Course 02158

Concurrent Data Structures

Hans Henrik Løvengreen

DTU Compute

The Realm of Shared Variables

- Originates in multiprogramming on mono/multi-processors

Examples

- Threads sharing a common address space
- Processes sharing a common file (data base).
- Distributed objects

Observations

- Execution seen as interleaving of atomic actions
- Atomicity of anything but read/writes must be implemented

Selection of Granularity

- Atomic actions must be large to ensure *semantic consistency*
- Atomic actions should be small to allow for parallelism
- Starting point:

  Operations on data types should be atomic

Concurrent Objects

- A *concurrent object* is a shared data structure with atomic operations
- *monitor* = concurrent object with atomicity by locking
- Monitors should always be the first choice
- Sometimes monitors provide too little synchronization
- Sometimes monitors provide too much synchronization

Concurrent Data Types and Data Structures

Basic Notions

- A *data type* is a set of (abstract) *values* with *operations* on these
- Typically given by a *signature* (interface) + *operation specifications*
- A *data structure* is a concrete implementation of a data type
- The operations may be implemented with particular *qualities*. eg. *complexity*
- The implementation may be given as a *class*/*object*
- Example: Java `List` implemented by `ArrayList` og `LinkedList`

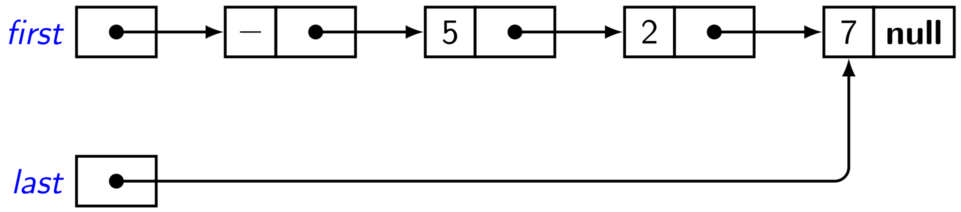
Concurrent Data Types

- A *concurrent data type* also specifies concurrent operation applications
- Operations may simply be prescribed to be *atomic* (=*thread-safe*?)
- A *concurrent data structure* implements a concurrent data type

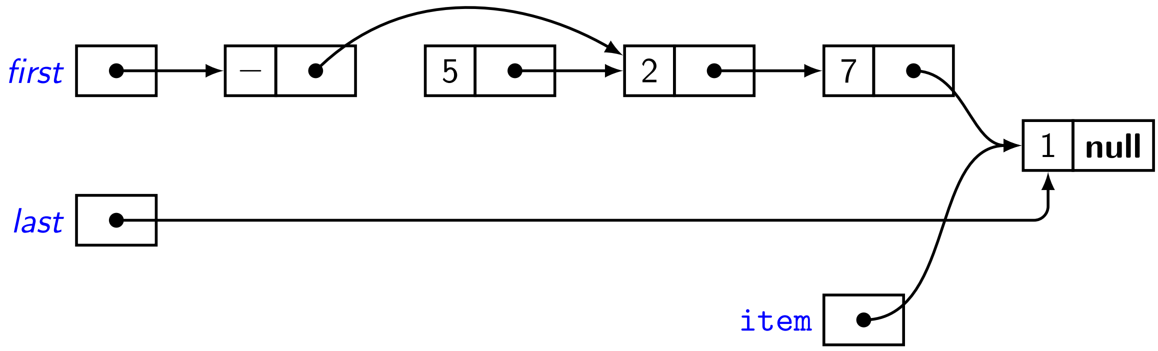
Monitor Issues

- Deadlocks
  - Recursive locks
  - Locking hierarchy
  - Deadlock detection/retry
  - Combining monitors
- Locking of whole structure — degrades concurrency
  - Apply *R/W locking* of operations
  - *Narrow* the locking — exclude data pre and post processing
  - Use *individual locking* of parts (aka *fine grained locking*)
- Synchronization overhead
  - Prevent sharing by outer critical region or single thread server (GUI)
  - Use *non-blocking synchronization* (at low or high level)
  - Use *spin locks* (for short critical sections on multiprocessors)

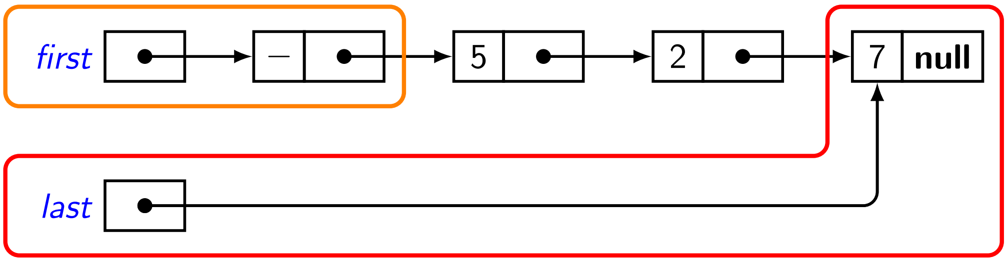
Linked List



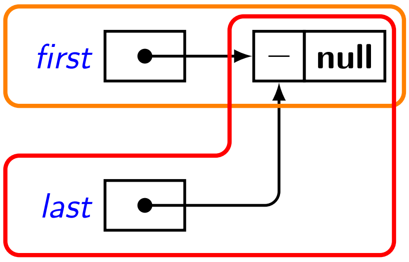
Linked List



Linked List with Concurrent Operations



Linked List with Concurrent Operations — empty list



Buffer with concurrent operations (in Java)

```java
class Elem { public int val; public Elem next; }

class ConcurrentBuffer {
  Elem first = new Elem(); // Sentinel object
  Elem last = first;
  Object fReg = new Object();
  Object lReg = new Object();

  void put(int v) {
    e = new Elem();
    e.val = v;
    synchronized (lReg) {
      last.next = e;
      last = e;
      lReg.notify();
    }
  }

  int get() throws InterruptedException {
    int res;
    synchronized (fReg) {
      if (first.next==null)
        synchronized (lReg) {
          while (first.next==null) lReg.wait();
        }
      res = first.next.val;
      first.next = first.next.next;
    }
    return res;
  }

}
```

Non-Blocking Synchronization

- Atomic operations avoiding cascaded delays ("blocking") by not using locks
- Operations do not wait, but may *fail*
- Depends on atomic *read-check-modify* instructions

Example with Compare-and-Swap

- $\texttt{CAS}(\textbf{var}\ \texttt{x}, \texttt{old}, \texttt{new}) = \langle\, \textbf{var}\ \texttt{r} := \texttt{x};\ \textbf{if}\ \texttt{x} = \texttt{old}\ \textbf{then}\ \texttt{x} := \texttt{new};\ \textbf{return}\ \texttt{r}\,\rangle$
- Atomic increment $\langle\, x := x + 1\,\rangle$:

```
repeat
  t := x;
until CAS(x, t, t + 1) = t;
```

- Can be appplied to any update function on x
- Can be extended to larger data types by updating pointers to structures.

Properties of Non-Blocking operations

- Assume a concurrent object with non-blocking operations

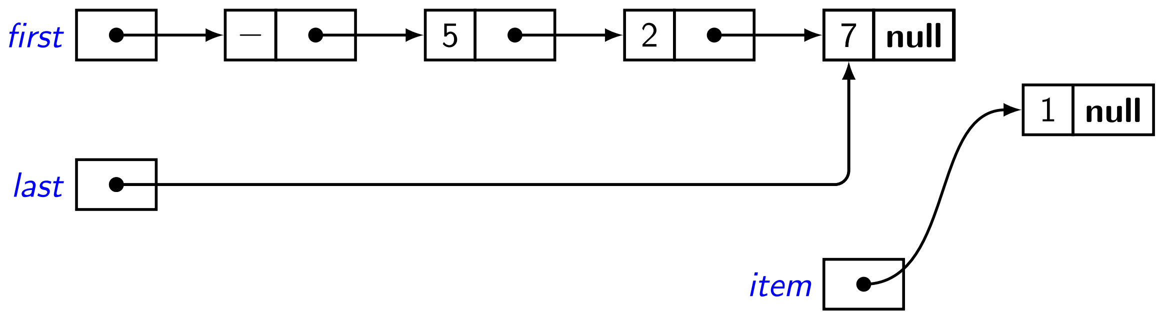
Correctness (safety)

- Each operation must fulfil its *sequential specification* when executed alone
- Any overlapping execution of operations must be *linearizable*

Progress notions (liveness)

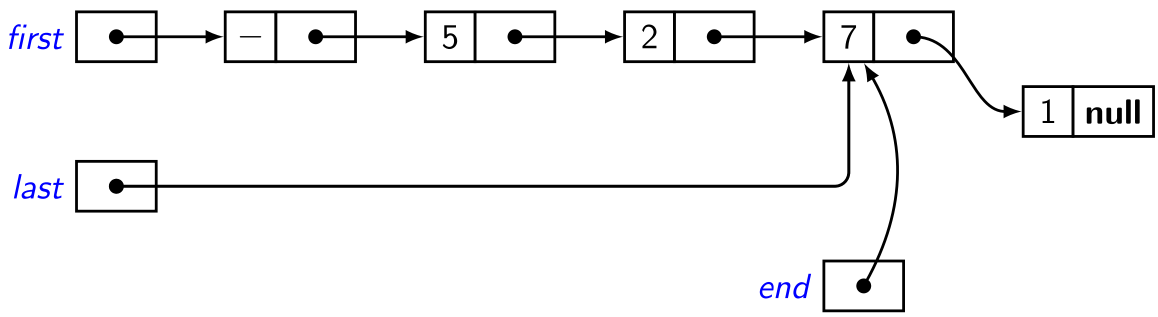
- **Obstruction-free** Each operation will succeed when executed *in isolation*
- **Lock-free** Some active operation will succeed
- **Wait-free** Any operation will succeed

Linked List — non-blocking



CAS(*last*.*next*,**null**, *item*)

Linked List — non-blocking (cleanup)



end := *last*
if *end*.*next* ≠**null then**
  CAS(*last*, *end*, *end*.*next*)

Atomic Scalar Types in Java

- Provide simple atomic operations on `Integer`, `Boolean`, `Long`, and `Reference` types and arrays of these
- Operations include: `set`, `get`, `add/sub` and combinations of these
- Based upon *read-check-modify* primitive

Example: AtomicInteger

- Selected operations on AtomicInteger `x` with value v

`x.addAndGet(`d`)`	$\langle\, v := v + d;\ \ \mathbf{return}\ v\,\rangle$
`x.getAndIncrement()`	$\langle\, \mathbf{var}\ old = v;\ \ v := v + 1\ \ \mathbf{return}\ old\,\rangle$
`x.compareAndSet(`old`,`new`)`	$\langle\, \mathbf{if}\ v\ =\ old\ \mathbf{then}\ v\ :=\ new;\ \mathbf{return}\ true\ \mathbf{else}\ \mathbf{return}\ false\,\rangle$

Thread-safe Data Structures in Java

- Standard Collection classes are **not** thread-safe
- A number of thread-safe substitutions are provided
- Wrapping into a synchronized context Collections.synchronized*XXX*
- Some efficiently implemented by *lock-free synchronization*

Classes

- Atomic operations, non-blocking:
  ConcurrentHashMap, ConcurrentLinkedQueue
- Atomic operations, potentially blocking:
  ArrayBlockingQueue (N-buffer)
  LinkedBlockingQueue (Unbounded buffer)
  SynchronousQueue (CSP-channel)
  PriorityBlockingQueue
  DelayQueue

Playground Field I (erroneous)

- ```
  public class Field {

      Map<Pos,Semaphore> tiles = new HashMap<Pos,Semaphore>();


      public void enter(int no, Pos pos) throws InterruptedException {

        if (tiles.get(pos)==null) tiles.put(pos, new Semaphore(1));

        tiles.get(pos).P();

      }

      public void leave(Pos pos) {

        tiles.get(pos).V();

      }
  }
  ```

Playground Field II (erroneous)

- ```
  public class Field {

      Map<Pos,Semaphore> tiles = new HashMap<Pos,Semaphore>();
      Semaphore mutex = new Semaphore(1);

      public void enter(int no, Pos pos) throws InterruptedException {
        mutex.P();
        if (tiles.get(pos)==null) tiles.put(pos, new Semaphore(1));
        mutex.V();
        tiles.get(pos).P();

      }

      public void leave(Pos pos) {

        tiles.get(pos).V();

      }
  }
  ```

Playground Field III (deadlocks)

- ```
  public class Field {

      Map<Pos,Semaphore> tiles = new HashMap<Pos,Semaphore>();
      Semaphore mutex = new Semaphore(1);

      public void enter(int no, Pos pos) throws InterruptedException {
        mutex.P();
        if (tiles.get(pos)==null) tiles.put(pos, new Semaphore(1));

        tiles.get(pos).P()
        mutex.V();
      }

      public void leave(Pos pos) {
        mutex.P();
        tiles.get(pos).V()
        mutex.V();
      }
  }
  ```

Playground Field IV (correct)

- ```
  public class Field {

      Map<Pos,Semaphore> tiles = new HashMap<Pos,Semaphore>();
      Semaphore mutex = new Semaphore(1);

      public void enter(int no, Pos pos) throws InterruptedException {
        mutex.P():
        if (tiles.get(pos)==null) tiles.put(pos, new Semaphore(1));
        Semaphore sem = tiles.get(pos);
        mutex.V();
        sem.P();
      }

      public void leave(Pos pos) {
        mutex.P();
        tiles.get(pos).V();
        mutex.V();
      }
  }
  ```

Playground Field — using thread-safe map I (erroneous)

- ```
  public class Field {
      Map<Pos,Semaphore> tiles=Collections.synchronizedMap(new HashMap<>());

      public void enter(int no, Pos pos) throws InterruptedException {
        if (tiles.get(pos)==null) tiles.put(pos, new Semaphore(1));
        tiles.get(pos).P();
      }
      public void leave(Pos pos) {
        tiles.get(pos).V();
      }
  }
  ```

Playground Field — using thread-safe map II (ineffecient)

- ```
  public class Field {
      Map<Pos,Semaphore> tiles=Collections.synchronizedMap(new HashMap<>());

      public void enter(int no, Pos pos) throws InterruptedException {
        tiles.putIfAbsent(pos, new Semaphore(1));
        tiles.get(pos).P();
      }
      public void leave(Pos pos) {
        tiles.get(pos).V();
      }
  }
  ```

Playground Field — using thread-safe map III (correct)

- ```
  public class Field {
      Map<Pos,Semaphore> tiles=Collections.synchronizedMap(new HashMap<>());

      public void enter(int no, Pos pos) throws InterruptedException {
        if (tiles.get(pos)==null) tiles.putIfAbsent(pos, new Semaphore(1));
        tiles.get(pos).P();
      }
      public void leave(Pos pos) {
        tiles.get(pos).V();
      }
  }
  ```

Playground Field — using thread-safe map IV (efficient)

- ```
  public class Field {
      Map<Pos,Semaphore> tiles = new ConcurrentHashMap<Pos, Semaphore>();

      public void enter(int no, Pos pos) throws InterruptedException {
        if (tiles.get(pos)==null) tiles.putIfAbsent(pos, new Semaphore(1));
        tiles.get(pos).P();
      }
      public void leave(Pos pos) {
        tiles.get(pos).V();
      }
  }
  ```

Playground Field — using thread-safe map V

- ```java
  public class Field {
      Map<Pos,Semaphore> tiles = new ConcurrentHashMap<Pos, Semaphore>();

      public void enter(int no, Pos pos) throws InterruptedException {
        tiles.computeIfAbsent(pos, (p) -> new Semaphore(1));
        tiles.get(pos).P();
      }
      public void leave(Pos pos) {
        tiles.get(pos).V();
      }
  }
  ```

Spin Locks

- Busy waiting (*spinning*) is injurious on mono-processors!
- On *multi-processors*, spinning may be faster than context switch
- If process holding lock is descheduled, spinning wastes CPU time
- OS may internally use spin locks which prevent descheduling
- Locks with *bounded spin* may be provided to user programs
- Example: *Futex* locks in Linux
  ```

Critical Regions with Shared Variables

Test-and-Set Solution

- Let $\mathtt{TS}(\textbf{var}\ \mathtt{X}) = \langle \textbf{var}\ \mathtt{r};\ \ (\mathtt{r}, \mathtt{X}) := (\mathtt{X}, 1);\ \ \textbf{return}\ \mathtt{r} \rangle$
- $\textbf{var}\ \mathtt{lock} : \mathtt{int} := 0;$

process P_1
  loop
    while $\mathtt{TS}(\mathtt{lock}) = 1$ **do skip**;
    critical section$_1$;
    $\mathtt{lock} := 0$;
    noncritical section$_1$
  end loop

process P_2
  loop
    while $\mathtt{TS}(\mathtt{lock}) = 1$ **do skip**;
    critical section$_2$;
    $\mathtt{lock} := 0$;
    noncritical section$_2$
  end loop

Critical Regions with Shared Variables

Non-spinning Test-and-Set Solution

- Let $\mathtt{TS}(\textbf{var}\ \mathtt{X}) = \langle \textbf{var}\ \mathtt{r};\ \ (\mathtt{r}, \mathtt{X}) := (\mathtt{X}, 1);\ \ \textbf{return}\ \mathtt{r} \rangle$
- $\textbf{var}\ \mathtt{lock} : \mathtt{int} := 0;\ \ \mathtt{s} : \mathtt{semaphore} := 0;$

process P_1
  loop
    while $\mathtt{TS}(\mathtt{lock}) = 1$ **do** $\mathtt{P}(\mathtt{s})$;
    critical section$_1$;
    $\mathtt{lock} := 0$;
    $\mathtt{V}(\mathtt{s})$
    noncritical section$_1$
  end loop

process P_2
  loop
    while $\mathtt{TS}(\mathtt{lock}) = 1$ **do** $\mathtt{P}(\mathtt{s})$;
    critical section$_2$;
    $\mathtt{lock} := 0$;
    $\mathtt{V}(\mathtt{s})$;
    noncritical section$_2$
  end loop

- Uses at least one semaphore operation for each entry/exit
- Semaphore should be binary (to avoid piling up)

Critical Regions with Shared Variables

Non-spinning Test-and-Set Attempt

- Let $\mathtt{TS}(\textbf{var}\ \mathtt{X}) = \langle\, \textbf{var}\ \mathtt{r};\ \ (\mathtt{r}, \mathtt{X}) := (\mathtt{X}, 1);\ \ \textbf{return}\ \mathtt{r} \,\rangle$
- $\textbf{var}\ \mathtt{lock} : \mathtt{int} := 0;\ \ \mathtt{s} : \mathtt{semaphore} := 0;\ \ \mathtt{waiting} : \mathtt{bool} := \mathtt{false};$

```
process P1
  loop
    while TS(lock) = 1 do
      {waiting := true; P(s)};
    critical section1;
    lock := 0;
    if waiting then
      {waiting := false; V(s)};
    noncritical section1
  end loop
```

```
process P2
  loop
    while TS(lock) = 1 do
      {waiting := true; P(s)};
    critical section2;
    lock := 0;
    if waiting then
      {waiting := false; V(s)};
    noncritical section2
  end loop
```

- Suffers from *race conditions*

Futex (Fast user-space mutual exclusion)

Idea

- Combine user-space atomic operations with OS-based blocking
- Atomic test and wait using *read-check-modify* principle

Data Type

- $\textbf{type}\ \mathtt{futex} = \ \textbf{struct}\ \{\ \underbrace{\mathtt{val} : \mathtt{int};}_{\text{userspace}}\ \underbrace{\mathtt{queue} : \mathtt{condition\_queue}}_{\text{kernelspace}}\ \}$
- Fields related by hash-table based *mapping*

Kernel operations

- $\mathtt{wait\_if\_eq}(\textbf{var}\ \mathtt{f} : \mathtt{futex}, \mathtt{old} : \mathtt{int}) \triangleq \langle\, \textbf{if}\ \mathtt{f.val} = \mathtt{old}\ \textbf{then}\ \mathtt{wait}(\mathtt{f.queue}) \,\rangle$
- $\mathtt{wake}(\textbf{var}\ \mathtt{f} : \mathtt{futex}, \mathtt{n} : \mathtt{int}) \triangleq \langle\, \textbf{for}\ 1..\mathtt{n}\ \textbf{do}\ \mathtt{signal}(\mathtt{f.queue}) \,\rangle$

Futex Example: Critical Region

- Let $\mathtt{TS}(\mathbf{var}\ \mathtt{X}) = \langle\, \mathbf{var}\ \mathtt{r};\ \ (\mathtt{r},\mathtt{X}) := (\mathtt{X},1);\ \ \mathbf{return}\ \mathtt{r}\,\rangle$
- **var** f : futex; waiting : int := 0;

  f.val := 0;

```
process P1
  loop
    while TS(f.val) = 1 do
      ⟨waiting++⟩;
      wait_if_eq(f,1);
      ⟨waiting--⟩;
  critical section1;
  f.val := 0;
  if waiting > 0 then wake(f,1);
  noncritical section1
end loop
```

```
process P2
  loop
    while TS(f.val) = 1 do
      ⟨waiting++⟩;
      wait_if_eq(f,1);
      ⟨waiting--⟩;
  critical section1;
  f.val := 0;
  if waiting > 0 then wake(f,1);
  noncritical section2
end loop
```

Shared Data Summary

- Shared data is an *efficient* way to interact within one machine
- Behaviour is modelled as *interleaving* of atomic actions
- Correctness: no assumption about execution speed must be made
- Operations on shared data must always be *protected*
- A *monitor solution* should always be considered first
- *Invariants* should be established to guide the design
- If performance problems occur, monitors may be optimized
- Basic synchronization principles generalize to:
  - Persistent data (databases)
  - Distributed objects

  But there issues of *failure* have to taken into account