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) {
    bool mine = false;
    atomic {
        while (!mine) {
            atomic {
                if (m->count==0) {
                    m->count=1;
                    // wake up threads
                    if (mine) m->count++; else
                    if (m->tid==0) {
                        m->tid = thread_id();
                    }
                }
                mine = true; m->count=1;
            }
            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

```c
struct monitor { int tid; int count; int cond; int cond2;... };
```
**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; int cond2; ... ; }
  ```

- 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 $a.q$
  - 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 $a$
  - if $s$ is empty, it wakes up one thread from $c$

**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 $c$ (unless $a.q$ is empty)
  - if a thread leaves, it wakes up one thread waiting on $c$

- 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 $c$ (unless $a.q$ is empty)
- If a thread leaves, it wakes up one thread waiting on $c$
- Signalled threads compete for the monitor
  - Assuming FIFO ordering on $c$, threads who tried to enter between `wait` and `notify` will run first
  - Need additional queue $s$ if waiting threads should have priority
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;
    wait(m->cond);
    bool next_to_enter;
    do {
        Atomic {            // wake up other threads
            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);
}
```

```c
void cond_notify(mon_t *m) {
}
```

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`
- 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 more complex implementation (queue data structure for signaled threads)
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() {
        enter();
        // body of f
        leave();
    }
}
```

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

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.)

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()`
- → both block indefinitely
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()
- ∞ both block indefinitely

How can this situation be avoided?

Treatment of Deadlocks

Observation: Deadlocks occur if the following four conditions hold [Coffman et al. (1971) Coffman, Elphick, and Shoshan]:

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
3. **prevention**: design programs to be deadlock-free
4. **avoidance**: use additional information about a program that allows the OS to schedule threads so that they do not deadlock

The treatment of deadlocks can be:

1. **avoidance**: use additional information about a program that allows the OS to schedule threads so that they do not deadlock
2. **prevention**: design programs to be deadlock-free
3. **detection**: check within OS for a cycle, requires ability to preempt
4. **preemption**: design programs to be deadlock-free
5. **ignorance**: for the lack of better approaches, can be reasonable if deadlocks are rare

---

Observation: Deadlocks occur if the following four conditions hold [Coffman et al. (1971) Coffman, Elphick, and Shoshan]:

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. **avoidance**: use additional information about a program that allows the OS to schedule threads so that they do not deadlock
2. **prevention**: design programs to be deadlock-free
3. **detection**: check within OS for a cycle, requires ability to preempt
4. **ignorance**: for the lack of better approaches, can be reasonable if deadlocks are rare

---

Observation: Deadlocks occur if the following four conditions hold [Coffman et al. (1971) Coffman, Elphick, and Shoshan]:

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. **avoidance**: use additional information about a program that allows the OS to schedule threads so that they do not deadlock
2. **prevention**: design programs to be deadlock-free
3. **detection**: check within OS for a cycle, requires ability to preempt
4. **ignorance**: for the lack of better approaches, can be reasonable if deadlocks are rare

---

Observation: Deadlocks occur if the following four conditions hold [Coffman et al. (1971) Coffman, Elphick, and Shoshan]:

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. **avoidance**: use additional information about a program that allows the OS to schedule threads so that they do not deadlock
2. **prevention**: design programs to be deadlock-free
3. **detection**: check within OS for a cycle, requires ability to preempt
4. **ignorance**: for the lack of better approaches, can be reasonable if deadlocks are rare
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 \( p \).

---

Treatment of Deadlocks

**Observation:** Deadlocks occur if the following four conditions hold (Coffman et al. 1971) [Coffman, Elphick, and Shoshant]:

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
3. **prevention:** design programs to be deadlock-free
4. **avoidance:** use additional information about a program that allows the OS to schedule threads so that they do not deadlock

---

** prevention** is the only safe approach on standard operating systems

- can be achieved using lock-free algorithms
- but what about algorithms that require locking?

---

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 \( p \).

---

**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

\[
\sigma^0 = \sigma
\]
\[
\sigma^{i+1} = \{ (x_1,x_3) \mid \exists x_2 \in X, (x_1,x_2) \in \sigma^i \land (x_2,x_3) \in \sigma \}
\]
**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

\[
\sigma^0 = \sigma \\
\sigma^{i+1} = \{ (x_1,x_3) \mid \exists x_2 \in X : (x_1,x_2) \in \sigma^i \land (x_2,x_3) \in \sigma \}
\]

Each time a lock is acquired, we track the lock set at $p$:

**Definition (lock order)**

Define $\prec \subseteq L \times L$ such that $l \prec l'$ if $l' \in \lambda(p)$ and the statement at $p$ is of the form `wait(l')` or `monitor.enter(l')`. Define the strict lock order $\prec_\neq = \prec^+$.

---

**Freedom of Deadlock**

The following holds for a program with mutexes and monitors:

**Theorem (freedom of deadlock)**

If there exists no $a \in L$ with $a \prec a$ then the program is free of deadlocks.

Suppose a program blocks on semaphores (mutexes) $L_S$ and on monitors $L_M$ such that $L = L_S \cup L_M$.

**Theorem (freedom of deadlock for monitors)**

If $\forall a \in L_M, a \neq a$ and $\forall a \in L_M, b \in L : a \prec b \land b \prec a \Rightarrow a = b$ then the program is free of deadlocks.

**Note:** the set $L$ contains *instances* of a lock.

- the set of lock instances can vary at runtime
- if we statically want to ensure that deadlocks cannot occur:
  - summarize every lock/monitor that may have several instances into one
  - a summary lock/monitor $\tilde{a} \in L_M$ represents several concrete ones
  - thus, if $\tilde{a} \prec \tilde{a}$ then this might not be a self-cycle
  - require that $\tilde{a} \neq \tilde{a}$ for all summarized monitors $\tilde{a} \in L_M$
Avoiding Deadlocks in Practice

How can we verify that a program contains no deadlocks?
1. identify mutex locks $L_S$ and summarized monitor locks $L_M^* \subseteq L_M$
2. identify non-summary monitor locks $L_M^* = L_M \setminus L_M^*$
3. sort locks into ascending order according to lock sets
4. check that no cycles exist except for self-cycles of non-summary monitors

Avoiding Deadlocks in Practice

How can we verify that a program contains no deadlocks?
1. identify mutex locks $L_S$ and summarized monitor locks $L_M^* \subseteq L_M$
2. identify non-summary monitor locks $L_M^* = L_M \setminus L_M^*$
3. sort locks into ascending order according to lock sets
4. check that no cycles exist except for self-cycles of non-summary monitors

What to do when the lock order contains a cycle?
- determining which locks may be acquired at each program point is undecidable -- lock sets are an approximation
- an array of locks in $L_S$: lock in increasing array index sequence
- if $l \in \lambda(P)$ exists $l' < l$ is to be acquired -- change program: release $l$, acquire $l'$, then acquire $l$ again -- inefficient
- if a lock set contains a summarized lock $\bar{a}$ and $\bar{a}$ is to be acquired, we're stuck

Locks Roundup

an example for the latter is the $\text{Foo}$ class: two instances of the same class call each other
Atomic Execution and Locks

Consider replacing the specific locks with atomic annotations:

```java
stack: removal
void pop() {
    ...
    wait(q->t);
    ...
    if (*) { signal(q->t); return; }
    ...
    if (c) wait(q->s);
    ...
    if (c) signal(q->s);
    signal(q->t);
}
```

- nested atomic blocks still describe one atomic execution
- locks convey additional information over atomic
- locks cannot easily be recovered from atomic declarations

Outlook

Writing atomic annotations around sequences of statements is a convenient way of programming.

Idea of mutexes: Implement atomic sections with locks:
- a single lock could be used to protect all atomic blocks
- more concurrency is possible by using several locks
- some statements might modify variables that are never read by other threads → no lock required
- statements in one atomic block might access variables in a different order to another atomic block → deadlock possible with locks implementation
- creating too many locks can decrease the performance, especially when required to release locks in λ(t) when acquiring
Concurrency across Languages

In most systems programming languages (C, C++, ...) we have

- the ability to use *atomic* operations
- we can implement *wait-free* algorithms

In Java, C# and other higher-level languages

- provide monitors and possibly other concepts
- often simplify the programming but incur the same problems

<table>
<thead>
<tr>
<th>language</th>
<th>barriers</th>
<th>wait-free</th>
<th>semaphore</th>
<th>mutex</th>
<th>monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>(a)</td>
</tr>
<tr>
<td>Java, C#</td>
<td>-</td>
<td>(b)</td>
<td>(c)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(a) some pthread implementations allow a *reentrant* attribute
(b) newer API extensions (Java.util.concurrent.atomic.*, System.Threading.Interlocked resp.)
(c) simulate semaphores using an object with two *synchronized* methods

Summary

Classification of concurrency algorithms:

- *wait-free*, *lock-free*, *locked*  
  - next on the agenda: *transactional*

*Wait-free* algorithms:

- never block, always succeed, never deadlock, no starvation
- very limited in expressivity

*Lock-free* algorithms:

- never block, may fail, never deadlock, may starve
- invariant may only span a few bytes (8 on Intel)

*Locking* algorithms:

- can guard arbitrary code
- can use several locks to enable more fine grained concurrency
- may deadlock
- semaphores are not *re-entrant* monitors are
- use algorithm that is best fit
E. G. Coffman, M. Elphick, and A. Shoshani.
System deadlocks.
ISSN 0360-0300.

T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.