Script **generated by TTT**

Title: Petter: Programmiersprachenh (07.11.2018)

Date: Wed Nov 07 14:08:44 CET 2018

Duration: 89:08 min

Pages: 31

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Atomic Executions, Locks and Monitors

Dr. Michael Petter
Winter 2018

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

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

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 Bumper-Pointer Allocation

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

Bumper Pointer Allocation

```
char heap[2^20];
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`

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;
}
```


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.
  - this operation 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.
  - this operation is called *compare-and-swap*

Lock-Free Algorithms

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.
  - this operation 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.
  - this operation 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.

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

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

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);
}
```


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);
}
```


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

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

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) {
    // wake up threads
    atomic {
      m->tid=0;
  } }
}
```


Condition Variables

✓ Monitors simplify the construction of thread-safe resources.

Still: Efficiency problem when using resource to synchronize:

- if a thread t waits for a data structure to be filled:
  - t will call e.g. `pop()` and obtain `-1`
  - t then has to call again, until an element is available

⚠ t is busy waiting and produces contention on the lock

Condition Variables

✓ Monitors simplify the construction of thread-safe resources.

Still: Efficiency problem when using resource to synchronize:

- if a thread t waits for a data structure to be filled:
  - t will call e.g. `pop()` and obtain `-1`
  - t then has to call again, until an element is available
  - ⚠ t is busy waiting and produces contention on the lock

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

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

Define these two functions:

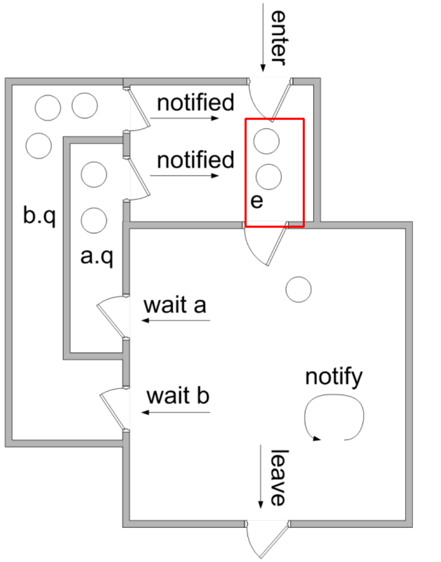
1. `wait` for the condition to become true
   - called while being *inside* the monitor
   - temporarily *releases* the monitor and blocks
   - when *signalled*, re-acquires the monitor and returns
2. `signal` waiting threads that they may be able to proceed
   - one/all waiting threads that called *wait*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

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

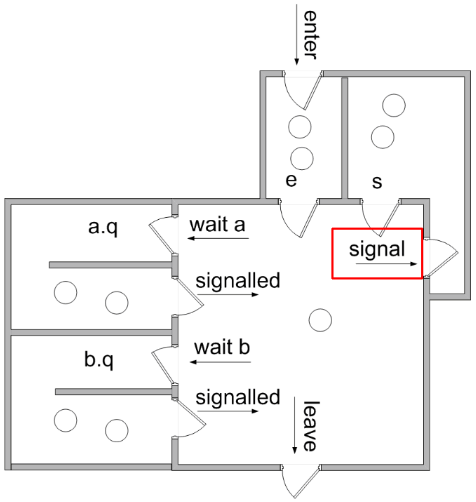


- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `notify` for a adds one thread from $a.q$ to e (unless $a.q$ is empty)
- if a thread leaves, it wakes up one thread waiting on e

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

Signal-And-Urgent-Wait Semantics

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



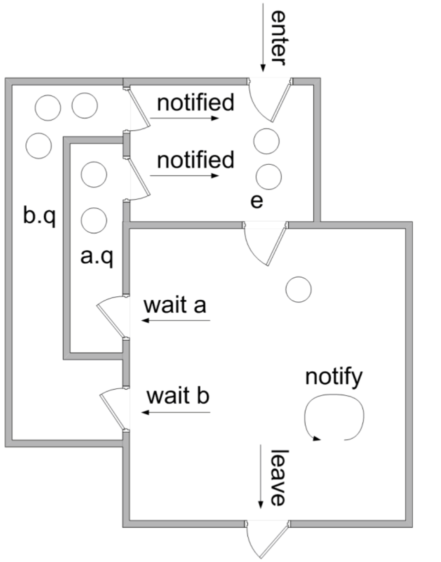
source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

- 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 form the a queue is woken up
- `signal` on a is a no-op if $a.q$ is empty
- if a thread leaves, it wakes up one thread waiting on s
- if s is empty, it wakes up one thread from e

⇝ queue s has priority over e

Signal-And-Continue Semantics

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



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

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

Implementing Condition Variables

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

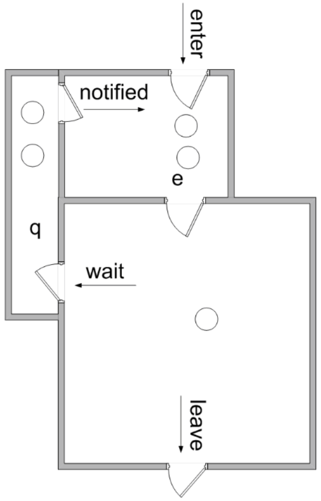
⇝ a *notified* thread is simply woken up and competes for the monitor

```
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);
}
```


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

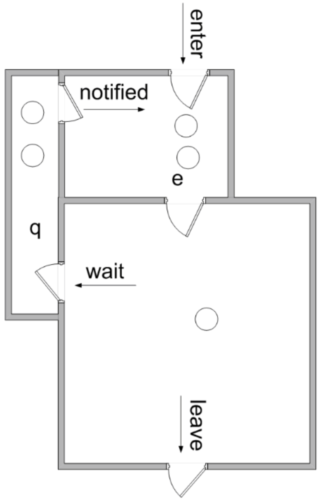
```
class C {
  public void f() {
    monitor_enter(this);
    // body of f
    monitor_leave(this);
  }}
```

with `Object` containing:

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


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   this.wait();
    monitor_leave(this);
  }}
```

this.notify

with `Object` containing:

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


Deadlocks