TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Atomic Executions, Locks and Monitors

Dr. Michael Petter
Winter 2019

Why Memory Barriers are not Enough

Often, *multiple memory locations* may only be modified exclusively by one thread during a computation.

- use barriers to implement automata that ensure *mutual exclusion*

⇝ generalize the re-occurring *concept* of enforcing mutual exclusion

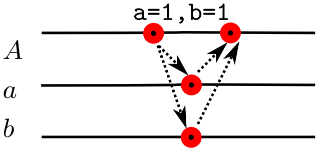
Why Memory Barriers are not Enough

Often, *multiple memory locations* may only be modified exclusively by one thread during a computation.

- use barriers to implement automata that ensure *mutual exclusion*

⇝ generalize the re-occurring *concept* of enforcing mutual exclusion

Needed: interaction with *multiple memory locations* within a *single step*:



Atomic Executions

A concurrent program consists of several threads that share *resources*:

- resources can be *memory locations* or *memory mapped I/O*
  - a file can be modified through a shared handle, e.g.
- usually *invariants* must be retained wrt. resources
  - e.g. a head and tail pointer must delimit a linked list
  - an invariant may span *multiple* resources
  - during an update, the invariant may be temporarily *locally broken*

⇝ multiple resources must be updated together to ensure the invariant

Atomic Executions

A concurrent program consists of several threads that share *resources*:

- resources can be *memory locations* or *memory mapped I/O*
  - a file can be modified through a shared handle, e.g.
- usually *invariants* must be retained wrt. resources
  - e.g. a head and tail pointer must delimit a linked list
  - an invariant may span *multiple* resources
  - during an update, the invariant may be temporarily *locally broken*

⇝ multiple resources must be updated together to ensure the invariant

Ideally, a sequence of operations that update shared resources should be *atomic* [Harris et al.(2010)Harris, Larus, and Rajwar]. This would ensure that the invariant never seems to be broken.

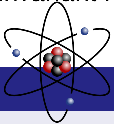
Atomic Executions

A concurrent program consists of several threads that share *resources*:

- resources can be *memory locations* or *memory mapped I/O*
  - a file can be modified through a shared handle, e.g.
- usually *invariants* must be retained wrt. resources
  - e.g. a head and tail pointer must delimit a linked list
  - an invariant may span *multiple* resources
  - during an update, the invariant may be temporarily *locally broken*

⇝ multiple resources must be updated together to ensure the invariant

Ideally, a sequence of operations that update shared resources should be *atomic* [Harris et al.(2010)Harris, Larus, and Rajwar]. This would ensure that the invariant never seems to be broken.



Definition (Atomic Execution)

A computation forms an *atomic execution* if its effect can only be *observed* as a single transformation on the memory.

Overview

We will address the *established* ways of managing synchronization. The presented techniques

- are available on most platforms
- likely to be found in most existing (concurrent) software
- provide solutions to common concurrency tasks
- are the source of common concurrency problems

The techniques are applicable to C, C++ (pthread), Java, C# and other imperative languages.

Overview

We will address the *established* ways of managing synchronization. The presented techniques

- are available on most platforms
- likely to be found in most existing (concurrent) software
- provide solutions to common concurrency tasks
- are the source of common concurrency problems

The techniques are applicable to C, C++ (pthread), Java, C# and other imperative languages.

Learning Outcomes

1. Principle of Atomic Executions
2. Wait-Free Algorithms based on Atomic Operations
3. Locks: Mutex, Semaphore, and Monitor
4. Deadlocks: Concept and Prevention

Wait-Free Atomic Executions

Wait-Free Updates

Which operations on a CPU are atomic? (`j`,`k` and `tmp` are registers)

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

Wait-Free Updates

Which operations on a CPU are atomic? (`j`,`k` and `tmp` are registers)

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

Answer:

- none by default (even without store and invalidate buffers, *why?*)

Wait-Free Updates

Which operations on a CPU are atomic? (`j`,`k` and `tmp` are registers)

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

Answer:

- none by default (even without store and invalidate buffers, *why?*)

⚠ The load and store (even `i++`'s) may be interleaved with a store from another processor.

Wait-Free Updates

Which operations on a CPU are atomic? (`j`,`k` and `tmp` are registers)

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

Answer:

- none by default (even without store and invalidate buffers, *why?*)

⚠ The load and store (even `i++`'s) may be interleaved with a store from another processor.

All of the programs *can* be made atomic executions (e.g. on x86):

- `i` must be in memory
- Idea: *lock the cache bus* for an address for the duration of an instruction

Wait-Free Updates

Which operations on a CPU are atomic? (`j`,`k` and `tmp` are registers)

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

Answer:

- none by default (even without store and invalidate buffers, *why?*)

⚠ The load and store (even `i++`’s) may be interleaved with a store from another processor.

All of the programs *can* be made atomic executions (e.g. on x86):

- `i` must be in memory
- Idea: *lock the cache bus* for an address for the duration of an instruction

Program 1

```
lock inc [addr_i]
```

Program 2 (fetch-and-add)

```
mov eax,reg_k
lock xadd [addr_i],eax
mov reg_j,eax
```

Program 3 (atomic-exchange)

```
lock xchg [addr_i],reg_j
```

Wait-Free Bumper-Pointer Allocation

Garbage collectors often use a *bumper pointer* to allocated memory:

Bumper Pointer Allocation

```
char heap[1<<20];
char* firstFree = &heap[0];

char* alloc(int size) {
  char* start = firstFree;
  firstFree = firstFree + size;

  if (start+size>sizeof(heap)) garbage_collect();
  return start;
}
```

- `firstFree` points to the first unused byte
- each allocation reserves the next `size` bytes in `heap`

Wait-Free Bumper-Pointer Allocation

Garbage collectors often use a *bumper pointer* to allocated memory:

Bumper Pointer Allocation

```
char heap[1<<20];
char* firstFree = &heap[0];

char* alloc(int size) {
  char* start;
  asm("lock; xadd %0, %1" :"=r"(start),"=m"(firstFree):
       "0"(size),"m"(firstFree) :"memory");
  if (start+size>sizeof(heap)) garbage_collect();
  return start;
}
```

- `firstFree` points to the first unused byte
- each allocation reserves the next `size` bytes in `heap`

Thread-safe implementation:

- `alloc`'s core functionality matches Program 2: fetch-and-add

⇝ inline assembler (GCC/AT&T syntax in the example)

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++;
}
```

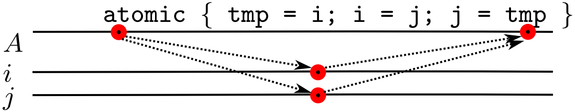
Program 2

```
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++;
}
```

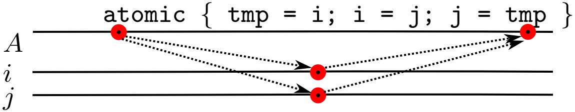
Program 2

```
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
- often, there are instructions 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 $v \in \{0, 1\}$ and returning its previous state.
  - the operation implementing programs 4 and 5 is called *set-and-test*
- the third case generalizes this to setting a variable `i` to the value of `j`, if `i`'s old value is equal to `k`'s.
  - the operation implementing program 6 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
- often, there are instructions 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 $v \in \{0, 1\}$ and returning its previous state.
  - the operation implementing programs 4 and 5 is called *set-and-test*
- the third case generalizes this to setting a variable `i` to the value of `j`, if `i`'s old value is equal to `k`'s.
  - the operation implementing program 6 is called *compare-and-swap*

⇝ use as *building blocks* for algorithms that can *fail*

Lock-Free Algorithms

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. compute 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. compute 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. compute 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
- compute 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. compute 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
- compute a new value
- perform a compare-and-swap operation on these n bytes

$\rightsquigarrow$ computing new value must be *repeatable* or *pure*

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

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

⇝ Lock-Free instructions as *building blocks* for *Locks*

Locked Atomic Executions

Locks

Definition (Lock)

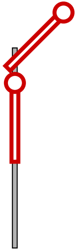
A lock is a data structure that

- can be *acquired* and *released*
- ensures *mutual exclusion*: only one thread may hold the lock at a time
- *blocks* other threads attempts to acquire while held by a different thread
- protects a *critical section*: a piece of code that may produce incorrect results when entered concurrently from several threads

⚠ may *deadlock* the program

Semaphores and Mutexes

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

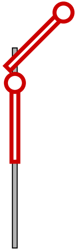


```
void signal(int *s) {
  atomic { *s = *s + 1; }
}
```

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

Semaphores and Mutexes

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



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

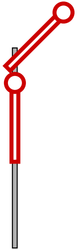
```
void wait(int *s) {
  bool avail;
  do {
    atomic {
      avail = *s>0;
      if (avail) (*s)--;
    }
  } while (!avail);
}
```

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

- a thread *acquiring* a resource executes `wait()`
- if a resource is still available, `wait()` returns
- once a thread finishes using a resource, it calls `signal()` to *release*

Semaphores and Mutexes

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



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

```
void wait(int *s) {
  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 *acquiring* a resource executes `wait()`
- if a resource is still available, `wait()` returns
- once a thread finishes using a resource, it calls `signal()` to *release*

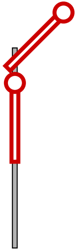
Special case: initializing with $s = 1$ gives a *binary* semaphore:

- can be used to block and unblock a thread
- can be used to protect a single resource

⇝ in this case the data structure is also called *mutex*

Implementation of Semaphores

A *semaphore* does not have to wait busily:

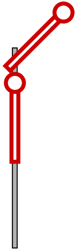


```
void signal(int *s) {
  atomic { *s = *s + 1; }
  wake(s);
}
```

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

Implementation of Semaphores

A *semaphore* does not have to wait busily:



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

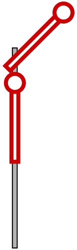
```
void wait(int *s) {
  bool avail;
  do {
    atomic {
      avail = *s>0;
      if (avail) (*s)--;
    }
    if (!avail) de_schedule(s);
  } while (!avail);
}
```

Busy waiting is avoided:

- a thread failing to decrease `*s` executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when `*s` becomes non-zero, usually by *monitoring writes* to `s` (⇝ *FUTEX_WAIT*)
- once a thread calls `wake(s)`, 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(int *s) {
  atomic { *s = *s + 1; }
  wake(s);
}
```

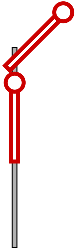
```
void wait(int *s) {
  bool avail;
  do {
    atomic {
      avail = *s>0;
      if (avail) (*s)--;
    }
    if (!avail) de_schedule(s);
  } while (!avail);
}
```

In general, the implementation is more complicated

- `wait()` may busy wait for a few iterations
  - avoids de-scheduling if the lock is released frequently
  - better throughput for semaphores that are held for a short time
- `wake(s)` informs the scheduler that `s` has been written to

Practical Implementation of Semaphores

Certain optimisations are possible:



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

```
void wait(int *s) {
  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
  - avoids de-scheduling if the lock is released frequently
  - better throughput for semaphores that are held for a short time
- `wake(s)` informs the scheduler that `s` has been written to

⇝ using a semaphore with a single core reduces to

```
if (*s) (*s)--; /* critical section */ (*s)++;
```

Mutexes

One common use of semaphores is to guarantee mutual exclusion.

⇝ in this case, a binary semaphore is also called a *mutex*

e.g. add a lock to the double-ended queue data structure

⚠ decide what needs protection and what not

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

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

E.g. a thread t waits for a data structure to be filled

- t will call `pop()` and obtain `-1`
- t then has to call again, until an element is available

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

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

E.g. a thread t waits for a data structure to be filled

- ► t will call `pop()` and obtain `-1`
- ► t then has to call again, until an element is available

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

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 by the current thread, proceed

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

E.g. a thread t waits for a data structure to be filled

- t will call `pop()` and obtain `-1`
- t then has to call again, until an element is available

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

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 by the current thread, proceed

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

Implementation of a Basic Monitor

A monitor contains a semaphore `count` and the id `tid` of the occupying thread:

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

Implementation of a Basic Monitor

A monitor contains a semaphore `count` and the id `tid` of the occupying thread:

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

```
void monitor_leave(mon_t *m) {
  m->count--;
  if (m->count==0) {
    atomic {
      m->tid=0;
    }
    wake(&m->tid);
  }
}
```

Condition Variables

✓ Monitors simplify the construction of thread-safe resources.

Still: Efficiency problem when using resource to synchronize:

E.g. a thread t waits for a data structure to be filled:

- t will call `pop()` and obtain `-1`
- t then has to call again, until an element is available

$\rightsquigarrow$ 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:

E.g. a thread t waits for a data structure to be filled:

- t will call `pop()` and obtain `-1`
- t then has to call again, until an element is available

⇝ t is busy waiting and produces contention on the lock

Idea: create a *condition variable* on which to block while waiting:

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

Condition Variables

✓ Monitors simplify the construction of thread-safe resources.

Still: Efficiency problem when using resource to synchronize:

E.g. a thread t waits for a data structure to be filled:

- t will call `pop()` and obtain `-1`
- t then has to call again, until an element is available

⇝ t is busy waiting and produces contention on the lock

Idea: create a *condition variable* on which to block while waiting:

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

Define these two functions:

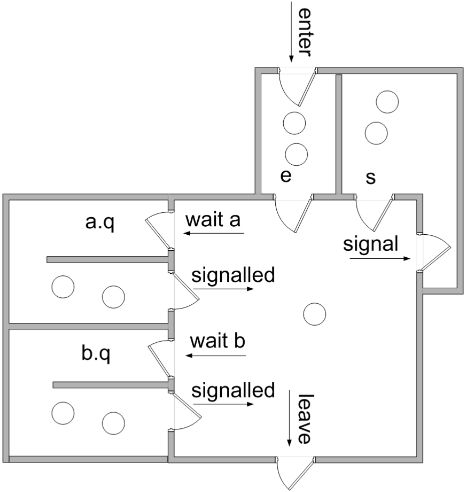
1. `wait` for the condition to become true
   - called while being *inside* the monitor
   - temporarily *releases* the monitor and blocks
   - when *signalled*, re-acquires the monitor and returns
2. `signal` waiting threads that they may be able to proceed
   - one/all waiting threads that called *wait* 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 queue for each condition c and a suspended queue s:

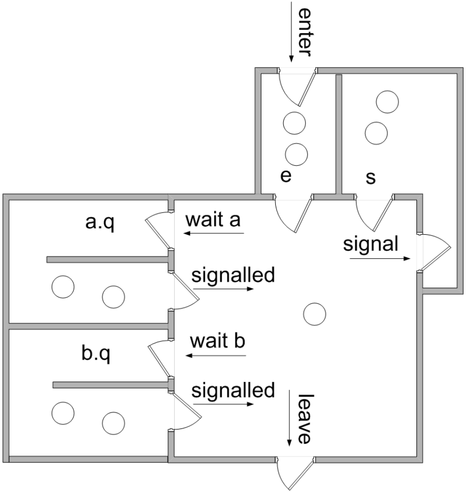


source: `http://en.wikipedia.org/wiki/Monitor_(synchronization)`

- 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 from 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 queue for each condition c and a suspended queue s:



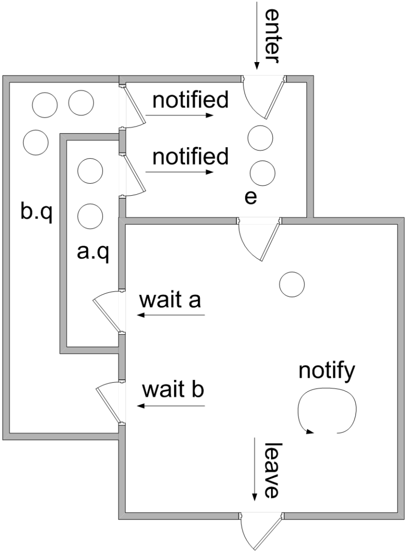
source: `http://en.wikipedia.org/wiki/Monitor_(synchronization)`

- 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 from 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

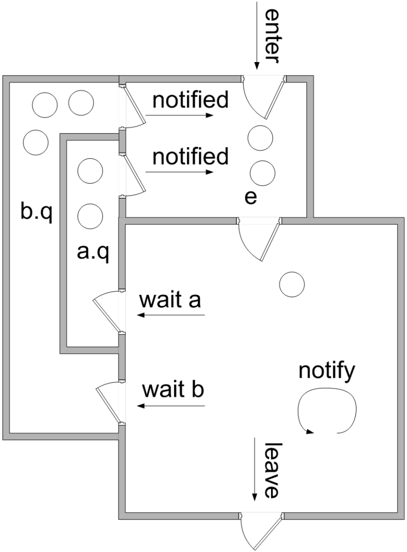
Here, the `signal` function is usually called `notify`.



- 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

- 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

Implementing Condition Variables

We implement the simpler *signal-and-continue* semantics for a single condition variable:

⇝ 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:

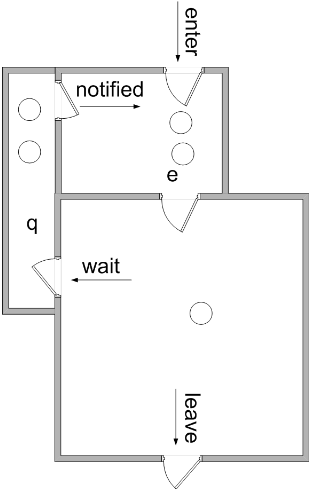
1. `notify`: wakes up exactly one thread waiting on condition variable
2. `notifyAll`: wakes up all threads waiting on a condition variable

⚠ an implementation often becomes easier if `notify` means *notify some*

⇝ programmer should assume that thread is not the only one woken up

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(this);
    // body of f
    monitor_leave(this);
  }}
```

with `Object` containing:

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

Deadlocks

Deadlocks with Monitors

Definition (Deadlock)

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

Deadlocks with Monitors

Definition (Deadlock)

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

Consider this Java class:

```
class Foo {
  public Foo other = null;
  public synchronized void bar() {
    ... if (*) other.bar(); ...
  }
}
```

and two instances:

```
  Foo a = new 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

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

Sequence leading to a deadlock:

- threads A and B execute `a.bar()` and `b.bar()`
- `a.bar()` acquires the monitor of `a`
- `b.bar()` acquires the monitor of `b`
- A happens to execute `other.bar()`
- A blocks on the monitor of b
- B happens to execute `other.bar()`
- ⇝ both *block* indefinitely

How can this situation be avoided?

Treatment of Deadlocks

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

Treatment of Deadlocks

Observation: 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

Observation: 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 achieved using *lock-free* algorithms
- but what about algorithms that require locking?

Deadlock Prevention through Partial Order

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

Definition (lock sets)

Let L denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at p, i.e. 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 are *partially ordered*.

Definition (lock sets)

Let L denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at p, i.e. 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

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

Deadlock Prevention through Partial Order

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

Definition (lock sets)

Let L denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at p, i.e. 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

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

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

Definition (lock order)

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

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/monitor $\bar{a} \in L_M$ represents several concrete ones
  - thus, if $\bar{a} \prec \bar{a}$ then this might not be a self-cycle

⇝ require that $\bar{a} \not\prec \bar{a}$ for all summarized monitors $\bar{a} \in L_M$

Inferring locksets and lockset order in practice

⚠ fix a representation for locksets

⇝ in our case: L comprises all lines, where any object is created.

```
0:  Foo  a = new Foo();
1:  Foo  b = new Foo();
2:  a.other = b;
3:  b.other = a;
4:
5:
6:  bar(&a); || bar(&b);
7:
```

```
 8:  void bar(this) {
 9:    monitor_enter(this);
10:    if (*) {
11:        ...
12:        bar(&other);
13:        ...
14:    }
15:    monitor_leave(this);
16:  }
```

Lockorder | ◁ |

Inferring locksets and lockset order in practice

⚠ fix a representation for locksets

⇝ in our case: L comprises all lines, where any object is created.

```
0:  Foo  a = new Foo();
1:  Foo  b = new Foo();
2:  a.other = b;
3:  b.other = a;
4:
5:
6:  bar(&a); || bar(&b);
7:
```

```
8:  void bar(this) {
9:    monitor_enter(this);
10:   if (*) {
11:       ...
12:       bar(&other);
13:       ...
14:   }
15:   monitor_leave(this);
16: }
```

$\lambda(8) = \{\}$

Inferring locksets and lockset order in practice

⚠ fix a representation for locksets

⇝ in our case: L comprises all lines, where any object is created.

this $= \{\&a, \&b\}$

```
0:   Foo  a = new Foo();
1:   Foo  b = new Foo();
2:   a.other = b;
3:   b.other = a;
4:
5:
6:   bar(&a); || bar(&b);
7:
```

$\lambda(9) = \{l_0, l_1\}$

```
8:   void bar(this) {
9:     monitor_enter(this);
0:     if (*) {
11:          ...
12:          bar(&other);
13:          ...
14:    }
15:    monitor_leave(this);
16:  }
```

Lockorder ◁

Inferring locksets and lockset order in practice

⚠ fix a representation for locksets

⇝ in our case: L comprises all lines, where any object is created.

$\texttt{this} = \{\&a, \&b\}$

```
 0:  Foo  a = new Foo();
 1:  Foo  b = new Foo();
 2:  a.other = b;
 3:  b.other = a;
 4:
 5:
 6:  bar(&a); || bar(&b);
 7:
```

$\lambda(11) = \{\mathtt{l}_0, \mathtt{l}_1\}$

```
 8:  void bar(this) {
 9:    monitor_enter(this);
  :    if (*) {
11:        ...
12:        bar(&other);
13:        ...
14:    }
15:    monitor_leave(this);
16:  }
```

Inferring locksets and lockset order in practice

⚠ fix a representation for locksets

⇝ in our case: L comprises all lines, where any object is created.

$\texttt{this} = \{\&a, \&b\}$

```
0:  Foo  a = new Foo();
1:  Foo  b = new Foo();
2:  a.other = b;
3:  b.other = a;
4:
5:
6:  bar(&a); || bar(&b);
7:
```

$\lambda(11) = \{\texttt{l}_0, \texttt{l}_1\}$

```
 8: void bar(this) {
 9:   monitor_enter(this);
  :   if (*) {
11:     ...
12:     bar(&other);
13:     ...
14:   }
15:   mon
16: }
```

$\texttt{other} = \{\&a, \&b\}$

Lockorder ◁

Inferring locksets and lockset order in practice

⚠ fix a representation for locksets

⇝ in our case: L comprises all lines, where any object is created.

```
 0:  Foo  a = new Foo();
 1:  Foo  b = new Foo();
 2:  a.other = b;
 3:  b.other = a;
 4:
 5:
 6:  bar(&a); || bar(&b);
 7:
```

```
 8:  void bar(this) {
       monitor_enter(this);
 0:    if (*) {
11:        ...
12:        bar(&other);
13:        ...
14:    }
15:    mon
16:  }
```

$\texttt{this} = \{\&a, \&b\}$

$\lambda(8) = \{\texttt{l}_0, \texttt{l}_1\}$

$\texttt{other} = \{\&a, \&b\}$

Lockorder	◁	

Inferring locksets and lockset order in practice

⚠ fix a representation for locksets

⇝ in our case: L comprises all lines, where any object is created.

```
0:   Foo  a = new Foo();
1:   Foo  b = new Foo();
2:   a.other = b;
3:   b.other = a;
4:
5:
6:   bar(&a); || bar(&b);
7:
```

```
8:   void bar(this) {
9:     monitor_enter(this);
0:     if (∗) {
11:        ...
12:        bar(&other);
13:        ...
14:    }
15:    mon
16:  }
```

this $= \{\&a, \&b\}$

$\lambda(9) = \{l_0, l_1\}$

other $= \{\&a, \&b\}$

Lockorder	$\lhd$	$\langle l_0, l_1\rangle, \langle l_1, l_0\rangle$

Avoiding Deadlocks in Practice

⚠ What to do when the lock order contains a cycle?

- determining which locks may be acquired at each program point is undecidable
  $\rightsquigarrow$ 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
  $\rightsquigarrow$ change program: release l, acquire l', then acquire l again
  ⚠ inefficient
- if a lock set contains a summarized lock $\bar{a}$ and $\bar{a}$ is to be acquired, we're stuck

Locks Roundup

Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

stack: removal

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

Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

stack: removal

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

- nested `atomic` blocks still describe one atomic execution

⇝ locks convey additional information over `atomic`

- locks cannot easily be recovered from `atomic` declarations

Outlook

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

Outlook

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

Idea of mutexes: Implement `atomic` sections with locks:

- a single lock could be used to protect all `atomic` blocks
- more concurrency is possible by using several locks
- some statements might modify variables that are never read by other threads $\leadsto$ no lock required
- statements in one `atomic` block might access variables in a different order to another `atomic` block $\leadsto$ 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

Outlook

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

Idea of mutexes: Implement `atomic` sections with locks:

- a single lock could be used to protect all `atomic` blocks
- more concurrency is possible by using several locks
- some statements might modify variables that are never read by other threads $\leadsto$ no lock required
- statements in one `atomic` block might access variables in a different order to another `atomic` block $\leadsto$ 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

$\leadsto$ creating locks automatically is non-trivial and, thus, not standard in programming 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

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)	(c)	✓	✓

(a) some pthread implementations allow a *reentrant* attribute

(b) newer API extensions (`java.util.concurrent.atomic.*` and `System.Threading.Interlocked` resp.)

(c) simulate semaphores using an object with two `synchronized` methods

Summary

Classification of concurrency algorithms:

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

Wait-free algorithms:

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

Lock-free algorithms:

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

Locking algorithms:

- can guard arbitrary code
- can use several locks to enable more fine grained concurrency
- may deadlock
- semaphores are not re-entrant, monitors are

⇝ use algorithm that is best fit

References

- E. G. Coffman, M. Elphick, and A. Shoshani.
  System deadlocks.
  ACM Comput. Surv., 3(2):67–78, June 1971.
  ISSN 0360-0300.
- T. Harris, J. Larus, and R. Rajwar.
  Transactional memory, 2nd edition.
  Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.