1 Concurrency

1.1 Problems

bad interleaving multiple methods executed at the same time which causes results not possible if only one thread were running (due to inconsistent state within the methods)

data races weird behaviour usually caused by compiler optimizations (out of order execution, memory reordering) and threads that don't see the same value (due to CPU caches)

1.2 Blocked Threads

deadlock processes are blocked as all of them are waiting for another to proceed

livelock competing processes detect a potential deadlock but make no progress while trying to resolve it (constantly changing)

starvation process is perpetually denied necessary resources

convoying thread holding a resource R is descheduled while other threads queue up waiting for R

priority inversion lower priority thread holds a resource R that a high priority thread is waiting on

memory model describes the interactions of threads through memory and their shared use of the data.

1.3 Solutions

thread-local resource only accessible by one thread

immutable only read access allowed

synchronization mutual exclusion for safe write operations

1.4 Registers

register basic memory object, shared or not, read and write operations

SWMR (single writer multiple reader), only one concurrent writer but multiple concurrent readers allowed

safe register any read not concurrent with a write returns the current operation, any read concurrent with a write can return any value of the domain of the register

Hint: the volatile keyword does not only prevent threads from saving that instance in the cache but also prevent out-of-order execution.

1.5 Hardware Support

Most hardware provides atomic registers via the instructions Test and Set/Compare and Swap:

```
boolean TAS(memref s)
atomic
    if (mem[s] == 0) {
        mem[s] = 1;
        return true;
        }
    else
        return false;
```

```
int CAS (memref a, int old, int new)
atomic
    oldval = mem[a];
    if (old == oldval)
        Mem[a] = new;
    return oldval;
```

1.6 Deadlock avoidance

2 phase locking check if holding a lock would introduce a deadlock, retry if so
resource ordering make sure the locks are never held in a way that would cause a deadlock

1.7 Dining philosphers

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.
Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before getting both of them.

1.8 Lock Granularity

1.8.1 Coarse Grained

procedure lock whole data structure
advantages easy to implement
disadvantages bottleneck, locks whole data structure

1.8.2 Fine Grained

procedure hand-over-hand traversal
disadvantages lots of (un)locking

1.8.3 Optimistic List

procedure 1. find element 2. lock it and predecessor 3. verify
advantages no contention on traversals, traversals are wait free, less lock acquisitions
disadvantages need to traverse list twice, contains() needs to acquire locks

1.8.4 Lazy List

procedure remove nodes lazily (marking it first)
advantages traverses only once, contains() never blocks

1.8.5 Lockfree List

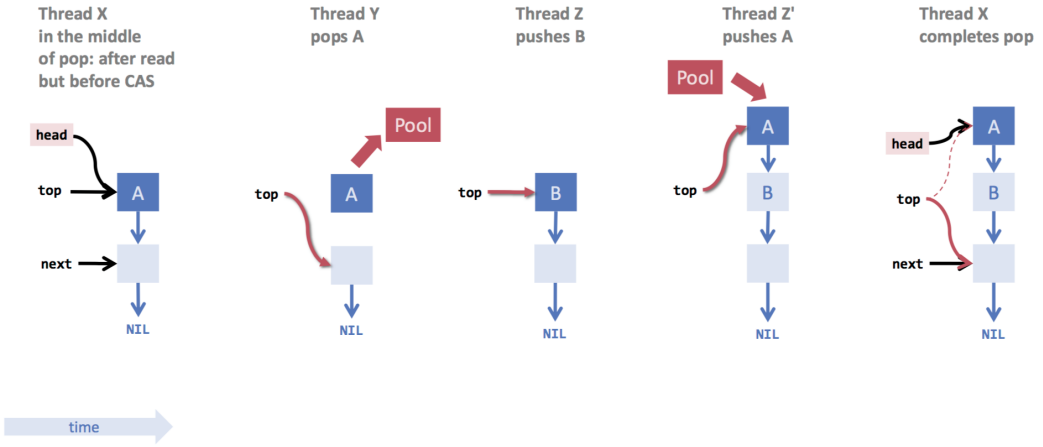
procedure use DCAS (double CAS) to implement lazy list in a lock free manner (allows us to atomically check two conditions at once)

1.8.6 Lockfree Programming

lock-freedom at least one algorithm always makes progress even if other algorithms run concurrently. Implies systemwide progress but not freedom from starvation. Deadlock-free by design.
wait-freedom all algorithms return in a finite time. Implies freedom from starvation and lock-freedom.
non-blocking failure or suspension of one thread cannot cause failure or suspension of another thread

ABA-Problem The ABA problem occurs when one activity fails to recognise that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overall state has not been changed.

Can be solved using DCAS (not available on most platforms), pointer tagging (only delays it) or hazard pointers.



	non-blocking	**blocking**
everyone makes progress	wait-free	starvation-free
someone makes progress	lock-free	deadlock-free

2 Implementation

	Fair	**Performance**	**Remark**
Lock	no	wasteful if held for a longer period due to contention, context switching	can be improved using an (exponential) back-off, not waitfree
Read-Write Lock	-	only useful if there are way more reads than writes	fairness needs to be implemented, starvation of the writes a problem
Semaphore			

2.1 2-Thread Locks

2.1.1 Decker

```
volatile boolean wantp=false, wantq=false, integer turn= 1
```

Process P

```
loop
    non-critical section
    wantp = true
    while (wantq){
        if (turn == 2){
            wantp = false;
            while(turn!=1);
            wantp = true }}
    critical section
    turn = 2
    wantp = false
```

only when q tries to get lock

and q has preference

let q proceed

and wait

and try again

Process Q

```
loop
    non-critical section
    wantq = true
    while (wantp){
        if (turn == 1){
            wantq = false
            while(turn != 2);
            wantq = true }}
    critical section
    turn = 1
    wantq = false
```

2.1.2 Peterson

```
let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1
```

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical sectio
    flag[P] = false
```

I am interested

but you go first

We both are interested

And you go first

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

2.2 n-Thread Locks

2.2.1 Filter Lock

Basically uses n Peterson locks to filter n threads for the critical section. It's not fair.

2.2.2 Bakery Algorithm

```
integer array[0..n-1] label = [0,...,0]
boolean array[0..n-1] flag =  [false, ..., false]
```

SWMR «ticket number»

SWMR «I want the lock»

```
lock(me):
   flag[me] = true;
   label[me] = max(label[0], ... , label[n-1]) + 1;
   while (∃k ≠ me: flag[k] && (k,label[k]) ≪ (me,label[me])) {};

unlock(me):
   flag[me] = false;
```

$(k, l_k) \ll (j, l_j) \Leftrightarrow l_k < l_j \text{ or } (l_k = l_j \text{ and } k < j)$

2.2.3 Spin Lock using CAS

Acquire (lock)

```
while (CAS(lock, 0, 1) != 0);
```

Release (lock)

```
CAS(lock, 1, 0);
```

ignore result

A spin lock is rather inefficient (at least if the lock is expected to be held for a longer time period). To counteract this, an **(exponential) backoff** can be implemented, so that the fighting threads don't fight over the register so much.

3 Linearization

invocation/response method call split into two events
pending invocation invocation that doesn't have a matching response (yet)
history H sequence of invocations and responses
object projection H|q history H filtered by all invocations/responses addressing q
thread projection H|B history H filtered by all invocations/responses executed on B
complete subhistories history H without its pending invocations
sequential history method calls of different threads do not interleave
well formed history per thread projections are sequential
equivalent histories per thread projections are equal (e.g. H|A = G|A, H|B = G|B)
legal history for every object x H|x adheres to the sequential specification of x

4 Message Passing

Actors send messages in parallel which makes them more isolated (e.g. Akka for Scala, Erlang)
CSP Communicating sequential processes, send messages sequentially, more tied together, rendezvous are possible (e.g. Go)

MPI communication protocol for parallel processes, messages may be sent asynchronously or synchronously

4.1 Consensus Protocol

wait-free returns in finite time for each thread
consistent all threads decide on the same value
valid the common decision is an input from one thread