1 Locking a Linked List

Typical OS data structure code
Prepend an element to a linked list:

\[
\begin{align*}
\text{node->next} &= \text{head}; \\
\text{head} &= \text{node};
\end{align*}
\]

Potential race condition

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{node-&gt;next} = \text{head};</td>
<td>\text{node-&gt;next} = \text{head};</td>
</tr>
<tr>
<td>\text{head} = \text{node};</td>
<td>\text{head} = \text{node};</td>
</tr>
</tbody>
</table>

Disable interrupts

\[
\begin{align*}
\text{local_irq_disable;}
\text{node->next} &= \text{head}; \\
\text{head} &= \text{node};
\text{local_irq_enable;}
\end{align*}
\]

- Incredibly cheap
- Couple of instructions

Problems with disabling interrupts

- Only for short critical sections
- Leave them off too long:
- Losing hardware interrupts
• “Why does my clock run slow?”
• Coarse-grained (only one “lock”)
• Insufficient for multiprocessors
• Still need real synchronization

Disable software interrupts

```c
local_bh_disable();
node->next = head;
head = node;
local_bh_enable();
```

• Queue up interrupts
• Handle when re-enabled
• Don’t lose hardware interrupts

Problems with disabling software interrupts

• More expensive
• Queue maintenance
• Doesn’t protect against access from hardware interrupts
• Still only for short critical sections
• Can cause process hangs if not released
• Coarse-grained (still only one “lock”)
• Still insufficient for multiprocessors
• Still need real synchronization

Busywaiting with a spinlock

```c
static DEFINE_SPINLOCK(list_lock);
spin_lock(&list_lock);
node->next = head;
head = node;
spin_unlock(&list_lock);
```

• Assumes every thread has a processor
• Nothing better to do than wait
• Spinning impacts other processors
• Variants reduce impact, add queuing overhead
Blocking with a semaphore

```c
static DECLARE_MUTEX(list_lock);
while(down_interruptible(&list_lock) == -EINVAL)
;
node->next = head;
head = node;
up(&list_lock);
```

- Queuing overhead
- Goes through scheduler
- Can block

Potential Correctness Problems with Mutual Exclusion

- Deadlocks
- Priority Inversion

Locking summary

- Provides correctness (if used correctly)
- Adds overhead
- Causes problems
- Various different tradeoffs
- Alternatives?
- Shared data structures need synchronization, right?

2 Implementing a Spinlock

Spinlock implementation

```c
void spin_lock(spinlock_t *lock)
{
    while(lock->counter != 0)
    ;
    lock->counter = 1;
}
```
Potential race condition

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>while(lock-&gt;counter != 0)</code> ;</td>
<td><code>while(lock-&gt;counter != 0)</code> ;</td>
</tr>
<tr>
<td><code>lock-&gt;counter = 1;</code></td>
<td><code>lock-&gt;counter = 1;</code></td>
</tr>
</tbody>
</table>

Fixing this race condition?

- Can’t just disable interrupts: insufficient for multiprocessors
- Can’t use semaphores: we don’t want to block

Locking the lock

```c
void spin_lock(spinlock_t *lock)
{
    spin_lock(lock->lock);
    while(lock->counter != 0)
    {
        lock->counter = 1;
    }
    spin_unlock(lock->lock);
}
```

Something wrong here!

Fixing the semaphore

- Alternatives?
- Shared data structures need synchronization, right?

### 3 Atomic Instructions

Atomic instructions

- One instruction, indivisible
- Bus locking protocols
- No interleaving
- What do we have available?
Test and set

- Set value to 1, return previous value

```c
void spin_lock(spinlock_t *lock)
{
    while(atomic_test_and_set(lock->counter))
        ;
}
```

Exchange

- Swap two values atomically
- Can implement test and set with exchange:
  - Exchange with variable containing 1
  - Return variable

Synchronization from atomic instructions

- Fundamental synchronization mechanism
- Build synchronization abstractions on them
- Build OS data structures on the synchronization abstractions

Abstractions?

From the Exokernel paper:

The exokernel architecture is founded on and motivated by a single, simple, and old observation: the lower the level of a primitive, the more efficiently it can be implemented, and the more latitude it grants to implementors of higher-level abstractions.

Using atomic instructions directly?

- Skip implementing synchronization abstractions
- Build OS data structures on atomic instructions
4 Linked list revisited

Linked list revisited
Prepend an element to a linked list:

```c
node->next = head;
head = node;
```

- What would work here?
- Atomic arithmetic doesn’t help
- Atomic test and set doesn’t help
- Would atomic exchange work?

Linked list with exchange?

```c
node->next = node;
atomic_exchange(node->next, head)
```

- This won’t work.
- It references two memory operands
- Atomic exchange can only handle one memory operand

How to fix linked list?

- Can’t reference two memory operands
- Need to change head in memory
- Need to verify head still has previous value
- If we load head separately, we can race
- How can we do this with one instruction?

Compare and Swap
Atomically:

```c
CAS(register old, register new, memory location)
{
    if(*location == old)
    {
        *location = new;
        return SUCCEED;
    }
    return FAIL;
}
```
Lock-free linked list

do
{
    node->next = old_head = head;
} while(CAS(old_head, node, &head) == FAIL);

5 Synthesis

Synthesis

- Completely lock-free operating system
- Data structures and OS services build on compare and swap

Lock-free vs wait-free

- Lock-free not the same as wait-free
- Lock-free, or non-blocking, allows starvation
- Wait-free prevents starvation
- Wait-free has higher cost
- Authors argued improbability of starvation

Lock-free structures in Synthesis

- Lists: insertion, deletion, traversal
- Recognizes the problem of freeing elements currently in use
- Uses reference counts to defer destruction
- Stacks
- Queues

Lock-free OS abstractions in Synthesis

- Threads
- Virtual memory
- Console I/O
- File system I/O
Lock-free threading

- Each thread has a Thread Table Entry (TTE)
- Run queue for each priority
- Fixed number of run queues
- Allocates timeslices to queues in defined pattern (0,1,0,2,0,1,0,3,...)
- Uses compare and swap to mark a TTE as in-use
- Other CPUs skip in-use TTEs
- Thread operations (suspend, resume, signalling) use lock-free flags in the TTEs

Compare and Double Swap

- Two compare and swap instructions in one
- Two old values, two new values, and two memory locations
- Used for several key algorithms in Synthesis:
  - List element deletion
  - List traversal with reference counts
  - Stack array push
  - Thread signalling
- Not supported on any modern architecture

6 Summary

Summary

- Need to protect shared OS structures
- Synchronization abstractions build on atomic instructions
- Cut out the middleman
- Implement OS structures with atomic instructions
- Achieves good performance
- Compare and swap works well
- Compare and double swap helps, but not generally available