CMPT 300
Operating System I
Synchronization Tools– Chapter 6

Dr. Hazra Imran

Summer 2022

Synchronization Hardware

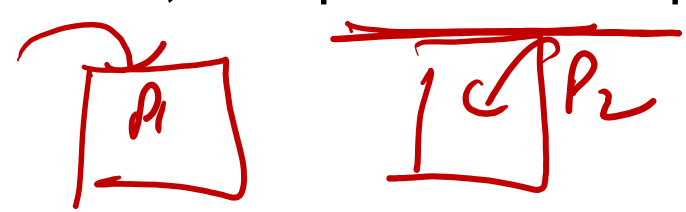
- Uniprocessors: could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptable

TAS

- Two types of atomic hardware instructions
  - test memory word and set value, TestAndSet()
  - swap contents of two memory words, Compare and Swap()

sort of sequences



Test and Set Instruction

$\boxed{100}$ target
1000 address

Effective behavior, but within a single instruction:

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

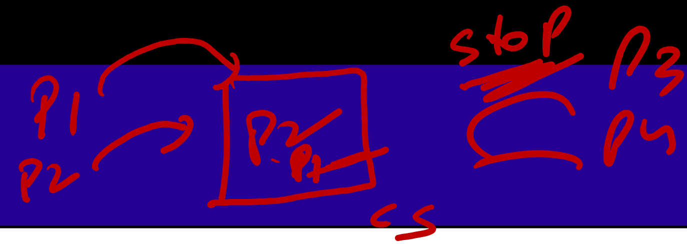
1. Executed atomically
2. Returns target's current value
3. Set the target's value to TRUE

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);
```

test_and_set()




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

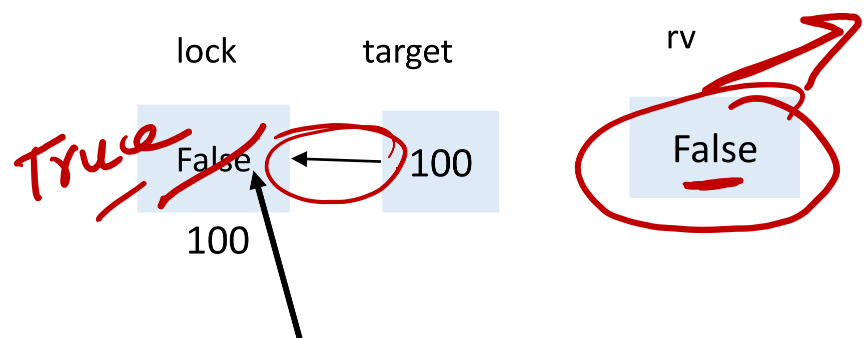
```
boolean test_and_set (boolean
*target)
 {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Set the new value

Return the previous value stored in *target

Shared Boolean variable **lock**

- False: no process is in critical section
- True: one process is in critical section



This will be set to True (new value)

Mutual exclusion using test_and_set()

P2

false

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

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

Set the new value

Return the previous value stored in *target

	lock	target	rv
P1	True (100)	100	False

	lock	target	rv
P2	True (100)	100	True

TAS P1

TAS P2

For P2, true will be return

test_and_set()

Does test_and_set() satisfy our Critical Section Properties?

- Mutual exclusion: Yes No No Guarantee
- Progress: Yes No No Guarantee
- Bounded wait: Yes No No Guarantee

Compare-and-Swap

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

Compare the actual current value with the expected value

compare

Update with the new value if the current value matches expected value

swap

1. Executed atomically
2. Returns the original value of passed parameter "value"
3. Set the variable "value" to the value of the passed parameter "new_value", but only if "value" =="expected".

   That is, the swap takes place only under this condition.

Critical Sections with compare_and_swap()

lock = false/0

- Shared integer "lock" initialized to 0;

expected newvalue

- Solution:

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

     /* critical section */

  lock = 0;
     /* remainder section */
} while (true);
```

Solution using Compare-and-Swap

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

Compare the actual current value with the expected value

Update with the new value if the current value matches expected value

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

Shared variable **lock** is initialized to 0

compare_and_swap()

Does compare_and_swap() satisfy our Critical Section Properties?

- Mutual exclusion: Yes No No Guarantee
- Progress: Yes No No Guarantee
- Bounded wait: Yes No No Guarantee

Lock()/Unlock()

- Threads pair up calls to lock() and unlock()
  - between lock() and unlock(), the calling thread **holds** the lock
    - at most one thread can hold a lock at a time
- What happens if the calls aren't paired
  - I lock, but forgot to unlock?
- What happens if the two threads lock() different locks (lock A and B)

T1

lock A

lock B

Shared

writes

Implementing locks

- Problem is that implementation of locks has critical sections, too!
  - the lock/unlock() must be **atomic**
    - atomic - executes as though it could not be interrupted
    - code that executes “all or nothing”

- Need help from the hardware
  - atomic instructions
    - test-and-set, compare-and-swap, …

Mutex Locks (Spinlocks)

busy waiting

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

```
release() {
   available = true;
}
```

- Both acquire and release must be done <u>atomically</u>
- Keeps spinning when waiting for a lock
  - May waste CPU cycles
  - No context switching occurs when thread is spinning
    - Useful especially when CS is small code
- Widely used on multiprocessor systems
  - A thread keeps spinning on one processor (waiting for lock)
  - While another thread performs CS on another processor, which will eventually release the lock for the spinning thread

Semaphores

- Semaphore *S* – integer variable

- Can only be accessed via two indivisible (atomic) operations
  - wait() and signal(), originally called P() and V() respectively.
  - Also known as down() and up()

```
wait(S) {
    while (S <= 0)
    ;// busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

Semaphores: Usage

- **Counting semaphore**: integer value can range over an unrestricted domain
  - Can solve a wider range of synchronization problems
  - But, can still implement a Binary Semaphore

- **Binary semaphore**: integer value can range only between 0 and 1
  - Same as a mutex lock

Semaphores: Usage

Consider two concurrent processes: P1 and P2

- S1 (part of P1) must happen before S2 (part of P2)
- Semaphore “synch” is initialized to 0

```
P1:
   // other code
   S1;
   signal(synch);
   // other code

P2:
   // other code
   wait(synch);
   S2;
   // other code
```

```
wait(S) {
     while (S <= 0)
     ;// busy wait
     S--;
}

signal(S) {
     S++;
}
```

Semaphore Details

- Implementations of **wait()** and **signal()** must guarantee that the same semaphore variable is not accessed by more than one process at the same time

- wait and signal become CS (must be protected)
  - Disable interrupts (uniprocessor systems only)
  - Too expensive to disable interrupt for each core for multicores
  - Busy waiting or spinlocks (multiprocessor systems)

- With their use, we can still have the busy waiting problem. Just got shifted from application-level CS entry to semaphore's wait and signal commands, which are very short

- **Question:** So why not do the **busy waiting** in the user's critical section?

- Ans: User application may run for a long time, busy waiting can be costly

Semaphore Implementation with no Busy Waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer): semaphore variable
  - pointer to a FIFO queue of processes waiting on the semaphore
- Two internal operations

  Block - suspend the process that invokes it (place the process in the waiting queue)

  Wakeup - resume the execution of blocked process (remove one of the processes from the waiting queue and place it in the ready queue)

```
typedef struct{
   int value;
   struct process *list;
} semaphore;
```

Additional wait list

Semaphore Implementation with no Busy Waiting

3 2 1 0 -1

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

No enough resource instances, must wait

1

Suspend the process (Move to waiting queue)

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Check if we can wake up a process that's been waiting

Schedule the process (Move to the ready queue)

Using Semaphores: Mutual Exclusion

0 1

Semaphore lock(1); // initial semaphore value = 1

Time

Process 1	Process 2
`wait (lock)`	`wait (lock)`
`Critical Section 1`	**`Critical Section 2`**
`signal (lock);`	`signal (lock);`

- Process 1 to execute wait () and lock is decremented to 0
- Process 2 waits until lock becomes > 0, which happens only when the first process executes signal
- Value of semaphore indicates number of waiting processes
  - lock = 0 means 1 process may be waiting
  - lock = 1 means no process is waiting

Semaphores

Let S and Q be two semaphores initialized to 1

Time

P0	P1
wait(S);	wait (Q);
wait(Q);	wait (S);
.	.
.	.
signal(S);	signal (Q);
signal(Q);	signal (S);

Deadlock

P0 S

P1 Q

Question: What could go wrong here?

Be Careful When Using Semaphores

- Incorrect order accessing multiple semaphores across processes
  - May cause deadlock

- signal (mutex) …. wait (mutex)
  - Multiple processes can access CS at the same time

- wait (mutex) … wait (mutex)
  - Processes may block for ever

- Forgetting wait (mutex) or signal (mutex)
  - Various problems, inconsistent data, …

Summary

- Synchronization: coordinate access to shared data
- Race condition
  - Multiple processes/threads manipulating shared data and result depends on execution order
- Critical section problem
  - Three requirements: mutual exclusion, progress, bounded waiting
  - Software: Peterson’s Algorithm
  - Hardware: Test_and_set, Compare-and-Swap and others
    - Busy waiting (or spinlocks)
  - Mutexes, semaphores

NC