Script generated by TTT

Title: Simon: Programmiersprachen (15.11.2013)

Date: Fri Nov 15 14:14:18 CET 2013

Duration: 90:34 min

Pages: 96

Marking Statements as Atomic

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

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

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

Program 3
```
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```


Marking Statements as Atomic

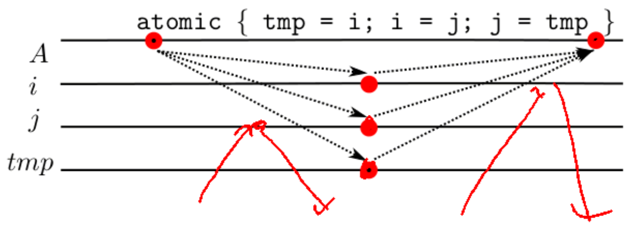
Rather than writing assembler: use made-up keyword `atomic`:

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

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

Program 3
```
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```

The statements in an `atomic` block execute as *atomic execution*:



Marking Statements as Atomic

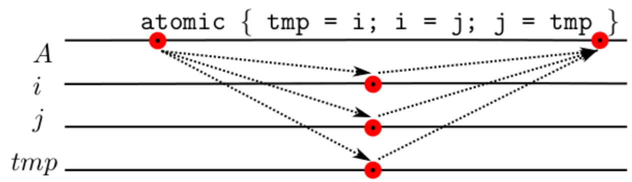
Rather than writing assembler: use made-up keyword `atomic`:

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

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

Program 3
```
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```

The statements in an `atomic` block execute as *atomic execution*:



- `atomic` only translatable when a corresponding atomic CPU instruction exist
- the notion of requesting *atomic execution* is a general concept

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:

- no control flow possible, no behavioral change depending on data

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:

- no control flow possible, no behavioral change depending on data
- instructions often exist that execute an operation conditionally

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:

- no control flow possible, no behavioral change depending on data
- instructions often exist that execute an operation conditionally

Program 4

```
atomic {
  r = b;
  b = 0;
}
```

Program 5

```
atomic {
  r = b;
  b = 1;
}
```

Program 6

```
atomic {
  r = (k==i);
  if (r) i = j;
}
```

Operations *update* a memory cell and *return* the previous value.

- the first two operations can be seen as setting a flag `b` to $v \in \{0, 1\}$ if `b` not already contains v
  - this operation is called *modify-and-test*
- the third case generalizes this to arbitrary values
  - this operation is called *compare-and-swap*

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:

- no control flow possible, no behavioral change depending on data
- instructions often exist that execute an operation conditionally

Program 4

```
atomic {
  r = b;
  b = 0;
}
```

Program 5

```
atomic {
  r = b;
  b = 1;
}
```

Program 6

```
atomic {
  r = (k==i);
  if (r) i = j;
}
```

Operations *update* a memory cell and *return* the previous value.

- the first two operations can be seen as setting a flag `b` to $v \in \{0, 1\}$ if `b` not already contains v
  - this operation is called *modify-and-test*
- the third case generalizes this to arbitrary values
  - this operation is called *compare-and-swap*

$\leadsto$ use as building blocks for algorithms that can *fail*

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.

Lock-Free Algorithms

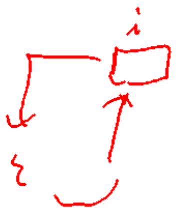
If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.
Common usage pattern for *compare and swap*:

1. read the initial value in i into k (using memory barriers)
2. calculate a new value $j = f(k)$
3. update i to j if $i = k$ still holds
4. go to first step if $i \neq k$ meanwhile

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.
Common usage pattern for *compare and swap*:

1. read the initial value in i into k (using memory barriers)
2. calculate a new value $j = f(k)$
3. update i to j if $i = k$ still holds
4. go to first step if $i \neq k$ meanwhile



⚠ note: $i = k$ must imply that no thread has updated i

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.
Common usage pattern for *compare and swap*:

1. read the initial value in i into k (using memory barriers)
2. calculate a new value $j = f(k)$
3. update i to j if $i = k$ still holds
4. go to first step if $i \neq k$ meanwhile

⚠ note: $i = k$ must imply that no thread has updated i

$\leadsto$ general recipe for *lock-free* algorithms

- given a compare-and-swap operation for n bytes
- try to group variables for which an invariant must hold into n bytes
- read these bytes atomically
- calculate a new value
- perform a compare-and-swap operation on these n bytes

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.
Common usage pattern for *compare and swap*:

1. read the initial value in i into k (using memory barriers)
2. calculate a new value $j = f(k)$
3. update i to j if $i = k$ still holds
4. go to first step if $i \neq k$ meanwhile

⚠ note: $i = k$ must imply that no thread has updated i

⇝ general recipe for *lock-free* algorithms

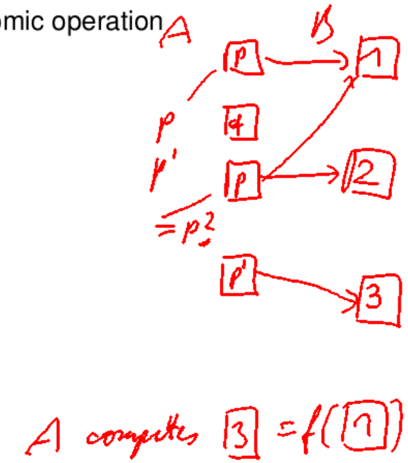
- given a compare-and-swap operation for n bytes
- try to group variables for which an invariant must hold into n bytes
- read these bytes atomically
- calculate a new value
- perform a compare-and-swap operation on these n bytes

⇝ calculating new value must be *repeatable*

Limitations of Wait- and Lock-Free Algorithms

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

- restricted to the semantics of a *single* atomic operation



Limitations of Wait- and Lock-Free Algorithms

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

- restricted to the semantics of a *single* atomic operation

Limitations of Wait- and Lock-Free Algorithms

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

- restricted to the semantics of a *single* atomic operation
- set of atomic operations is architecture specific, but often includes

Limitations of Wait- and Lock-Free Algorithms

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

- restricted to the semantics of a *single* atomic operation
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register

Limitations of Wait- and Lock-Free Algorithms

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

- restricted to the semantics of a *single* atomic operation
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory

Limitations of Wait- and Lock-Free Algorithms

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

- restricted to the semantics of a *single* atomic operation
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand

⇝ only very simple algorithms can be implemented, for instance

Limitations of Wait- and Lock-Free Algorithms

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

- restricted to the semantics of a *single* atomic operation
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand

⇝ only very simple algorithms can be implemented, for instance

binary semaphores : a flag that can be acquired (set) if free (unset) and released

counting semaphores : an integer that can be decreased if non-zero and increased

Limitations of Wait- and Lock-Free Algorithms

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

- restricted to the semantics of a *single* atomic operation
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand

⇝ only very simple algorithms can be implemented, for instance

binary semaphores : a flag that can be acquired (set) if free (unset) and released

counting semaphores : an integer that can be decreased if non-zero and increased

mutex : ensures mutual exclusion using a binary semaphore

Limitations of Wait- and Lock-Free Algorithms

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

- restricted to the semantics of a *single* atomic operation
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand

⇝ only very simple algorithms can be implemented, for instance

binary semaphores : a flag that can be acquired (set) if free (unset) and released

counting semaphores : an integer that can be decreased if non-zero and increased

mutex : ensures mutual exclusion using a binary semaphore

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 operation
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand

⇝ only very simple algorithms can be implemented, for instance

binary semaphores : a flag that can be acquired (set) if free (unset) and released

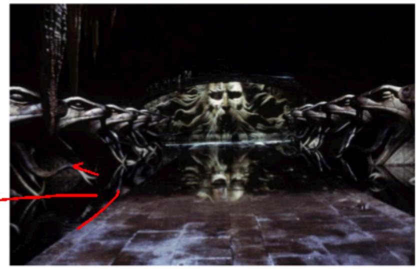
counting semaphores : an integer that can be decreased if non-zero and increased

mutex : ensures mutual exclusion using a binary semaphore

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*.

Locks

 ⇒ 

A lock is a data structure that

- protects a *critical section*: a piece of code that may produce incorrect results when executed concurrently from several threads
- it ensures *mutual exclusion*: no two threads execute at once
- *block* other threads as soon as one thread executes the critical section
- can be *acquired* and *released*
- may *deadlock* the program

Semaphores and Mutexes

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

```
void signal() {                 void wait() {
  atomic { s = s + 1; }           bool avail;
}                                 do {
                                    atomic {
                                      avail = s>0;
                                      if (avail) s--;
                                    }
                                  } while (!avail);
                                }
```

free/release

acquire

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

Semaphores and Mutexes

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

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

Semaphores and Mutexes

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

```
void signal() {                 void wait() {
  atomic { s = s + 1; }           bool avail;
}                                 do {
                                    atomic {
                                      avail = s>0;
                                      if (avail) s--;
                                    }
                                  } while (!avail);
                                }
```

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

- a thread requiring a resource executes `wait()`
- if a resource is still available, `wait()` returns

Semaphores and Mutexes

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

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

Semaphores and Mutexes

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

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

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

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

- a thread requiring a resource executes `wait()`
- if a resource is still available, `wait()` returns
- once a thread finishes using a resource, it calls `signal()`
- (choosing which available resource to use requires more synchr.)

Special case: initializing with $s = 1$ gives a *binary* semaphore:

- can be used to block and unblock a thread

Implementation of Semaphores

A *semaphore* does not have to busy wait:

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

```
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:

Implementation of Semaphores

A *semaphore* does not have to busy wait:

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

```
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()`

Implementation of Semaphores

A *semaphore* does not have to busy wait:

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

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

Busy waiting is avoided by placing waiting threads into queue:

- a thread failing to decrease s executes `de_schedule()`

- `de_schedule()` enters the operating system and adds the waiting thread into a queue of threads *waiting for a write* to memory address `&s`
- once a thread calls `signal()`, the first thread t waiting on `&s` is extracted

Implementation of Semaphores

A *semaphore* does not have to busy wait:

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

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

Busy waiting is avoided by placing waiting threads into queue:

- a thread failing to decrease s executes `de_schedule()`
- `de_schedule()` enters the operating system and adds the waiting thread into a queue of threads *waiting for a write* to memory address `&s`
- once a thread calls `signal()`, the first thread t waiting on `&s` is extracted
- the operating system lets t return from its call to `de_schedule()`

Practical Implementation of Semaphores

Certain optimisations are possible:

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

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:

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

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

In general, the implementation is more complicated

- `wait()` may busy wait for a few iterations
  - saves de-scheduling if the lock is released frequently

Practical Implementation of Semaphores

Certain optimisations are possible:

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

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

In general, the implementation is more complicated

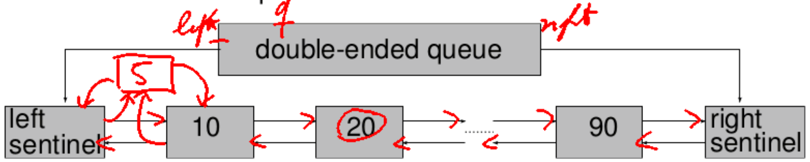
- `wait()` may busy wait for a few iterations
  - saves de-scheduling if the lock is released frequently
  - better throughput for semaphores that are held for a short time
- `signal()` might have to inform the OS that `s` has been written

⇝ 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:



double-ended queue: adding an element

```
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*

Mutexes

One common use of semaphores is to guarantee mutual exclusion.

- in this case, a binary semaphore is also called a *mutex*
- add a lock to the double-ended queue data structure

Implementing the Removal

By using the same lock `q->s`, we can write a thread-safe `PopRight`:

double-ended queue: removal

```
int PopRight(DQueue* q) {
  QNode* oldRightNode;
  QNode* leftSentinel = q->left;
  QNode* rightSentinel = q->right;
  wait(q->s); // wait to enter the critical section
  oldRightNode = rightSentinel->left;
  if (oldRightNode==leftSentinel) { signal(q->s); return -1; }
  QNode* newRightNode = oldRightNode->left;
  newRightNode->right = rightSentinel;
  rightSentinel->left = newRightNode;
  signal(q->s); // signal that we're done
  int val = oldRightNode->val;
  free(oldRightNode);
  return val;
}
```

Implementing the Removal

By using the same lock q->s, we can write a thread-safe PopRight:

double-ended queue: removal

```
int PopRight(DQueue* q) {
  QNode* oldRightNode;
  QNode* leftSentinel = q->left;
  QNode* rightSentinel = q->right;
  wait(q->s); // wait to enter the critical section
  oldRightNode = rightSentinel->left;
  if (oldRightNode==leftSentinel) { signal(q->s); return -1; }
  QNode* newRightNode = oldRightNode->left;
  newRightNode->right = rightSentinel;
  rightSentinel->left = newRightNode;
  signal(q->s); // signal that we're done
  int val = oldRightNode->val;
  free(oldRightNode);
  return val;
}
```

- 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:

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

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:

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`

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

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
   - ⚠ t is busy waiting and produces contention on the lock

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
   - ⚠ 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
   - ⚠ 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
   - ⚠ 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

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
   - ⚠ 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

⇝ need a way to release the lock after the return of the last recursive call

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:

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

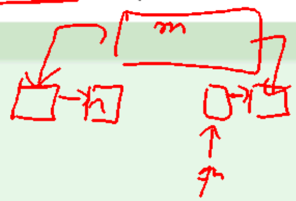
```
void monitor_enter(mon_t *m) {
  bool mine = false;
  while (!mine) {
    atomic {
      mine = thread_id()==m->tid;
      if (mine) m->count++; else
        if (m->tid==0) {
          mine = true; m->count=1;
          m->tid = thread_id();
        }
    };
    if (!mine) de_schedule(&m->tid);}}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

```
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:

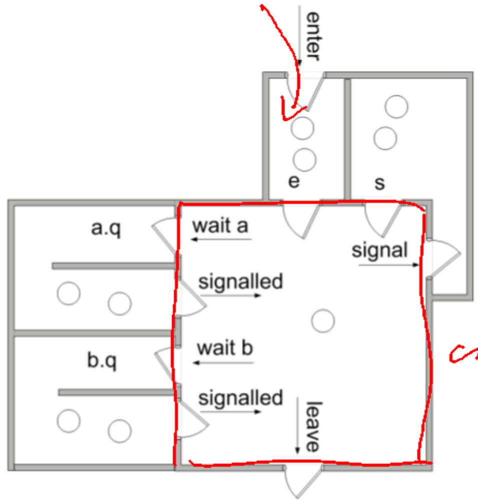
     signal-and-urgent-wait : the *signalling* thread suspends and continues once the *signalled* thread has released the monitor

     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

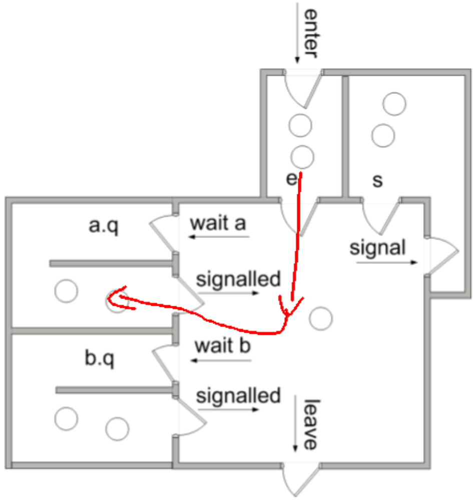


source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

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 $a.q$

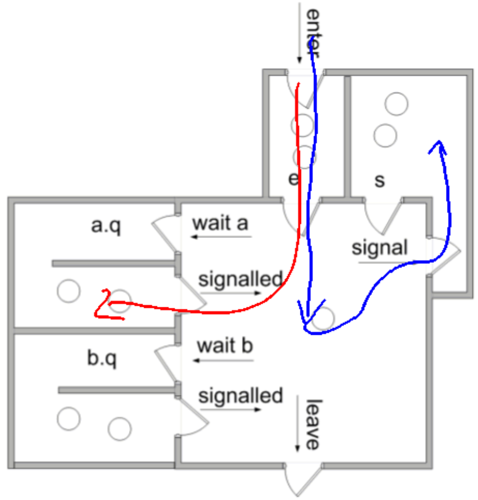


source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

Signal-And-Urgent-Wait Semantics

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

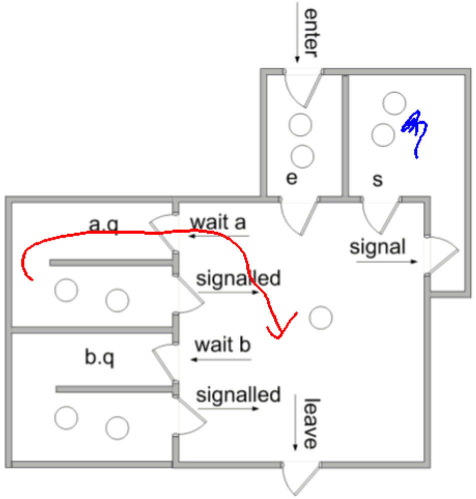
- a thread who tries to enter a monitor is added to queue e if the monitor is occupied
- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `signal` for a adds thread to queue s (suspended)



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:

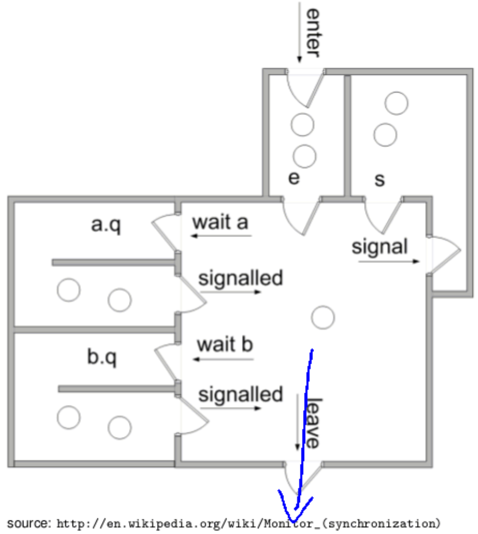


- a thread who tries to enter a monitor is added to queue e if the monitor is occupied
- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `signal` for a adds thread to queue s (suspended)
- one thread form the a queue is woken up

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:

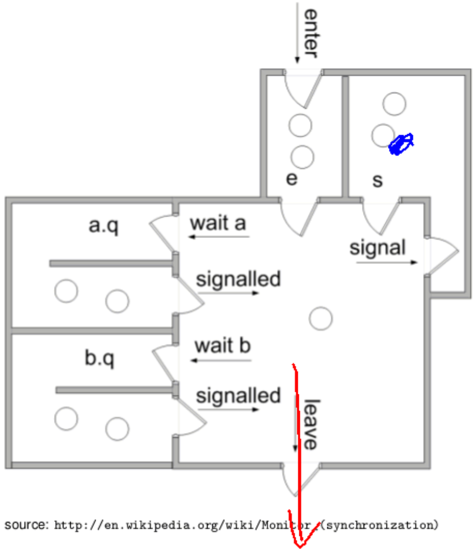


- a thread who tries to enter a monitor is added to queue e if the monitor is occupied
- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `signal` for a adds thread to queue s (suspended)
- one thread form the a queue is woken up
- `signal` on a is a no-op if $a.q$ is empty

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:

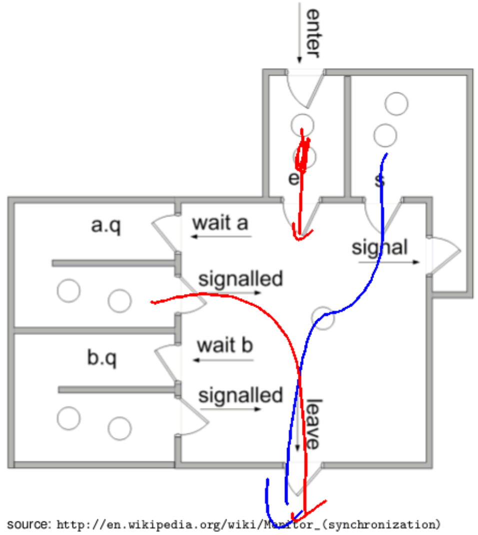


- a thread who tries to enter a monitor is added to queue e if the monitor is occupied
- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `signal` for a adds thread to queue s (suspended)
- one thread form the a queue is woken up
- `signal` on a is a no-op if $a.q$ is empty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

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:

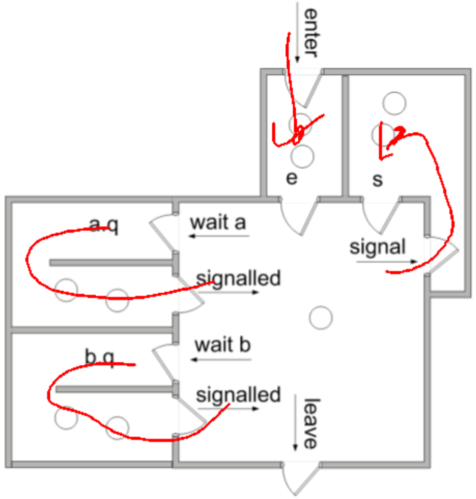


- a thread who tries to enter a monitor is added to queue e if the monitor is occupied
- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `signal` for a adds thread to queue s (suspended)
- one thread form the a queue is woken up
- `signal` on a is a no-op if $a.q$ is empty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

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:



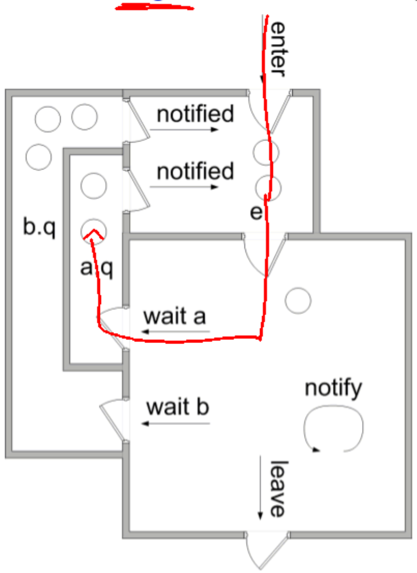
source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

- 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

$\leadsto$ queue s has priority over e

Signal-And-Continue Semantics

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

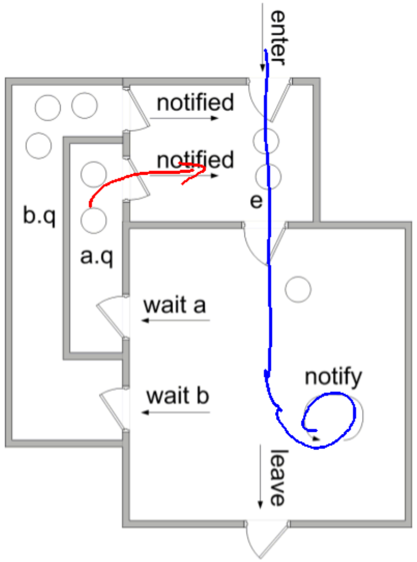


source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

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

Signal-And-Continue Semantics

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

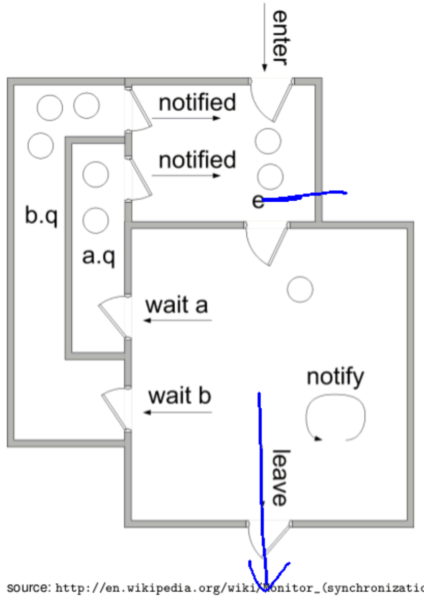


source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

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

Signal-And-Continue Semantics

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

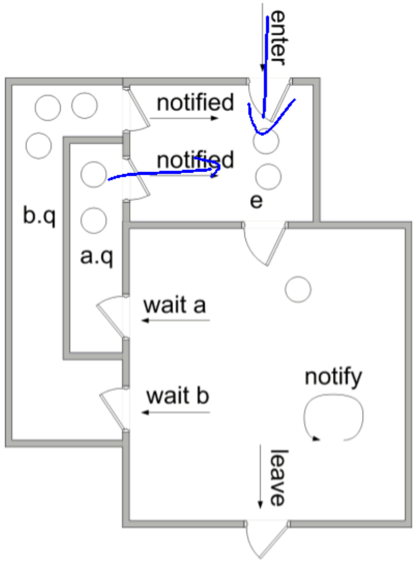


source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

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

Signal-And-Continue Semantics

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



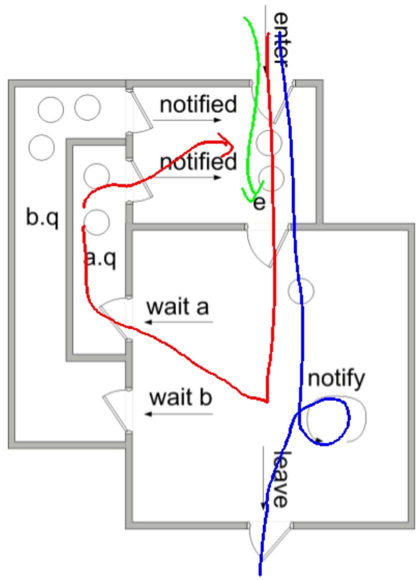
- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `notify` for a adds one thread from $a.q$ to e (unless $a.q$ is empty)
- if a thread leaves, it wakes up one thread waiting on e

⇝ signalled threads compete for the monitor

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

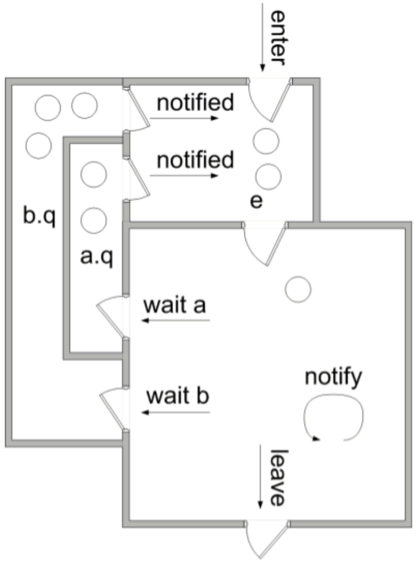
⇝ signalled threads compete for the monitor

- 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;
  de_schedule(&m->cond);
  bool next_to_enter;
  do {
    atomic {
      next_to_enter = m->tid==0;
      if (next_to_enter) {
        m->tid = thread_id();
        m->count = old_count;
      }
    }
    if (!next_to_enter) de_schedule(&m->tid);
  } while (!next_to_enter);
}
```

```
void cond_notify(mon_t *m) {
  // wake up other threads
  m->cond = 1;
}
```

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

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

What about the priority of notified threads?

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

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 semaphore to block threads that are waiting

A semaphore can be implemented using a monitor:

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

Monitor versus Semaphores

A monitor can be implemented using semaphores:

- protect each queue with a mutex
- use a semaphore to block threads that are waiting

A semaphore can be implemented using a monitor:

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

A note on the history of monitors:

- condition variables were meant to be associated with a predicate p
- signalling a variables would only wake up a thread if p is true

Monitor versus Semaphores

A monitor can be implemented using semaphores:

- protect each queue with a mutex
- use a semaphore to block threads that are waiting

A semaphore can be implemented using a monitor:

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

A note on the history of monitors:

- condition variables were meant to be associated with a predicate p
- signalling a variables would only wake up a thread if p is true
- $\leadsto$ 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
- $\rightsquigarrow$ 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
  - $\rightsquigarrow$ 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 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 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 semaphore to block threads that are waiting

A semaphore can be implemented using a monitor:

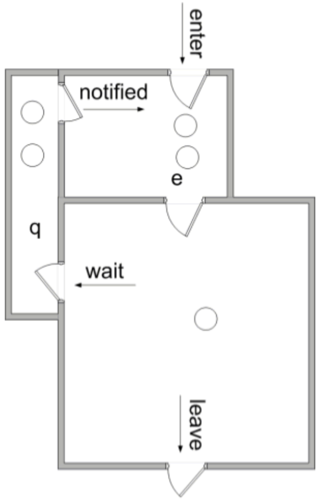
- protect the semaphore variable s with a monitor
- implement `wait` by calling `cond_wait` if $s = 0$

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 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

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.)

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`
- `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()`
- $\leadsto$ both *block* indefinitely