Lock-Free Concurrent Data Structures, CAS and the ABA-Problem

Dr. Wolfgang Koch

Friedrich Schiller University Jena

Department of Mathematics and
Computer Science

Jena, Germany

wolfgang.koch@uni-jena.de

Motivation

Today almost all PCs and Laptops have a **multi-core** (e.g. quad-core) processor

using SMP (symmetric multiprocessing)
with shared memory and cache coherence

But most **software has not changed** adequately, doesn't use much multithreading

Lock-free synchronization in **concurrent data structures** for a long time has been a research topic only in the area of mainframes and supercomputers

Only in recent years it obtained a common attention (the subject in not really new, so many of the basic articles are rather old)

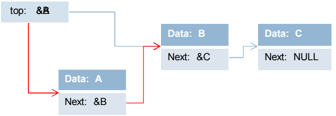
Blocking Queue

Example: “Blocking” LIFO Queue (a Stack):

```
shared Node * top;
shared Lock lock;

void push(Node *node)
{
  Node *t;        // local pointer

  acquire(&lock);
   t = top;
   node->Next = t;
   top = node;
  release(&lock);
}
```



Locks, Critical Section

`shared Lock lock;` Why do we need the lock?

We protect a **critical section**:

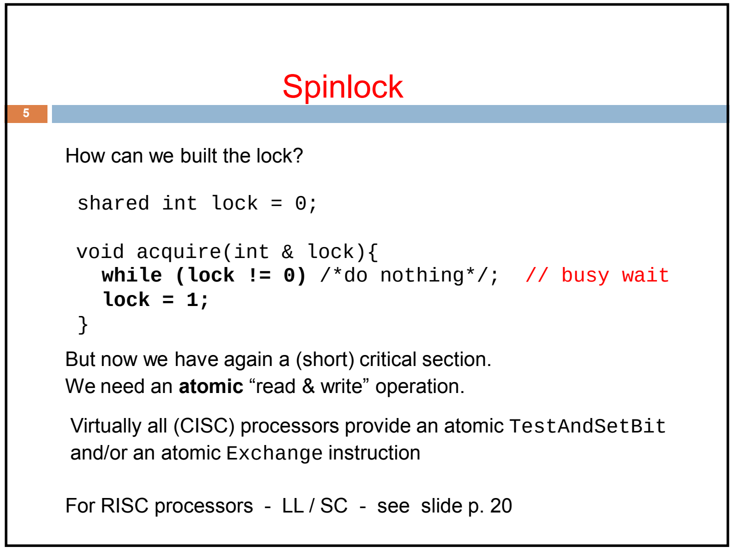
```
      acquire(&lock); … release(&lock);
```

without the lock (2 or more threads may interfere):

```
t1=top;
                      t2=top;
node1->Next = t1;
top = node1;
                      node2->Next = t2;   // =t1, not node1
                      top = node2;
```

Now top points to node2 (and node2 to old *top) – node1 is lost!

`t=top; … top=node;` is a **critical section** – **at most 1** thread may be in the section at the same time (**mutual exclusion**, mutex)



Spinlock

How can we built the lock?

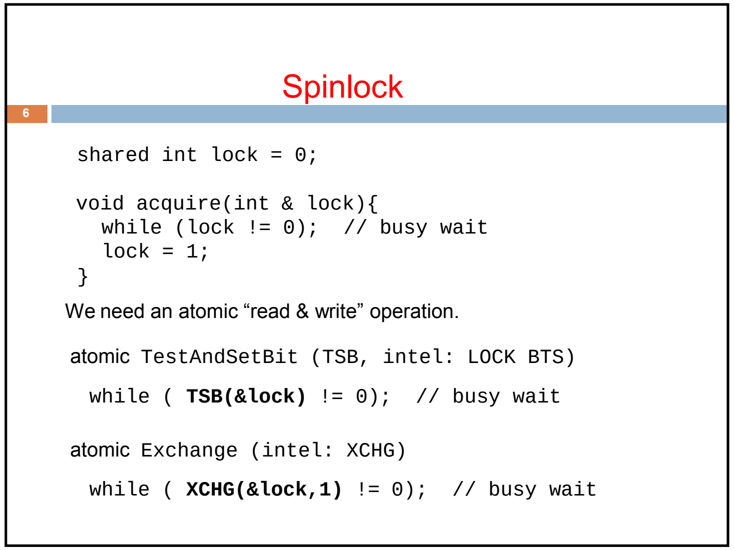
```
shared int lock = 0;

void acquire(int & lock){
  while (lock != 0) /*do nothing*/;  // busy wait
  lock = 1;
}
```

But now we have again a (short) critical section.
We need an **atomic** “read & write” operation.

Virtually all (CISC) processors provide an atomic `TestAndSetBit` and/or an atomic `Exchange` instruction

For RISC processors - LL / SC - see slide p. 20



Spinlock

```
shared int lock = 0;

void acquire(int & lock){
  while (lock != 0);  // busy wait
  lock = 1;
}
```

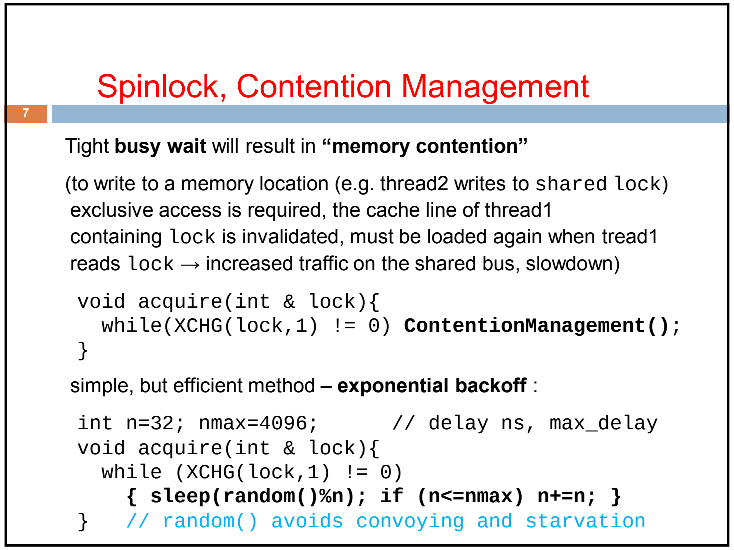
We need an atomic “read & write” operation.

atomic `TestAndSetBit (TSB, intel: LOCK BTS)`

```
while ( TSB(&lock) != 0);  // busy wait
```

atomic `Exchange (intel: XCHG)`

```
while ( XCHG(&lock,1) != 0);  // busy wait
```



Spinlock, Contention Management

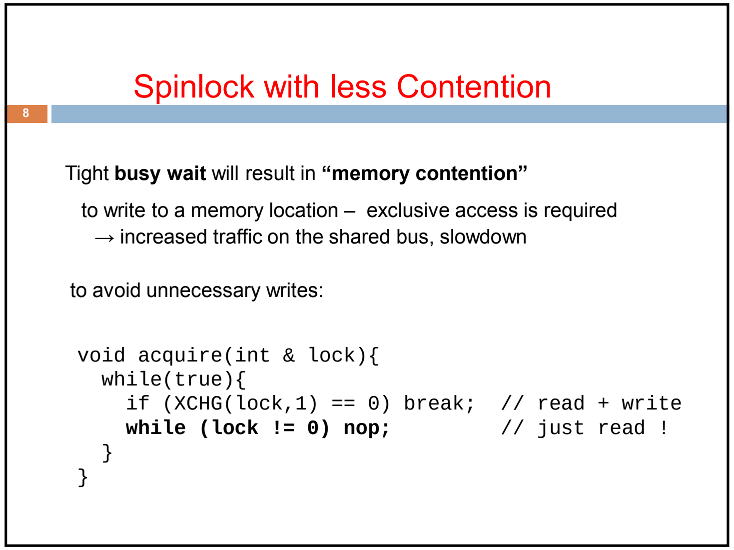
Tight **busy wait** will result in **“memory contention”**

(to write to a memory location (e.g. thread2 writes to `shared lock`) exclusive access is required, the cache line of thread1 containing `lock` is invalidated, must be loaded again when tread1 reads `lock` → increased traffic on the shared bus, slowdown)

```
void acquire(int & lock){
  while(XCHG(lock,1) != 0) ContentionManagement();
}
```

simple, but efficient method – **exponential backoff** :

```
int n=32; nmax=4096;      // delay ns, max_delay
void acquire(int & lock){
  while (XCHG(lock,1) != 0)
    { sleep(random()%n); if (n<=nmax) n+=n; }
}   // random() avoids convoying and starvation
```



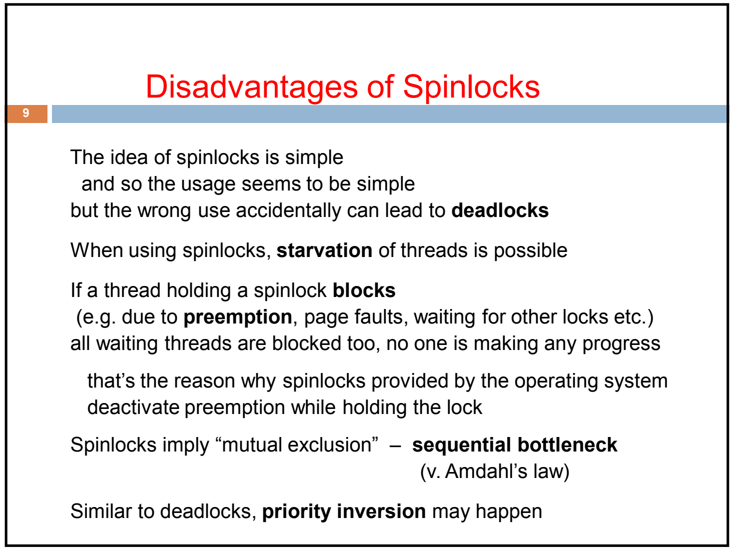
Spinlock with less Contention

Tight **busy wait** will result in **“memory contention”**

to write to a memory location – exclusive access is required
→ increased traffic on the shared bus, slowdown

to avoid unnecessary writes:

```
void acquire(int & lock){
  while(true){
    if (XCHG(lock,1) == 0) break;  // read + write
    while (lock != 0) nop;         // just read !
  }
}
```



Disadvantages of Spinlocks

The idea of spinlocks is simple
and so the usage seems to be simple
but the wrong use accidentally can lead to **deadlocks**

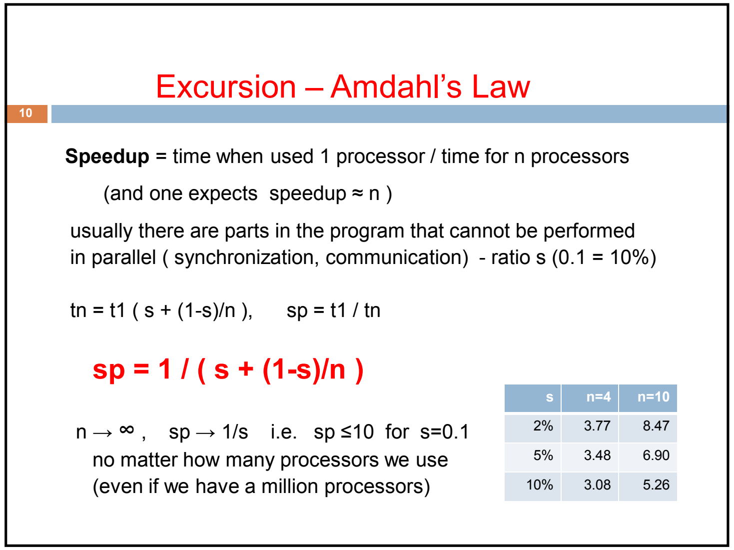
When using spinlocks, **starvation** of threads is possible

If a thread holding a spinlock **blocks**
(e.g. due to **preemption**, page faults, waiting for other locks etc.)
all waiting threads are blocked too, no one is making any progress

that's the reason why spinlocks provided by the operating system
deactivate preemption while holding the lock

Spinlocks imply "mutual exclusion" – **sequential bottleneck**
(v. Amdahl's law)

Similar to deadlocks, **priority inversion** may happen



Excursion – Amdahl's Law

Speedup = time when used 1 processor / time for n processors

(and one expects speedup ≈ n)

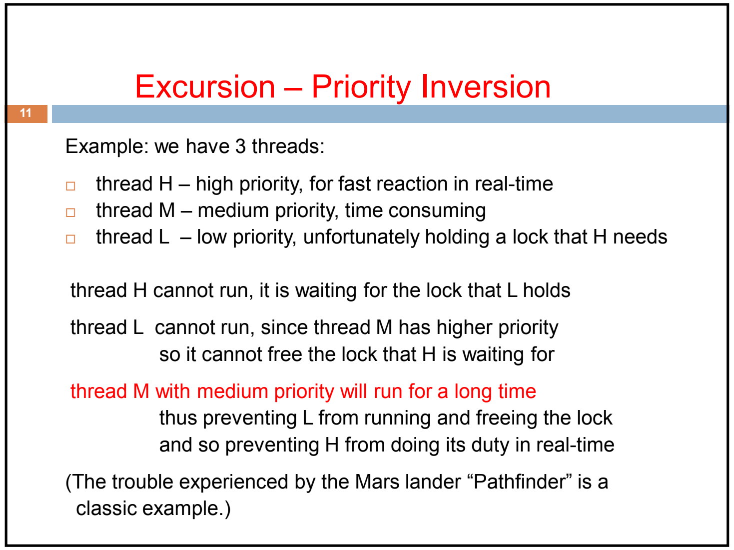
usually there are parts in the program that cannot be performed
in parallel (synchronization, communication) - ratio s (0.1 = 10%)

tn = t1 (s + (1-s)/n), sp = t1 / tn

$$sp = 1 / (s + (1-s)/n)$$

$n \to \infty$, $sp \to 1/s$ i.e. $sp \le 10$ for $s=0.1$
no matter how many processors we use
(even if we have a million processors)

s	n=4	n=10
2%	3.77	8.47
5%	3.48	6.90
10%	3.08	5.26



Excursion – Priority Inversion

Example: we have 3 threads:

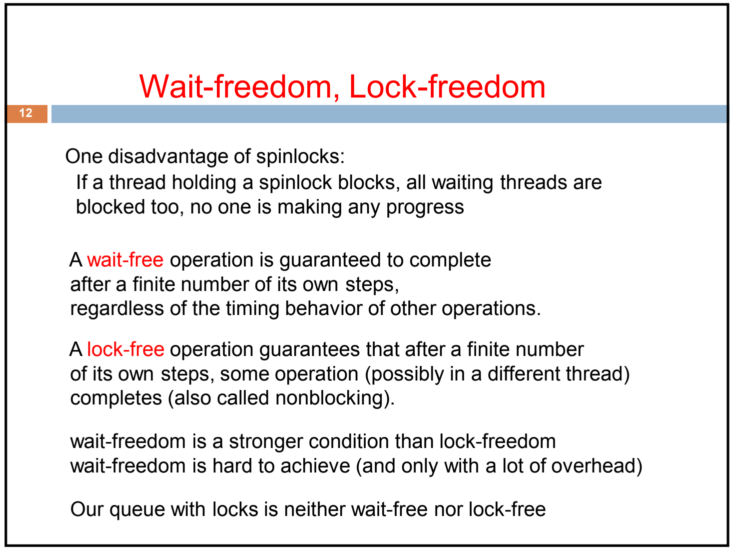
- thread H – high priority, for fast reaction in real-time
- thread M – medium priority, time consuming
- thread L – low priority, unfortunately holding a lock that H needs

thread H cannot run, it is waiting for the lock that L holds

thread L cannot run, since thread M has higher priority
so it cannot free the lock that H is waiting for

thread M with medium priority will run for a long time
thus preventing L from running and freeing the lock
and so preventing H from doing its duty in real-time

(The trouble experienced by the Mars lander "Pathfinder" is a classic example.)



Wait-freedom, Lock-freedom

One disadvantage of spinlocks:
If a thread holding a spinlock blocks, all waiting threads are
blocked too, no one is making any progress

A wait-free operation is guaranteed to complete
after a finite number of its own steps,
regardless of the timing behavior of other operations.

A lock-free operation guarantees that after a finite number
of its own steps, some operation (possibly in a different thread)
completes (also called nonblocking).

wait-freedom is a stronger condition than lock-freedom
wait-freedom is hard to achieve (and only with a lot of overhead)

Our queue with locks is neither wait-free nor lock-free

Lock-free method

Disadvantages of spinlocks (slide p. 9) – request for a lock-free method
make changes on a copy, then set the copy into effect
in a single atomic step - if the original has not changed

```
boolean try_push(Node *node)
{ boolean  res;
  Node     *t;          // local pointer

  t = top;              // local copy
  node->Next = t;       // still private node
  // top = node;        // global – Danger!
  atomic( if (top==t) {top=node; res=true;}
          else res=false;     // try again
        )
  return res;
}
```

Lock-free method, CAS

. . . set the copy into effect in a single atomic step
if the original has not changed

```
atomic( if (top==t) {top=node; res=true;}
        else res=false;     // try again
      )
```

We need an **atomic primitive** that accomplishes this task
(TSB and XCHG are not strong enough)

IBM introduced `CompareAndSwap (CAS)` in 1970 in the IBM 370

```
res = CAS(&top, t, node);
```

Compare-and-Swap - CAS

IBM introduced `CompareAndSwap (CAS)` in 1970 in the IBM 370
(in some other processors called `CompareAndSet`) – boolean CAS

```
type - longword or pointer

boolean CAS(type * mem, type exp, type new)
{ atomic(
    if (*mem == exp){*mem=new; return true;}
    else return false;   //and leave mem untouched
  )
}
```

In Intel processors (starting with i486 – 1989) there we find a variant
of CAS (called `CMPXCHG` – Compare and Exchange) that returns
the old value of mem in register EAX, and a boolean result in the Z-flag

Intel - CMPXCHG

In Intel processors (starting with i486) there we find another variant
of CAS (called `CMPXCHG` – Compare and Exchange) that returns
the old value of mem in register EAX, and a boolean result in the Z-flag

```
int CAS(void **mem, void *old, void *new)
{
  int res;

  asm("lock cmpxchg %3,%1; mov $0,%0; jnz 1f; inc %0; 1:"
  : "=a" (res) : "m" (*mem), "a" (old), "d" (new) );

  return res;
}
```

(there is also an atomic instruction `CMPXCHG8B` – for double long values)

Lock-free methods

```
boolean try_push(Node *node)
{ Node *t;          // local pointer
  t = top;
  node->Next = t;
  return CAS(&top,t,node);
}

void push(Node *node)
{ Node *t;          // local pointer
  while(true){
    t = top;
    node->Next = t;
    if (CAS(&top,t,node)) break;
  }
}
```

Lock-free methods

```
void push(Node *node)
{ Node *t;
  while(true){
    t = top;
    node->Next = t;
    if (CAS(&top,t,node)) break;
  }
}
```

is lock-free:

if CAS succeeds, our thread completes the push-operation
if CAS fails, it failed because another thread has changed top
so the CAS of that other thread succeeded
the other tread has completed its (push-) operation

Lock-free pop-operation

```
Node * pop(void)
{
  Node *t, *next;

  while(true){
    t = top;
    if (t == NULL) break;    // empty stack
    next = t->Next;
    if (CAS(&top,t,next)) break;   // lock-free
  }
  return t;
}
```

There might be a problem: we use a pointer to a node (`t->Next`), but that node may be freed meanwhile by another thread (in systems without garbage collection) – problem of **data persistence**.

In addition: the ABA-problem

RISC Processors – LL / SC

Are there atomic “read & write” instructions like XCHG or CAS on RISC processors too?

No – RISC instructions can **either read or write** but **not both read and write** in one single instruction.

Instead of atomic CAS –
RISC processors (e.g. MIPS) provide a pair of instructions:

LL – load linked (from a memory location to a register)

SC – store conditional (a register to a memory location)
when there was no write to that memory location
since the last LL (ideal SC ↔ practical, weak SC)
otherwise leaves memory untouched
returns a boolean result in a register

RISC Processors – LL / SC

RISC processors provide a pair of instructions: LL / SC

```
type - longword or pointer

type    LL(type * mem);
boolean SC(type * mem, type new);

void push(Node *node)
{ Node *t;

  while(true){
    t = LL(&top);               // t = top;
    node->Next = t;
    if (SC(&top,node)) break;  // CAS(&top,t,node))
  }
}
```

LL / SC – CAS – ABA-problem

RISC processors provide a pair of instructions: LL / SC

```
while(true){
  t = LL(&top);
  node->Next = t;
  if (SC(&top,node)) break;  // CAS(&top,t,node))
}
```

LL/SC is stronger than CAS:

in case top has changed from one value, say A to B and the back to A (ABA-problem)

CAS erroneously succeeds,
but **SC** fails (prevents the ABA-problem)

ABA-problem

Is ABA really a problem ? (the value has not changed)

Yes – of cause – the data structure may have changed.

Imagine, we have a stack:

top --> A --> B --> C --> /

thread1 - pop():

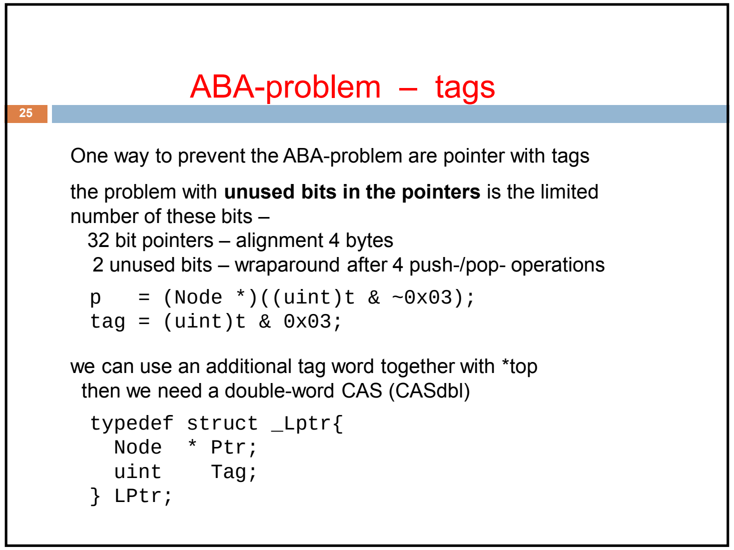
```
t = top;          // top  = &A
next = t->Next;   // next = &B
            // thread2: A=pop, B=pop, push A
            // top --> A --> C --> /
if (CAS(&top,t,next)) break;   // succeeds !
            // top --> B --> ??   -- Error !!
```

ABA-problem – short tags

One way to prevent the ABA-problem are pointer with tags (e.g. un-used bits in the pointers) which are incremented in each push or pop

```
void push(Node *node)
{ Node *t, *p;
  uint tag;           // unsigned int

  while(true){
    t = top;          // pointer + tag
    p   = (Node *)((uint)t & ~0x03);
    tag = (uint)t & 0x03; tag = (tag+1) & 0x03;
    node->Next = p;
    node = (Node *)((uint)node | tag);
    if (CAS(&top,t,node)) break;
  }
}
```



ABA-problem – tags

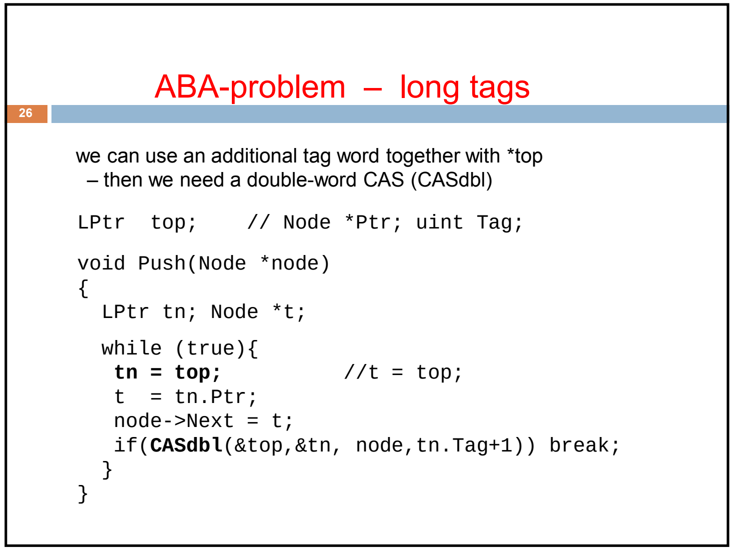
One way to prevent the ABA-problem are pointer with tags

the problem with **unused bits in the pointers** is the limited number of these bits –
 32 bit pointers – alignment 4 bytes
 2 unused bits – wraparound after 4 push-/pop- operations

```
p   = (Node *)((uint)t & ~0x03);
tag = (uint)t & 0x03;
```

we can use an additional tag word together with *top
 then we need a double-word CAS (CASdbl)

```
typedef struct _Lptr{
  Node  * Ptr;
  uint    Tag;
} LPtr;
```

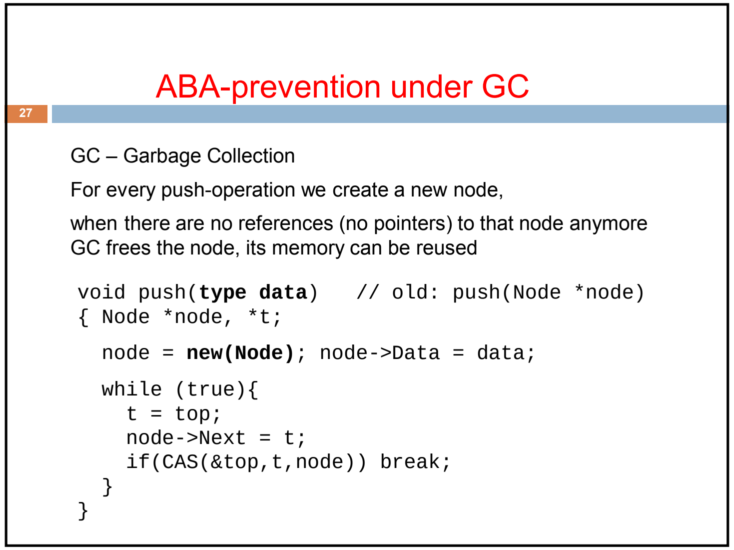


ABA-problem – long tags

we can use an additional tag word together with *top
 – then we need a double-word CAS (CASdbl)

```
LPtr  top;    // Node *Ptr; uint Tag;

void Push(Node *node)
{
  LPtr tn; Node *t;
  while (true){
   tn = top;          //t = top;
   t  = tn.Ptr;
   node->Next = t;
   if(CASdbl(&top,&tn, node,tn.Tag+1)) break;
  }
}
```



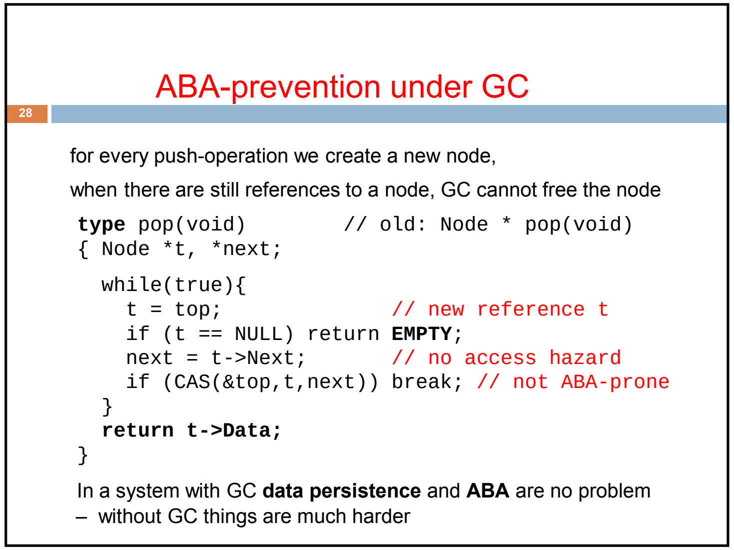
ABA-prevention under GC

GC – Garbage Collection

For every push-operation we create a new node,

when there are no references (no pointers) to that node anymore
GC frees the node, its memory can be reused

```
void push(type data)   // old: push(Node *node)
{ Node *node, *t;
  node = new(Node); node->Data = data;
  while (true){
    t = top;
    node->Next = t;
    if(CAS(&top,t,node)) break;
  }
}
```



ABA-prevention under GC

for every push-operation we create a new node,

when there are still references to a node, GC cannot free the node

```
type pop(void)        // old: Node * pop(void)
{ Node *t, *next;
  while(true){
    t = top;              // new reference t
    if (t == NULL) return EMPTY;
    next = t->Next;       // no access hazard
    if (CAS(&top,t,next)) break; // not ABA-prone
  }
  return t->Data;
}
```

In a system with GC **data persistence** and **ABA** are no problem
– without GC things are much harder

Reference Counter

Garbage Collection usually works with a **reference counter**

A data item can be freed not until there a no longer references to that item, i.e. not until the reference counter is zero.

We can include a reference counter (RC) in our nodes
– but there is a problem:

```
void ReleaseNode(Node *p)
{
  int rc = atomicDecrement(& p->RC);  // atomic!
  if (rc == 0) delete(p);    // Danger !!
}
```

danger: other threads can reserve the node just now, in the gap between `(rc==0)` and `delete()`, but it nevertheless will be deleted

Reference Counter

```
if(rc == 0) /*gap*/ delete(p);
```

danger: other threads can reserve the node just now, but it nevertheless will be deleted, while in use by the other threads

We might fix this, when we allow reservation only for nodes with RC greater than zero
– but there is a more serious problem:

```
Node * ReserveNode(Node *p)
{
  int rc = p->RC;     // access hazard !
  if (rc > 0) ...
```

In order to test and increment the RC for a pointer (reserve a node) we must use this pointer – but to use the pointer it must be reserved. We are in a doom loop, in a ”circulus vitiosus“.

GC – Reference Counter

In order to test and increment the RC for a pointer (reserve a node) we must use this pointer – but to use the pointer it must be reserved. We are in a doom loop, in a ”circulus vitiosus“.

But then – how does GC work? (in Java, C#, Haskell, …)

it usually is not lock-free

and often uses stop-the-world techniques (Detlefs)

The reason that garbage collectors commonly “stop the world” is that some of these pointers are in threads' registers and/or stacks,

discovering these requires operating system support, and is difficult to do concurrently with the executing thread

Excursion – atomicDecrement

```
rc = atomicDecrement(& p->RC);
```

In Intel processors there are (atomic) Increment- and Decrement-instructions (`inc, dec`), but they do not supply a result.

There is also an atomic ExchangeAndAdd instruction (lock xadd) that delivers the old value (`rc = ExchangeAdd(&p->RC,-1)-1;`)

We can easily built an `atomicDecrement` using CAS:

```
int atomicDecrement(int *mem, int val)
{ int old;
  while (true){
    old = *mem;
    if(CAS(mem,old, old-val)) break;
  }
  return old-val;
}
```

Hazard Pointers

In order to built an own lock-free GC-like environment
and in consideration of the difficulties managing reference counters

M. M. Michael introduced Hazard Pointers:

```
Node * pop(void)
{ Node *t, *next;

  while(true){
    t = top; if (t == NULL) break;
    hp[thr] = t;
    if (t != top) continue;
    next = t->Next;         // no access hazard
    if (CAS(&top,t,next)) break;
  }
  return t;
}
```

Hazard Pointers

M. M. Michael introduced Hazard Pointers (one or two for each tread)

```
Node * pop(void)
{
    t = top; if (t == NULL) break;
    hp[thr] = t;   ...
    next = t->Next;    // no hazard
    ...
```

Caller:

```
Node * A = pop(); if (A == NULL) break;
data = A->Data;
hp[thr]=NULL; DeleteNode(A);
```

Hazard Pointers – DeleteNode

```
hp[thr]=NULL; DeleteNode(A);

int dcount = 0;     // static, per thread
Node * dlist[R];    // or a linked list

void DeleteNode(Node *node)
{
  dlist[dcount++] = node;
  if (dcount == R) Scan(dlist);
}
```

Scan first collects all non-null hazard pointers in a local data structure
(e.g. a hash table),
then checks each node in `dlist` against these hazard pointers;
if there is no match the node is deleted,
otherwise node remains in `dlist` until a subsequent Scan

FIFO Queue

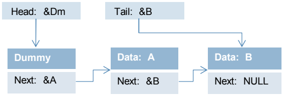
Two entry points (pointers): Node ***Head**, ***Tail**;

To avoid special cases (the empty queue)
the queue always includes a **dummy node** as the first node

Introduced by Michael and Scott (→ the MS-queue)
included in the standard JavaTM Concurrency Package (JSR-166)
correctness – linearizability proof by L. Groves

We **enqueue** at the **tail** (after the so far last node)

we **dequeue** at the **head** (unless the queue is empty)
we read the next node
after the dummy,
then this node becomes
the new dummy



FIFO Queue

We **enqueue** data at the **tail**

we create a new Node:

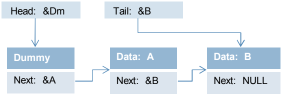
```
Node * node = new(Node);
node->Data  = data;
node->Next  = NULL;     // important!
```

to enqueue this node we have to **change two pointers**:

first – the Next-field of the so far last node (now NULL)

second – Tail

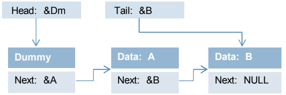
(not possible
in one single
atomic step)



FIFO Queue

To enqueue a node we have to **change two pointers**: Next and Tail
A first (incomplete) routine looks like this:

```
    void Enqueue(Type data)
    { Node *node, *t, *next;
1:    node = new(Node);
      node->Data = data; node->Next = NULL;
      while(true){
2:      t = Tail;
4:      if (CAS(&t->Next,NULL,node)) break;
      }
5:    CAS(&Tail,t,node);
    }
```



FIFO Queue

```
    while(true){
2:    t = Tail;
      next = t->Next;
3:    if (next != NULL) {CAS(&Tail,t,next); continue}
4:    if (CAS(&t->Next,NULL,node)) break;
    }
5:  CAS(&Tail,t,node);
```

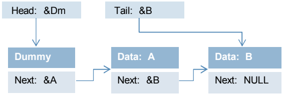
If one thread has performed step 4, but not yet step 5 (when it is blocked)
other threads cannot succeed in step 4 (t->Next != NULL)
→ the algorithm (so far – without step 3) is **not lock-free** !

To repair this, threads must be able to adjust Tail
(step 3 in our tread instead of step 5 in the blocking thread)
– our thread assists the obstructing tread

FIFO Queue

The complete, lock-free routine:

```
void Enqueue(Type data)
{ Node *node, *t, *next;
  node = new(Node);
  node->Data = data; node->Next = NULL;
  while(true){
    t = Tail;
    next = t->Next;
    if (next!=NULL) {CAS(&Tail,t,next); continue}
    if (CAS(&t->Next,NULL,node)) break;  //lin. point
  }
  CAS(&Tail,t,node);
}
```

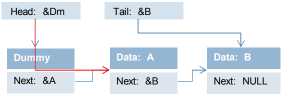


FIFO Queue

To dequeue we (usually) change only one pointer - Head
(step 3 is analogous to step 3 in Enqueue)

```
    Type Dequeue(void)
    {  Node *h, *t, *next; Type data;

       while(true){
         h = Head; t = Tail;
1:       next = h->Next;
2:       if (next == NULL) return EMPTY;
3:       if (h == t) { CAS(&Tail,t,next); continue}
         data = next->Data;      // next behind dummy
4:       if (CAS(&Head,h,next)) break;  // new dummy
       }
       return data;
    }
```



References, Shortlist

Nir Shavit
Data Structures in the Multicore Age
Communications of the ACM, Vol. 54, No. 3, March 2011, pp. 76-84

Mark Moir, Nir Shavit (Sun Microsystems Laboratories)
Concurrent Data Structures
Sun Microsystems Laboratories, CRC Press, 2001
32 p., 138 references!

Maged M. Michael (IBM, T.J. Watson Research Center)
Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects
IEEE Transactions on Parallel and Distributed Systems, Vol. 15, No. 6,
June 2004, pp. 491-504

David L. Detlefs, Paul A. Martin, Mark Moir, Guy L. Steele Jr. (Sun Laboratories)
Lock-Free Reference Counting
Proc. 20th Ann. ACM Symp. Principles of Distributed Computing, Aug. 2001

References, Shortlist

Maged M. Michael, Michael L. Scott
Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms
Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC ’96), New York, USA, ACM (1996)
pp. 267-275

Lindsay Groves
Verifying Michael and Scott’s Lock-Free Queue Algorithm using Trace Reduction
Computing: The Australasian Theory Symposium (CATS2008), Wollongong Australia 2008

M. M. Michael
High Performance Dynamic Lock-Free Hash Tables and List-Based Sets
Proceedings of the 14th annual ACM Symposium on Parallel Algorithms and Architectures, 2002. ACM Press, 2002. pp. 73–82.

References, Shortlist

Michael Spiegel, Paul F. Reynolds Jr.
Lock-Free Multiway Search Trees
39th International Conference on Parallel Processing, 2010
pp. 604-613

Maurice Herlihy (Digital Equipment Corporation, now Brown University, Rhode Island)
Wait-Free Synchronization
ACM Transactions on Programming Languages and Systems,
Vol. 11, No 1, January 1991, pp. 124-149

A fundamental paper on CAS etc. – Dijkstra Prize !

All the papers can be found as pdf-files in the internet.