Operating Systems
CS F372

Synchronization

BIJU K RAVEENDRAN

Synchronization Hardware

- Hardware is faster than the software & can have better efficiency
- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Processor is limited in its ability to interleave programs

- Generally too inefficient on multiprocessor systems
- Disabling interrupt in a multiprocessor machine is time consuming, as the message must pass to all the processors. This message passing delays entry into each critical section & system efficiency decreases.
- The system clock functionality may be disturbed, if the clock is kept updated by the interrupt.
- Modern machines provide special atomic (non-interruptable) hardware instructions
  - Performed in a single instruction cycle
  - Access to the memory location is blocked for any other instructions
  - Either test memory word and set value
  - Or swap contents of two memory words

Three forms of Hardware Support

1. Memory barriers
2. Hardware instructions
3. Atomic variables

Memory Barriers

- Memory models of an Architecture may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barriers

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
   memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words atomically (uninterruptibly.)
- Test-and-Set instruction
- Compare-and-Swap instruction

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
{  boolean rv = *target;
  *target = TRUE;
   return rv:
}
```

Solution: Shared boolean variable lock initialized to false.

```
do {
    while ( TestAndSet (&lock )) ;// do nothing
    //    critical section
    lock = FALSE;
   //      remainder section
    } while (TRUE);
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{      boolean temp = *a;
       *a = *b;
       *b = temp:
}
```

- Shared Boolean variable lock initialized to FALSE;
- Each process has a local Boolean variable key

```
do {    key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );
        //    critical section
        lock = FALSE;
        //      remainder section
} while (TRUE);
```

Mutual Exclusion Machine Instructions

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

Mutual Exclusion Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority (real-time) process needs it
      - The higher priority process will execute busy waiting in processor which leads to deadlock

Bounded waiting Mutual Exclusion with TestAndSet

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)   key = TestAndSet(&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);
```