EE5903 RTS
Chapter 6
Real-Time Process Synchronization

Bharadwaj Veeravalli

elebv@nus.edu.sg

Acknowledgments: Parts of the material are from the book, OS Concepts by Silberschatz, Galvin and Gagne, 9th Edition

Hardware Solutions

- Many systems provide hardware support for implementing the critical section code.
- Almost all solutions listed are based on idea of **locking** – That is, protecting critical regions via locks
- Uniprocessors – Disabling interrupts is one way – this means preemption is completely avoided;
- Multiprocessors – Not a scalable solution for multiprocessor systems;

Hardware solutions

General working logic:

```
do {
        acquire lock;
                enter CS;
        release lock;
                remainder section;
} while(True);
```

Only one process will acquire the lock and others busy wait on that lock

Modern architectures provide atomic hardware instructions – non-interruptible;

- Test&Set()
- Compare&Swap() / Swap()

Hardware Solution – Test&Set()

- **T&S()** is executed as an atomic operation; This does 2 things –

  (i) Test the current value of the lock

  (ii) Set the lock to true after acquiring

Both the above two steps are executed as though it is a single instruction – hence the name “atomic”;

- We say that a process has acquired the lock when it receives a **return value** as FALSE after T&S() execution;
- If the **return value** is TRUE then this means some other process has acquired the lock (and still has it);

Hardware Solution – T&S() (cont'd)

Definition:

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

Q 6.6*: Any problem with this Solution?*

Process i

```
while (true) {
    while (test_and_set(&lock))   /* Entry section */
       ; // wait until lock becomes equal false
    // here lock=true
           /* critical section */
    lock = false;   /* Release lock */
           /* remainder section */
```

T&S() - Assembly codes

```
EnS: // Entry Section
        TandS Reg,Lock
        CMP Reg, #0
        JNE EnS
        RET

ExS: // Exit section
        MOV Lock, #0
        RET
```

```
main:
Call EnS
Execute CS
Call ExS
```

Hardware Solution – Swap()

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

To be executed atomically!

Q: How to make use of this swap() for controlling lock acquisition?

Implementation of Entry and Exit sections:

- We have a **shared boolean variable** – lock (all processes use this)
- Each process uses a **local boolean variable** - key

Hardware Solution – Swap() (Cont'd…)

```
do{
        key = True;
        while(key == True)
  (*)       Swap(&lock, &key);
        Critical Section;
        lock = False; /* releasing the lock */
        Remainder section;
} while(True);
```

Entry Section

Shared variable lock is initialized to FALSE at t=0

Note:
lock – SHARED var
key – LOCAL var

Note: After swapping, in the next iteration while() becomes False and hence it enters CS.

Hardware Solutions - observations

- What do we observe in the two solutions?

  - Mutual Exclusion property is satisfied

  - Bounded Waiting time – Not satisfied;

Thus, a process k may be waiting but process q may be repeatedly executing CS indefinitely;

This can happen when its CPU time quantum does not expire even when it is in its remainder section and hence it can enter CS again!

How do we achieve this bounded waiting time property in the above two solutions?

Hardware Solution – Bounded waiting time solution in T&S()

Q 6.7: How do you bring in the bounded waiting time feature in the T&S() procedure? Show your implementation clearly.

Discuss clearly all possible cases on how a process acquires a lock, enters the CS, and releases it. Identify and convince yourself on the fairness of your implementation.

Hardware Solution – Bounded waiting time solution in T&S()

- First refer to the original T&S() soln

```
do {
        waiting[i] = True;
        key = True; // local lock var
        while(waiting[i] && key)
                key = T&S(&lock);
        waiting[i] = False;
```

Entry Section

- waiting[] is boolean shared array;
- key is a local var to store the current lock status;

```
        Critical Section;

        j = (i+1)%n; // who is the next to enter CS?
        while( ((j != i)&& !waiting[j]) // Find the next process waiting
                j=(j+1)%n
        if(j == i) lock = False; // no one is waiting
        else waiting[j] = False;

}while(True);
```

Exit Section

Hardware Solution – Bounded waiting time solution in T&S()

Key observations:

•Initially lock = False;
•Until a currently running process sees that there is no one waiting, the lock is "locked" (i.e., held at the state "True" forever;
•Thus, if some process j is waiting (= True) then that processes' waiting status is changed to "False", allowing it to enter CS section;
•If no one waits, the lock is released;
•More than one process waiting? – Choose as per a strategy that is in place! (priority driven, sequence #, largest load first, etc)
•A process once executed the CS is forced to leave thus enabling other waiting processes to enter CS, guaranteeing a bounded waiting time for others!

Hardware Solution – Bounded waiting time solution in Swap()

Q 6.8: How do you bring in the bounded waiting time feature in the Swap() procedure? Show your implementation clearly.

Discuss clearly all possible cases on how a process acquires a lock, enters the CS, and releases it. Identify and convince yourself on the fairness of your implementation.

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers; OS designers build software tools to solve critical section problem

- Simplest is **mutex** lock - Protect a critical section by:

  - First: **acquire()** a lock;
  - Then: **release()** the lock;
  - Boolean variable indicating if lock is available or not;

Mutex Locks

- **acquire**()

```
{
    while(!available)
        ; /* busy wait */
    available = false;
}
```

release()

```
{
    available = true;
}
```

```
while(true) {
    acquire()
    critical section
    release()
   remainder section
}
```

Remarks: Calls to **acquire()** and **release()** must be atomic; Usually implemented via hardware atomic instructions;

But this solution requires **busy waiting**; This lock therefore referred to as a **spinlock;**

Avoiding busy waiting – *How*?

Is there a solution which will not allow a process to *busy wait* if it is not in CS?

We can force a process to “sleep” so long as it cannot enter CS. If another process needs CS then it is “awaken” and it will be allowed to enter the CS.

Such a solution is achieved by the use of “*Semaphores*”