Operating Systems (CSE531)
Lecture # 13

Manish Shrivastava
LTRC, IIIT Hyderabad

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- Used to to illustrate the power of synchronization techniques
- We assume that the buffer consists of n buffers, each capable of holding an item.
- The mutex semaphore provides mutual exclusion access to buffer which is initialized to the value 1.
- The **empty** and **full** semaphores count the number of empty and full buffers which are initialized to n and zero respectively.
- Shared data
  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);
```

- **Producer is producing full buffers for the consumer and consumer is producing empty buffers for the consumer.**

Readers-Writers Problem

- Problem: A data object (file or record) is shared among several concurrent processes.
  - Some want to read and others want to update it.
- **Readers:** processes interested in reading.
- **Writers:** processes interested in writing.
- Two readers can access shared data object simultaneously.
- But a writer and reader can access shared data object simultaneously
  - problems may occur!
- To protect from these problems, writers should have an exclusive access to the shared object.
- This synchronization problem is referred to as readers-writers problem.
- It is a different kind of synchronization problem.
- The readers-writers problem has several variations.
  - Simple one: No reader will be kept waiting unless writer has obtained permission to write.

Readers-Writers Problem

- Semaphores used: **mutex** and **wrt**
- The semaphore **wrt** is common to reader and writer.
- Semaphore **mutex** is used to update **readcount**.
- **readcount** keeps track of how many are reading the object.

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0

Readers-Writers Problem Writer Process

wait(wrt);

…

writing is performed

…

signal(wrt);

Readers-Writers Problem 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):
```

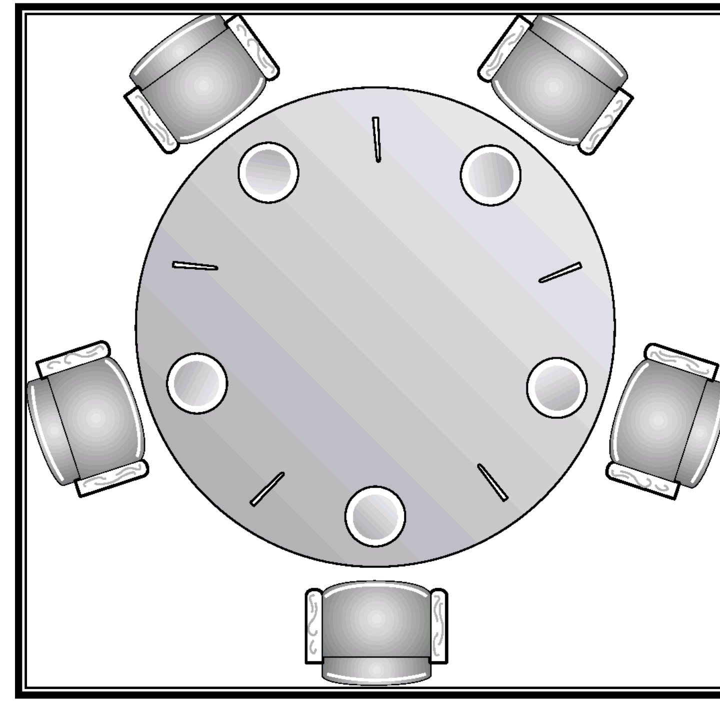
- **Writers can be starved if there is a continuous sequence of readers.**

Readers-Writers Problem

- Can the producer/consumer problem is considered as case of the readers/writers problem with a writer is a producer and reader is a consumer ?
- The answer is no
- The producer is not just a writer
  - It must read queue pointers to determine where to write the next item and it must determine if the buffer is full.
- Similarly the consumer is not a reader
  - It must adjust queue pointers to show that it has removed a unit from the buffer.

Dining-Philosophers Problem

- Five philosophers spend their lives thinking and eating.
- They share a common circular table surrounded by five chairs.
- Five single chopsticks are available.
- Whenever a philosopher wants to eat, he tries to pick up two chopsticks that are closest to him/her.
- A philosopher can not pick the chopstick in the hand of neighbor.
- After finishing, the philosopher puts back the chopsticks and starts thinking.



- It is simple representation of the need to allocate several resources among several processes in a **deadlock and starvation free manner.**

Dining-Philosophers Problem

- Shared data

  semaphore chopstick[5];
  Initially all values are 1

- Philosopher *i*:

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

- **The solution creates a deadlock**

Barbershop Problem

- 3 barbers, each with a barber chair
  - Haircuts may take varying amounts of time
- Sofa can hold 4 customers, max of 20 in shop
- Customers wait outside if necessary
- When a chair is empty:
  - Customer sitting longest on sofa is served
  - Customer standing the longest sits down
- After haircut, go to cashier for payment
  - Only one cash register
  - Algorithm has a separate cashier, but often barbers also take payment
    - This is also a critical section

Barbershop Problem

- The main body of the program activates 50 customers, 3 barbers, and the cashier process. Synchronization operators.
  - Shop and sofa capacity: the capacity of shop and the capacity of the sofa are governed by the semaphores **max_capacity** and **sofa.**
    - When customer enters max_capacity decremented by one.
    - When a customer leaves it is incremented.
    - Wait and signal operations are surround the actions of sitting and getting_up from sofa.
  - **Barber chair capacity:**
    - There are three barber chairs; the semaphore barber_chair assures that no more than three customers attempt to obtain service at a time.
    - A customer will not get up from the sofa until at least one chair is free.
  - **Ensuring customers are in the barber chair:** The semaphore cust_ready provides a wakeup signal for a sleeping barber indicating that the customer has just taken the chair.
  - **Holding customers in barber chair:** once seated the customer remain in the chair until the barber gives the signal that haircut is complete, using the semaphore finished.
  - **Limiting one customer to a barber chair:** the semaphore barber_chair is intended to limit the number of customers in barber chairs to three. The semaphore leave_b_chair is used to synchronize sitting.
  - **Paying and receiving:** payment and receipt semaphores are used to synchronize the operations.
  - **Coordinating barber and cashier functions:** To save money the barber shop does not employ a separate cashier. Each barber is required to perform that task when not cutting hair. The semaphore coord ensures the barbers perform only one task at a time.

Barbershop Problem

Semaphore	Wait operation	Signal operation
max_capacity	Customer waits for a room to enter shop.	Exiting customer signals customer waiting to enter
sofa	Customer waits for seat on sofa	Customer leaving sofa signals customer waiting for sofa
barber_chair	Customer waits for empty barber chair	Barber signals when that barber's chair is empty
Cust_read	Barber waits until customer is in the chair	Customer signals barber that customer is in the chair
finished	Customer waits until his haircut is complete.	Barber signals when done cutting hair of his customer.
leave_b_chair	Barber waits until customer gets up from the chair	Customer signals barber when customer gets up from chair.
payment	Cashier waits for a customer to pay	Customer signals cashier that he has paid.
receipt	Customer waits for a receipt for a payment	Cashier signals that payment has been accepted.
coord	Wait for a barber resource to be free to be free perform either the hair cutting or cashiering function.	Signal that a barber resource is free.

Barbershop

```
program     barbershop1;
var         max_capacity: semaphore (:=20);
            sofa: semaphore (:=4);
            barber_chair, coord: semaphore (:=3);
            cust_ready, leave_b_chair, payment, receipt: semaphore (:=0)
```

```
procedure customer;
var custnr: integer;
begin
   wait (max_capacity );
   enter shop;
   wait( sofa );
   sit on sofa;
   wait( barber_chair );
   get up from sofa;
   signal( sofa );
   sit in barber chair;
   wait( mutex2 );
  signal( cust_ready );
   wait( finished[custnr] );
   leave barber chair;
   signal( leave_b_chair );
   pay;
   signal( payment );
   wait( receipt );
   exit shop;
```

```
procedure barber;
var b_cust: integer
begin
   repeat
      wait( cust_ready );
      wait( coord );
      cut hair;
      signal( coord );
      signal( finsihed[b_cust] );
      wait( leave_b_chair );
      signal( barber_chair );
   forever
end;
```

```
procedure cashier;
begin
   repeat
      wait( payment );
      wait( coord );
      accept payment;
      signal( coord );
      signal( receipt );
   forever
end;
```

```
Void main()
{
 count=0;
Parbegin {customer… 50 times,…customer,
Barber, barber,barber, cashier)
}
```

Barbershop Problem

- The preceding solution is unfair.
- The customers are servers in the order they enter the shop.
- If one barber is very fast and one of the customer is quite bald.
- The problem can be solved with more semaphores.
  - We assign unique customer number is to each customer.
  - The semaphore mutex1 protects access to global variable count.
- The semaphore finished is refined to be an array of 50 semaphores.
  - Once a customer seated in a barber chair, he executes wait(finished[custnt]) to wait in his own unique semaphore.
- Please see the solution in Willium Stallling book (pp 229-234)

Fair Barbershop

```
program     barbershop2;
var         max_capacity: semaphore (:=20);
            sofa: semaphore (:=4);
            barber_chair, coord: semaphore (:=3);
            mutex1, mutex2: semaphore (:=1);
            cust_ready, leave_b_chair, payment, receipt: semaphore (:=0)
            finished: array [1..50] of semaphore (:=0);
            count: integer;
```

```
procedure customer;
var custnr: integer;
begin
   wait (max_capacity );
   enter shop;
   wait( mutex1 );
   count := count + 1;
   custnr := count;
   signal( mutex1 );
   wait( sofa );
   sit on sofa;
   wait( barber_chair );
   get up from sofa;
   signal( sofa );
   sit in barber chair;
   wait( mutex2 );
   enqueue1( custnr );
   signal( cust_ready );
   signal( mutex2 );
   wait( finished[custnr] );
   leave barber chair;
   signal( leave_b_chair );
   pay;
```

```
procedure barber;
var b_cust: integer
begin
   repeat
      wait( cust_ready );
      wait( mutex2 );
      dequeue1( b_cust );
      signal( mutex2 );
      wait( coord );
      cut hair;
      signal( coord );
      signal( finsihed[b_cust] );
      wait( leave_b_chair );
      signal( barber_chair );
   forever
end;
```

```
procedure cashier;
begin
   repeat
      wait( payment );
      wait( coord );
      accept payment;
      signal( coord );
      signal( receipt );
   forever
end;
```

```
Void main()
{
 count=0;
Parbegin {customer… 50 times,…customer,
Barber, barber,barber, cashier)
}
```

Implementation of Semaphores

- wait and signal operations are atomic.
- Good Solution: implement through hardware or firmware.
- Other solutions
  - Ensure that only process manipulates “wait” and “signal” operations.
  - One can use Dekker’s algorithm or Peterson’s algorithm
    - Substantial processing overhead
  - Use one of the hardware supported schemes
    - Test and set
    - disabling interrupts (single processor)
- The wait and signal code is very short the amount of busy waiting involved is short.

Two possible implementations of Semaphores

```
Wait(s)
{
   while(!testset(s.flag)
      /* do nothing */
    s.count--;
    if  (s.count <0)
      {
       place this process in s.queue;
       block this process (set s.flag to 0)
      }
    else
      s.flag=0;
 }
```

```
Signal(s)
{
   while(!testset(s.flag)
      /* do nothing */
    s.count++;
    if  (s.count <= 0)
      {
       remove a process P from  s.queue;
       Place a process P in the ready list
      }
      s.flag=0;
 }
```

With TestSet Instruction

```
Wait(s)
{
   Inhibit interrupts
    s.count--;
    if  (s.count <0)
      {
       place this process in s.queue;
       block this process allow interrupts
      }
    else
      allow interrupts;
 }
```

```
Signal(s)
{
   Inhibit interrupts;
    s.count++;
    if  (s.count <= 0)
      {
       remove a process P from  s.queue;
       Place a process P in the ready list
      }
      allow interrupts;
 }
```

With Interrupts

Problem with semaphores

- Incorrect use may result in timing errors
- These errors are difficult to detect as these occur if only particular sequence occurs.
- Missing or reverse order.
- It is difficult to produce correct program using semaphores.
- The wait and signal operations are scattered throughout the program and it is difficult to see overall effect of these operations on the semaphores.

Problem with semaphores (cont.)

- Problems
  - Suppose a process interchanges the order in which wait and signal operations on the semaphore are executed

    signal (mutex)
    CS
    wait (mutex)

    - Several processes may be executing in their CS simultaneously.
  - Suppose that a process replaces signal(mutex) with wait(mutex)

signal (mutex)	wait(mutex)
CS	CS
signal (mutex)	wait(mutex)

    - Deadlock will occur.
  - Suppose a process omits wait(mutex) or signal(mutex) or both.
    - ME is violated or deadlock occurs.
- A critical region and monitor concept is introduced to address this problem

THANK YOU