left;
    newNode->right = rightSentinel;
    rightSentinel->left = newNode;
    signal(q->s); // signal that we're done
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```

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

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

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:
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
  - $\triangledown$ $t$ is busy waiting and produces contention on the lock

Monitor: a mechanism to address these problems:
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
  - △ 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

Implementation of a Basic Monitor

A monitor contains a mutex s 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 the monitored procedure recursively

```c
void monitor_enter(mon_t* mon) { 
  atomic {
    bool mine = false; 
    while (!mine) { 
      atomic {
        mine = thread_id() == mon->tid; 
        if (mine) mon->count++; else 
          if (mon->tid==0) { 
            mine = true; mon->count=1; 
          }
        mon->tid = thread_id(); 
      }
    } 
    if (!mine) de_schedule(mon->tid); 
  }
}
```

```c
void monitor_leave(mon_t* m) {
  atomic {
    if (m->count==0) { 
      m->count--; 
    } else 
      m->tid=0; 
  }
}
```
Rewriting the Queue using Monitors

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

double-ended queue: monitor version

```c
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:

Implementation of a Basic Monitor

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

```c
typedef struct monitor mon_t;
struct monitor { int tid; int count; };
void monitor_init(m 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(m mon_t *m) {
    bool mine = false;
    atomic {
        m->count--;
        while (!mine) {
            atomic {
                mine = thread_id()===m->tid;
                if (mine) m->count++;
                else m->tid=0;
            } // wake up threads
            if (m->tid==0) {
                m->count=1;
            }
            m->tid = thread_id();
        }
    }
    if (!mine) de_schedule(&m->tid);
}
```

Rewriting the Queue using Monitors

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

double-ended queue: monitor version

```c
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:
- the function passed to `ForAll` can invoke `PushLeft`
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->right; qn=qn->right)
        (*callback)(data, qn->val);
    monitor_leave(q>m);
}
```

Recursive calls possible:
- the function passed to `ForAll` can invoke `PushLeft`
- example: `ForAll(q,g,&PushLeft)` duplicates entries

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. `PopRight` and obtain `-1`
  - `t` then has to call again, until an element is available
  - `⚠️` `t` is busy waiting and produces contention on the lock

```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. `PopRight` 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:

- **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

- **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

Signal-And-Urgent-Wait Semantics

 Requires one queues for each condition $c$ and a suspended queue $s$:

- a thread who tries to enter a monitor is added to queue $c$ if the monitor is occupied
- a call to `wait` on condition $a$ adds thread to the queue $a.q$
Signal-And-Urgent-Wait Semantics

Requires one queues for each condition $c$ and a suspended queue $s$:

- A thread who tries to enter a monitor is added to queue $e$ if the monitor is occupied
- A call to `wait` on condition $a$ adds thread to the queue $a.q$
- A call to `signal` for $a$ adds thread to queue $s$ (suspended)
- One thread form the $a$ queue is woken up
- `signal` on $a$ is a no-op if $a.q$ is empty
- If a thread leaves, it wakes up one thread waiting on $s$

![Diagram](http://en.wikipedia.org/wiki/Monitor_(synchronization))
Signal-And-Urgent-Wait Semantics

Requires one queues for each condition $e$ and a suspended queue $s$:

- a thread who tries to enter a monitor is added to queue $e$ if the monitor is occupied
- a call to `wait` on condition $a$ adds thread to the queue $a.q$
- a call to `signal` for $a$ adds thread to queue $s$ (suspended)
- one thread form the $a$ queue is woken up
- `signal` on $a$ is a no-op if $a.q$ is empty
- if a thread leaves, it wakes up one thread waiting on $s$
- if $s$ is empty, it wakes up one thread from $e$ on queue $s$ has priority over $e$

Signal-And-Continue Semantics

Here, the `signal` function is usually called `notify`.

- a call to `wait` on condition $a$ adds thread to the queue $a.q$
- a call to `notify` for $a$ adds one thread from $a.q$ to $e$ (unless $a.q$ is empty)

Signal-And-Continue Semantics

Here, the `signal` function is usually called `notify`.

- a call to `wait` on condition $a$ adds thread to the queue $a.q$
- a call to `notify` for $a$ adds one thread from $a.q$ to $e$ (unless $a.q$ is empty)
- if a thread leaves, it wakes up one thread waiting on $e$
Signal-And-Continue Semantics

Here, the `signal` function is usually called `notify`.

- A call to `wait` on condition `a` adds thread to the queue `a.q`
- A call to `notify` for `a` adds one thread from `a.q` to `e` (unless `a.q` is empty)
- If a thread leaves, it wakes up one thread waiting on `e`

→ Signalled threads compete for the monitor

Signal-And-Continue Semantics

Here, the `signal` function is usually called `notify`.

- A call to `wait` on condition `a` adds thread to the queue `a.q`
- A call to `notify` for `a` adds one thread from `a.q` to `e` (unless `a.q` is empty)
- If a thread leaves, it wakes up one thread waiting on `e`

→ Signalled threads compete for the monitor

- Assuming FIFO ordering on `e`, threads who tried to enter between `wait` and `notify` will run first

Implementing Condition Variables

We implement the simpler `signal-and-continue` semantics:

- A `notified` thread is simply woken up and competes for the monitor

```c
void cond_wait(mon_t *m) {
    assert(m->tid==thread_id());
    int old_count = m->count;
    m->tid = 0;
    de_schedule(&m->cond);
    bool next_to_enter;
    do {
        atomic {
            next_to_enter = m->tid==0;
            if (next_to_enter) {
                m->tid = thread_id();
                m->count = old_count;
            }
        }
        if (!next_to_enter) de_schedule(&m->tid);
    } while (!next_to_enter);
}
```

- `cond_notify`:

```c
void cond_notify(mon_t *m) {
    m->cond = 1;
    m->tid = thread_id();
}
```
A Note on Notify

With *signal-and-continue* semantics, two notify functions exist:

- **notify**: wakes up exactly one thread waiting on condition variable
- **notifyAll**: wakes up all threads waiting on a condition variable

⚠️ an implementation often becomes easier if notify means *notify some*

→ programmer should assume that thread is not the only one woken up

What about the priority of notified threads?

- a notified thread is likely to block immediately on &m->tid
A Note on Notify

With signal-and-continue semantics, two notify functions exist:

- notify: wakes up exactly one thread waiting on a condition variable
- notifyAll: wakes up all threads waiting on a condition variable

Implementing PopRight with Monitors

We use the monitor q->m and the condition variable q->c. PopRight:

```
double-ended queue: removal

int PopRight(DQueue* q, int val) {
    QNode* oldRightNode;
    monitor_enter(q->m); // wait to enter the critical section
    QNode* rightSentinel = q->right;
    oldRightNode = rightSentinel->left;
    if (oldRightNode==leftSentinel) { cond_wait(q->c); goto L; }
    QNode* newNode = oldRightNode->left;
    newNode->right = rightSentinel;
    rightSentinel->left = newNode;
    monitor_leave(q->m); // signal that we’re done
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```

- if the queue is empty, wait on q->c
- use a loop, in case the thread is woken up spuriously

Monitor versus Semaphores

A monitor can be implemented using semaphores:

- protect each queue with a mutex

an implementation often becomes easier if notify means notify some

What about the priority of notified threads?

- a notified thread is likely to block immediately on &m->tid
- notified threads compete for the monitor with other threads
- if OS implements FIFO order: notified threads will run after threads that
  tried to enter since wait was called
- giving priority to waiting threads requires better interface to OS
Monitor versus Semaphores

A monitor can be implemented using semaphores:
- protect each queue with a mutex
- use a semaphore to block threads that are waiting

A semaphore can be implemented using a monitor:
- protect the semaphore variable $s$ with a monitor
- implement `wait` by calling `cond.wait` if $s = 0$

A note on the history of monitors:
- condition variables were meant to be associated with a predicate $p$
- signalling a variables would only wake up a thread if $p$ is true
Monitor versus Semaphores

A monitor can be implemented using semaphores:
- protect each queue with a mutex
- use a semaphore to block threads that are waiting

A semaphore can be implemented using a monitor:
- protect the semaphore variable \( s \) with a monitor
- implement `wait` by calling `cond.wait` if \( s = 0 \)

A note on the history of monitors:
- condition variables were meant to be associated with a predicate \( p \)
- signalling a variables would only wake up a thread if \( p \) is true
- difficult implement general conditions
  - OS would have to run code to determine if \( p \) holds
  - OS would have to ensure atomicity
  - problematic if \( p \) is implemented by arbitrary code
  - wake up thread and have it check the predicate itself
- create condition variable for each set of threads with the same \( p \)

Monitor versus Semaphores

A monitor can be implemented using semaphores:
- protect each queue with a mutex
- use a semaphore to block threads that are waiting

A semaphore can be implemented using a monitor:
- protect the semaphore variable \( s \) with a monitor
- implement `wait` by calling `cond.wait` if \( s = 0 \)

A note on the history of monitors:
- condition variables were meant to be associated with a predicate \( p \)
- signalling a variables would only wake up a thread if \( p \) is true
- difficult implement general conditions
  - OS would have to run code to determine if \( p \) holds
  - OS would have to ensure atomicity
  - problematic if \( p \) is implemented by arbitrary code
  - wake up thread and have it check the predicate itself
- create condition variable for each set of threads with the same \( p \)
- notify variable if the predicate may have changed

Monitors with a Single Condition Variable

Monitors with a single condition variable are built into Java and C#:

```java
class C {
    public synchronized void f() {
        // body of f
    }
}
```

is equivalent to

```java
class C {
    public void f() {
        monitor_enter();
        // body of f
        monitor_leave();
    }
}
```

with Object containing:

```java
private int mon_var;
private int mon_count;
private int cond_var;
protected void monitor_enter();
promtected void monitor_leave();
```
Deadlocks with Monitors

**Definition (Deadlock)**
A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

Consider this Java class:
```java
class Foo {
    public Foo other = null;
    public synchronized void bar() {
        ... if (*) other.bar(); ...
    }
}
```

and two instances:
```java
Foo a = new Foo();
Foo b = new Foo();
a.other = b; b.other = a;
// in parallel:
a.bar() || b.bar();
```

Sequence leading to a deadlock:
```java
threads A and B execute a.bar() and b.bar()
```

Deadlocks with Monitors

**Definition (Deadlock)**
A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

Consider this Java class:
```java
class Foo {
    public Foo other = null;
    public synchronized void bar() {
        ... if (*) other.bar(); ...
    }
}
```

and two instances:
```java
Foo a = new Foo();
Foo b = new Foo();
a.other = b; b.other = a;
// in parallel:
a.bar() || b.bar();
```
**Definition (Deadlock)**

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

Consider this Java class:

```java
class Foo {
    public Foo other = null;
    public synchronized void bar() {
        ... if (*) other.bar(); ... 
    }
}
```

and two instances:

```java
Foo a = new Foo();
Foo b = new Foo();
a.other = b; b.other = a;
// in parallel:
a.bar() || b.bar();
```

Sequence leading to a deadlock:

- threads \( A \) and \( B \) execute \( a.bar() \) and \( b.bar() \)
- \( a.bar() \) acquires the monitor of \( a \)
- \( b.bar() \) acquires the monitor of \( b \)
- \( A \) happens to execute \( other.bar() \)
- \( A \) blocks on the monitor of \( b \)
- \( B \) happens to execute \( other.bar() \)
- \( \rightarrow \) both block indefinitely