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

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 a monitored procedure recursively

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

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

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

---

Rewriting the Queue using Monitors
Instead of the mutex, we can now use monitors to protect the queue:

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

Recursive calls possible:
- the function passed to `ForAll` can invoke `PushLeft`
- example: `ForAll(q,q,&PushLeft)` duplicates entries
- using monitor instead of mutex ensures that recursive call does not block
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
    - $\triangle$ $t$ 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; }
```

---

**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

**Diagram:**

- a.q
- b.q
- wait a
- signalled
- signal
- wait b
- leave

SOURCE: [http://en.wikipedia.org/wiki/Monitor_(synchronization)]
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 \texttt{wait} on condition \( a \) adds thread to the queue \( a.q \)
- A call to \texttt{signal} for \( a \) adds thread to queue \( s \) (suspended)
- One thread form the \( a \) queue is woken up

![Diagram](http://en.wikipedia.org/wiki/Monitor_(synchronization))
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 $s$.
- If $s$ is empty, it wakes up one thread from $e$.

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 `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 $e$.

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

→ Signalled threads compete for the monitor.

Source: http://en.wikipedia.org/wiki/Monitor_(synchronization)
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 `c`

\[ \sim \] 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 {
            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);
}
```

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`

\[ \sim \] programmer should assume that thread is not the only one woken up
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 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`

Implementing PopRight with Monitors

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

```
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;
}
```
Implementing PopRight with Monitors

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

```c
int PopRight(DQueue* q, int val) {
    QNode* oldRightNode;
    monitor_enter(q->m); // wait to enter the critical section
    L: 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;
}
```

- 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
- use a binary 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 binary semaphore to block threads that are waiting

A semaphore can be implemented using a monitor:
- protect the semaphore variable with a monitor
Monitor versus Semaphores
A monitor can be implemented using semaphores:
- protect each queue with a mutex
- use a binary 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 binary 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
- \( \sim \) difficult to 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
  - \( \sim \) 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();
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:

```java
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();
```
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:

class Foo {
    public Foo other = null;
    public synchronized void bar() {
        a.bar() acquires the monitor of a
        ... 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();

Sequence leading to a deadlock:

1. threads A and B execute a.bar() and b.bar()
2. a.bar() acquires the monitor of a
3. ... if (*) other.bar(); ...
4. b.bar() acquires the monitor of b
5. A happens to execute other.bar()
6. A blocks on the monitor of b
7. B happens to execute other.bar()
8. 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:

class Foo {
    public Foo other = null;
    public synchronized void bar() {
        a.bar() acquires the monitor of a
        ... 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();

Sequence leading to a deadlock:

1. threads A and B execute a.bar() and b.bar()
2. a.bar() acquires the monitor of a
3. ... if (*) other.bar(); ...
4. b.bar() acquires the monitor of b
5. A happens to execute other.bar()
Treatment of Deadlocks

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

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 form processes
4. **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
- **avoidance**: use additional information about a program that allows the OS to schedule threads so that they do not deadlock

---

Atomic Executions, Locks and Monitors  Locked Atomic Executions  3 of 4

---

Atomic Executions, Locks and Monitors  Locked Atomic Executions  4 of 4
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

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

Each time a lock is acquired, we track the lock set at \( p \):

Definition (lock order)
Define \( \subseteq \subseteq L \times L \) such that \( l \triangleleft l' \) iff \( l \in \lambda(p) \) and the statement at \( p \) is of the form \( \text{wait}(l') \) or \( \text{monitor_enter}(l') \). Define the strict lock order \( \triangleleft \triangleleft \).

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

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

Each time a lock is acquired, we track the lock set at \( p \):

Definition (lock order)
Define \( \subseteq \subseteq L \times L \) such that \( l \triangleleft l' \) iff \( l \in \lambda(p) \) and the statement at \( p \) is of the form \( \text{wait}(l') \) or \( \text{monitor_enter}(l') \). Define the strict lock order \( \triangleleft \triangleleft \).

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 \( a \triangleleft a \) then the program is free of deadlocks.
**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 $\forall a \in L_S . a \neq a$ and $\forall a \in L_M . b \in L . a \prec b \wedge b \prec a \Rightarrow a = b$ then the program is free of deadlocks.

**Avoiding Deadlocks in Practice**

How can we verify that program contains no deadlocks?

- identify mutex locks $L_S$ and summarized monitor locks $L_M^* \subseteq L_M$

---

**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 $\forall a \in L_S . a \neq a$ and $\forall a \in L_M . b \in L . a \prec b \wedge b \prec a \Rightarrow a = b$ 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 $a \in L_M$ represents several concrete ones
  - thus, if $a \prec a$ then this might not be a self-cycle
- require that $a \neq a$ for all summarized monitors $a \in L_M$

**Avoiding Deadlocks in Practice**

How can we verify that program contains no deadlocks?

- identify mutex locks $L_S$ and summarized monitor locks $L_M^* \subseteq L_M$
- identify non-summary monitor locks $L_M^\# = L_M \setminus L_M^*$
Avoiding Deadlocks in Practice

How can we verify that program contains no deadlocks?

- Identify mutex locks $L_S$ and summarized monitor locks $L_M^* \subseteq L_M$
- Identify non-summary monitor locks $L_M^* = L_M \setminus L_M^*$
- Sort locks into ascending order according to lock sets
- Check that no cycles exist except for self-cycles of non-summary monitors

⚠️ What to do when lock order contains cycle? $
\bar{a}_M < a_{M+1}$

- Determining which locks may be acquired at each program point is undecidable \( \sim \) 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 \( \sim \) change program: release $l$, acquire $l'$, then acquire $l$ again \( \sim \) inefficient
- If a lock set contains a summarized lock $\bar{a}$ and $\bar{a}$ is to be acquired, we’re stuck

4/41 Atomic Executions, Locks and Monitors

Locked Atomic Executions

Avoiding Deadlocks in Practice

How can we verify that program contains no deadlocks?

- Identify mutex locks $L_S$ and summarized monitor locks $L_M^* \subseteq L_M$
- Identify non-summary monitor locks $L_M^* = L_M \setminus L_M^*$
- Sort locks into ascending order according to lock sets
- Check that no cycles exist except for self-cycles of non-summary monitors

⚠️ What to do when lock order contains cycle?

- Determining which locks may be acquired at each program point is undecidable \( \sim \) 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 \( \sim \) change program: release $l$, acquire $l'$, then acquire $l$ again \( \sim \) inefficient
- If a lock set contains a summarized lock $\bar{a}$ and $\bar{a}$ is to be acquired, we’re stuck

An example for the latter is the Foo class: two instances of the same class call each other

4/41 Atomic Executions, Locks and Monitors

Locked Atomic Executions
Refining the Queue: Concurrent Access

Add a second lock $s \rightarrow t$ to allow concurrent removal:

```c
int PopRight(DQueue* q) {
    QNode* oldRightNode = q->right;
    wait(q->t); // wait to enter the critical section
    QNode* rightSentinel = q->right;
    oldRightNode = rightSentinel->left;
    if (oldRightNode == leftSentinel) { signal(q->t); return -1; }
    QNode* newRightNode = oldRightNode->left;
    int c = newRightNode == leftSentinel;
    if (c) wait(q->s);
    newRightNode->right = rightSentinel;
    rightSentinel->left = newRightNode;
    if (c) signal(q->s);
    signal(q->t); // signal that we're done
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```

Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

```c
void PopRight() {
    ...
    wait(q->t);
    ...
    if (*) { signal(q->t); return; }
    ...
    if (c) wait(q->s);
    ...
    if (c) signal(q->s);
    signal(q->t);
}
```

Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

```c
void PopRight() {
    ... wait(q->t);
    ... if (*) { signal(q->t); return; }
    ... if (c) wait(q->s);
    ... if (c) signal(q->s);
    signal(q->t);
}
```

Atomic Execution and Locks

Consider replacing the specific locks with atomic annotations:

```c
void PopRight() {
    ... wait(q->t); atomic
    ...
    if (*) { signal(q->t); return; }
    ...
    if (c) wait(q->s); atomic
    ...
    if (c) signal(q->s);
    signal(q->t); 3
}
```
Atomic Execution and Locks

Consider replacing the specific locks with \texttt{atomic} annotations:

```c
/* double-ended queue: removal */
void PopRight() {
    ...
    wait(q->t);
    ...
    if (*) { signal(q->t); return; }
    ...
    if (c) wait(q->s);
    ...
    if (c) signal(q->s);
    signal(q->t);
}
```

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

Outlook

Writing \texttt{atomic} annotations around sequences of statements is a convenient way of programming.

Concurrency across Languages

In most systems programming languages (C,C++) we have
- the ability to use \texttt{atomic} operations

\textit{Idea of mutexes:} Implement \texttt{atomic} sections with locks:
- a single lock could be used to protect all \texttt{atomic} blocks
- more concurrency is possible by using several locks
  - see the \texttt{PushLeft,PopRight} example
- some statements might modify variables that are never read by other threads \texttt{atomic} lock required
- statements in one \texttt{atomic} block might access variables in a different order to another \texttt{atomic} block \texttt{atomic} lock required to release locks in \texttt{lambda} when acquiring /
Concurrence 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-lock-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>✓/✓/✓</td>
</tr>
<tr>
<td>Java, C#</td>
<td>-</td>
<td>-</td>
<td>(b)</td>
<td>✓</td>
<td>✓/✓/✓</td>
</tr>
</tbody>
</table>

(a) Some pthread implementations allow a *reentrant* attribute
(b) 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 what they can do

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