Other Approaches

Arnav Gupta

December 10, 2024

Contents

1 Atomic (Lock-Free) Data Structure

Lock free data structure have operations, which are critical sections, but performed without **ownership**.

For example, add/remove node without any blocking duration (operation takes constant atomic time).

Lock-free is still locking since it spins for conceptual lock, and so it busy waits (starvation).

If eventual progress guaranteed, called **wait free**.

1.1 Compare and Set Instruction

Compare-and-set (assign) instruction performs atomic compare and conditional assignment (CAS — compare and swap):

- if compare/assign returns true, lock stops and lock is set to closed
- if compare/assign returns false, lock executes until other thread sets lock to open

```
bool CAS( int & val, int comp, int nval ) {
  // begin atomic
  if ( val == comp ) {
    val = nval;
    return true;
  }
  return false;
  // end atomic
}
```

Alternative implementation assigns comparison value with value when not equal.

```
bool CAS( int & val, int & comp, int nval ) {
  // begin atomic
  if ( val == comp ) {
    val = nval;
    return true;
  }
  comp = val;
  return false;
```

```
  // end atomic
}
```

Assignment when unequal useful to restart operations with new changed value.

1.2 Lock-Free Stack

Build stack with lock-free `push` and `pop` operations.

Use CAS to automatically update `top` pointer when nodes pushed or popped concurrently.

```
void Stack::push( Node & n ) {
  for ( ;; ) {
    n.next = top;
    if ( CAS( top, n.next, &n ) ) {
      break;
    }
  } // top = &n
}

Node * Stack::pop() {
  Node * t;
  for ( ;; ) {
    t = top;
    if ( t == nullptr ) return t; // empty list
    if ( CAS( top, t, t->next ) ) {
      return t;
    }
  } // top = t->next
}
```

1.3 ABA Problem

Pathological failure for series of pops and pushes.

Issue occurs when time is sliced before CAS but after getting `t->next`.

Causes corrupted stack.

1.4 Hardware Fix

Probabilistic solution for stack exists using double-wide CAVD instruction, which compares and assigns 64/128-bit values for 32/64-bit architectures.

```
bool CAVD( uintS_t &val, uintS_t &comp, uintS_t nval ) {
  // begin atomic
  if ( val == comp ) {
    val = nval;
    return true;
  }
  comp = val;
  return false;
  // end atomic
}
```

Associate counter (ticket) with header node and increment counter in `push`, so `pop` can detect ABA if node re-pushed.

Ticket strategy only probabilistic correct since counter is finite, for the case of the counter wrapping around.

No CAS program ensures eventual progress, so rule 5 broken.

1.5 Hardware/Software Fix

Fixing ABA with CAS/V and more code is complex, as well as implementing more complex data structures.

All solutions require complex determination of when a node has no references (like garbage collection):

- each thread maintains a list of accessed nodes, called **hazard pointers**
- thread updates hazard pointers while threads are reading them
- thread removes a node by hiding it on a private list and periodically scans hazard lists of other threads for references to that node
- if no pointers found, node can be freed

Lock-free has no ownership (hold-and-wait) and so no deadlock.

Lock-free can only handle limited set of critical sections. Lock can protect arbitrarily complex critical section.

Lock-free is no cure, and performance is unclear.

Best may be to combine lock and lock-free.

2 Exotic Atomic Instruction

VAX computer has instructions to atomically insert and remove a node to/from the head/tail of a circular doubly linked list.

MIPS processor has 2 instructions that generalize atomic read/write cycle: `LL` (load locked) and `SC` (store conditional)

- `LL` loads (reads) a value from memory into a register, and sets a hardware **reservation** on the memory from which the value is fetched
  - register value can be modified, even moved to another register
- `SC` instruction stores (writes) new value back to original or another memory location
  - store is conditional and occurs only if no interrupt, execution, or write has occurred at `LL` reservation
  - failure indicated by setting register containing value to be stored to 0
- using `LL` and `SC` does not suffer from the ABA problem

Most architectures support weak LL/SC:

- reservation granularity may be cache line or memory block rather than word
- no nesting or interleaving of LL/SC pairs
- prohibit memory access between `LL` and `SC`

Cannot implement atomic swap of 2 memory locations as 2 reservations are necessary (register to memory swap is possible)

Hardware transactional memory allows 4, 6, 8 reservations.

Like DB **transaction** that optimistically executed change, and either commits changes or rolls back and restarts if interference:

- `SPECULATE`: start speculative region and clear zero flag, next instruction checks for abort and branches to retry

- `LOCK`: `MOV` instructions indicate location for atomic access, but moves not visible to other CPUs
- `COMMIT`: end speculative region
  - if no conflict, make `MOV` instructions visible to other CPUs
  - if conflict to any move locations, set failure, discard reservations, and restore registers back to instruction following `SPECULATE`

Can implement several data structures without ABA problem.

Software Transactional Memory (STM) allows any number of reservations:

- atomic blocks of arbitrary size
- records all memory locations read and written, and all values mutated
  - bookkeeping costs and rollbacks typically result in performance degradation
- alternative implementation inserts locks to protect shared access
  - finding all access is difficult and ordering lock acquisition is complex

3 General-Purpose GPU (GPGPU)

Graphical Processing Unit (GPU) is a coprocessor to main computer, with separate memory and processors.

GPU is Single-Instruction Multiple-Data (Thread) (SIMDT) architecture vs usual Multiple-Instruction Multiple-Data.

For branching code, all threads test the condition (create mask of true and false):

- `true` threads execute "then" instructions, false threads execute NOP
- `false` threads execute "else" instructions, true threads execute NOP

Critical path is time to execute both clauses of `if` (no speedup).

Complex contortions to eliminate different forms of branching.

GPU structure:

- kernel manages multiple blocks (loaded/controlled by CPU)

- block executes the same code
  - may be barrier-synchronized
  - synchronization among blocks is finishing kernel and launching new one
- warp synchronizes execution (one instruction decoder per warp)
- thread computes value

Instead of cache to optimize latency in warp, large register file is used to optimize throughput. GPUs have enough duplicate registers to store state of several warps.

Kernel is memory-bound, so data layout extremely important for performance consideration

Warps scheduled to run when required data loaded from memory.

CPU sets up GPU memory, loads memory, launches code, and retrieves results.

Most modern multi-core CPUs have similar model using vector-processing. Can simulate warps and use concurrency framework to simulate blocks.

4 Concurrency Languages

4.1 Ada 95

Provides restricted implicit (automatic) signal.

`when` clause only to be used at start of entry routine, not within.

`when` expression can contain only global object variables. Parameter or local variables are disallowed (no direct service).

Direct service is only possible when restrictions are eliminated.

Provides `task` type with task main and declarations. Allows for Accepts and `when` guards with mutex members.

`select` is external scheduling and only appears in `task` main. Ada has no internal scheduling mechanism (no condition variables).

`requeue` statement can be used to make blocking call to another (usually non-public) mutex member of the object. Original call is re-blocked on that

mutex member's entry queue, which can be subsequently accepted when it is appropriate to restart it.

All `requeue` techniques suffer the problem of dealing with accumulated temporary results:

- if a call must be postponed, its temporary results must be returned and bundled with initial parameters before forwarding to the mutex member handling the next step
- or temporary results must be re-computed at the next step (if possible)

In contrast, waiting on a condition variable automatically saves execution location and any partially computed state.

4.2 SR/Concurrent C++

SR and concurrent C++ have tasks with external scheduling using an accept statement, but no condition variables or requeue statement.

To ameliorate lack of internal scheduling, add a `when` and `by` clauses on the `accept` statement.

`when` clause is allowed to reference caller's arguments via parameters of mutex members. Done by placing `when` after `accept` clause so param names are defined.

`when` referencing parameter means implicit search of waiting tasks on mutex queue, and so locking mutex queue.

Select longest waiting if multiple true `when` clauses.

`by` clause calculated for each true `when` clause and the minimum `by` clause is selected.

Select longest waiting if multiple `by` clauses with same minimum.

`by` clause exacerbates execution cost of computing `accept` clause

While `when` and `by` remove some internal scheduling and/or requeues, constructing expressions can be complex.

Still exist situations that cannot be handled, like if selection criteria involves multiple params:

- selection criteria involves information from other mutex queues

Often simplest to unconditionally accept a call allowing arbitrary examination, and possibly postpone (internal scheduling).

4.3 Java

Concurrency constructs largely derived from Modula-3.

```
class Thread implements Runnable {
    public Thread();
    public Thread(String name);
    public String getName();
    public void setName(String name);
    public void run(); // uC++ main
    public synchronized void start();
    public static Thread currentThread();
    public static void yield();
    public final void join();
}
```

`Thread` is like `uBaseTask`, and all tasks must explicitly inherit from it. Thread starts in member `run`.

Java requires explicit starting of a thread by calling `start` after thread's declaration. Coding convention is to start thread or inheritance is precluded (can only start thread once).

Termination synchronization accomplished by calling `join`.

Returning result on thread termination is accomplished by members returning values from the task's global variables.

Like μC++, when the task's thread terminates, it becomes an object, allowing the call to `result` to retrieve a result.

While it is possible to have public `synchronized` members of a task, there is no mechanism to manage direct calls (no accept statement). So this requires complex emulation of external scheduling with internal scheduling for direct communication.

4.4 Go

Non-object-oriented, light-weight non-preemptive threads (called goroutines).

Has cooperative scheduling, so implicitly inserts yields at safe points (not interrupt based). Busy waiting only on multicore.

`go` statement (like start/fork) creates new user thread running in routine.

Arguments may be passed to goroutine, but return value discarded.

Cannot reference goroutine object since no direct communication.

All threads terminate silently when program terminates.

Threads synchronize/communicate via **channel** (CSP), so different from routine call.

Channel: typed shared buffer with 0 to N elements:

```
ch1 := make( chan int, 100 ) // integer channel with buffer size 100
ch2 := make( chan string )  // string chnanle with buffer size 0
ch3 := make( chan chan string ) // channel of channel of strings
```

If buffer size more than 0, up to N asynchronous calls, otherwise, synchronous call.

Operator `<-` performs send/receive, so:

- send: `ch1 <- 1`
- receiver: `s <- ch2`

Channel can be constrained to only send and receive, otherwise bi-directional. More like futures and $\tilde{}_{\text{Select}\_}$, and asynchronous call.

Also has mutual exclusion locks, synchronization locks, singleton-pattern locks, readers/writers locks, and countdown locks.

Also has atomic operations for:

- adding signed and unsigned integers and pointers to integers
- compare and swap signed and unsigned integers and pointers to integers
- load and store signed and unsigned integers and pointers to integers

4.5 C++11 Concurrency

C++11 library can be sound as C++ now has strong memory model (SC).

Compile with

```
g++ -std=c++11 -pthread ...
```

Thread creation uses start/wait (fork/join) approach:

```
class thread {
  public:
    template <class Fn, class ... Args>
      explicit thread( Fn && fn, Args &&... args );
    void join(); // termination synchronization
    bool joinable() const; // true => joined, else false
    void detach();  // independent lifetime
    id get_id const; // thread id
};
```

Passing multiple arguments using C++11's variadic template feature to provide a type-safe call chain via thread constructor to the callable routine.

Any entity that is callable (functor) may be started:

Thread starts implicitly at point of declaration. Instead of `join`, thread can run independently by detaching.

Beware dangling pointers to local variables. Deallocating thread object before join or detach is an error.

Usable locks include mutual exclusion locks (mutex, recursive, timed, recursive-timed) and condition variables.

```
class mutex {
  public:
    void lock(); // acquire
    void unlock(); // release
    bool try_lock(); // nonblocking acquire
};

class condition_variable {
  public:
    void notify_one(); // unblock one
    void notify_all(); // unblock all
    void wait( mutex &lock ); // atomic block + release lock
};
```

Scheduling is no-priority nonblocking, so barging. This means `wait` statements must be in while loops to recheck conditions.

Also has futures, with `get` member to block and get answer and `async` call with function and params.

Also has atomic types and operations. Atomic types include flag, bool, char (signed and unsigned as well), short (signed and unsigned as well), int (signed and unsigned as well), long/llong (signed and unsigned as well), wchar, address, and template atomic type.

Atomic types allow for most C++ operations to be atomic including increment, decrement, increase, decrease, `&=`, `|=`, `^=`, store, load, exchange, set, compare/exchange, and fetch add/sub/and/or/xor.

5 Threads and Locks Library

5.1 java.util.concurrent

Java library is sound because of memory model and language is concurrent aware.

Synchronizers include `Semaphore` (counting), `CountDownLatch`, `CyclicBarrier`, `Exchanger`, `Condition`, `Lock`, and `ReadWriteLock`.

Can use locks to build a monitor with multiple condition variables.

`Condition` is nested class within `ReentrantLock`, so condition implicitly knows its associated (monitor) lock.

Scheduling still no-priority non-blocking (barging), so `wait` statements must be in `while` loops to recheck condition.

No connection with implicit condition variable of an object.

Do not mix implicit and explicit condition variables.

Executor/Future are both actor-like:

- executor is a server with 1+ worker tasks (worker pool)
- future is a closure with work for executor (callable) and place for result
- call to executor `submit` is asynchronous and returns a future
- result retrieved using `get` routine, which may block until result inserted by executor

μC++ also has fixed thread-pool executor (used with actors).

Also has collections for different types of synchronous and blocking queues, maps, sets, and list. Can create threads that interact indirectly through atomic data structures.

Also has atomic types using compare-and-set (lock-free). Includes atomic versions of boolean, integer, int array, long, long array, reference (templated), and reference array (templated).

5.2 Pthreads

C libraries built around routine abstraction and mutex/condition locks:

```
int pthread_create( pthread_t * new_thread_ID,
                    void * (*start_func)(void *),
                    void * arg);
int pthread_join( pthread_t target_thread,
                  void ** status );
pthread_t pthread_self( void );
int pthread_yield( void );

// mutex lock and cond vars also available
```

Thread starts in routine `start_func` via `pthread_create`, where initialization is single `void *` value.

Termination synchronization is performed by calling `pthread_join`. Return a result on thread termination by passing back a single `void *` value from `pthread_join`.

All C library approaches have type-unsafe communication with tasks.

No external scheduling, so complex direct-communication emulation.

Internal scheduling is no-priority non-blocking (barging), so `wait` statements must be in while loops to recheck conditions.

Explicit calls are necessary to `ctor` and `dtor` before/after use of pthreads since no constructors or destructors in C.

All locks must be initialized and finalized.

Mutual exclusion must be explicitly defined where needed. Condition locks should only be accessed within mutual exclusion.

`pthread_cond_wait` atomically blocks thread and releases mutex lock, which is necessary to close race condition on baton passing.

6 OpenMP

Shared memory, implicit thread management (programmer hints), 1 to 1 threading model (kernel threads), and some explicit locking.

Communicate with compiler with `#pragma` directives.

Uses fork/join model:

- fork: initial thread creates a team of parallel threads (including itself)
- each thread executes the statements in the region construct
- join: when team threads complete, synchronize, and terminate, except initial thread which continues

Can do COBEGIN/COEND where each thread executes a different section using `#pragma omp section` after defining how many threads to use in the following parallel sections.

`for` directive specifies loop iteration executed by a team of threads (CO-FOR). In this case, sequential code is directly converted to concurrent with the pragma.

Variable outside a section are shared, variables inside are thread private. Programmer is responsible for sharing in vector/matrix multiplication.

Can use barrier with `barrier` directive after defining threads (no section). Without `omp` section, all threads run same block (like parallel for).

Barrier's trigger is the number of block threads.

Also has critical section and atomic directives.