2. Mutex Without Special Atomic Instructions

Q: Is it possible to arrange mutual exclusion through clever use of atomic load & store instructions?

A: Yes, although the solutions are complex & have serious disadvantages

Classic solutions:

- Strict Alternation
- Peterson's solution
- Dekker's solution

Strict Alternation, I

Assumes hardware-atomic:

- Assignment (write) to `turn_to_run`
- Read of `turn_to_run` (during `while` test)

Thread 0:

```
int turn_to_run;     /* shared variable */

while (TRUE) {
   while (turn_to_run != 0)
      ;     /* wait */
   <critical section>
   turn_to_run = 1;
   <other, non-critical code>
}
```

Thread 1:

```
while (TRUE) {
   while (turn_to_run != 1)
      ;     /* wait */
   <critical section>
   turn_to_run = 0;
   <other, non-critical code>
}
```

Strict Alternation, II

Violates condition #3: non-CS thread can block others

(E.g., 0 can't enter its CS until 1 has had its turn; what if 0 needs to enter its CS more often than 1?)

Violates condition #5: must have fixed number of threads, *known at program creation time*

Peterson's Solution, I

Same assumptions as strict alternation: atomic read/write of int

CS protocol (for process 0):

```
enter(0)
<critical section>
leave(0)
```

Peterson's Solution, II

```
/* shared variables */
int turn_to_wait;
int interested[2];

enter(proc) {
   int other = 1 - proc;     /* assumes 0,1 */
   interested[proc] = TRUE;
   turn_to_wait = proc;
   while ((turn_to_wait == proc) &&
          (interested[other] == TRUE))
      ;   /* wait by looping */
}


leave(proc) {
   interested[proc] = FALSE;
}
```

Peterson's Solution

Unlike strict alternation, `turn_to_wait` tells whose turn it is to WAIT; if both execute `enter()` simultaneously, whichever sets `turn_to_wait` LAST will wait

Improvement over Strict Alternation: does not violate condition #3

But: requires known set of threads

Dekker’s Solution

Weaker assumption – assumes only atomic READ of an int

More complicated than Peterson’s solution

Assuming atomic write is not trivial: some multiprocessors do not implement it

Dekker's Solution: Enter

```
/* shared variables */
int turn_to_wait;
int interested[2] = {FALSE, FALSE};

enter(proc) {
   int other = 1 - proc;    /* assumes 0,1 */

   interested[proc] = TRUE;
   turn_to_wait = proc;
   while (interested[other] == TRUE) {
       if (turn_to_wait == other) {
           interested[proc] = FALSE;
           while (turn_to_wait == other)
               ;
           interested[proc] = TRUE;
       }
   }
}
```

Dekker's Solution: Leave

```
leave(proc) {
   int other;
   other = 1 - proc;

   turn_to_wait = other;
   interested[proc] = FALSE;
}
```

3. Locking

An int can be manipulated atomically by special lock & unlock instructions, whereas, on most architectures, most larger data structures can't

Idea: associate an int with a larger data structure; value of this integer indicates which thread may access the data structure

How it works:

1. get associated lock

2. perform critical transformation/view on large data structure

3. drop lock

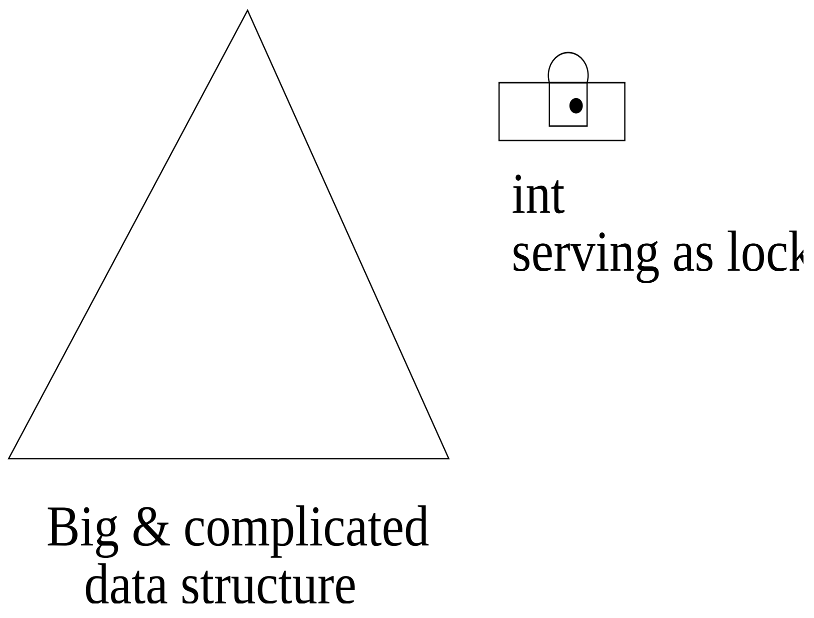
3. Locking

Merely a *convention*: all threads accessing large data structure agree to get/drop associated lock

Depends on special hardware atomic instructions; e.g., "test-and-set;" "Load Linked and Store Conditional;" others

Can get fancy by having multiple "lock modes;" e.g., read & write to allow either single writer or multiple readers

Illustration: Locking



3. Locking

Q: How to implement lock/unlock?

A: With special atomic instructions mentioned below

3. Locking: Evaluation

Satisfies all 5 conditions

Easy to understand, simple to use

Disadvantage: blocked thread's "spin wait" wastes its CPU time slice

4. Special Instructions

To support locking, IBM Series 360 computers introduced **test-and-set** instruction

Later, other architectures included special atomic instructions capable of supporting locking AND atomic manipulation of certain common & important data structures:

- Compare-and-swap (Motorola 68K)
- Load-linked & Store-conditional (MIPS)

4. Test-and-Set Semantics, I

TAS(mem) does these actions **atomically**:

```
mem tested & CCs set /* test */
mem = 1;             /* set */
```

Later instructions then test whether `mem` was 0 or non-0

Aside: Condition Code Bits

Note: “CCs” on preceding page means **condition codes**

Older architectures had condition code bits to indicate that the previous instruction produced a result that was equal to, less than, or greater than zero. Conditional jump instructions jumped based on CC bit values.

With many instructions accessing the condition code bits, condition codes complicate pipelining and parallelization

Consequently, newer architectures don’t have CC bits; conditional instructions jump based on register values

4. Test-and-Set Semantics, II

Other architectures copied & slightly redefined the instruction

Sometimes defined as TAS(reg, mem):

```
mem tested & CCs set
mem = reg;            /* helpful value, like PID */
```

Or even TAS(reg, mem):

```
reg = mem;            /* test reg rather than CC */
mem = 1;
```

And finally TAS(reg, mem):

```
reg = mem;            /* atomic swap; test reg */
mem = reg;
```

Use of Test-and-Set for Locking

To implement a lock using *first* definition:

- `mem` is single word storing lock value
- 0 if free, 1 if held
- to lock:

```
do {
    TAS(mem)
} while (CC indicates lock was non-zero);
```

- to unlock: `mem = 0;`

Assumes hardware-atomic: TAS instruction, store to single word