Concurrent Cheatsheet

by The Boys

L1 - Introduction

1 Why Concurrency ?

- *Abstraction* : Separating different tasks, without worrying about when to execute them.
- *Responsiveness* : Providing a responsive user interface, with different tasks executing independently.
- *Performance* : Splitting complex tasks in multiple units, and assign each unit to a different processor.

2 Concurrency vs. Parallelism

You can have *concurrency* without physical parallelism : operating systems running on single-processor single-core systems. *Parallelism* is mainly about speeding up computations by taking advantage of redundant hardware.

- *Concurrency* : Nondeterministic composition of independently executing units (logical parallelism).
- *Parallelism* : Efficient execution of fractions of a complex task on multiple processing units (physical parallelism).

Note : Challenges with Concurrency

- *Fairness* : Everyone gets to do their laundry.
- *Mutual exclusion* : Machines are operated by at most one user.

Note : Challenges with Parallelism

- *Load balancing* : Distribute load evenly over machines/-rooms.

3 Amdahl's Law

We have n processors that can run in parallel. How much speedup can we achieve ? The impact of introducing parallelism is limited by the fraction p of a program that can be parallelized :

$$speedup_{max} = \frac{1}{(1-p)+p/n}$$

L2 - Races locks and semaphores

4 Data Races

A **race condition** is a situation where the correctness of a concurrent program depends on the specific execution.
A **data race** is a situation where two or more threads simultaneously attempt to read from or write to a shared memory location without proper synchronization. This occurs when two concurrent threads :

- Access a shared memory location
- At least one access is a write
- The threads use no explicit synchronization mechanism to protect the shared data

5 Locks

Locks, also called mutexes, guarantee mutual exclusion. One thread can at most acquire the lock. Other threads block on acquiring.

```java
interface Lock {
    void lock(); // acquire lock
    void unlock(); // release lock
}
```

Example of using a lock in java :

```java
Lock lock; // shared with other synchronizing threads
lock.lock(); // entry protocol
try {
    // mutual exclusive code.
}
finally { // lock released even if an exception is thrown
    lock.unlock(); // exit protocol
}
```

The `synchronized` keyword is also used for locks in java :

```java
synchronized(this) {
    // the exclusive code is the block's content
}

synchronized Type method() { // uses lock by default
    // the exclusive code is the whole method body
}
```

Important : Built-in locks, and all lock implementations in `java.util.concurrent.locks` are **re-entrant** : a thread holding a lock can lock it again without causing a **deadlock** !

6 Semaphores

A semaphore acts as a lock but only blocks when the count is 0. The count is set to a value C (capacity) at initialization.

```java
interface Semaphore {
    int count(); // current value of counter
    void up(); // increment counter (signal)
    void down(); // decrement counter (wait)
}

Semaphore sem = new Semaphore(1); // One thread can acquire
```

Note : Binary semaphore ($capacity = 1$)
Binary semaphores are very similar to locks with one difference :
In a semaphore, a thread may decrement the counter to 0 and then let another thread increment it to 1. Thus (binary) semaphores support transferring of permissions.

7 Barriers

A barrier is a form of synchronization where there is a point (the barrier) in a program's execution that all threads in a group have to reach before any of them is allowed to continue.

8 Deadlocks

the situation where a group of threads wait forever because each of them is waiting for resources that are held by another thread in the group (circular dependency).
Conditions for a deadlock to occur :

- *Mutual exclusion* : threads may have exclusive access to the shared resources.
- *Hold and wait* : a thread may request one resource while holding another one.
- *No preemption :* resources cannot forcibly be released from threads that hold them.

- *Circular wait* : two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding.

9 Starvation

Starvation is the situation where a thread is perpetually denied access to a resource it requests (The progam is not fair). Avoiding starvation requires the scheduler to “give every thread a chance to execute”.

10 Fairness

Fairness means that every thread has a chance to execute.

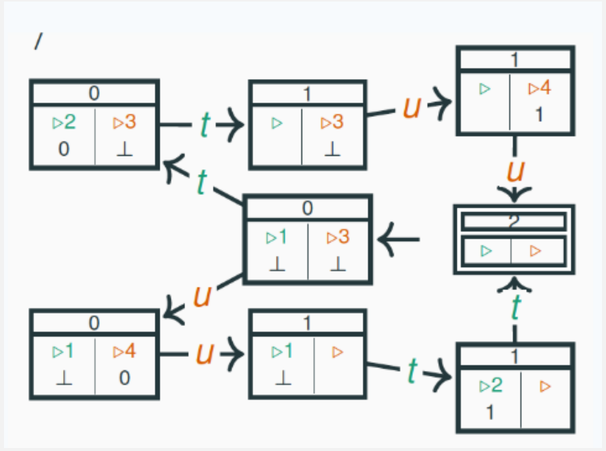
- *Weak fairness* : if a thread continuously requests (that is, without interruptions) access to a resource, then access is granted eventually (or infinitely often)
- *Strong fairness* : if a thread requests access to a resource infinitely often, then access is granted eventually (or infinitely often)

L3 - Models of concurrency and synchronization algorithms

11 State/transition diagrams and transition tables

States in a diagram capture possible program states.
Transitions connect states according to execution order.
The structural properties of a diagram capture semantic properties of the program :

- *Mutual exclusion* : there are no states where two threads are in their critical section.
- *Deadlock freedom* : for every (non-final) state, there is an outgoing transition.
- *Starvation freedom* : there is no (looping) path such that a thread never enters its critical section while trying to do so.
- *No race conditions* : all the final states have the same (correct) result.



State diagram

CURRENT	NEXT WITH t	NEXT WITH u
$\langle 0, \rhd 1, \bot, \rhd 3, \bot\rangle$	$\langle 0, \rhd 2, 0, \rhd 3, \bot\rangle$	$\langle 0, \rhd 1, \bot, \rhd 4, 0\rangle$
$\langle 0, \rhd 2, 0, \rhd 3, \bot\rangle$	$\langle 1, \rhd\ ,\ , \rhd 3, \bot\rangle$	—
$\langle 0, \rhd 1, \bot, \rhd 4, 0\rangle$	—	$\langle 1, \rhd 1, \bot, \rhd\ ,\ \rangle$
$\langle 1, \rhd\ ,\ , \rhd 3, \bot\rangle$	—	$\langle 1, \rhd\ ,\ , \rhd 4, 1\rangle$
$\langle 1, \rhd 1, \bot, \rhd\ ,\ \rangle$	$\langle 1, \rhd 2, 1, \rhd\ ,\ \rangle$	—
$\langle 1, \rhd\ ,\ , \rhd 4, 1\rangle$	—	$\langle 2, \rhd\ ,\ , \rhd\ ,\ \rangle$
$\langle 1, \rhd 2, 1, \rhd\ ,\ \rangle$	$\langle 2, \rhd\ ,\ , \rhd\ ,\ \rangle$	—
$\langle 2, \rhd\ ,\ , \rhd\ ,\ \rangle$	—	—

Transition table

12 Peterson’s algorithm

Algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication. (no locks)

Table title	
Process A	Process B
non-critical section	non-critical section
`readyA = True`	`readyB = True`
`turn = B`	`turn = A`
await($!readyB \lor turn = A$)	await($!readyA \lor turn = B$)
critical section	critical section
`readyA = False`	`readyB = False`

13 Busy waiting

Also known as spinning, or busy looping is a process synchronization technique in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution.

14 Bounded Waiting

When a thread t is waiting to enter its critical section, the maximum number of times other arriving threads are allowed to enter their critical section before t is bounded by a function of the number of contending threads.

- *r-bounded waiting* : When a thread t is waiting to enter its critical section, the maximum number of times other arriving threads are allowed to enter their critical section before t is less than $r + 1$.
- *First-come-first-served* : 0-bounded waiting.

15 Volatile fields (java)

Accessing a field (attribute) declared as `volatile` forces synchronization ! By using `volatile` we ensure the variable changes at runtime and that the compiler should not cache its value for any reason.

Important : Java does not support arrays whose elements are volatile ! Workarounds :

- Use an object of class `AtomicIntegerArray` in package `java.util.concurrent.atomic`
- Explicitly guard accesses to shared arrays with a lock

L4 - Synchronization problems with semaphores

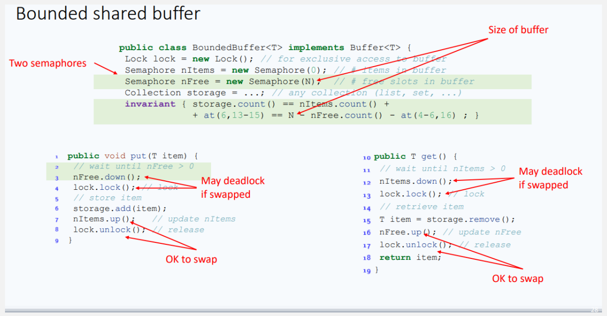
16 Producer-consumer problem

Producers and consumer exchange items through a shared buffer :

- Producers asynchronously produce items and store them in buffer
- Consumers asynchronously consume items after removing them from buffer

Producer-consumer problem : implement Buffer such that :

- Producers and consumers access the buffer in mutual exclusion
- Consumers block when the buffer is empty
- Producers block when the buffer is full (bounded buffer variant)



Bounded shared buffer

```
public class BoundedBuffer<T> implements Buffer<T> {
 Lock lock = new Lock(); // for exclusive access to buffer
 Semaphore nItems = new Semaphore(0); // # items in buffer
 Semaphore nFree = new Semaphore(N);  // # free slots in buffer
 Collection storage = ...; // any collection (list, set, ...)
 invariant { storage.count() == nItems.count() +
             + at(6,13-15) == N - nFree.count() - at(4-6,16) ; }
```

```
1 public void put(T item) {
2  // wait until nFree > 0
3  nFree.down();
4  lock.lock(); // lock
5  // store item
6  storage.add(item);
7  nItems.up();   // update nItems
8  lock.unlock(); // release
9 }
```

```
10 public T get() {
11  // wait until nItems > 0
12  nItems.down();
13  lock.lock(); // lock
14  // retrieve item
15  T item = storage.remove();
16  nFree.up(); // update nFree
17  lock.unlock(); // release
18  return item;
19 }
```

Below are a implementation of producer-consumer with an unbounded buffer using monitors.
Producer-consumer with a bounded buffer (capacity is the maximum size) uses two condition variables

```
monitor class BoundedMonitorBuffer<T> extends
    MonitorBuffer<T> {

    Condition notFull = new Condition(); // signal when not
    full

    public void put(T item) {
        if (storage.count() == capacity)
            notFull.wait(); // wait until buffer not full
            super.put(item); // do as in
    MonitorBuffer.put(item)
    }
    public T get() {
        T item = super.get(); // do as in MonitorBuffer.get()
        notFull.signal() // signal buffer not full
        return item;
    }
}
```

17 Dining philosophers

Five philosophers are sitting around a dinner table, with a fork in between each pair of adjacent philosophers. Each philosopher alternates between thinking (non critical section) and eating (critical section). In order to eat, a philosopher needs pick up the two forks that lie to the philopher's left and right. Since the forks are shared, there is a synchronization problem between philosophers (threads).

Properties of a good solution :

- Support an arbitrary number of philosophers
- Deadlock freedom
- Reasonable efficiency : eating in parallel still possible
- Starvation freedom

Below are one solution

```
// Dining philosophers solution
// Limiting the number of philosophers active at the table
    to M < N ensures that there are enough resources for
    everyone at the table, thus avoiding deadlock
public class SeatingTable implements Table {
    Lock[] forks = new Lock[N];
    Semaphore seats = new Semaphore(M); // # available seats

    public void getForks(int k) {
        seats.down();   // get a seat
        forks[left(k)].lock();  // pick up left fork
        forks[right(k)].lock(); // pick up right fork
    }

    public void putForks(int k) {
        forks[left(k)].unlock();    // put down left fork
        forks[right(k)].unlock();   // put down right fork
        seats.up();                 // leave seat
    }
}
```

18 Readers-writers

Readers and writers concurrently access shared data :

- Readers may execute concurrently with other readers, but need to exclude writers
- Writers need to exclude both readers and other writers

Other properties that a good solution should have :

- Support an arbitrary number of readers and writers
- No starvation of readers or writers

The problem captures situations common in databases, filesystems, and other situations where accesses to shared data may be inconsistent. Here is a fair and starvation free solution :

```
public class FairBoard<T> extends SyncBoard<T> {
    // held by the next thread to go
    Semaphore baton = new Semaphore(1, true); // fair binary
     sem.
    public T read() {
        baton.down(); // wait for my turn
        baton.up(); // release a waiting thread
        return super.read(); // read() as in SyncBoard
    }
    public void write(T msg) {
        baton.down(); // wait for my turn
        super.write(msg); // write() as in SyncBoard
        baton.up(); // release a waiting thread
    }
}
```

L5 - Monitors

Monitors are like a special way of making sure different parts of a computer program don't interfere with each other. They use concepts like classes, objects, and keeping information private.
In a monitor class :

- Attributes are shared variables, which all threads running on the monitor can see and modify
- Methods define critical sections, with the built-in guarantee that at most one thread is active on a monitor at any time

Mutual exclusion for n threads accessing their critical sections is straightforward to achieve using monitors : every monitor method executes uninterruptedly because at most one thread is running on a monitor at any time. A proper monitor implementation also guarantees starvation freedom.
Turning a pseudo-code monitor class into a proper Java class is straightforward :

- Mark all attributes as `private`
- Add locking to all public methods !

```
// Example of a shared counter that is free from race
// conditions, and thus is a monitor.
public class Counter {
    private int count = 0; // attribute, private
    public synchronized void increment() { // critical
     section
        count = count + 1; }
    public synchronized void decrement() { // critical
     section
        count = count - 1; }
}
```

Barrier (also called rendezvous)

A barrier is a form of synchronization where there is a point (the barrier) in a program's execution that all threads in a group have to reach before any of them is allowed to continue. A solution to the barrier synchronization problem for 2 threads with binary semaphores :

```
// Psevdocode, both started as taken.
Semaphore [] done = new Semaphore(0), new Semaphore(0)
t0                             t1
//code before barrier      |  //code before barrier
done[t0].up();             |  done[t1].up();
done[t1].down();           |  done[t0].down();
//code after barrier       |  //code after barrier
// The solution deadlocks if both t0 and t1 perform down
    before up
```

There is also solutions for barriers that has n-thread, both single and reusable barriers. Here a solution for barrier for n threads in pseudocode

```
int nDone = 0;  // number of threads that reached the barrier
Lock lock = new Lock();     // mutal exclusion for nDone
Semaphore open = new Semaphore();    // 1 if barrier is open
---------------  thread k  ----------------------
// code before barrier
lock.lock();                // lock nDone
nDone = nDone + 1;          // I'm Done
if (nDone == n) open.up();  // If i am last, open barrier
lock.unlock();              // Unlock lock
open.down();                // Wait till all done, or go if
    barrier is open
open(up);                   // let the next one go
// code after barrier
```

L6 - Parallelizing computations

A number of factors challenge designing correct and efficient parallelizations :

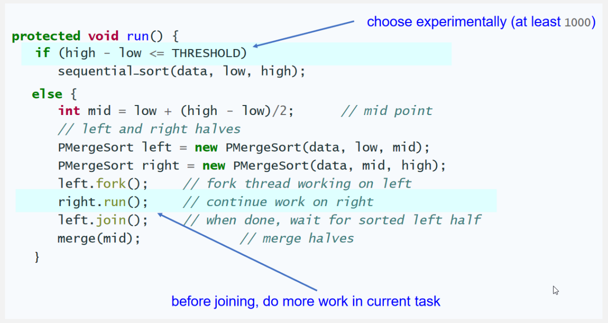
- *Sequential dependencies* : When tasks must wait for others to finish, limiting parallelism.
- *Synchronization costs* : Overhead in managing shared data among threads. (cost to sync)
- *Spawning costs* : Overhead in creating and managing threads. (cost to spawn)
- *Error proneness and composability* : Concurrent code is more error-prone and less composable.

19 Fork/join parallelism

Naturally capture sequential dependencies :

- Fork : Initially, a large task is "forked" or divided into smaller subtasks. These subtasks can be processed independently and concurrently.
- Parallel Execution : Each subtask is processed in parallel, ideally on separate processor cores or threads. This parallel execution allows for improved performance, as multiple tasks are being worked on simultaneously.
- Join : After all the subtasks have been completed, the results are "joined" or combined to produce the final result of the original, larger task.

MergeSort example with fork/join. If the length of the list is greater than 1000 it will fork a new thread :

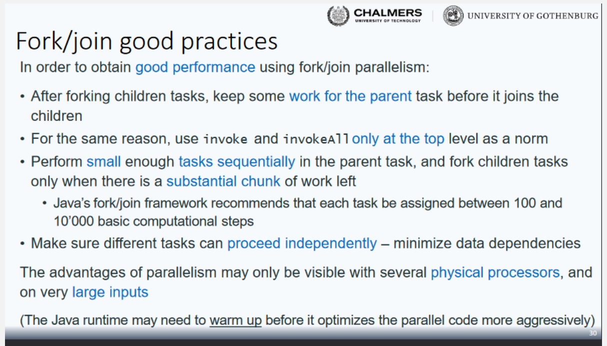


```
protected void run() {
  if (high - low <= THRESHOLD)
     sequential_sort(data, low, high);
  else {
     int mid = low + (high - low)/2;      // mid point
     // left and right halves
     PMergeSort left = new PMergeSort(data, low, mid);
     PMergeSort right = new PMergeSort(data, mid, high);
     left.fork();    // fork thread working on left
     right.run();    // continue work on right
     left.join();    // when done, wait for sorted left half
     merge(mid);            // merge halves
  }
```

choose experimentally (at least 1000)

before joining, do more work in current task

MergeSort with fork/join



CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Fork/join good practices

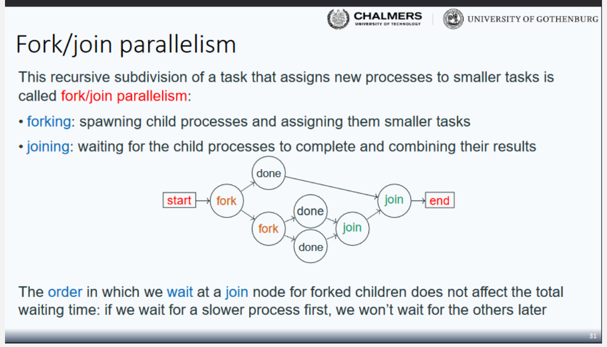
In order to obtain good performance using fork/join parallelism:

- After forking children tasks, keep some work for the parent task before it joins the children
- For the same reason, use `invoke` and `invokeAll` only at the top level as a norm
- Perform small enough tasks sequentially in the parent task, and fork children tasks only when there is a substantial chunk of work left
  - Java's fork/join framework recommends that each task be assigned between 100 and 10'000 basic computational steps
- Make sure different tasks can proceed independently – minimize data dependencies

The advantages of parallelism may only be visible with several physical processors, and on very large inputs

(The Java runtime may need to warm up before it optimizes the parallel code more aggressively)

30



CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Fork/join parallelism

This recursive subdivision of a task that assigns new processes to smaller tasks is called fork/join parallelism:

- forking: spawning child processes and assigning them smaller tasks
- joining: waiting for the child processes to complete and combining their results

The order in which we wait at a join node for forked children does not affect the total waiting time: if we wait for a slower process first, we won't wait for the others later

31

Good practices

20 Pools and work stealing

Process pools are a technique to address the problem of using an appropriate number of processes. A pool creates a number of worker processes upon initialization The number of workers is chosen according to the actual available resources to run them in parallel – a detail which pool users need not know about :

- As long as more work is available, the pool deals a work assignment to a worker that is available.
- The pool collects the results of the workers' computations.
- When all work is completed, the pool terminates and returns the overall result.

This kind of pool is called a dealing pool : it actively deals work to workers.

20.1 Workers

Workers are threads that run as long as the pool that created them does. They are either waiting for work or performing tasks. (until the pool is terminated)

20.2 Dealing or stealing ?

Dealing pools work well if :

- the workload can be split in even chunks, and
- the workload does not change over time.(users send new tasks to be done)

A stealing pool associates a queue to every worker process. The pool distributes new tasks by adding them to the workers' queues. When a worker becomes idle :

- First, it gets the next task from its own queue
- If its queue is empty, it can directly steal tasks from the queue of another worker that is currently busy

With this approach, workers adjust dynamically to the current working conditions without requiring a supervisor that can reliably predict the workload required by each task.
With stealing, the pool may even send all tasks to one default thread, letting other idle threads steal directly from it, simplifying the pool and reducing the synchronization costs it incurs.

L10 - Parallel linked lists

A straightforward way to make SequentialSet work correctly under concurrency is using a lock to ensure that at most one thread at a time is operating on the structure.

21 Concurrent set with coarse-grained locking

Every method add, remove, and has simply works as follows :

- acquires the lock on the set
- performs the operation as in SequentialSet
- releases the lock on the set

```
// Example of adding (pseudo)
public boolean add(T item) {
    lock.lock(); // lock whole set
    try {
        return super.add(item); // execute add while locking
    } finally {
        lock.unlock(); // done: release lock
    }
}
```

Access to the set is essentially sequential – missing opportunities for parallelization. If contention is high (many threads accessing the set concurrently), CoarseSet is quite slow.

22 Concurrent set with fine-grained locking

Rather than locking the whole linked list at once, we add a lock to each node. Then, threads only lock the individual nodes on which they are operating.

```
public class FineSet<T> extends SequentialSet<T> {
    // empty set
    public FineSet() {
        // smallest key
        head = new LockableNode<>(Integer.MIN_VALUE);
        // largest key
        tail = new LockableNode<>(Integer.MAX_VALUE);
        head.setNext(tail);
    }
// ...
```

Each node includes a lock object, and lock and unlock methods that access the lock.

```
class LockableNode<T> extends SequentialNode<T>{
    private Lock lock = new ReentrantLock();
    void lock() { lock.lock(); } // lock node
    void unlock() { lock.unlock(); } // unlock node
}
```

Example of a find fuction with fine-grained locking(pseudo) :

```
protected Node<T>, Node<T> find(Node<T> start, int key) {
    Node<T> pred, curr;     // predecessor and current node
     in iteration
    pred = start;           // from start node
    pred.lock();            // lock pred node
    curr = start.next();
    curr.lock();            // lock curr node
    while (curr.key < key) {
        pred.unlock();      // unlock pred node
        pred = curr;
        curr = curr.next(); // move to next node
        curr.lock();        // lock next node
    }                       // until curr.key >= key
    return (pred, curr);    // return position
}
```

When performing add, remove, find, and so on, we use **hand-over-hand locking** :

- Always keep at least one node locked to prevent interference between threads.
- Locking two nodes at once is sufficient to prevent problems with conflicting operations.

The hand-over-hand locking protocol may be quite slow, as it involves a significant number of lock operations.

23 Concurrent set with optimistic locking

We validate a position after finding it while the nodes are locked ; to verify that no interference took place. The "next" node needs to be volatile, because we aer not using locks !

```
public class OptimisticSet<T> extends SequentialSet<T> {
    public FineSet() {
        head = new ReadWriteNode<>(Integer.MIN_VALUE);
        tail = new ReadWriteNode<>(Integer.MAX_VALUE);
        head.setNext(tail);
    }
    // is (pred, curr) a valid position?
    protected boolean valid(Node<T> pred, Node<T> curr){
        Node<T> node = head; // start from head
        // does pred point to curr?
        while (node.key() <= pred.key()) {
            if (node == pred) return pred.next() == curr;
            node = node.next(); // continue to the next node
        } // until node.pred > pred.key
        return false;
    }
// ...
```

In OptimisticSet, operations work as follows :

- Find the item's position inside the list without locking.
- Lock the position's nodes pred and curr
- Validate the position while the nodes are locked :
  - if the position is valid, perform the operation while the nodes are locked, then release locks.
  - if the position is invalid, release locks and repeat the operation from scratch.

This approach is optimistic because it works well when validation is often successful. (so we don't have to repeat operations)
OptimisticSet is not starvation free : a thread t may fail validation forever.

24 Lazy node removal

A way to atomically share the information that a node is being removed, but without locking. To this end, each node includes a flag `valid` with setters and getters. If this flag is false it means the node is removed/being removed.

25 Lock free access

access the information of both attributes `valid` and `next` atomically : every node includes an attribute `nextValid` of type `AtomicMarkableReference<Node<T>>` , which provides methods to both update a reference and mark it, atomically.

L11 - Parallel Queues and Lock-free programming

We use linked lists to implement a lock-free queue data structures with interface :

```
interface Queue<T> {
void enqueue(T item); // add item to back of queue
T dequeue() throws EmptyException; // remove and return item
    in front of the queue,EmptyException if queue is empty
}
```

Atomic references , To implement data structures that are correct under concurrent access without using any locks we need to rely on synchronization primitives more powerful than just reading and writing shared variables We are going to use a variant of the compare-and-set operation :

```
class AtomicReference<V> {
V get(); // current reference
void set(V newRef); // set reference to newRef
// if reference == expectRef, set to newRef and return true
// otherwise, do not change reference and return false
boolean compareAndSet(V expectRef, V newRef);}
```

Nodes, The underlying implementations of queues use singly-linked lists, which are made of chains of nodes.
Every node :

- stores an item– its value
- points to the next nod in the chain

To build a lock-free implementation, nextis a reference that supports compare-and-set operations (thus,need not be volatile)

```
class QNode<T>{
T value; // value of node
AtomicReference<QNode<T>> next; // next node in chain
QNode(T value) {
this.value = value;
next = new AtomicReference<>(null); }
}
```

Queues as chains of nodes A list with a pair of head and tail references implements a queue :

- a sentinel node points to the first element to be dequeued
- a sentinel node points to the first element to be dequeued
- head points to the sentinel (front of queue)
- tail points to the latest enqueued element (back of queue), or the sentinel if the queue is empty
- The sentinel (also called “dummy node”) ensures that head and tail are never null

```
class LockFreeQueue<T> implements Queue<T>
{
// access to front and back of queue
protected AtomicReference<QNode<T>> head;
protected AtomicReference<QNode<T>> tail;
// constructor creating empty queue
public LockFreeQueue() {
// value of sentinel does not matter
QNode<T> sentinel = new QNode<>();
head = new AtomicReference<>(sentinel);
tail = new AtomicReference<>(sentinel);
}}
```

Enqueue operation , The method enqueue adds a new node to the back of a queue – where tail points. It requires two updates that modify the linked structure :

- update last : make the last node in the queue point to the new node
- update tail : make tail point to the new node

Each update is individually atomic (it uses compare-and-set), but another thread may interfere between the two updates :

- repeat update last until success
- try update tail once
- the implementation should be able to deal with a “half finished” enqueue operation (tail not updated yet), and finish the job – this technique is called helping

Dequeue operation The method dequeue removes the node at the head of a queue (where the sentinel points) Unlike enqueue, dequeueing only requires one update to the linked structure :

- update head : make head point the node previously pointed to by the sentinel ; the same node becomes the new sentinel and is also returned

The update is atomic (it uses compare-and-set), but other threads may be updating the head concurrently :

- repeat update head until success
- if you detect a “half finished” enqueue operation – with the tail pointing to the sentinel about to be removed – help by moving the tail forward

Code of Enqueue and dequeue operation

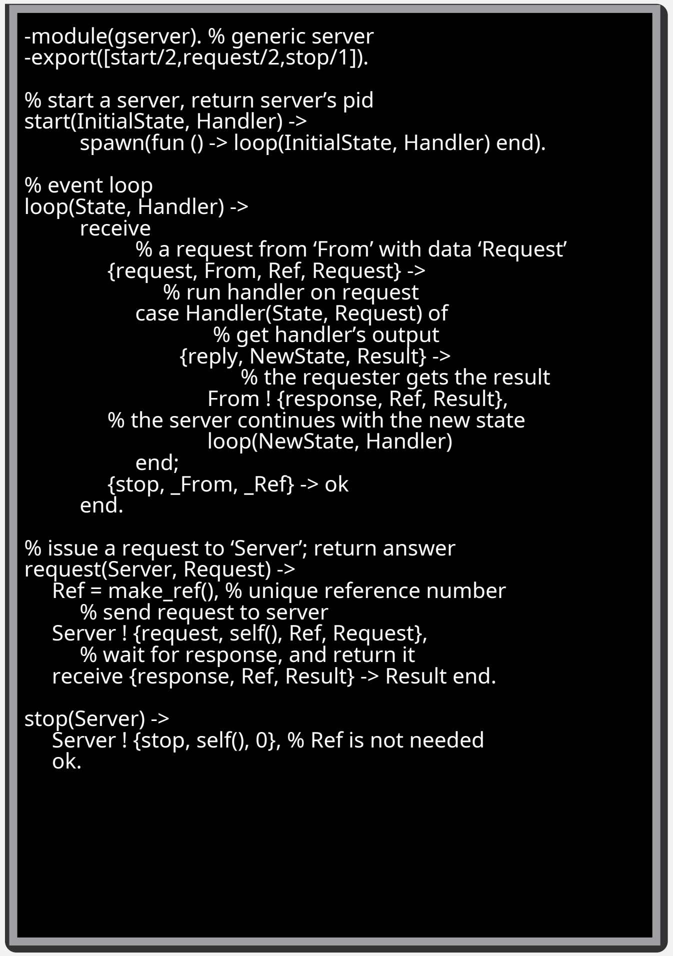
```
public void enqueue(T value) {
QNode<T> node = new QNode<>(value); // new node
while (true) { // nodes at back of queue
QNode<T> last = tail.get();
QNode<T> nextToLast = last.next.get();
if (last == tail.get()) { // if tail points to last
if (nextToLast == null) { // and if last has no successor
// make last point to new node
if (last.next.compareAndSet(nextToLast, node))
// if last.next updated, try once to update tail
{ tail.compareAndSet(last, node); return; }
} else // valid successor: try to update tail and repeat
{ tail.compareAndSet(last, nextToLast); } } } }
```

```
public T dequeue() throws EmptyException {
while (true){ // nodes at front, back of queue
QNode<T> sentinel = head.get();
last = tail.get();
first = sentinel.next.get();
if (sentinel == head.get()) { // if head points to sentinel
if (sentinel == last) { // if tail also points to sentinel
if (first == null) { // empty queue: raise exception
throw new EmptyException(); }
// non-empty: update tail, repeat
tail.compareAndSet(last, first); }
else { T value = first.value; // tail not point to sentinel
// make head point to first; retry until success
if (head.compareAndSet(sentinel, first)) return value; }}}}
```

Garbage collection saves the day , If we were using a language without garbage collection – where objects can be recycled – the following problem could occur :

- t is about to CAS head from sentinel node a to node b : head.compareAndSet(sentinel,first)
- u dequeues b andx
- u enqueues a again (the very same node), enqueues y, enqueues p, and then dequeues a again, so that the samenode a becomes the sentinel again
- t completes CAS successfully (head still points to t's local reference sentinel), but node b is nowd is connected !

Erlang - Ping Pong



```
-module(gserver). % generic server
-export([start/2,request/2,stop/1]).

% start a server, return server's pid
start(InitialState, Handler) ->
        spawn(fun () -> loop(InitialState, Handler) end).

% event loop
loop(State, Handler) ->
        receive
                % a request from 'From' with data 'Request'
            {request, From, Ref, Request} ->
                    % run handler on request
                case Handler(State, Request) of
                        % get handler's output
                    {reply, NewState, Result} ->
                            % the requester gets the result
                        From ! {response, Ref, Result},
            % the server continues with the new state
                        loop(NewState, Handler)
                end;
            {stop, _From, _Ref} -> ok
        end.

% issue a request to 'Server'; return answer
request(Server, Request) ->
    Ref = make_ref(), % unique reference number
        % send request to server
    Server ! {request, self(), Ref, Request},
        % wait for response, and return it
    receive {response, Ref, Result} -> Result end.

stop(Server) ->
    Server ! {stop, self(), 0}, % Ref is not needed
    ok.
```

Erlang - State

```
-module(state).
-export([main/0]).

% Simple server that can receive two types of messages:
%    {hello, From, Name}, where From is the PID of the sender
%    {reqs, From},  where From is the PID of the sender
% If we get {hello, From, Name}:
%    - Already communicated with From and same Name return:
    "Already talked with you"
%    - Already communicated with From and diff Name return:
    "No you're not"
%    - First communication with From return: "Hello mate!"
% If we get {reqs, From}, we return the total number of
    contacts we have until that point.

server(Contacts,List) ->
    receive
        {hello, From, Name} ->
            search(Contacts,List,List,From,Name);
        {reqs, From} ->
            From ! Contacts,
            server(Contacts,List)
    end.

search(Contacts,OldList,[],From,Name) ->
   From ! "Hello mate!",
   server(Contacts+1, [{From,Name} | OldList]);
search(Contacts,OldList,[{From,Name}|_Tail],From,Name) ->
   From ! "Already talked with you",
   server(Contacts,OldList);

    search(Contacts,OldList,[{From,_OtherName}|_Tail],From,_Name)
    ->
   From ! "No you're not",
   server(Contacts,OldList);
search(Contacts,OldList,[_Head|Tail],From,Name) ->
   search(Contacts,OldList,Tail,From,Name).


client(Server,Name,OtherName) ->
   Server ! { hello,self(),Name},
   receive Msg1 -> io:fwrite("~s~n", [Msg1]) end,
   Server ! { hello, self(),OtherName},
   receive Msg2 -> io:fwrite("~s~n", [Msg2]) end.

main() ->
    Server = spawn(fun() -> server(0,[]) end),
    Xs = lists:seq(1, 10), % Creates a list from 1 to 10
    [ spawn(fun() -> client(Server,X,X+1) end) || X <- Xs],
    ok.
```

Erlang - Genserver Implementation

```
-module(server_genserver).
-export([main/0,multiply/2,add/2,subtract/2,
square/1,count/0,start_server/0,stop_server/0]).
%-import(genserver,[start/3,request/2,stop/1]).

% We have our server become a genserver (read genserver.erl
    to see the
% implementation)
start_server() ->
    genserver:start(server,0,fun my_handler/2).

% The server loop. Now no recursion, it looks like an
    ordinary function
my_handler(N,{count}) ->
    { reply, N, N };
my_handler(N,{mul, Num1, Num2}) ->
    { reply, Num1 * Num2, N+2};
my_handler(N,{add,Num1, Num2}) ->
    { reply, Num1 + Num2, N+1};
my_handler(N,{sub,Num1, Num2}) ->
    { reply, Num1 - Num2, N};
my_handler(N,{square, Num }) ->
    { reply, Num * Num, N+2}.

multiply(X,Y) ->
    genserver:request(server,{mul, X, Y}).

add(X,Y) ->
    genserver:request(server,{add, X, Y}).

subtract(X,Y) ->
    genserver:request(server,{sub, X, Y}).

square(X) ->
    genserver:request(server,{square, X}).

count() ->
    genserver:request(server,{count}).

stop_server() ->
    genserver:stop(server).

main() ->
    start_server(),
    multiply(3,square(2)).
```

Dictionary

- **Atomic** - An operation is atomic if the operation can't be interrupted by another thread.
- **Traces** - The sequence of states that two or more threads had when executing a concurrent program. The order the threads executed each instruction.
- **Process** - an independent unit of execution – the abstraction of a running sequential program. Does not share memory with other processes. Is ready, blocked or running in the scheduler.
- **Thread** - A thread is a lightweight process – an independent unit of execution in the same program space. Does share memory with other threads.
- **Shared Memory vs message passing** - Shared memory = Java threads, Message passing = Erlang. In shared memory, threads have access to the same memory space.
- **Mutual Exclusion - Property** : No more than 1 thread is in its critical section at any given time. A solution to mutual exclusion problems should include freedom from starvation and freedom from deadlock.
- **Semaphores** - A semaphore is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system.
- **Volatile** - The volatile keyword indicates that a field might be modified by multiple threads that are executing at the same time.
- **Petersons solution** - is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.
- **Data races** - A data race occurs when two or more threads or processes access shared data concurrently, and at least one of them performs a write operation (modification) on that data. Data races are typically associated with low-level programming languages like C and C++ but can also occur in other multi-threaded environments.
- **Race condition** - The behavior of a program depends on the order in which threads or processes execute their tasks, leading to unintended consequences.
- **Monitor** - A monitor is a high-level synchronization construct that provides a way to control access to shared resources or critical sections of code. Monitors are used to ensure that only one thread or process can execute a specific section of code at a time, preventing race conditions and data races.
- **Barrier** - A barrier is a synchronization mechanism that is used to coordinate a group of threads or processes, forcing them to wait at a specific point until all threads or processes have reached that point. Once all participants have arrived at the barrier, they can proceed.
- **Producer/Consumer** - exchanges items through a shared buffer. Producers asynchronously produce items and store them in the buffer. Consumers asynchronously consume items after removing them from the buffer.
- **Bounded Waiting** - When a thread t is waiting to enter its critical section, the maximum number of times arriving.
- **Fairness** - Fairness means that a thread is granted access to a resource eventually. (Starvation free).
- **Starvation** - It's the situation where a thread is perpetually denied access to a resource it requests.
- **Actor model** - Erlang is based on this. Actors are abstractions of processes. NO shared state between actors.
- **Signaling** - A thread can signal that it is executing inside the monitor (where there can only be one thread). This makes other threads unable to execute inside the monitor.Signal and continue : thread s continues executing and thread u (blocked) is moved to the entry que for the monitor. Signal and wait : s is moved to the entry que and u resumes executing (silently gets the monitors lock).
- **wait()** - Pauses a thread and also releases synchronization
- **CAS** - Compare and set
- **Message Passing** - The way Erlang passes comands and variables around
- **Fork/join pools** - A ForkJoinPool is a specialized type of thread pool provided by Java's Fork/Join Framework. It is designed to efficiently execute tasks that can be split into subtasks and run in parallel, making it especially well-suited for parallel computing and tasks that involve recursive decomposition.
- **find()** - Returns 2 locked objects, hand-overReturne-hand, dont use validation
- **coarse lock** - Locks all shared data

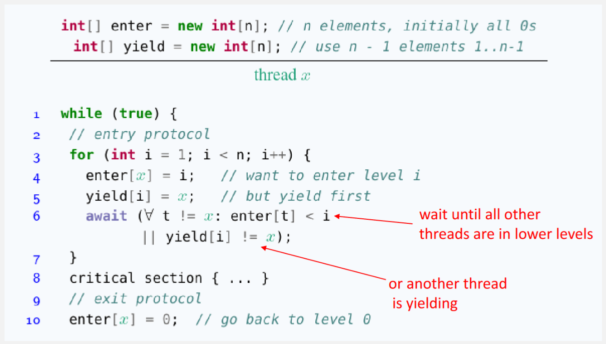
Deadlock

- **Mutual Exclusion** - threads may have exclusive access to the shared resources
- **Hold and wait** - a thread may request one recourse while holding another
- **No preemption** - - resources cannot forcibly be released from threads that hold them
- **Circular wait** - two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding.

Pages for lecture

1. Ben-Ari 1, 2.1-2.3.
2. Ben-Ari 2.4-2.7, 2.12, 6.1-6.5, 6.9.
3. Ben-Ari 3.1-3.9.
4. Ben-Ari 6.5-6.9
5. Ben-Ari 7.1-7.3, 7.5.

Peterons n threads



```
int[] enter = new int[n]; // n elements, initially all 0s
 int[] yield = new int[n]; // use n - 1 elements 1..n-1
```

thread x

```
while (true) {
 // entry protocol
 for (int i = 1; i < n; i++) {
   enter[x] = i;   // want to enter level i
   yield[i] = x;   // but yield first
   await (∀ t != x: enter[t] < i
          || yield[i] != x);
 }
 critical section { ... }
 // exit protocol
 enter[x] = 0;  // go back to level 0
```

wait until all other threads are in lower levels
or another thread is yielding

Petersons for n threads