Definition: Sequential Consistency

Definition (Sequential Consistency Condition for Multi-Processors):
The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Uh?

Definition (Sequential Consistency Condition for Multi-Processors):
The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Uh? The result of an n-threaded program does not change
**Definition: Sequential Consistency**

**Definition (Sequential Consistency Condition for Multi-Processors)**

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Uh? The result of an n-threaded program does not change
- all operations \( \forall p_0^1, p_1^1, \ldots \) and \( \forall p_0^2, p_1^2, \ldots \) and \( \ldots \)\( \forall p_0^k, p_1^k, \ldots \)
- are executed in a total order \( \exists C. C[p_j^1] < C[p_k^1] \) for all \( i, j, k, l \)
- where \( j = l \) implies \( i < k \)

Idea for showing that a system is not sequentially consistent:
**Definition: Sequential Consistency**

**Definition (Sequential Consistency Condition for Multi-Processors)**

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

**Uh?** The result of an $n$-threaded program does not change

- all operations $p_0, p_1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$
- are executed in a total order $\exists C: C(p_j^i) < C(p_j^k)$ for all $i, j, k, l$
- where $j = l$ implies $i < k$

**Idea for showing that a system is not sequentially consistent:**
- assume program executes correctly under sequential consistency
- pick an execution $\bullet$ and a total ordering of all operations $\bullet$

**Memory Consistency**

**Sequential Consistency**

12 / 36
Definition: Sequential Consistency

Definition (Sequential Consistency Condition for Multi-Processors)
The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Uh? The result of an n-threaded program does not change
- all operations \( \forall p_i, p_j, \ldots, p_k, p_l, \ldots \)
- are executed in a total order \( \exists C : C(p_i) < C(p_j) \) for all \( i, j, k, l \)
- where \( j = l \) implies \( i < k \)

Idea for showing that a system is not sequentially consistent:
- assuming program executes correctly under sequential consistency
- pick an execution \( \bullet \) and a total ordering of all operations \( \bullet \)
- add extra processes for a more realistic model
- the original order \( \bullet \) becomes a partial order \( \rightarrow \)
- show that total orderings \( C' \) exist for \( \rightarrow \) for which the result differ

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

Limitations of Wait- and Lock-Free Algorithms

Wait-/Lock-Free algorithms are severely limited in terms of their computation:
- restricted to the semantics of a single atomic operations
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand
Limitations of Wait- and Lock-Free Algorithms

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

- restricted to the semantics of a single atomic operations
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand
  - only very simple algorithms can be implemented, for instance
    - binary semaphores: a flag that can be acquired (set) if free (unset) and released
    - 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 operations
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand
  - only very simple algorithms can be implemented, for instance
    - binary semaphores: a flag that can be acquired (set) if free (unset) and released
    - counting semaphores: an integer that can be decreased if non-zero and increased
    - mutex: ensures mutual exclusion using a binary semaphore
    - monitor: ensures mutual exclusion using a binary semaphore, allows other threads to block until the next release of the resource
Limitations of Wait- and Lock-Free Algorithms

Wait-/Lock-Free algorithms are severely limited in terms of their computation:
- restricted to the semantics of a single atomic operations
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand
- only very simple algorithms can be implemented, for instance
  - binary semaphores: a flag that can be acquired (set) if free (unset) and released
  - counting semaphores: an integer that can be decreased if non-zero and increased
    - mutex: ensures mutual exclusion using a binary semaphore
    - 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.

Semaphores and Mutexes

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

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

void signal() {
    atomic {
        s = s + 1;
        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()
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
- can be used to protect a single resource

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
- can be used to protect a single resource

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
- can be used to protect a single resource

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
- can be used to protect a single resource
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:

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
- 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 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:

```c
void signal() {
    atomic {
        s = s + 1;
    }
}
```
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 waiting on &s is extracted

Implementation of Semaphores

```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 waiting on &s is extracted

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
Practical Implementation of Semaphores

Certain optimisations are possible:

```c
void wait() {
    bool avail;
    do { atomic {
        avail = s>0;
        atomic { s = s + 1; }
            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

Practical Implementation of Semaphores

Certain optimisations are possible:

```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

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--; 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:

```
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
- decide what needs protection and what not

```
void PushLeft(DQueue* q, int val) {
    QNode *qn = 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;
    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 \( q \rightarrow s \), we can write a thread-safe `PopRight`:

```c
int PopRight(DQueue* q, int val) {
    QNode* oldRightNode;
    wait(q->s); // wait to enter the critical section
    QNode* rightSentinel = q->right;
    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;
}
```

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:

- 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
- becomes problematic in recursive calls: it blocks
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 in recursive calls: it blocks
3. 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
   - \( \Delta \) \( t \) 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:

1. is a re-occurring pattern, should be generalized
2. becomes problematic in recursive calls: it blocks
3. 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
   - \( \Delta t \) 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

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 in recursive calls: it blocks
3. 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
   - \( \Delta t \) 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

Implementation of a Basic Monitor

A monitor contains a mutex \( m \) and the thread currently occupying it:

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

Define monitor enter and leave:

void monitor_enter(monitor_t* m) {
    atomic {
        bool mine = false;
        while (!mine) {
            atomic {
                mine = thread_id()==m->tid;
                if (mine) m->count++; else
                if (m->count==0) {
                    m->tid = thread_id();
                    mine = true;
                    m->count = 1;
                }
            } // atomic
            if (mine) de_schedule(&m->tid);
        }
    } // atomic
}

void monitor leave(monitor_t* m) {
    atomic {
        if (m->count==0) {
            m->count--;
            if (m->count==0) {
                m->tid=0;
                // wake up threads
            }
        }
    } // atomic
}
```
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`
- example: `ForAll(q,q,&PushLeft)` duplicates entries

Condition Variables

✓ Monitors simplify the construction of thread-safe resources.

Still: Efficiency problem when using resource to synchronize:

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

- Monitors simplify the construction of thread-safe resources.
- Still: Efficiency problem when using resource to synchronize:
  - if a thread \( i \) waits for a data structure to be filled:
    - \( i \) will call e.g. \texttt{PopRight} and obtain \(-1\)
    - \( i \) then has to call again, until an element is available
    - \( \Delta \) \( i \) is busy waiting and produces contention on the lock
- Idea: create a \textit{condition variable} on which to block while waiting:
  ```c
  struct monitor { int s; int tid; int count; int cond; }; 
  ```

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 thread who tries to enter a monitor is added to queue \( e \) if the monitor is occupied
- a call to \texttt{wait} on condition \( a \) adds thread to the queue \( a.q \)

Source: [https://en.wikipedia.org/wiki/Monitor_%28computer_science%29](https://en.wikipedia.org/wiki/Monitor_%28computer_science%29)
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 $a,q$
- a call to `signal` for $a$ adds thread to queue $s$ (suspended)
- if a thread leaves, it wakes up one thread waiting on $s$

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 e adds thread to the queue a.q
- a call to signal for e adds thread to queue s (suspended)
- one thread form the e 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 e
- if s is empty, it wakes up one thread from e

\[ \text{\sim} \text{ queue } s \text{ has priority over } e \]

Signal-And-Continue Semantics
Here, the signal function is usually called notify.
- a call to wait on condition e adds thread to the queue a.q
- a call to notify for e 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
- \( \text{\sim} \) 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 \( a \cdot q \)
- A call to notify for a adds one thread from \( a \cdot q \) to \( e \) (unless \( a \cdot q \) 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->count = 0; m->tid = 0;
    de_schedule(&m->cond);
    do {
        atomic {
            next_to_enter = m->tid=0;
            if (next_to_enter) {
                m->tid = thread_id();
                m->count = old_count;
            }
        } while (!next_to_enter);
    } while (!next_to_enter);
}
```

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
A Note on Notify

With \textit{signal-and-continue} semantics, two notify functions exist:

1. \texttt{notify}: wakes up exactly one thread waiting on condition variable
2. \texttt{notifyAll}: wakes up all threads waiting on a condition variable

\textbf{⚠️} an implementation often becomes easier if \texttt{notify} means \texttt{notify some}.

\texttt{~\~}\textit{programmer should assume that thread is not the only one woken up.}

What about the priority of notified threads?

\begin{itemize}
  \item a notified thread is likely to block immediately on \texttt{&m->tid}
  \item \texttt{~\~} notified threads compete for the monitor with other threads
  \item if OS implements FIFO order: notified threads will run \texttt{after} threads that tried to enter since \texttt{wait} was called
\end{itemize}
Implementing PopRight with Monitors

We use the monitor $q \rightarrow m$ and the condition variable $q \rightarrow c$. PopRight:

```c
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* newRightNode = oldRightNode->left;
    newRightNode->right = rightSentinel;
    rightSentinel->left = newRightNode;
    monitor_leave(q->m); // signal that we're done
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```

Monitor versus Semaphores

A monitor can be implemented using semaphores:

- protect each queue with a mutex

if the queue is empty, wait on $q \rightarrow c$
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
- use a semaphore to block threads that are waiting
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 \texttt{wait} by calling \texttt{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
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

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
    - `wait` up thread and have it check the predicate itself
  - create condition variable for each set of threads with the same $p$
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:

- private int `mon_var`;
- private int `mon_count`;
- private int `cond_var`;
- protected void `monitor_enter()`;
- protected 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:

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

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

**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()

**Treatment of Deadlocks**
Deadlocks occur if the following four conditions hold [1]:
1. **mutual exclusion**: processes require exclusive access
2. **wait for**: a process holds resources while waiting for more
3. **no preemption**: resources cannot be taken away from processes
4. **circular wait**: waiting processes form a cycle

The occurrence of deadlocks can be:
1. **ignored**: for the lack of better approaches, can be reasonable if deadlocks are rare
2. **detection**: check within OS for a cycle, requires ability to preempt
Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [1]:

- mutual exclusion: processes require exclusive access
- wait for: a process holds resources while waiting for more
- no preemption: resources cannot be taken away from processes
- circular wait: waiting processes form a cycle

The occurrence of deadlocks can be:

- ignored: for the lack of better approaches, can be reasonable if deadlocks are rare
- detection: check within OS for a cycle, requires ability to preempt
- prevention: design programs to be deadlock-free

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be partially ordered.

Definition (lock sets)

Let \( L \) denote the set of locks. We call \( L(p) \subseteq L \) the lock set at \( p \), that is, the set of locks that may be in the "acquired" state at program point \( p \).
Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be *partially ordered.*

Definition (lock sets)

Let $L$ denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at $p$, that is, the set of locks that may be in the “acquired” state at program point $p$.

We require the transitive closure $\sigma^+$ of a relation $\sigma$:

Definition (transitive closure)

Let $\sigma \subseteq X \times X$ be a relation. Its transitive closure is $\sigma^+ = \bigcup_{i \in \mathbb{N}} \sigma^i$ where

\[
\begin{align*}
\sigma^0 &= \sigma \\
\sigma^{i+1} &= \sigma^i \cup \{ (x_1, x_3) | \exists x_2 \in X, (x_1, x_2) \in \sigma^i \land (x_2, x_3) \in \sigma^i \}
\end{align*}
\]