Computer Science & Engineering Department
I. I. T. Kharagpur

Operating System: CS33007
3rd Year CSE: 5th Semester (*Autumn 2006 - 2007*)
Lecture IX (Mutual Exclusion of CS)

Goutam Biswas *Date:* 23rd - 28th August, 2006

1 Architectural Support

A *critical section (CS)* may be implemented by *disabling interrupt* at the beginning of the CS code and then enabling it when the control comes out of it. This will stop preemption of a process when the control is within a CS. But this is not possible within a user process as these operations are privileged.

This method can be used within the kernel code provided the length of the critical section is short. If the interrupt is kept disabled for a longer period, time critical responses may suffer.

This method also has problem in a multiprocessor SMP system.

1.1 Special Machine Mnstructions

Execution of an instruction is *atomic* for a uniprocessor system. The interrupt is not serviced before the completion of the current instruction.

But for a multiprocessor system (SMP) the atomicity of a machine instruction is not enough. Two processors may try to access a shared memory location within one instruction cycle unless the location is locked.

We assume that there is hardware support to make access to a memory location atomic. There are machine instructions e.g. *test and set* or *exchange*.

1. The semantics of `test&set` is as follows:

```
int test&set(int i) {
     if(i==0) { i=1; return 1; }
     else return 0 ;
}
```

   This operation is *atomic*. Pentium provides *bit test and set* (`bts`) operation.

```
lock                    # Not req. for uni processor.
   btsl $0, <memory>       # CF <-- bit 0 of <memory>
                           # bit 0 of memory <-- 1
```

   Mutual exclusion using `test&set`.

```
int lock = 0 ;   // Shared data

/* Process i */

while(!test&set(lock)) ;  // busy wait
/* critical section */
lock = 0 ;
```

2. Semantics of `exchange` operation.

   ```
   void exchange(int *regP, int *varP) {
        int temp ;
        temp = *regP; *regP = *varP; *varP = temp ;
   }
   ```

   This operation is also atomic. Pentium processor provides the instruction `xchg`.

   ```
   xchgl %eax, <memory>   # eax <--> <memory>
   ```

   Mutual exclusion using `exchange`.

   ```
   int lock = 0 ;

   /* Process i */

   int local = 1 ;
   while(local != 0) exchange(&local, &lock) ;;
   /* Critical Section */
   lock = 0 ;
   ```

1.2 Use of `xchgl`

The machine instruction `xchgl` is used to make producer consumer operations *atomic*.

```
/************************************************************
 * Producer-Consumer Problem on shared memory between
 * processes.  Atomicity by 'xchg'
 * $ cc -Wall sharedProdCons2.c queue.o
 * **********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>
#include "queue.h"
```

```
void producer(queue *) ;
void consumer(queue *) ;

int main() {
     int shmID, chID1, chID2, status ;
     struct shmid_ds buff ;
     queue *qP ;
     int *lockP ;

     shmID = shmget(IPC_PRIVATE, sizeof(queue)+4, IPC_CREAT | 0777);
     if(shmID == -1) {
          printf("Error in shmget\n") ;
          exit(0) ;
     }

     qP = (queue *) shmat(shmID, 0, 0777) ;
     initQ(qP) ;
     lockP = (int *) (qP + 1) ;
     *lockP = 0 ; // Lock is zero

     if((chID1 = fork()) != 0) { // Parent
          if((chID2 = fork()) != 0) { // Parent now

               waitpid(chID1, &status, 0) ;
               waitpid(chID2, &status, 0) ;

               shmdt(qP) ;
               shmctl(shmID, IPC_RMID, &buff) ;
          }
          else { // Child 2
               queue *qP ;

               qP = (queue *) shmat(shmID, 0, 0777) ;
               consumer(qP) ;
               shmdt(qP) ;
          }
     }
     else { // Child I
          queue *qP ;

          qP = (queue *) shmat(shmID, 0, 0777) ;
          producer(qP) ;
          shmdt(qP) ;
     }

     return 0 ;
}
```

```
void producer(queue *qP){
     int count = 1, *lockP ;

        lockP = (int *) (qP + 1) ;
        srand(getpid()) ;
        while(1) {
          int data, added = 1, err ;

          if(added) {
               data = rand() ;
               added = 0 ;
          }
          __asm__ __volatile__ (
               "movl $1, %%eax \n\t"
               ".MyLbl1:     \n\t"
               "xchgl %%eax, 0(%%ebx) \n\t"
               "cmpl $0, %%eax \n\t"
               "jne .MyLbl1     \n\t"
               :
               :"b"(lockP)
               :"%eax"
          ) ;
          err = addQ(qP, data) ;
          *lockP = 0 ; // make lock = 0
          if(err == OK) {
               added = 1 ;
               printf("Produced Data %d: %d\n", count++, data) ;
          }
        }
}

void consumer(queue *qP) {
        int count = 1, *lockP ;

        lockP = (int *) (qP + 1) ;
        while(1) {
          int data, err ;

          __asm__ __volatile__ (
               "movl $1, %%eax \n\t"
               ".MyLbl2:     \n\t"
               "xchgl %%eax, 0(%%ebx) \n\t"
               "cmpl $0, %%eax \n\t"
               "jne .MyLbl2     \n\t"
               :
               :"b"(lockP)
               :"%eax"
              ) ;
          err = frontQ(qP, &data) ;
          if(err == OK) deleteQ(qP) ;
          *lockP = 0 ; // Lock is 0
          if(err == OK)
              printf("\t\t\tConsumed Data %d: %d\n", count++, data) ;
            }
}
```

1.3 Use of bts

The machine instruction `bts` (test and set) is used to have mutual exclusion in producer consumer problem.

```
/**************************************************************
 * Producer-Consumer Problem on shared memory between
 * processes.  Atomicity by 'btsl'
 * $ cc -Wall sharedProdCons3.c queue.o
 * ************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>
#include "queue.h"

void producer(queue *) ;
void consumer(queue *) ;

int main() {
     int shmID, chID1, chID2, status ;
     struct shmid_ds buff ;
     queue *qP ;
     int *lockP ;

     shmID = shmget(IPC_PRIVATE, sizeof(queue)+4, IPC_CREAT | 0777);
     if(shmID == -1) {
          printf("Error in shmget\n") ;
          exit(0) ;
     }

     qP = (queue *) shmat(shmID, 0, 0777) ;
     initQ(qP) ;
     lockP = (int *) (qP + 1) ;
```

```
    *lockP = 0 ; // Lock is zero

    if((chID1 = fork()) != 0) { // Parent
        if((chID2 = fork()) != 0) { // Parent now

            waitpid(chID1, &status, 0) ;
            waitpid(chID2, &status, 0) ;

            shmdt(qP) ;
            shmctl(shmID, IPC_RMID, &buff) ;
        }
        else { // Child 2
            queue *qP ;

            qP = (queue *) shmat(shmID, 0, 0777) ;
            consumer(qP) ;
            shmdt(qP) ;
        }
    }
    else { // Child I
        queue *qP ;

        qP = (queue *) shmat(shmID, 0, 0777) ;
        producer(qP) ;
        shmdt(qP) ;
    }

    return 0 ;
}

void producer(queue *qP){
    int count = 1, *lockP ;

    lockP = (int *) (qP + 1) ;
    while(1) {
        int data, added = 1, err ;

        srand(getpid()) ;
        if(added) {
            data = rand() ;
            added = 0 ;
        }
        __asm__ __volatile__ (
            ".MyLbl1:     \n\t"
            "lock          \n\t"
            "btsl $0, 0(%%ebx) \n\t"
            "jc .MyLbl1     \n\t"
```

```
                :
                :"b"(lockP)
                :"%eax"
               ) ;
                  err = addQ(qP, data) ;
          *lockP = 0 ; // make lock = 0
          if(err == OK) {
                     added = 1 ;
                      printf("Produced Data %d: %d\n", count++, data) ;
          }
      }
}

void consumer(queue *qP) {
     int count = 1, *lockP ;

     lockP = (int *) (qP + 1) ;
     while(1) {
          int data, err ;

          __asm__ __volatile__ (
                ".MyLbl2:     \n\t"
                "lock         \n\t"
                "btsl $0, 0(%%ebx) \n\t"
                "jc .MyLbl2     \n\t"
                :
                :"b"(lockP)
                :"%eax"
          ) ;
          err = frontQ(qP, &data) ;
          if(err == OK) err = deleteQ(qP) ;
          *lockP = 0 ; // Lock is 0
          if(err == OK)
               printf("\t\t\tConsumed Data %d: %d\n", count++, data) ;
          }
}
```

1.4 Locking a pthread

We can use `xchg` to lock a thread.

```
/*******************************************************************
 * The producer consumer problem using a thread
 * Any global variable is shared between two pthreads
 * cc -Wall -c queue.c
 * cc -Wall -O1 -lpthread threadProdCons2.c
 * Not using thread lock
 * ************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include "queue.h"

queue q ;
// pthread_mutex_t makeAtomic = PTHREAD_MUTEX_INITIALIZER ;
int lock ;

void * thread1(void *) ;
void * thread2(void *) ;

void lockMe(int *lockP){
 __asm__ __volatile__ (
         "movl $1, %%eax \n\t"
         ".MyLbl2:       \n\t"
         "lock           \n\t"
         "xchgl %%eax, 0(%%ebx) \n\t"
         "cmpl $0, %%eax \n\t"
         "jne .MyLbl2    \n\t"
         :
         :"b"(lockP)
         :"%eax"
  ) ;
}

inline void unlockMe(int *lockP){ *lockP = 0; }

void producer(){
     int count = 1 ;
     while(1) {
          int data = rand() ;

          while(isFullQ(&q));
          lockMe(&lock) ;
          // pthread_mutex_lock(&makeAtomic) ;
          addQ(&q, data) ;
          // pthread_mutex_unlock(&makeAtomic) ;
          unlockMe(&lock) ;
          printf("Produced Data %d: %d\n", count++, data) ;
     }

}
```

```
void consumer() {
     int count = 1 ;
     while(1) {
          int data ;

          while(isEmptyQ(&q)) ;
          lockMe(&lock) ;
          // pthread_mutex_lock(&makeAtomic) ;
          frontQ(&q, &data) ;
          deleteQ(&q) ;
          unlockMe(&lock) ;
          // pthread_mutex_unlock(&makeAtomic) ;
          printf("\t\t\tConsumed Data %d: %d\n", count++, data) ;
     }
}

int main() {
     pthread_t thID1, thID2 ;

     initQ(&q) ;
     lock = 0 ;
     pthread_create(&thID1, NULL, thread1, NULL) ;
     pthread_create(&thID2, NULL, thread2, NULL) ;
     pthread_join(thID2, NULL) ;
     pthread_join(thID1, NULL) ;
     return 0;
}

void *thread1(void *vp) {
     producer() ;
     return NULL ;
}

void *thread2(void *vp) {
     consumer() ;
     return NULL ;
}
```

References

[1]