

# CS510 Concurrent Systems

Jonathan Walpole



# A Lock-Free Multiprocessor OS Kernel

# The Synthesis Kernel

A research project at Columbia University

## Synthesis V.0

- Uniprocessor (Motorola 68020)
- No virtual memory

## 1991 - Synthesis V.1

- Dual 68030s
- virtual memory, threads, etc
- Lock-free kernel

# Locking

Why do kernels normally use locks?

Locks support a concurrent programming style based on mutual exclusion

- Acquire lock on entry to critical sections
- Release lock on exit
- Block or spin if lock is held
- Only one thread at a time executes the critical section

Locks prevent concurrent access and enable sequential reasoning about critical section code

# Why Not Use Locking?

## Granularity decisions

- Simplicity vs performance
- Increasingly poor performance (superscalar CPUs)

## Complicates composition

- Need to know the locks I'm holding before calling a function
- Need to know if its safe to call while holding those locks?

## Risk of deadlock

## Propagates thread failures to other threads

- What if I crash while holding a lock?

# Is There an Alternative?

Use lock-free, “optimistic” synchronization

- Execute the critical section unconstrained, and check at the end to see if you were the only one
- If so, continue. If not roll back and retry

Synthesis uses no locks at all!

Goal: Show that Lock-Free synchronization is...

- Sufficient for all OS synchronization needs
- Practical
- High performance

# Locking is Pessimistic

Murphy's law:

- "If it can go wrong, it will..."

In concurrent programming:

- "If we can have a race condition, we will..."
- "If another thread could mess us up, it will..."

Solution:

- Hide the resources behind locked doors
- Make everyone wait until we're done
- That is...if there was anyone at all
- We pay the same cost either way

# Optimistic Synchronization

The common case is often little or no contention

- Or at least it should be!
- Do we really need to shut out the whole world?
- Why not proceed optimistically and only incur cost if we encounter contention?

If there's little contention, there's no starvation

- So we don't need to be "wait-free" which guarantees no starvation
- Lock-free is easier and cheaper than wait-free

Small critical sections really help performance



# How Does It Work?

## Copy

- Write down any state we need in order to retry
- 

## Do the work

- Perform the computation

## Atomically “test and commit” or retry

- Compare saved assumptions with the actual state of the world
- If different, *undo work*, and *start over* with new state
- If preconditions still hold, commit the results and continue
- This is where the work becomes visible to the world (ideally)

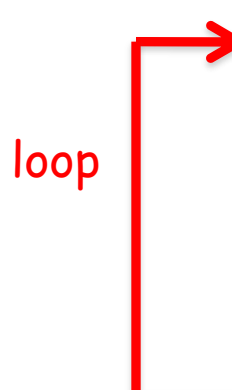
# Example – Stack Pop

```
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;  
}
```

# Example – Stack Pop

```
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;  
}
```

loop



# Example – Stack Pop

```
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;  
}
```

*Locals -  
won't change!*

*Global - may  
change any  
time!*

*"Atomic"  
read-modify-write  
instruction*


# CAS

## CAS – single word Compare and Swap

- An atomic read-modify-write instruction
- Semantics of the single atomic instruction are:


```
CAS(copy, update, mem_addr)
{
    if (*mem_addr == copy) {
        *mem_addr = update;
        return SUCCESS;
    } else
        return FAIL;
}
```

# Example – Stack Pop

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

# Example – Stack Pop

```
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;  
}
```

Do Work 

# Example – Stack Pop

```
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;  
}
```

**Test**





# Example – Stack Pop

```
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;  
}
```

**Copy** →

**Do Work** →

**Test** →

# What Made It Work?

It works because we can atomically commit the new stack pointer value and compare the old stack pointer with the one at commit time

This allows us to verify no other thread has accessed the stack concurrently with our operation


- i.e. since we took the copy
- Well, at least we know the address in the stack pointer is the same as it was when we started
  - Does this guarantee there was no concurrent activity?
  - Does it matter?
  - We have to be careful !

# Stack Push

```
Push(elem) {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP - 1;  
        old_val = *new_SP;  
        if(CAS2(old_SP, old_val, new_SP,  
elem, &SP, new_SP)  
            == FAIL) goto retry;  
}
```

# Stack Push

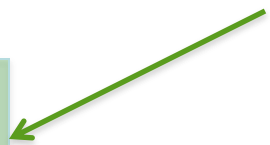
```
Push(elem) {  
    retry:  
    old_SP = SP;  
    new_SP = old_SP - 1;  
    old_val = *new_SP;  
    if(CAS2(old_SP, old_val, new_SP,  
elem, &SP, new_SP)  
        == FAIL) goto retry;  
}
```



# Stack Push

```
Push(elem) {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP - 1;  
        old_val = *new_SP;  
        if(CAS2(old_SP, old_val, new_SP,  
elem, &SP, new_SP)  
            == FAIL) goto retry;  
}
```

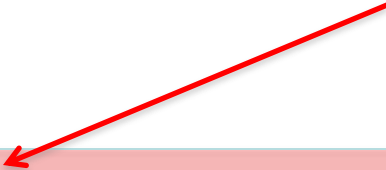
Do Work



# Stack Push

```
Push(elem) {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP - 1;  
        old_val = *new_SP;  
        if(CAS2(old_SP, old_val, new_SP,  
elem, &SP, new_SP)  
            == FAIL) goto retry;  
}
```

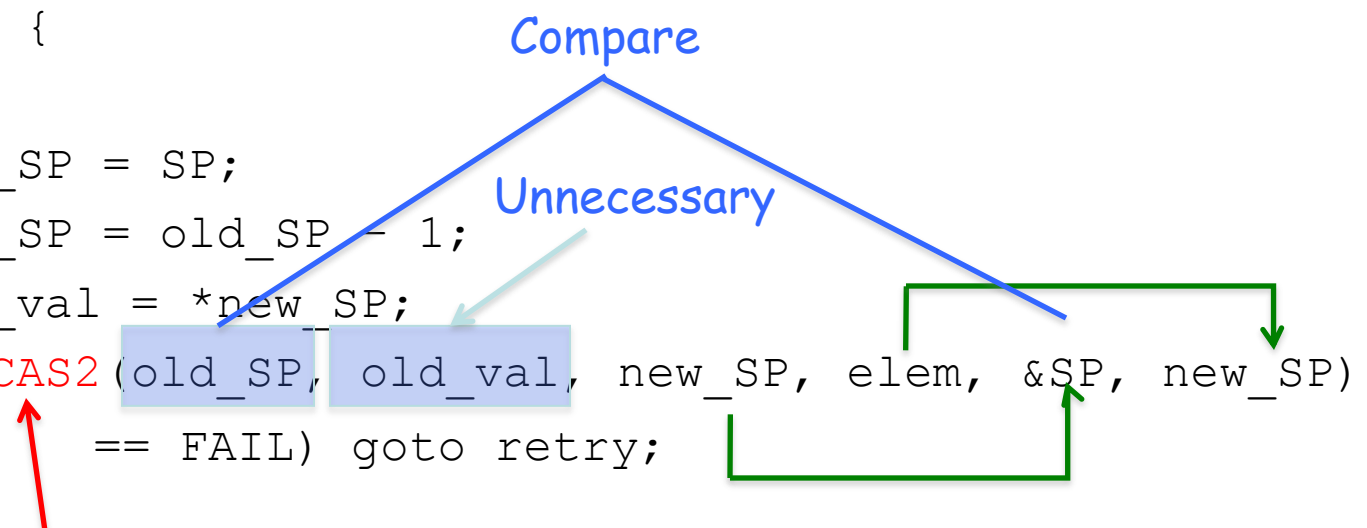
**Test and commit**



# Stack Push

```

Push(elem) {
  retry:
  old_SP = SP;
  new_SP = old_SP - 1;
  old_val = *new_SP;
  if(CAS2(old_SP, old_val, new_SP, elem, &SP, new_SP)
      == FAIL) goto retry;
}
  
```



Note: this is a **double** compare and swap!  
Its needed to atomically update both the  
new item and the new stack pointer

# CAS2

CAS2 = double compare and swap

- Sometimes referred to as DCAS

```
CAS2(copy1, copy2, update1, update2, addr1, addr2)
{
    if(addr1 == copy1 && addr2 == copy2) {
        *addr1 = update1;
        *addr2 = update2;
        return SUCCEED;
    } else
        return FAIL;
}
```



# Stack Push

```
Push(elem) {  
    retry:  
    Copy → old_SP = SP;  
           new_SP = old_SP - 1;  
    Do Work → old_val = *new_SP;  
           if(CAS2(old_SP, old_val, new_SP, elem, &SP, new_SP)  
    Test →     == FAIL) goto retry;  
}
```

# Synchronization in Synthesis

Saved state is only one or two words

Commit is done via

- Compare-and-Swap (CAS), or
- Double-Compare-and-Swap (CAS2 or DCAS)

Can we really do everything in only *two* words?

- Every synchronization problem in the Synthesis kernel is reduced to only needing to atomically touch two words at a time!
- Requires some very clever kernel architecture

# Approach

Build data structures that work concurrently

- Stacks
- Queues (array-based to avoid allocations)
- Linked lists

Then build the OS around these data structures

Concurrency is a first-class concern

# Its Trickier Than It Seems

List operations show insert and delete at the head

- This is the easy case
- What about insert and delete of interior nodes?
- Next pointers of deletable nodes are not safe to traverse, even the first time!
- Need reference counts and DCAS to atomically compare and update the count and pointer values
- This is expensive, so we may choose to defer deletes instead (more on this later in the course)

Specialized list and queue implementations can reduce the overheads

# The Fall-Back Position

If you can't reduce the work such that it requires atomic updates to two or less words:

- Create a single server thread and do the work sequentially on a single CPU
- Why is this faster than letting multiple CPUs try to do it concurrently?

Callers pack the requested operation into a message

- Send it to the server (using lock-free queues!)
- Wait for a response/callback/...
- The queue effectively serializes the operations

# Lock vs Lock-Free Critical Sections

```
Lock_based_Pop() {
```

```
    spin_lock(&lock);
```

```
    elem = *SP;
```

```
    SP = SP + 1;
```

```
    spin_unlock(&lock);
```

```
    return elem;
```

```
}
```

```
Lock_free_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;
```

```
}
```

# Conclusions

This is *really* intriguing!

Its possible to build an entire OS without locks!

But do you really want to?

- Does it add or remove complexity?
- What if hardware only gives you CAS and no DCAS?
- What if critical sections are large or long lived?
- What if contention is high?
- What if we can't undo the work?
- ... ?