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.**
  - **Not easy to generalize to more complex synchronization problems.**
  - **Complicated programming, not flexible to use.**
- **Semaphore: 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**

| $P_i$ | $P_j$ |
|---|---|
| **…** | **…** |
| **A** | *wait* **(flag) ;** |
| *signal* **(flag) ;** | **B** |
| **…** | **…** |

## Semaphore without busy-waiting

- **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:**
  - **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.**

## Semaphore without busy-waiting

- **Define a semaphore as a record**

```
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.
- **Normally this type of semaphore is implemented in kernel.**

## Semaphore without busy-waiting

- **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);
        }
```

## Semaphore Implementation(1)

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

```
wait(S) {

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

```
signal(S) {

  Disable_Interrupt ;
  S++ ;
  Enable_Interrupt ;
  return ;

}
```

## Semaphore 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**
- **Example: implement spinlock between two processes**
  - **Use Peterson's solution for process synchronization**
  - **Shared data:**

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

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

## Semaphore 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 ;
}
```

## Two Types of Semaphores

- ***Counting*** **semaphore – integer value can range over an unrestricted domain.**
- ***Binary*** **semaphore – integer value can range only between 0 and 1; simpler to implement by hardware.**
- **We can implement a counting semaphore *S* by using two binary semaphore.**

## Implementing counting semaphore *S* with Binary Semaphore

- **Data structures:**

  *binary-semaphore S1, S2;*
  *int C:*
- **Initialization:**

  *S1 = 1*
  *S2 = 0*
  *C* = **initial value of semaphore** *S*

## Implementing *S*

- ***wait(S)*** **operation**

```
wait(S1);
C--;
if (C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);
```

- ***signal(S)*** **operation**

```
wait(S1);
C ++;
if (C <= 0)
    signal(S2);
else
    signal(S1);
```

## Classic Problems of Synchronization

- **The Bounded-Buffer Problem**
- **The Readers-Writers Problem**
- **The Dining-Philosophers Problem**

## Bounded-Buffer 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 full, empty, mutex*;

  **Initially:**

  *full = 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(full);

} while (1);
```

## Bounded-Buffer Problem: Consumer Process

```
do {
        wait(full)
        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, the 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**

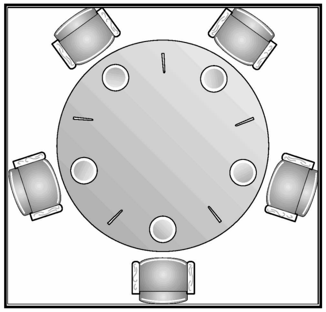
```
…
wait(wrt);
…
writing is performed
…
signal(wrt);
…
```

**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* **– indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.**
- *Deadlock* **– two or more processes are waiting indefinitely 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*);** |