OPERATING SYSTEM

Mutual Exclusion (H/W Support)

Dr Rahul Nagpal
Computer Science

OPERATING SYSTEM

Mutual Exclusion (H/W Support)

Dr. Rahul Nagpal
Computer Science

Outline

- Synchronization Hardware
- Test and Set
- Compare and Swap
- Mutex and Semaphor

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- 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

Solutions to Critical Section Problem

```
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".

```
do {
      acquire lock
              critical section
      release lock
              remainder section
} while (TRUE);
```

lock, initialized to FALSE

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

Solutions to CS Problem using compare & Swap

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

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.

- Shared integer "lock" initialized to 0;

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

Bounded-waiting Mutual Exclusion with test_and_set

```
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);
```

Bounded-waiting Mutual Exclusion with test_and_set

- 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
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Mutex Lock

- 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
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Semaphor

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

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

- Definition of the **`signal() operation`**

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

Semaphor Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphor Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time

- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied

- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphor 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)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

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

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

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1

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

- **Starvation** – **indefinite blocking** A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process - Solved via **priority-inheritance protocol**

THANK YOU

Dr Rahul Nagpal
Computer Science
rahulnagpal@pes.edu