Script **generated by TTT**

Title: Petter: Programmiersprachen (29.10.2014)

Date: Wed Oct 29 14:18:06 CET 2014

Duration: 85:35 min

Pages: 79

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

19.11. there is a lecture

Programming Languages

Concurrency: Atomic Executions, Locks and Monitors

Dr. Axel Simon and Dr. Michael Petter

Winter term 2014

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:

```
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;
  while (!mine) {
    atomic {
      mine = thread_id()==m->tid;
      if (mine) m->count++; else
        if (m->tid==0) {
          mine = true; m->count=1;
          m->tid = thread_id();
        }
    };
    if (!mine) de_schedule(&m->tid);}}
```

```
void monitor_leave(mon_t *m) {
  atomic {
    m->count--;
    if (m->count==0) {
      // wake up threads
      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

```
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:

Rewriting the Queue using Monitors

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

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`

Rewriting the Queue using Monitors

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

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:

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
  - ⚠ t 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 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:

```
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:

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


Define these two functions:

1. `wait` for the condition to become true
   - called while being *inside* the monitor
   - temporarily *releases* the monitor and blocks
   - when *signalled*, re-acquires the monitor and returns
2. `signal` waiting threads that they may be able to proceed
   - one/all waiting threads that called *wait* will be woken up, two possibilities:

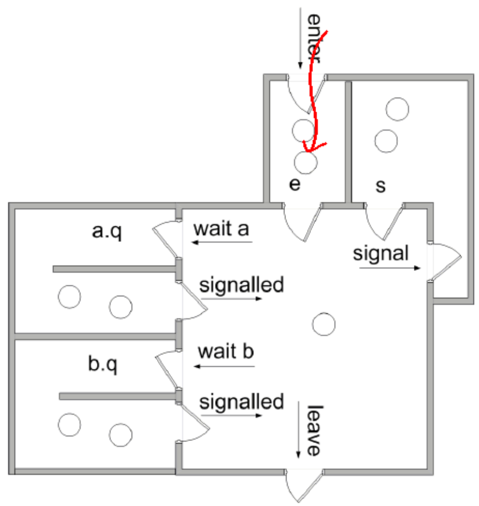
     signal-and-urgent-wait : the *signalling* thread suspends and continues once the *signalled* thread has released the monitor

     signal-and-continue the *signalling* thread continues, any *signalled* thread enters when the monitor becomes available

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

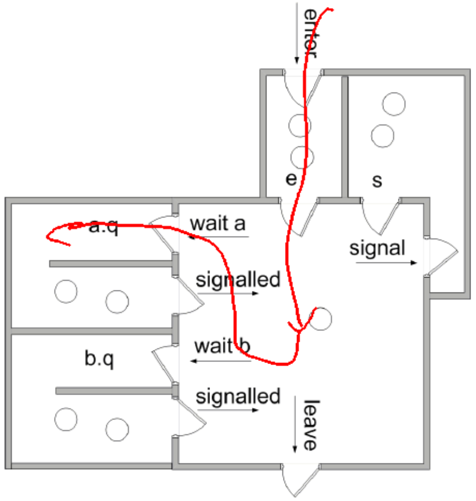


source: `http://en.wikipedia.org/wiki/Monitor_(synchronization)`

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 `wait` on condition a adds thread to the queue $a.q$

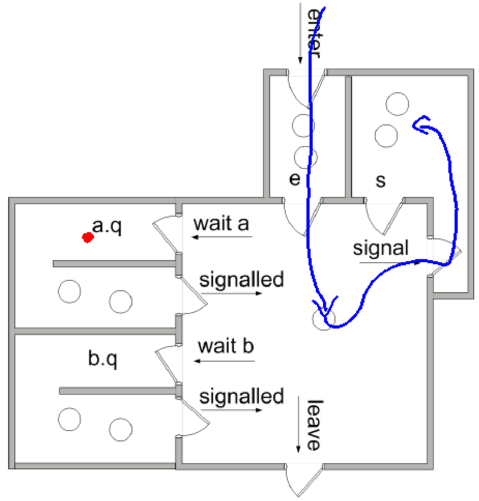


source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

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 `wait` on condition a adds thread to the queue $a.q$
- a call to `signal` for a adds thread to queue s (suspended)

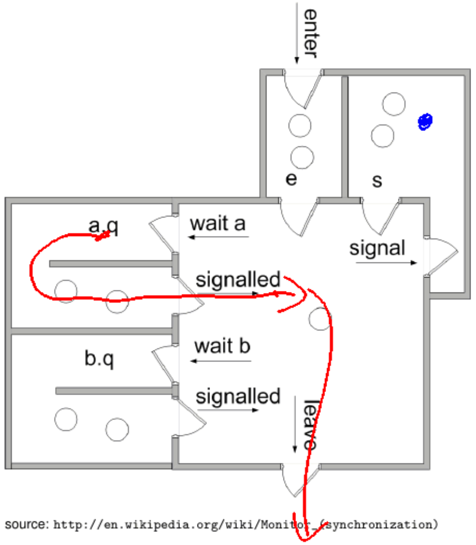


source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

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 `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

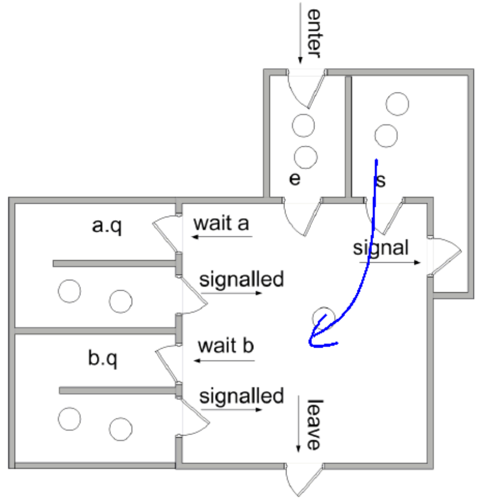


source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

Signal-And-Urgent-Wait Semantics

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

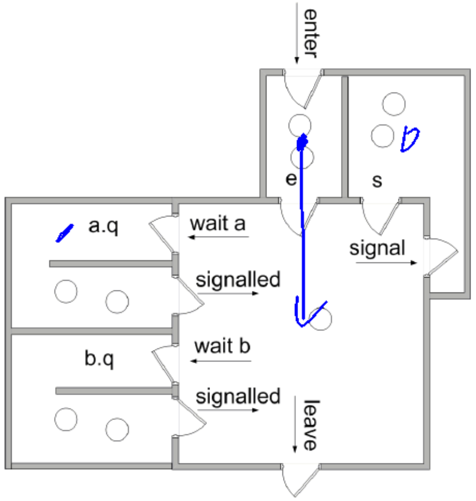
- a thread who tries to enter a monitor is added to queue e if the monitor is occupied
- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `signal` for a adds thread to queue s (suspended)
- one thread form the a queue is woken up
- `signal` on a is a no-op if $a.q$ is empty



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:

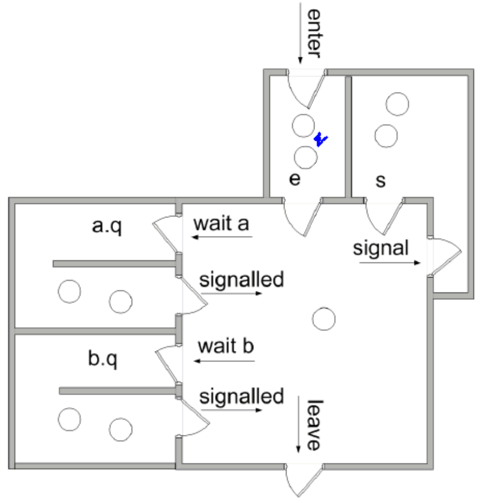


source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

- 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-Urgent-Wait Semantics

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



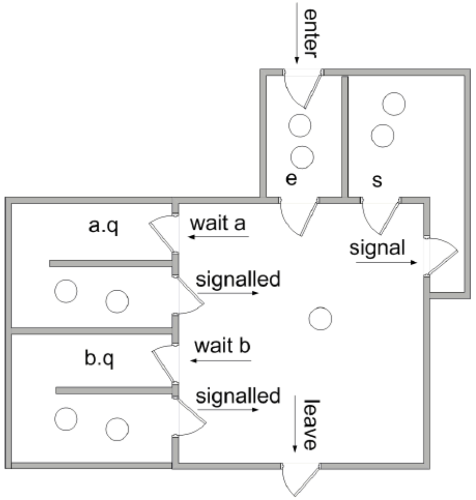
source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

- 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

$\rightsquigarrow$ queue s has priority over e

Signal-And-Urgent-Wait Semantics

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



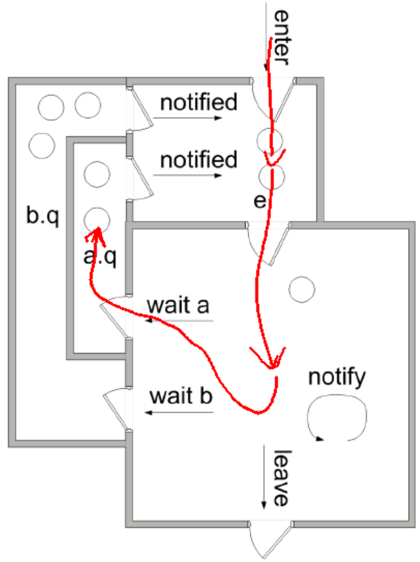
source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

- 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

$\rightsquigarrow$ queue s has priority over e

Signal-And-Continue Semantics

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

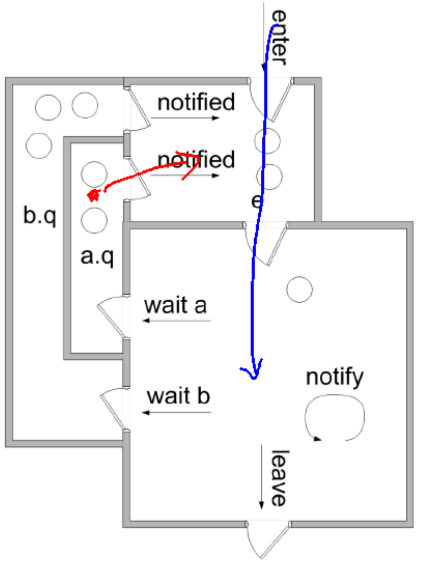


- a call to `wait` on condition a adds thread to the queue $a.q$

source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

Signal-And-Continue Semantics

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

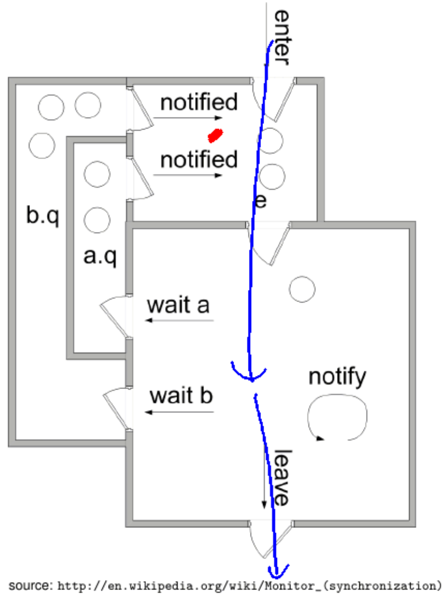


- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `notify` for a adds one thread from $a.q$ to e (unless $a.q$ is empty)

source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

Signal-And-Continue Semantics

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

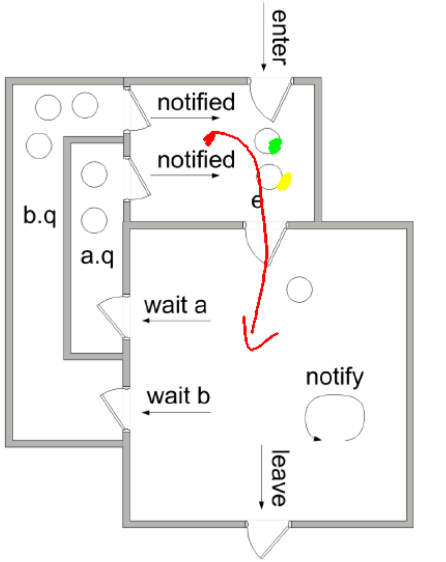


- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `notify` for a adds one thread from $a.q$ to e (unless $a.q$ is empty)
- if a thread leaves, it wakes up one thread waiting on e

source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

Signal-And-Continue Semantics

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



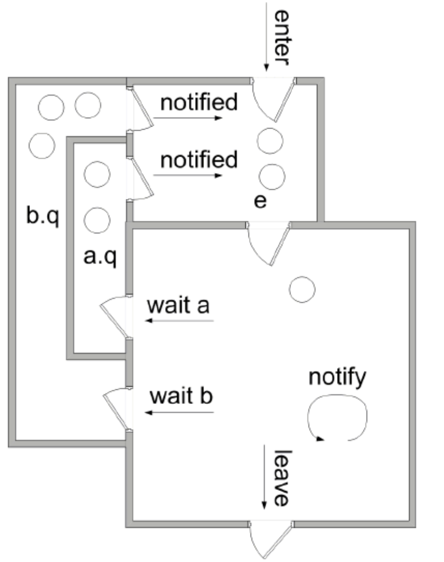
- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `notify` for a adds one thread from $a.q$ to e (unless $a.q$ is empty)
- if a thread leaves, it wakes up one thread waiting on e

⇝ 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 e

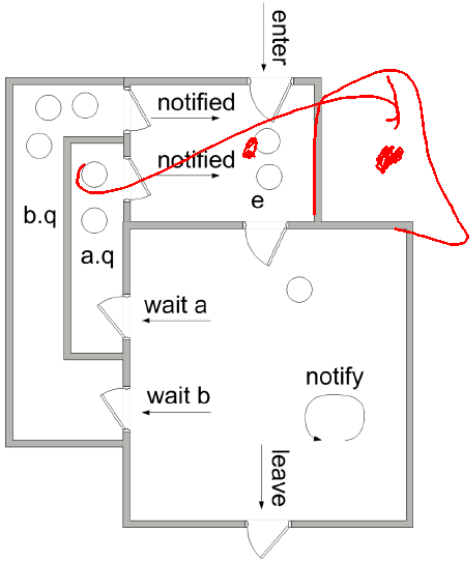
⇝ signalled threads compete for the monitor

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

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 e

⇝ signalled threads compete for the monitor

- assuming FIFO ordering on e, threads who tried to enter between `wait` and `notify` will run first
- need additional queue s if waiting threads should have priority

source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

Implementing Condition Variables

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

- a *notified* thread is simply woken up and competes for the monitor

```
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);}
```

```
void cond_notify(mon_t *m) {
  // wake up other threads
  signal(m->cond);
}
```


A Note on Notify

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

1. `notify`: wakes up exactly one thread waiting on condition variable
2. `notifyAll`: wakes up all threads waiting on a condition variable

A Note on Notify

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

1. `notify`: wakes up exactly one thread waiting on condition variable
2. `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

A Note on Notify

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

1. `notify`: wakes up exactly one thread waiting on condition variable
2. `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:

1. `notify`: wakes up exactly one thread waiting on condition variable
2. `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

A Note on Notify

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

1. `notify`: wakes up exactly one thread waiting on condition variable
2. `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

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
L: QNode* rightSentinel = q->right;
  oldRightNode = rightSentinel->left;
  if (oldRightNode==leftSentinel) { cond_wait(q->c); goto L; }
  QNode* newRightNode = oldRightNode->left;
  newRightNode->right = rightSentinel;
  rightSentingel->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:

double-ended queue: removal

```
  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;
    rightSentingel->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

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 s 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 $s = 0$

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

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 `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
- $\leadsto$ 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

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
- $\rightsquigarrow$ 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
  - $\rightsquigarrow$ wake up thread and have it check the predicate itself

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
- $\rightsquigarrow$ 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
  - $\rightsquigarrow$ 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 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
- $\rightsquigarrow$ 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
  - $\rightsquigarrow$ 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

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:

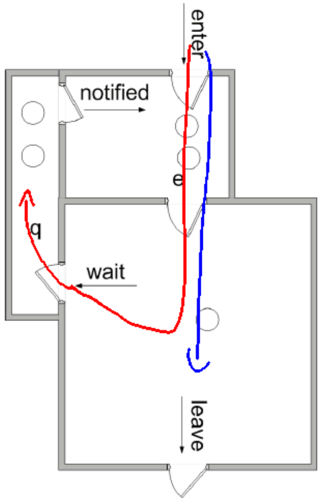
- protect the semaphore variable s with a monitor
- implement `wait` by calling `cond_wait` if $s = 0$

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
- $\rightsquigarrow$ 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
  - $\rightsquigarrow$ 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
- or, simpler: notify all threads each time any predicate changes
  - $\rightsquigarrow$ without predicates, a single condition variable suffices!

Monitors with a Single Condition Variable

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



source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

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

is equivalent to

```
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.)

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() {
    ... 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:

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() {
    ... 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:

- 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:

```
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();
```

Sequence leading to a deadlock:

- threads A and B execute a.bar() and b.bar()
- a.bar() acquires the monitor of a

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() {
    ... 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:

- 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()

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() {
    ... 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:

- 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()
- $\rightsquigarrow$ both *block* indefinitely

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:

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:

1. *ignored*: for the lack of better approaches, can be reasonable if deadlocks are rare

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:

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 [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:

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

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:

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 achieve 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.

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 σ^+ of a relation σ:

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

$$\begin{aligned} \sigma^0 &= \sigma \\ \sigma^{i+1} &= \{\langle x_1, x_3 \rangle \mid \exists x_2 \in X . \langle x_1, x_2 \rangle \in \sigma^i \wedge \langle x_2, x_3 \rangle \in \sigma^i\} \end{aligned}$$

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 σ^+ of a relation σ:

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

$$\begin{aligned} \sigma^0 &= \sigma \\ \sigma^{i+1} &= \{\langle x_1, x_3 \rangle \mid \exists x_2 \in X . \langle x_1, x_2 \rangle \in \sigma^i \wedge \langle x_2, x_3 \rangle \in \sigma^i\} \end{aligned}$$

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

Definition (lock order)

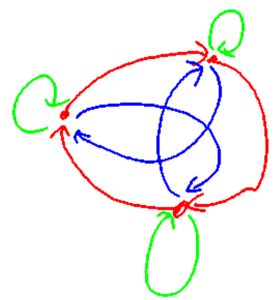
Define $\triangleleft \subseteq L \times L$ such that $l \triangleleft l'$ iff $l \in \lambda(p)$ and the statement at p is of the form `wait(l')` or `monitor_enter(l')`. Define the strict lock order $\prec = \triangleleft^+$.

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 $a \prec a$ 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 $\forall a \in L_S . a \not\prec a$ 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.

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 $\forall a \in L_S . a \not\prec a$ 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/moitor $\bar{a} \in L_M$ represents several concrete ones
  - thus, if $\bar{a} \prec \bar{a}$ then this might not be a self-cycle
  - $\leadsto$ require that $\bar{a} \not\prec \bar{a}$ for all summarized monitors $\bar{a} \in L_M$

Avoiding Deadlocks in Practice

How can we verify that program contains no deadlocks?

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

Avoiding Deadlocks in Practice

How can we verify that program contains no deadlocks?

- identify mutex locks L_S and summarized monitor locks $L^s_M \subseteq L_M$
- identify non-summary monitor locks $L^n_M = L_M \setminus L^s_M$

Avoiding Deadlocks in Practice

How can we verify that program contains no deadlocks?

- identify mutex locks L_S and summarized monitor locks $L^s_M \subseteq L_M$
- identify non-summary monitor locks $L^n_M = L_M \setminus L^s_M$
- sort locks into ascending order according to lock sets

Avoiding Deadlocks in Practice

How can we verify that program contains no deadlocks?

- identify mutex locks L_S and summarized monitor locks $L^s_M \subseteq L_M$
- identify non-summary monitor locks $L^n_M = L_M \setminus L^s_M$
- sort locks into ascending order according to lock sets
- check that no cycles exist except for self-cycles of non-summary monitors

Avoiding Deadlocks in Practice

How can we verify that program contains no deadlocks?

- identify mutex locks L_S and summarized monitor locks $L^s_M \subseteq L_M$
- identify non-summary monitor locks $L^n_M = L_M \setminus L^s_M$
- 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 $\leadsto$ lock sets are an approximation
- an array of locks in L_S: lock in increasing array index sequence
- if $l \in \lambda(P)$ exists $l' \prec l$ is to be acquired $\leadsto$ change program: release l, acquire l', then acquire l again $\leadsto$ inefficient
- if a lock set contains a summarized lock $\bar{a}$ and $\bar{a}$ is to be acquired, we're stuck

Avoiding Deadlocks in Practice

How can we verify that program contains no deadlocks?

- identify mutex locks L_S and summarized monitor locks $L^s_M \subseteq L_M$
- identify non-summary monitor locks $L^n_M = L_M \setminus L^s_M$
- 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 $\leadsto$ lock sets are an approximation
- an array of locks in L_S: lock in increasing array index sequence
- if $l \in \lambda(P)$ exists $l' \prec l$ is to be acquired $\leadsto$ change program: release l, acquire l', then acquire l again $\leadsto$ 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

Refining the Queue: Concurrent Access

Add a second lock `s->t` to allow concurrent removal:

double-ended queue: removal

```
int PopRight(DQueue* q) {
  QNode* oldRightNode;
  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:

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);
}
```


Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

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);
}
```

- in `PushLeft`, the lock set for s is empty
- here, the lock set of s is $\{t\}$
- $t \lhd s$ and transitive closure is $t \prec s$
- $\leadsto$ the program cannot deadlock

Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

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);
}
```

Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

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 `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.

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 use to protect all `atomic` blocks
- more concurrency is possible by using several locks
  - see the `PushLeft`, `PopRight` example
- 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 $\lambda(l)$ when acquiring l

Concurrency across Languages

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

- the ability to use *atomic* operations

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

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

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

language	barriers	wait-/lock-free	semaphore	mutex	monitor
C,C++	✓	✓	✓	✓	(a)
Java,C#	-	-	(b)	✓	✓

(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