Lock-free (wait-free) synchronization

EECE 494

Sathish Gopalakrishnan

Outline

- Locks can be problematic.
- How do we achieve synchronization without locks?
  - Scheduling
  - Non-blocking synchronization

Eliminating locks

- One use of locks is to coordinate multiple updates of single piece of state. How to remove locks here?
  - **Duplicate state so each instance only has a single writer.**
  - (Assumption: assignment is atomic.)

- Circular buffer
  - *Why do we need a lock for a circular buffer?*
  - To prevent loss of update to buf.n (buffer size). No other reason.
  - What is buf.n good for?
    - Signaling buf full and empty.
  - How else to check this?
    - Full: (buf.head - buf.tail) ==N
    - Empty: buf.head == buf.tail
- *Can we use these facts to eliminate locks in get/put?*

Lock-free sync (1 producer/1 consumer)

```
int head = 0, tail = 0;
char buf[N];
void put(char c) {
        while((buf.head - buf.tail) == N)
                wait();
        buf.buf[buf.head % N] = c;
        buf.head++;
        }

void get(void) {
        char c;
        while(buf.tail == buf.head)
                wait();
        c = buf.buf[buf.tail % N];
        buf.tail++;
        return c;
}
```

All shared variables have single writer (no lock needed):

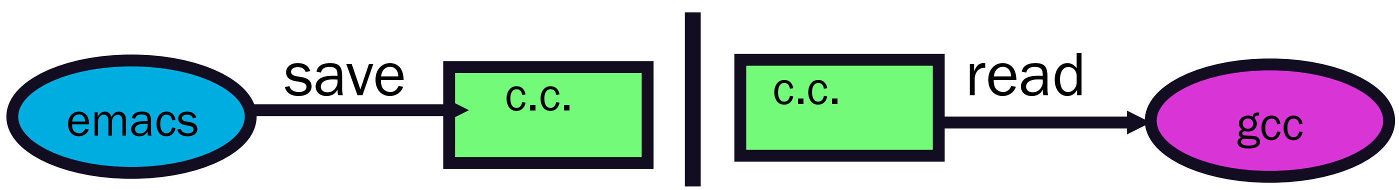
- head – producer
- tail – consumer
- buffer:
    - head != tail then no overlap and
    - buf[head] – producer
    - buf[tail] – consumer
- head = tail then
    - empty and consumer
    - waits until head != tail

invariants:

- not full: once not full true, can
    - only be changed by producer
- not empty: once not empty can
    - only be changed by consumer

Locks vs. Explicit Scheduling

- Race condition = bad interleaving of processes.
  - We've used locks to prevent bad interleaving
  - could use scheduler to enforce legal schedules.
- Examples:
  - doctor's appointment vs. emergency room
  - dinner reservation vs. showing up
  - run processes sequentially vs. acquire locks
- Tradeoffs?



Non-blocking synchronization

- How about getting correct interleaving by detecting and retrying when a bad interleaving occurred?
  - Don’t need locks to synchronize.
- Example: hits = hits + 1;
  - **A) Read hits into register R1.**
  - **B) Add 1 to R1 and store it in R2.**
  - **C) Atomically store R2 in hits only if hits==R1. (i.e. CAS)**
    - **If store didn’t write goto (A)**
- Can be extended to any data structure:
  - **A) Make copy of data structure, modify copy.**
  - **B) Use atomic word compare-and-swap to update pointer.**
  - **C) Goto A if some other thread beat you to the update.**

Non-blocking synchronization

- Other names:
  - Wait free synchronization, Lock free synchronization.
  - Optimistic concurrency control.
  - Initial work by Leslie Lamport, Maurice Herlihy.
- Modern machine have support for it:
  - **x86 CMPXCHG, CMPXCHG8B** – Compare and Exchange.
  - Someone wrote an entire OS with no locks!
- Useful properties:
  - Synchronizes with interrupt handlers.
  - Remove overhead (CPU/memory) locks.
  - Deals with failures better (e.g., process dies with locks).
- Issues:
  - Lots of retrying under high load.

A Lock-Free Multiprocessor OS Kernel

Henry Massalin and Calton Pu
Columbia University
June 1991

Where are we?

- Just to ground ourselves: Welcome to 1991
- Previously - Synthesis V.0
  - Uniprocessor (Motorola 68020)
  - No virtual memory
- 1991 - Synthesis V.1
  - Dual 68030s, virtual memory, supports threads
  - A significant step forward
  - A fairly ground-up OS

Example: Stack code

```
Pop() {
  retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if (CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
  return elem;
}
```

Stack code (Clarification)

```
Pop() {
  retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if (CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
  return elem;
}
```

local variables – can't change underneath us

"global" (at least to the data structure) …other threads can mutate this at any time

Stack code (Stages)

1. Write down preconditions
2. Do computation
3. Commit results (or retry)

```
Pop() {
  retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if (CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
  return elem;
}
```

- Specifically in the work by Massalin and Pu:
  - Saved state is only one or two words
  - Commit is done via Compare-and-Swap (CAS) or Double Compare-and-Swap (DCAS)
- Only two words?
  - They claimed that all OS synchronization problems can be solved while needing to touch only two words at a time
  - Impressive?

Massalin & Pu's approach

- Build data structures that work concurrently.
  - Stacks, queues, linked lists, …
- Then, build the OS around these data structures.

- If all else fails:
  - Create a single "server" thread for the task
    - Callers then…
      1. Pack the requested operation into a message
      2. Send it to the server (using lock-free queues)
      3. Wait for response/callback/…
  - The queue effectively serializes operations.

Impact

- This paper spawned a lot of research on DCAS.
  - Improved lock-free algorithms
  - The utility of DCAS
- DCAS is not supported on modern hardware.
  - "DCAS is not a silver bullet for non-blocking algorithm design."

What should you know?

- What is lock-free (wait-free) synchronization?
- How can it be achieved?
- What special instructions are needed?

- Read for better background:
  - **Wait-Free Synchronization**, Maurice Herlihy. ACM Transactions on Programming Languages and Systems, Vol. 13, No. 1, January 1991 (pages 124—149).
  - **A Lock-Free Multiprocessor OS Kernel**, Henry Massalin and Calton Pu. Operating Systems Review, Vol. 26, No. 2, 1992.
  - **Real-Time Computing with Lock-Free Shared Objects**, James H. Anderson, Srikanth Ramamurthy and Kevin Jeffay. IEEE Real-Time Systems Symposium, December 1995 (pages 28—37).

Conclusions

- Optimistic synchronization is effective...
  - when contention is low
  - when critical sections are small
  - as locking costs go up
- It's possible to build an entire OS without locks.
- Optimistic techniques are still applicable
  - though the implementation (DCAS) is not.