CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Synchronization Hardware

Recall the critical section problem

```
do {
```

entry section

critical section

exit section

remainder section

```
} while (true);
```

Solution must satisfy three criteria:

- **Mutual exclusion**
- **Progress**
- **Bounded waiting**

Saw a software solution that (kind of) worked for a specific case

In general, solutions will be based on **locking**

Hardware support will be key in solving this

Shared variable `lock`

```
do {

    while (lock == 1)
        ;

    lock = 1;

    # do work in critical section

    lock = 0;

} while (true);
```

How could this go wrong?

Need some way of testing the lock and aquiring it that cannot be interrupted

One solution would be to request that the OS disable interrupts during that process, but disabling interrupts is inefficient on a multiprocessor machine

Hardware offers operations for locking that can happen **atomically** -- they act as a single instruction

`test_and_set(&lock)` -- check whether lock is set and set lock (atomically)

`compare_and_swap(&lock)` -- read a value and optionally set it (atomically)

Description of instruction:

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}
```

Critical section problem solution using `test_and_set`

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);
```

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);
```

This solution does not satisfy all three of our requirements for a solution to the critical section problem. Why?

`test_and_set` can be used to provide such a solution, which we will see later

Description of instruction:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

Critical section problem solution using `compare_and_swap`

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
} while (true);
```

Similar to what we saw with `test_and_set`, this solution does not meet all of our criteria

Remember, the key is that these `test_and_set` and `compare_and_swap` "functions" are executed **atomically**

They are hardware instructions

Actually writing the functions as we've seen them would be useless

We next look at a solution that meets all three requirements

Assume n processes are sharing the variables

```
boolean waiting[n];
boolean lock;
```

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test and set(&lock);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */
} while (true);
```

`test_and_set` and `compare_and_swap` are low-level and not typically used by application programmers

We will see higher-level abstractions later