CS 452 Operating Systems

Process Synchronization

Dr. Denton Bobeldyk

Test and Set Lock

- Hardware based solution:
  - Provide atomic method to test and set a lock. (Recall atomic means all the steps will execute without the CPU swapping it out.)
    - lock = 0 (unlocked)
    - lock = 1 (locked)

Test and Set Lock

- How does it work?
  - Ready to enter critical section?
    - Test the lock to see what the value is
      - If locked = true, wait to enter the critical section
        - while(locked) {} / / do nothing

Which 'while locked do nothing' syntax is best?

```
while(locked) { } // do nothing

while(locked) ;  // do nothing

while(locked){
// do nothing
}

while(locked)
; // do nothing
```

Test and Set Lock - How does it work?

- Ready to enter critical section?
  - Test the lock to see what the value is
    - If lock = true, wait to enter the critical section.
    - If lock = false, set the lock value to true and enter our own critical section

Test and Set Lock

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

**The above statements execute atomically

Test and Set Lock

myLock is a boolean variable representing the lock we would like to acquire

while(test_and_set(myLock)) {} // do nothing while waiting for the return value to be false.

```
boolean test_and_set (boolean *myLock)
    {
        boolean returnValue = *myLock;
        *myLock = TRUE;
        return returnValue:
    }
```

❖

Test and Set Lock

Two options:
1) the value of myLock will be FALSE (lock is available, enter the critical section)
2) the value of myLock will be TRUE (lock is unavailable, wait for it to be available)

```
boolean test_and_set (boolean *myLock)
    {
        boolean returnValue = *myLock;
        *myLock = TRUE;
        return returnValue:
    }
```

Test and Set Lock

Two options:

1) the value of myLock will be FALSE (lock is available, lock and enter the critical section)

```
boolean test_and_set (boolean *myLock)    // myLock = FALSE
    {
        boolean returnValue = *myLock;    // set the returnValue = FALSE
        *myLock = TRUE;                   // set value of myLock = TRUE, locking it for our use


        return returnValue;               // return FALSE because we now have the lock and want to exit the while loop
    }
```

Test and Set Lock

Two options:
2) the value of myLock will be TRUE (lock is unavailable, loop in while loop)

```
boolean test_and_set (boolean *myLock)    // myLock = TRUE
    {
        boolean returnValue = *myLock;    // set the returnValue = TRUE
        *myLock = TRUE;
                        /* set value of myLock = TRUE, which is already its value*/

        return returnValue;
    }
                        // return TRUE because someone else is currently using it.
```

Test and Set Lock

myLock is a boolean variable representing the lock we would like to acquire

```
boolean test_and_set (boolean *myLock)
      {
           boolean returnValue = *myLock;
           *myLock = TRUE;
           return returnValue:
      }
```

If myLock = FALSE, what value is returned?

Test and Set Lock

myLock is a boolean variable representing the lock we would like to acquire

```
boolean test_and_set (boolean *myLock)
	{
		boolean returnValue = *myLock;
		*myLock = TRUE;
		return returnValue:
	}
```

If myLock = TRUE, what value is returned?

Test and Set Lock

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

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

Solutions to Critical Section - Group Exercise

- Are the following conditions for a solution to the critical section problem met?
  - Mutual Exclusion
  - Progress
  - Bounded Waiting

Solutions to Critical Section - Group Exercise

- **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Solutions to Critical Section

- Mutual Exclusion - Yes
- Progress - Yes
- Bounded Waiting - No

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);
```

❖

Test and Set Lock Group Exercise

- In a multi CPU system, how can we ensure that there is a sequential order applied to two processes attempting to execute test_and_set on the same lock?

Test and Set Lock in-Class Assignment

- Download word document, demonstrate execution of test and set lock

compare_and_swap

- Synchronization provided by hardware
- Must be executed atomically

compare_and_swap

- The first process to enter compare_and_swap will set the lock to true and enter its critical section
- Other calls to compare_and_swap will not succeed until the first process sets lock back to 0.

compare_and_swap Instruction

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

Value is the lock variable, expected is 0, new_value is 1

compare_and_swap Instruction

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

Create a temp variable and assign it the value of the lock variable (value)

compare_and_swap Instruction

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

Check if the lock is available by comparing *value and expected

compare_and_swap Instruction

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

Change the lock '*value' to 1

compare_and_swap Instruction

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

Return the value of temp which is the value of ***value** passed to the method.

compare_and_swap instruction example

```
compare_and_swap(int *myLock, 0, 1){
  int temp = *myLock;
  if(*myLock == 0){
    *myLock = 1;
  }
  return temp;
}
```

Pass in the lock

Set temp variable to the lock value

Check if the lock is available

If it is, set it to be locked now

return the original value of lock, if it was available, returning 0 will kick the process out of the while loop waiting for this lock

compare_and_swap vs. test_and_set

- So far, they look very similar, the advantage of compare_and_swap is that we have the capability to use an integer and not just one bit (bool)

compare_and_swap vs. test_and_set

- Small groups:
  - Determine some advantages compare_and_swap has over test_and_set

compare_and_swap Instruction

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

Value could be considered like the lock variable. Expected is the value you'd like it to be, new_value is the value that '*value' will become if the lock is acquired.

compare_and_swap Instruction

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

Create a temp variable and assign it the value of the lock variable (*value)

compare_and_swap Instruction

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

Check if the lock is available by comparing *value and expected

compare_and_swap Instruction

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

Change the lock '*value' to new_value. This could be assigning the lock to be 1

compare_and_swap Instruction

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

Return the value of temp which is the value of ***value** passed to the method.

Compare_and_swap Example Use

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

Mutex Locks

- compare_and_swap, test_and_set are hardware based solutions
- What about software solutions?

Mutex Locks

- compare_and_swap, test_and_set are hardware based solutions
- What about software solutions?
  - Simplest is mutex lock

Mutex Locks

- Protect a critical section by first acquiring a lock, then release the lock when complete.
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions
- This solution requires 'busy waiting', sometimes this lock is called a 'spin lock'

Mutex Locks

```
   acquire() {
       while (!available) { } /* busy wait */
       available = false;
    }
   release() {
       available = true;
    }
   do {
    acquire lock
       critical section
    release lock
      remainder section
 } while (true);
```

Semaphore

- A semaphore is an integer variable that is accessed using two atomic operations:
  - wait()
  - signal()

Semaphore

- A semaphore is an integer variable that is accessed using two atomic operations:
  - wait() proberen (Dutch word ‘to test’)
  - signal() verhogen (Dutch word ‘to increment’)

Semaphore

- A semaphore is an integer variable that is accessed using two atomic operations:
  - wait()   proberen (Dutch word ‘to test’)
  - signal()   verhogen (Dutch word ‘to increment’)

Does anyone know why we even mention the dutch words here?

Semaphore

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

Semaphore

- Counting semaphore
  - Allows for more than one process to access a lock
- Binary semaphore
  - A single process can access the lock

Semaphore

P1:

// some code

signal(semaphoreLock)

P2:

wait(semaphoreLock)

// some code

Semaphore

Semaphores - avoiding busy waiting (and wasting cpu cycles)

* Once the process executes ‘wait’ it can block itself. The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state.

* Control is transferred to the CPU scheduler which then selects another process to execute

Semaphore -support for no busy waiting

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

Semaphore -support for no busy waiting

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

Semaphore -support for no busy waiting

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

Semaphore Funtivity

- Using a scratchpad or piece of paper:
  - Write some pseudocode that:
    - Uses a semaphore to control access to a single resource
    - Allows 3 processes to access the resource at a time
    - Denies the 4th process access to the resource
    - Releases 1 process from the resource
    - Signals to the ‘4th’ resource that is now available