OPERATING SYSTEMS

Synchronization

Chandravva Hebbi
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9^{th} edition 2013 and some slides from 10^{th} edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9^{th} edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Principles of concurrency
Synchronization Hardware

Chandravva Hebbi
Department of Computer Science

Review of Peterson's solution

Code for process i

```
do{
flag[i]=TRUE
turn=j
while(flag[j]&&turn==j);//Do-nop
   critical section
flag[i]=FALSE;
Reminder section
}while(TRUE)
```

Code for process j

```
do{
flag[j]=TRUE
turn=i
while(flag[i]&&turn==i);//Do-nop
   critical section
flag[j]=FALSE;
Reminder section
}while(TRUE)
```

- **Principles of Concurrency**

- relative speed of execution of processes is not predictable.
- system interrupts are not predictable
- scheduling policies may vary

Synchronization Hardware

- Software based solutions are not guaranteed to work on modern computer architectures
- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks.
- synchronization can be done through Lock & Unlock technique
- Locking part is done in the Entry Section. After locking the process enter critical section.
- The process is moved to the Exit Section after it is done with execution in CS.
- Unlock is done in exit section.
- This process is designed in such a way that all the three conditions of the Critical Sections are satisfied

Synchronization Hardware

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

test_and_set Instruction

- Test and Set Lock (TSL) is a synchronization mechanism.
- It uses a test and set instruction to provide the synchronization among the processes executing concurrently.
- It is an instruction that returns the old value of a memory location and sets the memory location value to 1 as a single atomic operation.
- If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.

- Does this scheme provide mutual exclusion

lock=0

Process P0

```
while(true)
{
while(lock!=0);
lock=1;
Critical section
lock=0
Remainder section
}
```

Process P1

```
while(true)
{
while(lock!=0);
lock=1;
Critical section
lock=0
Remainder section
}
```

Execution sequence of the processes

```
lock=0
P0:while(lock!=0);
//context switching
P1:while(lock!=0);
lock=1;
//context switch
P0: lock=1;
Critical section
```

No mutual Exclusion

Test and Set is not an atomic operation

test_and_set Instruction

Definition:

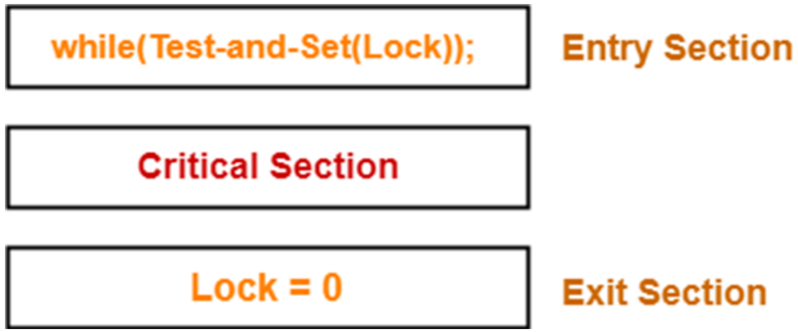
```
boolean test_and_set (boolean *target)
  {
      boolean rv = *target;
      *target = TRUE;
      return rv:
  }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
  while (test_and_set(&lock))
    ; /* do nothing */
      /* critical section */
  lock = false;
      /* remainder section */
} while (true);
```



compare_and_swap Instruction

Definition:

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

  if (*value == expected)
    *value = new_value;
return temp;
}
```

```
do{
while(compare_and_swap(&lock,0,1)!=0);
Critical section
lock=0
Remainder section
}while(true)
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if *value == expected. That is, the swap takes place only under this condition.
4. In the x86 (since 80486) and Itanium architectures this is implemented as compare and exchange (CMPXCHG) instruction

Solution using compare_and_swap()

- Shared integer “lock” initialized to 0;
- Solution:

```
do {
        while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
    /* critical section */
 lock = 0;
    /* remainder section */
} while (true);
```

Mutual exclusion is satisfied

Do not satisfy bounded waiting requirement

Bounded-waiting Mutual Exclusion with test_and_set

This test_and_set algorithm satisfies all the critical section requirements

The common data structures are

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);
```

THANK YOU

Chandravva Hebbi
Department of Computer Science Engineering
chandravvahebbi@pes.edu