cse

User-level Mutual Exclusion

THE UNIVERSITY OF NEW SOUTH WALES

cse

Lock-free?

- Avoid needing locking by using lock-free data structure
  - Still need short atomic sequences
    - compare-and-swap
- Lock-based data structure also need mutual exclusion to implement the lock primitive themselves.

THE UNIVERSITY OF NEW SOUTH WALES

cse

How do we provide efficient mutual exclusion to kernel-implemented threads at user-level

- Interrupt disabling?
- Syscalls?
- Processor Instructions?

THE UNIVERSITY OF NEW SOUTH WALES

cse

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

THE UNIVERSITY OF NEW SOUTH WALES

cse

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

THE UNIVERSITY OF NEW SOUTH WALES

cse

- 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

THE UNIVERSITY OF NEW SOUTH WALES

cse

- 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

cse

How to roll forward?

- Code re-writing
  - Re-write instruction after sequence to call back to interrupt handler
    - Cache issues – need to flush the instruction cache??

cse

- 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???

cse

- 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

cse

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

cse

Limiting Duration of Roll forward

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

cse

Implementations - Dynamic Registration Scheme With Jump

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

cse

Implementations - Dynamic Registration Scheme With Fault

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

THE UNIVERSITY OF NEW SOUTH WALES

cse

Implementations - Dynamic Registration Scheme With Fault

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:
```

THE UNIVERSITY OF NEW SOUTH WALES

cse

Results

Technique	DEC Alpha NULL	DEC Alpha LIFO	DEC Alpha FIFO	HP PA-RISC 1.1 NULL	HP PA-RISC 1.1 LIFO	HP PA-RISC 1.1 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