CSE 3221.3
Operating System Fundamentals

No.6

Process Synchronization(2)

Prof. Hui Jiang

Dept of Computer Science and Engineering

York University

Semaphores

· **Problems with the software solutions.**
  – **Complicated programming, not flexible to use.**
  – **Not easy to generalize to more complex synchronization problems.**

· *Semaphore* (a.k.a. *lock*): **an easy-to-use synchronization tool**
  – **An integer variable *S***
  – *wait(S)* **{**
      while (*S*<=0) ;
       S-- ;
    }
 – *signal(S)* **{**
      S++ ;
    }

Semaphore usage (1):
the n-process critical-section problem

- The n processes share a semaphore,

  Semaphore *mutex ;*   *// mutex* **is initialized to 1**.

  Process P$_i$
  ```
  do {

          wait(mutex);

                  critical section of Pi

          signal(mutex);

                  remainder section of Pi

  } while (1);
  ```

Semaphore usage (2):
as a General Synchronization Tool

- Execute *B* in *P$_j$* only after *A* executed in *P$_i$*
- Use semaphore *flag* initialized to 0

P$_i$

```
…
A
signal (flag) ;
…
```

P$_j$

```
…
wait (flag) ;
B
…
```
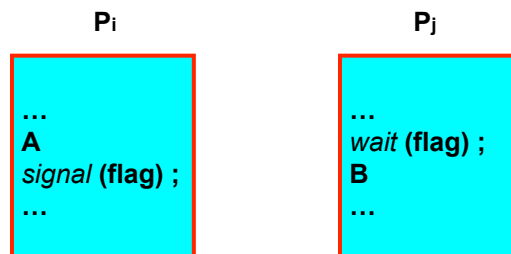
Spinlock vs. Sleeping Lock

- · **Previous definition of semaphore requires busy waiting.**
  - **It is called** *spinlock.*
  - *spinlock* **does not need context switch, but waste CPU cycles in a continuous loop.**
  - *spinlock* **is OK only for lock waiting is very short.**
- · **Semaphore without busy-waiting, called** *sleeping lock***:**
  - **In defining** *wait()***, rather than busy-waiting, the process makes system calls to block itself and switch back to waiting state, and put the process to a waiting queue associated with the semaphore. The control is transferred to CPU scheduler.**
  - **In defining** *signal(),* **the process makes system calls to pick a process in the waiting queue of the semaphore, wake it up by moving it to the ready queue to wait for CPU scheduling.**
  - **Sleeping Lock is good only for long waiting.**

Spinlock Implementation(1)

- · **In uni-processor machine, disabling interrupt before modifying semaphore.**

```
wait(S) {

  do {

    if(S>0) {
        S-- ;

        return ;
    }

  } while(1) ;
}
```

```
signal(S) {

    S++ ;

    return ;

}
```

Spinlock Implementation(1)

· **In uni-processor machine, disabling interrupt before modifying semaphore.**

```
wait(S) {

 do {
    Disable_Interrupt;
    if(S>0) {
       S-- ;
       Enable_Interrupt ;
       return ;
    }
    Enable_Interrupt ;
 } while(1) ;
}
```

```
signal(S) {

   Disable_Interrupt ;
   S++ ;
   Enable_Interrupt ;
   return ;

}
```

Spinlock Implementation(2)

· **In multi-processor machine, inhibiting interrupt of all processors is not easy and efficient.**
· **Use software solution to critical-section problems**
   – **e.g., bakery algorithm.**
   – **Treat** *wait()* **and** *signal()* **as critical sections.**
· **Or use hardware support if available:**
   – *TestAndSet()* **or** *Swap()*

· **Example: implement spinlock among two processes.**
   – **Use Peterson's algorithm for protection.**
   – **Shared data:**

   Semaphore *S ;* **Initially** *S***=1**

   boolean flag[2]; **initially** *flag [0] = flag [1] = false*.
   int turn;   **initially** *turn = 0* **or** *1*.

Spinlock Implementation(3)

```
wait(S) {
    int i=process_ID(); //0→P0, 1→P1
    int j=(i+1)%2 ;
 do {
  flag [ i ]:= true; //request to enter
  turn = j;
  while (flag [ j ] and turn = j) ;
  if (S >0) { //critical section
      S--;
     flag [ i ] = false;
     return ;
   } else {
     flag [ i ] = false;
   }
 } while (1);
}
```

```
signal(S) {
    int i=process_ID(); //0→P0, 1→P1
    int j=(i+1)%2 ;

    flag [ i ]:= true; //request to enter
    turn = j;
    while (flag [ j ] and turn = j) ;

     S++; //critical section

    flag [ i ] = false;

    return ;
}
```

Spinlock Implementation(2)

- **In multi-processor machine, inhibiting interrupt of all processors is not easy and efficient.**
- **Use software solution to critical-section problems**
  - **e.g., bakery algorithm.**
  - **Treat** *wait()* **and** *signal()* **as critical sections.**
- **Or use hardware support if available:**
  - *TestAndSet()* **or** *Swap()*

- **Example: implement spinlock between N processes.**
  - **Use Bakery algorithm for protection.**
  - **Shared data:**

  Semaphore *S ;* **Initially** *S*=1

  boolean choosing[N]; (**Initially** *false*)
  int number[N];   (**Initially** *0*)

Spinlock Implementation(3)

```
wait(S) {
    int i=process_ID();

    choosing[ i ] = true;
    number[ i ] = max(number[0], number[1],
    …, number [N – 1])+1;
    choosing[ i ] = false;
    for (j = 0; j < N; j++) {
        while (choosing[ j ]) ;
        while ((number[ j ] != 0) &&
                (number[ j ],j)< (number[ i ],i)) ;
    }
    if (S >0) { //critical section
        S--;
        number[i] = 0;
        return ;
    }
    number[i] = 0;
} while (1);
}
```

```
signal(S) {
    int i=process_ID();

    choosing[ i ] = true;
    number[ i ] = max(number[0], number[1],
        …, number [N – 1])+1;
    choosing[ i ] = false;
    for (j = 0; j < N; j++) {
        while (choosing[ j ]) ;
        while ((number[ j ] != 0) &&
                (number[ j ],j)< (number[ i ],i)) ;
    }

    S++; //critical section

    number[i] = 0;

    return ;
}
```

Sleeping Lock (I)

· **Define a sleeping lock as a structure:**

```
typedef struct {
    int value;   // Initialized to 1
    struct process *L;
} semaphore;
```

· **Assume two system calls:**
  – *block()* **suspends the process that invokes it.**
  – *wakeup(P)* **resumes the execution of a blocked process** P.

· **Equally applicable to multiple threads in one process.**

Sleeping Lock (II)

· **Semaphore operations now defined as:**

> *wait(S):*
>> *S.value--;*
>>
>> *if (S.value < 0) {*
>>> **add this process to** *S.L*;
>>> *block();*
>>
>> *}*

> *signal(S):*
>> *S.value++;*
>>
>> *if (S.value <= 0) {*
>>> **remove a process** *P* **from** *S.L*;
>>> *wakeup(P);*
>>
>> *}*

Two Types of Semaphores: Binary vs. Counting

· *Binary* **semaphore (a.k.a. mutex lock) – integer value can range only between 0 and 1; simpler to implement by hardware.**

· *Counting* **semaphore – integer value can range over an unrestricted domain.**

· **We can implement a counting semaphore** *S* **by using two binary semaphore.**

· **Binary semaphore is normally used as mutex lock.**

· **Counting semaphore can be used as shared counter, load controller, etc…**

Classical Synchronization Problems

- **The Bounded-Buffer P-C Problem**

- **The Readers-Writers Problem**

- **The Dining-Philosophers Problem**

Bounded–Buffer P-C Problem

- **A producer produces some data for a consumer to consume. They share a bounded-buffer for data transferring.**
- **Shared memory:**
  A buffer to hold at most *n* items
- **Shared data (three semaphores)**

Semaphore filled, empty; /*counting*/
Semaphore mutex;   /* binary */

Initially:

filled = 0, empty = n, mutex = 1

Bounded-Buffer Problem: Producer Process

```
do {
        …
        produce an item in nextp
        …
        wait(empty);
        wait(mutex);
        …
        add nextp to buffer
        …
        signal(mutex);
        signal(filled);

} while (1);
```

Bounded-Buffer Problem: Consumer Process

```
do {
        wait(filled)
        wait(mutex);
        …
        remove an item from buffer to nextc
        …
        signal(mutex);
        signal(empty);
        …
        consume the item in nextc
        …
} while (1);
```

The Readers-Writers Problem

· **Many processes concurrently access a data object**
  – **Readers: only read the data.**
  – **Writers: update and may write the data object.**

· **Only writer needs exclusive access of the data.**

· **The first readers-writers problem:**
  – **Unless a writer has already obtained permission to use the shared data, readers are always allowed to access data.**
  – **May starve a writer.**

· **The second readers-writer problem:**
  – **Once a writer is ready, the writer performs its write as soon as possible.**
  – **May starve a reader.**

The 1st Readers-Writers Problem

· **Use semaphore to implement 1st readers-writer problem**

· **Shared data:**

   int readcount = 0 ; **// keep track the number of readers**
   // accessing the data object

   Semaphore *mutex = 1 ;* **// mutually exclusive access to**
   // *readcount* **among readers**

   Semaphore *wrt = 1 ;* **// mutual exclusion to the data object**
   // used by every writer

   //also set by the 1st reader to read the data

   // and clear by the last reader to finish reading

The 1st Readers-Writers Problem

Writer Process

```
 …
wait(wrt);
 …
writing is performed
 …
signal(wrt);
 …
```

Reader Process

```
…
wait(mutex);
readcount++;
if (readcount == 1)   wait(wrt);
signal(mutex);
 …
reading is performed
…
wait(mutex);
readcount--;
if (readcount == 0)   signal(wrt);
signal(mutex);
…
```

The Dining-Philosophers Problem

- Five philosophers are thinking or eating
- Using only five chopsticks
- When thinking, no need for chopsticks.
- When eating, need two closest chopsticks.
- Can pick up only one chopsticks
- Can not get the one already in the hand of a neighbor.
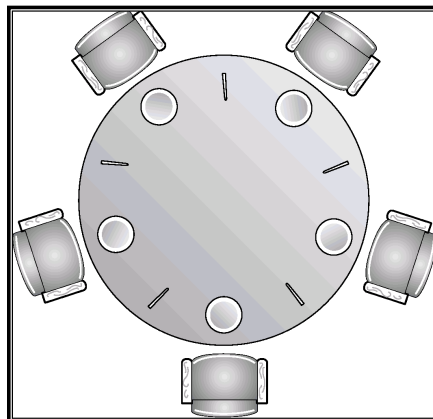
The Dining-Philosophers Problem: Semaphore Solution

· **Represent each chopstick with a semaphore**

      Semaphore *chopstick[5]*; **// Initialized to** *1*

Philosopher i
(i=0,1,2,3,4)

```
do {
    wait(chopstick[i]) ;
    wait(chopstick[(i+1) % 5]) ;
     …
     eat
     …
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
     …
     think
     …
} while (1);
```

Incorrect Semaphore Usage

Mistake 1:

```
…
signal(mutex) ;
…
Critical
Section
…
wait(mutex) ;
```

Mistake 2:

```
…
wait(mutex) ;
…
Critical
Section
…
wait(mutex) ;
```

Mistake 3:

```
…
wait(mutex) ;
…
Critical
Section
…
```

Mistake 4:

```
…
Critical
Section
…
signal(mutex) ;
```

Starvation and Deadlock

- *Starvation* **– infinite blocking. A process may never be removed from the semaphore queue in which it is suspended.**

- *Deadlock* **– two or more processes are waiting infinitely 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
wait(S);	*wait(Q);*
wait(Q);	*wait(S);*
⋮	⋮
signal(S);	*signal(Q);*
signal(Q)	*signal(S);*

double_rq_lock()
in Linux Kernel

```
double_rq_lock(struct runqueue *rq1,
   struct runqueue *rq2)
{
   if (rq1 == rq2)
      spinlock(&rq1->lock);
   else {
        if (rq1 < rq2) {
          spin_lock(&rq1->lock);
          spin_lock(&rq2->lock);
        } else {
          spin_lock(&rq2->lock);
          spin_lock(&rq1->lock);
        }
     }
}
```

Why not?

```
double_rq_lock(struct runqueue *rq1,
   struct runqueue *rq2)
{
    spin_lock(&rq1->lock);
    spin_lock(&rq2->lock);
}
```

```
struct runqueue *RdQ, *DevQ1, *DevQ2, …
```

P1	P2
…	…
double_rq_lock(RdQ,DevQ1);	double_rq_lock(DevQ1,RdQ);
…	…

---

double_rq_unlock()
in Linux Kernel

```
double_rq_unlock(struct runqueue *rq1,
   struct runqueue *rq2)
{
    spin_unlock(&rq1->lock);
      if (rq1 != rq2)
            spin_unlock(&rq2->lock);
}
```

Pthread Semaphore

· **Pthread semaphores for multi-threaded programming in Unix/Linux:**

– **Pthread Mutex Lock**
  (binary semaphore)

– **Pthread Semaphore**
  (general counting semaphore)

Pthread Mutex Lock

```
#include <pthread.h>

/*declare a mutex variable*/

pthread_mutex_t  mutex ;


/* create a mutex lock */

pthread_mutex_init (&mutex, NULL) ;


/* acquire the mutex lock  */

pthread_mutex_lock(&mutex) ;


/* release the mutex lock  */

pthread_mutex_unlock(&mutex) ;
```

Using Pthread Mutex Locks

· **Use mutex locks to solve critical section problems:**

```
#include <pthread.h>

pthread_mutex_t  mutex ;
…

pthread_mutex_init(&mutex, NULL) ;
…
pthread_mutex_lock(&mutex) ;

/*** critical section ***/

pthread_mutex_unlock(&mutex) ;
```

Pthread Semaphores

```
#include <semaphore.h>
/*declare a pthread semaphore*/
sem_t sem ;


/* create and initialize a semaphore */
sem_init (&sem, flag, initial_value) ;


/* wait() operation  */
sem_wait(&sem) ;


/* signal() operation  */
sem_post(&sem) ;
```

Using Pthread semaphore

· **Using Pthread semaphores for counters shared by multiple threads:**

```
#include <semaphore.h>

sem_t  counter ;

…

sem_init(&counter, 0, 0) ;  /* initially 0 */

…

sem_post(&counter) ;    /* increment */

…

sem_wait(&counter) ;    /* decrement */
```

volatile in multithread program

· **In multithread programming, a shared global variable must be declared as volatile to avoid compiler's optimization which may cause conflicts:**

```
volatile int data ;


volatile char buffer[100] ;
```

Process Synchronization
for multiple processes in Unix

· **In Unix, a shared global variable must be created with the following systems calls:**

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);

void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

nanosleep()

```
#include <time.h>

int nanosleep(const struct timespec *req,
              struct timespec *rem);

struct timespec
{
 time_t tv_sec; /* seconds */
 long tv_nsec; /* nanoseconds 0-999,999,999 */
};
```