Chapter 5: Process Synchronization (Part 2)

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Condition variables

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
  - Software-based solutions not guaranteed to be correct on modern comp architecture
    - Example: ***Peterson's solution*** to the CS problem
- All solutions in the following slides are based on idea of **locking**
  - Protecting critical regions via locks
- We can use some hardware support (if available) for protecting critical section code
- Uniprocessors – could disable interrupts
    - Solve CS problem by ***preventing interrupts*** while modifying shared data
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

Synchronization Hardware (cont'd)

- Normally, access to a memory location excludes other access to that same location
- **Extension:** designers have proposed machines instructions that perform two actions atomically (indivisible) on the same memory location (ex: reading and writing)
- The execution of such an instruction is mutually exclusive (even with multiple CPUs)
- They can be used simply to provide mutual exclusion but need more complex algorithms for satisfying the three requirements of the CS problem
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - The two locks are:
  - ***test memory word and set value***
  - Or ***swap contents of two memory words***
- Note that these are machine/assembly instructions, and are thus atomic.

1. test_and_set Instruction

- **test_and_set()** is a machine/assembly instruction but here we provide definition of it using a high level language code such as C.

  Definition:

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

1.Executed atomically; a ***non-interruptible unit*** of execution

2.Returns the original value of passed parameter in temp variable **rv**

3.Set the new value of passed parameter to “TRUE”.

- Thus, we can implement mutual exclusion on multi-CPU systems
  - See next slide

Solution using test_and_set()

- Shared Boolean variable lock, declared and initialized to FALSE
- Solution: for a process P_i

```
    bool lock FALSE
do {
    while (test_and_set(&lock))
    ; /* do nothing */
      /* critical section */
       lock = false; //release lock so another process use it.
              /* remainder section */
  } while (true);
```

```
boolean test_and_set (boolean *target)
     {
       boolean rv = *target;
       *target = TRUE;
       return rv: //if target was originally
                   FALSE, process will avail it
          }
```

2. compare_and_swap Instruction

- **compare_and_swap()** is a machine/assembly instruction but here we provide definition of it using a high level language code.

Definition:

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

    if (*value == expected) //lock is expected to be FALSE (0)
       *value = new_value; //set value to TRUE (1)
 return temp;           // return whether TRUE or FALSE
}
```

1.Executed atomically

2.Returns the original value of passed parameter “value” using **temp** variable

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.

- Thus, we can implement mutual exclusion on multi-CPU systems
    - See next slide

Solution using compare_and_swap instruction on multi-CPU System

- Shared integer "lock" initialized to 0;
- Solution: for a process P_i

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

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

      if (*value == expected) //lock is expected to be '0'
         *value = new_value;
    return temp;
   }
```

Hardware Solution

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- However, ***bounded waiting*** may not be satisfied.
  - If there are multiple processes trying to get into their critical sections, there is no guarantee of what order they will enter

Bounded-waiting Mutual Exclusion with test_and_set

```
do {/* Shared Boolean waiting[n] and lock initialized to false */
    waiting[i] = true;       //Process i is interested
    key = true;              //Assume another process holds the key for the lock
    while (waiting[i] && key) //loop as long as both are TRUE
        key = test_and_set(&lock); //set key to FALSE if Lock was FALSE
   waiting[i] = false; //Not interested anymore, but key for lock is still true

    /* critical section; enters only if waiting[i] or key == false */
//find one process waiting
    j = (i + 1) % n;
    while ((j != i) && !waiting[j]) //keep checking each process one by one
        j = (j + 1) % n; //Circular Search,i.e., i+1, i+2, .. n, 0, 1,...,(i-1)
    if (j == i) //If none is waiting
        lock = false; //open the lock.
    else //if process 'j' is waiting
        waiting[j] = false;//Let j access
    /* remainder section */
} while (true);
```

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

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- **Mutex Locks**
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- Simplest is mutex lock; short for mutual exclusion
- 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**

Mutual exclusion using acquire() and release()

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

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

```
release() { // available is boolean
         available = true;
   }
```

Busy Waiting Problem

- All of these solutions use ***busy waiting***.
- ***Busy waiting*** means a process waits by executing a tight loop to check the status/value of a variable.
- ***Busy waiting*** may be needed on a multiprocessor system; however, it wastes CPU cycles that some other processes may use productively.
- Even though some systems may allow users to use some atomic instructions, unless the system is lightly loaded, CPU and system performance can be low, although a programmer may “think” his/her program looks more efficient.
- So, we need better solutions.

Mutual Exclusion with Pthreads mutex

- Pthreads api provides functions for initializing, destroying, locking, and unlocking mutexes.

- The two important functions provided by Pthreads for mutual exclusion are:

- **pthread_mutex_lock()** - acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.

- **pthread_mutex_unlock()** - unlock a mutex variable. An error is returned if mutex is already unlocked or owned by another thread.

Mutex using pthreads

```
void *funcINC(){
pthread_mutex_lock( &mutex1 );
counter++; //critical section
printf("Counter value:%d\n",counter);
pthread_mutex_unlock( &mutex1 );
}
```

- We first declare and initialize mutex as below:

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

- We declare **counter** as a global integer variable which will be a shared resource between the two threads of the same process.
- Next we use **mutex** whenever a thread tries to modify the variable **counter**

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;
int main(){
pthread_t t1, t2;//two independent threads will execute functionINC
pthread_create( &t1, NULL, &funcINC, NULL)
      printf("Thread 1 created successfully: \n");
pthread_create( &t2, NULL, &funcINC, NULL)
      printf("Thread 2 created successfully: \n");
/* Wait till threads are complete before main continues. */
pthread_join( t1, NULL);
pthread_join( t2, NULL);
exit(0); }
```

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- **Semaphores**
- Classic Problems of Synchronization
- Monitors
- Condition Variables