CSC 4304 - Systems Programming
Fall 2010

LECTURE - XIV

CONCURRENT PROGRAMMING

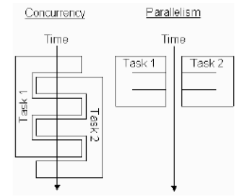
Tevfik Koşar

Louisiana State University

November 2nd, 2010

Concurrency Issues

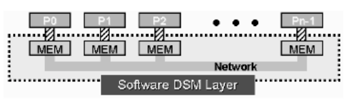
P1:	`X:=1`
P2:	`X:=0; X:=X+1`



- If programs are independent, the results are the same (X=1)
- If programs are executed concurrently and one program is `X:=1`, are results of P1 and P2 different
- **"interleaving" makes it difficult to deal with global properties from the local analysis!**
- assumption: access to the memory is atomic

Concurrency Issues

- Shared variables are an effective way to communicate between processes
- X:=X+1 is implemented as 3 different instructions
  - load the value of X to the register
  - increment the register
  - store the value of register to X
- Two processes updating same variable concurrently causes erroneous results
- Correctivity of the program needs that this updating will be indivisible (or atomic)
- Reading a variable can also be a critical section
  - e.g. reading four bytes that are not volatile



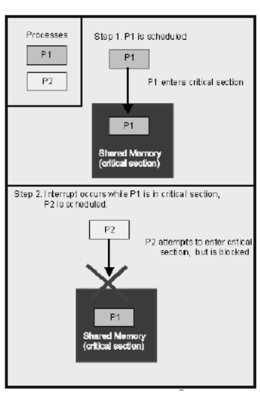

```
LD    AX, CARS
INC   AX
LD    CARS, AX
```


Synchronization

- Mechanism that allows the programmer to control the relative order in which operations occur in different threads or processes.

Critical Section

- A section (lines of program code) which should be seen as atomic is called as a **critical section**
- Synchronization which is needed for the implementation of critical section is called **mutual exclusion**



Busy Waiting

- Busy waiting is one possible way to implement synchronization (and a mutual exclusion)
- Processes writes and reads a shared variable

```
Process P1 (*waiting process*)
  ...
  while flag = down do
     null
  end;
  ...
end P1
```

```
Process P2 (* signalling process*)
   ...
   flag := up
   ...
end P2
```

- if `flag`=down can P1 continue first after P2 set the flag up
- Busy waiting is simple but inefficient
  - Draws CPU resources
  - Excessive traffic on bus (or network)

Suspend and Resume

- busy-waiting loops use precisious CPU time
- instead we can set the waiting process to halt
- suspend and resume operations
- can still lead to **race-condition**

```
process P1; (* waiting process *)
...
if flag = down do
    suspend
end;
flag := down
end P1

process P2; (* signalling process *)
...
flag := up
resume P1
end P2
```

- flag is a shared variable which is used to control the action
- testing and actions (suspend) should be done atomically

Mutual Exclusion

- Suppose we have 2 processes with the following structure

```
Process P
  loop
    entry protocol
      critical section
    exit protocol
    non-critical section
  end
end P
```

- In which way the protocol could be implemented so that we can guarantee the mutual exclusion?

Mutual Exclusion: Problem 1

```
Process P1
  loop
    flag1 := up
    while flag2 = up do
      null
    end
    <critical section>
    flag1 := down
    <non-critical section>
  end
end P1
```

```
Process P2
  loop
    flag2 := up
    while flag1 = up do
      null
    end
    <critical section>
    flag2 := down
    <non-critical section>
  end
end P1
```

- problem:
  - P1 sets `flag1 := up`
  - P2 sets `flag2 := up`
  - P2 controls `flag1`
  - P2 goes to busy-wait
  - P1 controls `flag2`
  - P1 goes to busy-wait
- problem
  - both processes stops in busy-wait loop and neither will come out of the loop
  - this is called as **livelock**
  - different from dead-lock, both processes run

Mutual Exclusion: Problem 2

- the problem is created when both processes announce that they will go to the critical section without checking that it is possible

```
Process P1
  loop
    while flag2 = up do
      null
    end
    flag1 := up
    <critical section>
    flag1 := down
    <non-critical section>
  end
end P1
```

```
Process P2
  loop
    while flag1 = up do
      null
    end
    flag2 := up
    <critical section>
    flag2 := down
    <non-critical section>
  end
end P1
```


Mutual Exclusion: Problem 2

- P1 and P2 are in not critical section (`flag1=flag2=down`)
- P1 tests `flag2` (that is down)
- P2 tests `flag1` (that is down)
- P2 set `flag2` (`flag2` is now up)
- P2 goes to the critical section
- P1 set `flag1` (`flag1` is now up)
- P1 goes to the critical section
- FAULT: P1 and P2 are simultanously in critical section

Mutual Exclusion: Problem 3

```
Process P1
  loop
    while turn = 2 do
      null
    end
    <critical section>
    turn := 2
    <non-critical section>
  end
end P1
```

```
Process P2
  loop
    while turn = 1 do
      null
    end
    <critical section>
    turn := 1
    <non-critical section>
  end
end P1
```

- turn must have value 1 or 2.
- if turn has the value 1, P1 cannot be indefinitely delayed, and P2 cannot enter its critical section
- turn can't get value 2 before P1 has been in its critical section
- a symmetric argument goes for P2
- problem:
  - if P1 breaks before critical section, P2 can never enter the critical section
  - needs that processes runs at the same speed (P1 can't be run 3 times at every time P2 runs)

Mutual Exclusion: Problem 3

- The problem is that a process cannot set its own flag and then test the other processes flag in one indivisable action
- one solution: create an extra variable `turn` which determines whose turn it is to go to the critical section

Mutual Exclusion: Solution

```
Process P1
  loop
    flag1 := up
    turn := 2
    while flag2 = up and turn = 2 do
      null
    end
    <critical section>
    flag1 := down
    <non-critical section>
  end
end P1
```

```
Process P2
  loop
    flag2 := up
    turn := 1
    while flag1 = up and turn = 1 do
      null
    end
    <critical section>
    flag2 := down
    <non-critical section>
  end
end P1
```

- Published by Gary L. Peterson at 1981
  - Problem defined (critical section problem) by Dijkstra at 1960s
  - First similar (but more complex) algorithm by Dekker at 1968
- Now we can guarantee that both processes never enters the critical section for an infinite time
  - and process which starts its pre-protocol will eventually enter the critical section will eventually enter, regardless of the behavior of the other process

POSIX Threads: MUTEX

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
                                                        *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- a new data type named pthread_mutex_t is designated for mutexes
- a mutex is like a key (to access the code section) that is handed to only one thread at a time
- the attribute of a mutex can be controlled by using the pthread_mutex_init() function
- the lock/unlock functions work in tandem

MUTEX Example

```
    #include <pthread.h>
...
pthread_mutex_t my_mutex;    // should be of global scope
...
int main()
{
    int tmp;
    ...
    // initialize the mutex
    tmp = pthread_mutex_init( &my_mutex, NULL );
    ...
    // create threads
    ...
    pthread_mutex_lock( &my_mutex );
        do_something_private();
    pthread_mutex_unlock( &my_mutex );
    ...
    return 0;
}
```

Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked. If so, it waits until it is unlocked. Otherwise, it takes the mutex, locks the succeeding code, then frees the mutex and unlocks the code when it's done.

Semaphores

- Semaphore *S* – integer variable
- Two standard operations modify wait() and signal()
  - Originally called P() and V()

```
- wait (S) {
      while S <= 0
               ; // no-op
        S--;
  }

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

- Less complicated
- Can only be accessed via two indivisible (atomic) operations

Semaphores

Int sum = 0;

Thread 1:	Thread 2:
int t;	int t;
wait(sem)	wait(sem);
sum = sum + x;	sum = sum + y;
t = sum;	t = sum;
….	…
signal(sem);	signal(sem);

Use of semaphores for thread synchronization!

POSIX: Semaphores

- creating a semaphore:
  int sem_init(sem_t *sem, int pshared, unsigned int value);
  - initializes a semaphore object pointed to by sem
  - pshared is a sharing option; a value of *0* means the semaphore is local to the calling process
  - gives an initial value value to the semaphore

- terminating a semaphore:
  int sem_destroy(sem_t *sem);
  - frees the resources allocated to the semaphore sem
  - usually called after pthread_join()
  - an error will occur if a semaphore is destroyed for which a thread is waiting

POSIX: Semaphores *(cont.)*

- semaphore control:
  int sem_post(sem_t *sem);
  int sem_wait(sem_t *sem);

  sem_post *atomically* increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)

  sem_wait *atomically* decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

Semaphore: Example

```
#include <pthread.h>
#include <semaphore.h>
...
void *thread_function( void *arg );
...
sem_t semaphore;        // also a global variable just like mutexes
...
int main()
{
    int tmp;
    ...
    // initialize the semaphore
    tmp = sem_init( &semaphore, 0, 0 );
    ...
    // create threads
    pthread_create( &thread[i], NULL, thread_function, NULL );
    ...
    while ( still_has_something_to_do() )
    {
        sem_post( &semaphore );
        ...
    }
    ...
    pthread_join( thread[i], NULL );
    sem_destroy( &semaphore );
    return 0;
}
```


Semaphore: Example *(cont.)*

```
void *thread_function( void *arg )
{
    sem_wait( &semaphore );
    perform_task_when_sem_open();
    ...
    pthread_exit( NULL );
}
```


Deadlock

No deadlock

```
P1
wait (S1);
 wait (S2);
.
.
.
signal(S2);
signal(S1)
```

```
P2
wait (S1);
 wait (S2);
.
.
.
signal(S2);
signal(S1)
```

Deadlock

```
P1
wait (S1);
 wait (S2);
.
.
.
signal(S2);
signal(S1)
```

```
P2
wait (S2);
 wait (S1);
.
.
.
signal(S1);
signal(S2)
```

- If processes synchronises with many semaphores, can "mystique" fault be created
- In other example, the program can in some spesific cases be deadlocked

Dining Philosophers Problem

- Five philosophers spend their time eating and thinking.
- They are sitting in front of a round table with spaghetti served.
- There are five plates at the table and five chopsticks set between the plates.
- Eating the spaghetti requires the use of two chopsticks which the philosophers pick up one at a time.
- Philosophers do not talk to each other.
- Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
Do {
    wait ( chopstick[i] );               //lock
    wait ( chopStick[ (i + 1) % 5] );   //lock

        //  eat

    signal ( chopstick[i] );             //unlock
    signal (chopstick[ (i + 1) % 5] );  //unlock

        //  think

} while (true) ;
```


To Prevent Deadlock

- Ensures mutual exclusion, but does not prevent deadlock
- Allow philosopher to pick up her chopsticks only if both chopsticks are available (i.e. in critical section)
- Use an asymmetric solution: an odd philosopher picks up first her left chopstick and then her right chopstick; and vice versa

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), J.Kim (KAIST), J. Schaumann (SIT), E. Cuansing (Purdue), and J. Vuori (JYU).