

User-level Uniprocessor Mutual Exclusion



Mutual Exclusion Overheads

- Locking implemented by:
 - interrupt disabling and enabling
 - not suitable for user-level
 - Hardware primitives (test and set)
 - not always available, not efficiently implemented
 - System calls
 - high overheads
- Trade-off between granularity of locking and locking overhead
 - Fine granularity
 - more potential parallelism
 - more locks and thus overhead



Can we avoid locking?

- Yes
 - in some cases
- Lock-free data structures
 - Need hardware help
 - compare-and-swap()
 - exchange()
 - test_and_set()



Background: Atomic Compare and Swap

```
bool compare_swap(addr, val, new)
{
    if (*addr == val) {
        *addr = new;
        return true;
    }
    return false;
}
```

addr = memory address

val = expected value

new = value to replace

r = success or failure



CAS Example

- Lock-free atomic increment

```
atomic_inc(int *addr)
{
    do {
        old = *addr;
        new = old + 1;
    } while (!compare_and_swap(addr, old, new));
}
```

- Lock-free does not preclude starvation
- Tricky to implement more complex structures



Lock-free the solution?

- Can avoid locking by using lock-free data structures
 - Still need short atomic sequences
 - compare-and-swap, etc, ..
 - not always provided by hardware
 - may be slow to execute
- Observe: Lock-based data structure also need mutual exclusion to implement the lock primitive themselves.



How do we provide efficient atomic sequences?

- Interrupt disabling?
- Syscalls?
- Processor Instructions?



The problem

add:

```
lw  r0, (r1)
```

```
add r0, r0, 1
```

```
sw  r0, (r1)
```



Optimistic Approach

- Assume the critical code runs atomically
 - *Atomic Sequence*
- If an interrupt occurs, OS recovers such that atomicity is preserved
- Two basic mechanisms
 - Rollback
 - Only single memory location update
 - Guarantee progress???
 - Rollforward



How does the OS know what is an atomic sequence?

- Designated sequences
 - Match well know sequences surrounding PC
 - Matching takes time
 - sequence may occur outside an atomic sequences
 - Rollback might break code
 - Rollforward okay
 - Sequences can be inlined
 - No overhead added to each sequence, overhead only on interruption



- Static Registration
 - All sequences are registered at program startup
 - No direct overhead to sequences themselves
 - Limited number of sequences
 - Reasonable to identify on interrupt
 - No inlining



- Dynamic Registration
 - Share a variable between kernel and user-level, set it while in an atomic sequence
 - Can inline, even synthesize sequences at runtime
 - Adds direct overhead to each sequence



How to roll forward?

- Problem: How to regain control after rolling forward to end of sequence
- Code re-writing
 - Re-write instruction after sequence to call back to interrupt handler
 - Cache issues – need to flush the instruction cache??



- Cloning
 - Two copies of each sequence
 - normal copy
 - modified copy that call back into interrupt handler
 - On interrupt, map PC in normal sequence into PC in modified
 - Mapping can be time consuming
 - Inlining???
 - Difficulties with PC relative offsets
 - Instruction encoding may change in clone.



- Computed Jump
 - Every sequence uses a computed jump at the end
 - Normal sequence simply jmp to next instruction
 - Interrupted sequence jumps to interrupt handler
 - Adds a jump to every sequence



- Controlled fault
 - Dummy instruction at end of each sequences
 - NOP for normal case
 - Fault for interrupt case
 - Example is read from (in)accessible page
 - Still adds an extra instruction

Limiting Duration of Roll forward

- Watchdog
- Restriction on code so termination can be inspected for



Implementations - Dynamic Registration Scheme With Jump

destAddr \leftarrow *addressOf*(theEnd)

inAS \leftarrow TRUE

\langle atomic sequence ... \rangle

inAS \leftarrow FALSE

jump *destAddr*

theEnd:

```
* lda  r4, inAS           # load address of inAS
* lda  r1, theEnd         # load address of theEnd into r1
* stl  zero, (r4)         # inAS <- TRUE (0 = TRUE)
  lda  r3, sharedCounter # load address of sharedCounter
  ldl  r2, (r3)           # load value of sharedCounter
  addl r2, 1, r2          # increment counter
  stl  r2, (r3)           # store back new value
* stl  r1, (r4)           # reset inAS to FALSE (not 0 = FALSE)
* jmp  (r1)               # jump to address stored in r1
```

theEnd:

Implementations - Dynamic Registration Scheme With Fault

```
                                destAddr ← addressOf(theEnd)
                                inAS ← TRUE
                                ⟨atomic sequence . . .⟩
theEnd:                          inAS ← *falseOrFault
```



Implementations – Hybrid registration – a hint-based approach

```
destAddr ← addressOf(theEnd)  
inAS ← TRUE  
⟨atomic sequence ...⟩  
jump destAddr
```

theEnd:

```
* lda  r1, theEnd          # load address of theEnd into r1  
  lda  r3, sharedCounter  # load address of sharedCounter  
  ldl  r2, (r3)           # load value of sharedCounter  
  addl r2, 1, r2          # increment counter  
  stl  r2, (r3)           # store back new value  
* jmp  (r1)              # jump to address stored in r1
```

theEnd:



Results

Technique	DEC Alpha			HP PA-RISC 1.1		
	NULL	LIFO	FIFO	NULL	LIFO	FIFO
sigprocmask	1682	3045	3363	1787	3578	3590
Dyn/Fault	13	27	24	12	24	27
Dyn/Jump	9	16	13	11	21	27
Hyb/Jump	6	5	6	5	8	12
DI	4	3	4	4	5	12
CIPL	4	5	6	14	24	29
splx	44	89	88	30	63	73
PALcode	≥ 13	≥ 13	≥ 13	n/a	n/a	n/a
LL/STC	n/a	≥ 118	≥ 118	n/a	n/a	n/a

Table 1: Overheads of Different Atomicity Schemes in Cycles



Benchmark Legend

- Sigprocmask – syscall based approach
- DI – disable all interrupts
- CIPL – set interrupt priority level
- SPLx – same as CIPL with function call
- PALcode – special Alpha processor call
- LL/SC – load link store conditional



Interrupt Delay

- Whenever an interrupt occurs, we need to check for atomic sequence.
 - Hyb/Jump
 - does r1 point to instruction after a jump
 - sequence ≤ 32 instructions
 - no backward jumps/branches
 - forward jump/branch targets within sequence
- Cost
 - $73 + N * 25$ cycles (N is length of sequence)
 - Only if we pre-empt in the small atomic sequence.

