Process Synchronization

(Chap-7)

Topics

- Background
- The Critical-Section Problem
  - ✓ Two-process Solution (Algorithm 1 , 2 & 3)
  - ✓ Multi-process Solution (Bakery Algorithm)
- Semaphores
  - ✓ Introduction
  - ✓ Semaphore Implementation
  - ✓ Types of Semaphore
- Classical Problems of Synchronization
- Monitors

Background

- Cooperating processes may either directly share a logical address space (i.e. both code and data), or be allowed to share data only through files.
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem (Chapter 4) allows at most *n* - 1 items in buffer at the same time. A solution, where all *N* buffers are used is not simple.
  - ✓ Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer (cont.)

- Producer process

```
item nextProduced;

while (1) {
  while (counter == BUFFER_SIZE)
      ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

Bounded-Buffer (cont.)

- Consumer process

```
item nextConsumed;

while (1) {
  while (counter == 0)
       ; /* do nothing */
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;
}
```

Bounded-Buffer (cont.)

- The producer and consumer routines will work properly when they run separately, but may not function correctly when executed concurrently.
- After concurrent execution of statements “counter++” and “counter--”, the value of counter will not be correct.
- That’s why the statements: “counter++” and “counter--” must be performed *atomically*.
- Atomic operation means an operation that completes in its entirety without interruption.
- The same can be proved in a machine language also by implementing the counter value by taking some registers.

Bounded-Buffer (cont.)

- The statement “count++” may be implemented in machine language as:

```
 register1 = counter
register1 = register1 + 1
counter = register1
```

- The statement “count--” may be implemented as:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Bounded-Buffer (cont.)

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Assume counter is initially 5. One interleaving of statements is:

Producer: $register_1$ = counter	{ $register_1 = 5$ }
producer: $register_1 = register_1 + 1$	{ $register_1 = 6$ }
consumer: $register_2$ = counter	{ $register_2 = 5$ }
consumer: $register_2 = register_2 – 1$	{ $register_2 = 4$ }
producer: counter = $register_1$	{ *counter* = 6 }
consumer: counter = $register_2$	{ *counter* = 4 }

The value of count may be either 4 or 6, where the correct result should be 5.

Race Condition

- **Race condition:** The situation where several processes access and manipulate the shared data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called race condition.

- To prevent race conditions, concurrent processes must be synchronized.

The Critical-Section Problem

- Consider a system consisting of n processes $\{P_0, P_1, \ldots P_{n-1}\}$.
- Each process has a code segment, called *"critical section"*, in which the process may changing common variables, updating a table, writing a file, and so on.
- **Problem (feature)** - ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
  - ✓ Thus the execution of critical sections by the processes is ***mutually exclusive***.
- Each process must request permission to enter its critical section. The section of code for request is *"entry section"*.
- The critical section may be followed by an *"exit section"*.
- The remaining code is *"remainder section"*.

Solution to Critical-Section Problem

A solution to the critical section problem must satisfy the following three requirements:

1. **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting:** There exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

General Structure of Process P_i

```
do {

        entry section

        critical section

        exit section

        reminder section
    } while (1);
```

Two-Process Solutions

- The algorithm is restricted only for two process.
- The processes are numbered P_0 and P_1.
- In general case, if one process is P_i then other one is P_j, where j = 1-i .
- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables: int turn;
  - The value of turn either 0 or 1 .
  - Initially, turn is set to 0 (i.e. turn = 0)
  - If turn == i, then *Pi* can enter its critical section
- The structure of Process *Pi*

```
do {
    while (turn != i) ;
        critical section
    turn = j;
        reminder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

Algorithm 2

- **Shared variables**
  - ✓ boolean flag[2];
  - ✓ initially **flag [0] = flag [1] = false.**
  - ✓ **If flag [i] = true** $\Rightarrow P_i$ is ready to enter its critical section
- **Process P_i**

```
do {
    flag[i] = true;
    while (flag[j]) ;
  critical section
    flag [i] = false;
        remainder section
} while (1);
```

Explanation of Algorithm 2

- Process P_i first sets flag[i] to be true, signaling that it is ready to enter its critical section. Then, P_i checks whether P_j is executing in its critical section or not.
- If P_j is executing in its critical section then P_i has to wait until P_j is not longer needed the critical section (i.e. until falg[j] was false).
- Now, P_i can enter to the critical section and by exiting from critical section P_i would set flag[i] to false, allowing the other process (if it is waiting) to enter its critical section.
- So, mutual-exclusion requirement satisfied, but progress requirements is not met. To illustrate the problem, consider the following execution sequence:
  - P_0 sets flag[0] = true & P_1 sets flag[1] = true
  - Now P_0 and P_1 are looping forever in their respective while statements.

Algorithm 3

- **Shared variables:**
  - ✓ By combining the key ideas of Algorithm 1 & 2 .
  - ✓ boolean flag[2];
  - ✓ int turn;
- **Process P_i**

```
        do {
            flag[i] = true;
            turn = j;
            while (flag[j] && turn == j) ;
                critical section
            flag[i] = false;
                remainder section
        } while (1);
```

Explanation of Algorithm 3

- Initially flag[0]=flag[1]=false, and the value of turn is immaterial (but is either 0 and 1).
- P_i will enters its CS iff either flag[j] == false or turn == i.
- If both P_i and P_j wants to execute in their CS at the same time, then flag[0]=flag[1]=true. But the value of turn could be either 0 or 1, allowing only one process to be in the CS at any point of time. Hence mutual exclusion is preserved.
- Similarly, progress and bounded waiting can be proved.
- As, when turn==i, P_i will enter in its CS and when turn==j, P_j will enter in its critical section and exiting from the critical section they will resets the flag value.
- So, after at most one entry (bounded waiting) another process will enter critical section (progress).

Multiple-Process Solutions

- This algorithm also known as ***bakery algorithm.***
- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section first.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 0,1,2,3,3,3,3,4,5...
- Since the process names are unique and totally ordered, this algorithm is completely deterministic.

Bakery Algorithm

- Common data structures (Shared data) are:

```
boolean choosing[n];
int number[n];
```

- Data structures are initialized to false and 0 respectively.
- For convenience, the following notations are used:
  - $(a,b) < (c,d)$ if $a < c$ or if $a == c$ and $b < d$
  - $\max(a_0,\ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0,\ldots, n - 1$.

Bakery Algorithm

```
do {
        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))) ;
                }

                critical section

        number[i] = 0;

                remainder section

} while (1);
```

Semaphores

- It is a synchronization tool used to generalize more complex problem.
- Semaphore *S* is an integer variable, apart from initialization, it can only be accessed via two standard atomic operations: wait
  (“P”-*proberen, to test*) and signal (“V”-*verhogen, to increment*).
- The classical definition of wait and signal in pseudocode are:

```
    wait (S){
         while (S≤ 0); // do no-op;
         S--;
    }
```

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

When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

In addition the testing of (*S*≤ 0) and its possible modification (S--) in wait(S), must also execute without interruption.

Usage

- **Critical Section of *n* Processes**
- n process share a semaphore mutex (mutual exclusion), initialized to 1.
- The structure of Process *Pi:*

```
do {
        wait(mutex);
            critical section

        signal(mutex);
            remainder section
    } while (1);
```

Mutual exclusion implementation with semaphore

Semaphore as a General Synchronization Tool

- Consider two currently running processes: P_1 with a statement S_1 and P_2 with a statement S_2.
- Suppose we require that S_2 be executed only after S_1 has completed.
- It can be implemented by using a common semaphore ***synch*** between P_1 and P_2 and initialized to 0.
- Now insert the statements

```
S1;
signal (synch);
```

in process P1

and the statements

```
wait (synch);
 S2;
```

in process P2.

- As synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after S1.

Semaphore as a General Synchronization Tool

Requirements:

- Execute *B* in P_j only after *A* executed in P_i

Solution:

- Use a semaphore *flag* initialized to 0
- Code:

```
        Pi                  Pj
        ⋮                   ⋮
        A              wait (flag);
  signal (flag);            B
```

Semaphore Implementation

- Problem in Petersen's algorithm as well as in semaphore is that, *busy waiting.* (continuously looping in entry code).
- Busy waiting wastes CPU cycles that other process can be used to improve productivity.
- This type of semaphore is also called *spinlock* (because the process "spins" while waiting for the lock).
- So, to overcome the busy waiting problem a slight modification is required in wait and signal semaphore.
- When a process executes wait operation and finds the value of semaphore is not positive, then instead of busy waiting it can *block* itself using *block()* system call.
- This blocked process can be restarted by a *wakeup* operation, when some other process executes signal operation.

Semaphore Implementation

- Define a semaphore as a “C” struct:

```
typedef struct {
        int value;
        struct process *L;
} semaphore;
```

- Each semaphore has an integer value and a list of processes.
- When a process must wait on a semaphore, it is added to the list of processes and then block.
- A signal operation removes one process from the list of waiting processes and awakens that process.

Implementation

- Semaphore operations now defined as:

```
void wait (semaphore S){
        S.value--;
        if (S.value < 0) {
                add this process to S.L;
                block();
        }
}
void signal (semaphore S){
        S.value++;
        if (S.value <= 0) {
                remove a process P from S.L;
                wakeup(P);
        }
}
```

Deadlock and Starvation

- **Deadlock:** Set of processes is in a deadlock state when every process in the set is waiting for an event that can caused only by another process in the set.
- Let *S* and *Q* be two semaphores initialized to 1.

```
      P0                      P1
   wait(S);                wait(Q);
   wait(Q);                wait(S);
      ⋮                       ⋮
   signal(S);              signal(Q);
   signal(Q)               signal(S);
```

- **Starvation** - indefinite blocking.
  - It can be occurred if we add and remove processes from the list associated with a semaphore in LIFO order.

Types of Semaphores

- Two types of semaphore:
- *Counting semaphore* - integer value can range over an unrestricted domain.
- *Binary semaphore* - integer value can range only between 0 and 1.can be simpler to implement.
- A binary semaphore is simpler to implement than counting semaphore depending on h/w architecture.

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1

Bounded-Buffer Problem (cont.)

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 (cont.)

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);
```

Readers-Writers Problem

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0

- No readers will be kept waiting unless a writer has already obtained permission to use shared object.

Readers-Writers Problem

Writer Process

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

Readers-Writers Problem (cont.)

Reader Process

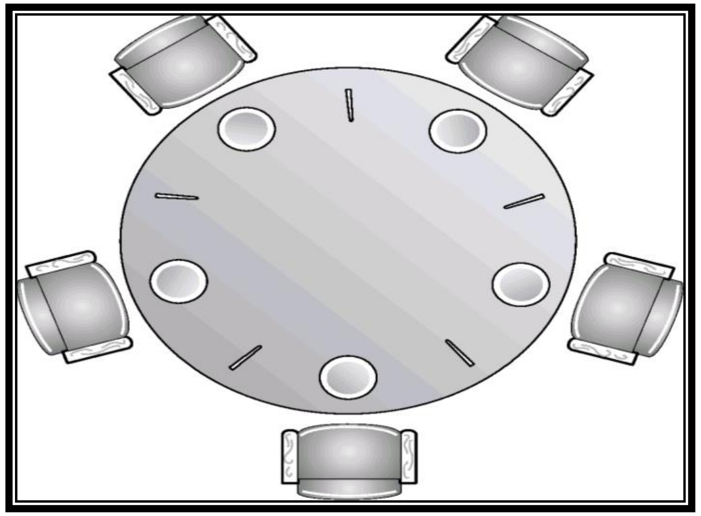
```
wait(mutex);
readcount++;
if (readcount == 1)
      wait(wrt);
signal(mutex);
      …
   reading is performed
      …
wait(mutex);
readcount--;
if (readcount == 0)
   signal(wrt);
signal(mutex):
```

Dining-Philosophers Problem

- Shared data

  semaphore chopstick[5];

  Initially all values are 1



- **Represent each chopstick by a semaphore.**
- **A philosopher tries to grab the chopstick by executing a wait operation on that semaphore & she releases her chopsticks by executing the signal operation on the appropriate semaphores.**

Dining-Philosophers Problem (cont.)

- **Philosopher *i*:**

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

Monitors

- Another high-level synchronization construct.
- A monitor is characterized by a set programmer defined operators.
- The representation of a monitor type consists of declaration of variables (whose value define the state of an instance of the type), as well as the bodies of procedures or functions that implement operations on the type.
- The general syntax of a monitor is shown in the next slide.
- The representation of a monitor type can't be used directly by the various processes.
- Thus, a procedure defined with in a monitor can access only those variables declared locally within the monitor and its formal parameters, and local variables of a monitor can be accessed by only the local procedures.
- The monitor construct ensures that only one process at a time can be active within the monitor.

Monitors (cont.)

```
monitor *monitor-name*
{
    shared variable declarations
    procedure body *P1* (…) {
        . . .
    }
    procedure body *P2* (…) {
        . . .
    }
    procedure body *Pn* (…) {
        . . .
    }
    {
        initialization code
    }
}
```

Monitors (cont.)

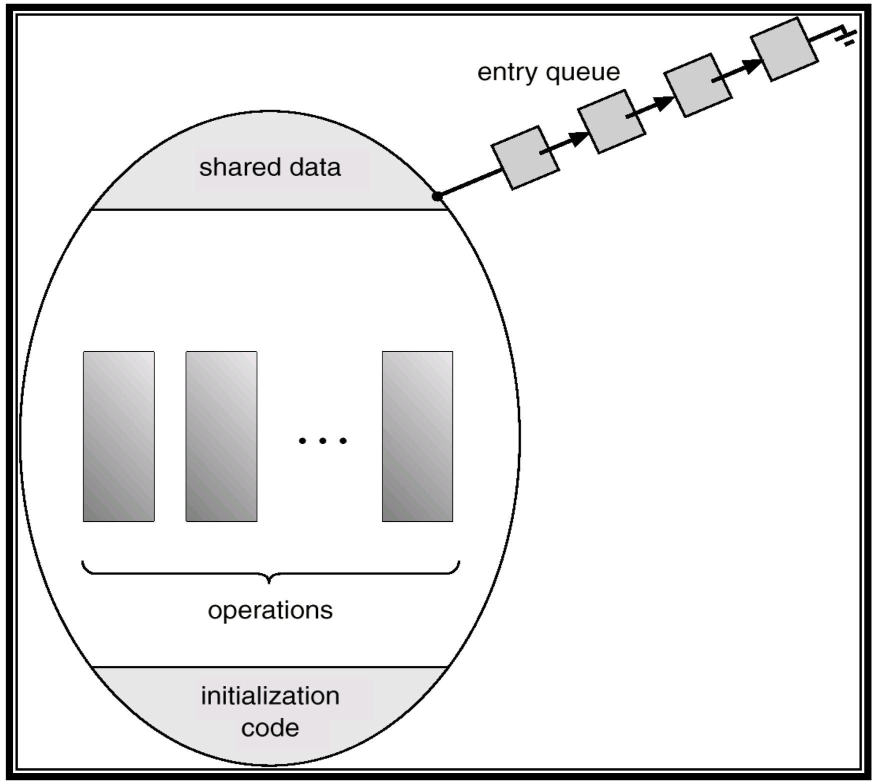
- Sometimes this monitor construct is not efficient for modeling some synchronization schemes, and for that we need to define additional synchronization schemes.
- These mechanism are provided by the condition construct.
- A programmer who needs to write his own synchronization schemes can define one or more variables of type *condition*.

```
condition x, y;
```

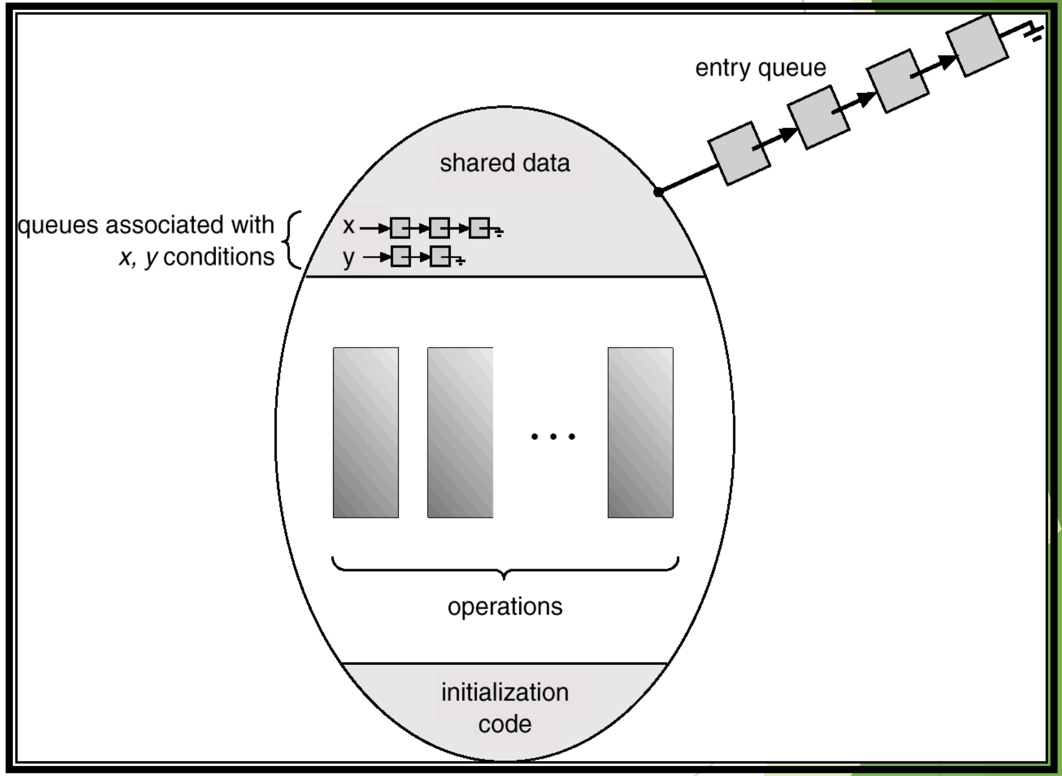
- The only operation that can be invoked on a condition variable are wait and signal.
- The operation x.wait() means that the process invoking this operation is suspended until another process invokes x.signal();
- The x.signal operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect; that is the state of x as is though the operation were never executed.
- Note: the signal operation associated with semaphores always affects the state of the semaphore.

Schematic View of a Monitor

Simple Monitor



Monitor With Condition Variables



Dining Philosophers Example

```
monitor dp
{
 enum {thinking, hungry, eating} state[5];
 condition self[5];
 void pickup(int i)        // following slides
 void putdown(int i)      // following slides
 void test(int i)        // following slides
 void init() {
      for (int i = 0; i < 5; i++)
           state[i] = thinking;
 }
```

Dining Philosophers (cont.)

```
void pickup(int i) {
      state[i] = hungry;
      test[i];
      if (state[i] != eating)
            self[i].wait();
}

void putdown(int i) {
      state[i] = thinking;
      // test left and right neighbors
      test((i+4) % 5);
      test((i+1) % 5);
}
```

Dining Philosophers (cont.)

```
void test(int i) {
        if ( (state[(i + 4) % 5] != eating) &&
          (state[i] == hungry) &&
          (state[(i + 1) % 5] != eating)) {
                state[i] = eating;
                self[i].signal();
}         }

  Void init() {
        for ( int i = 0; i < 5; i++ )
              state[i] = thinking;
  }
} /* End of monitor */
```

It will executed in this sequence:

```
dp.pickup(i);
        …
        eat
        …
dp.putdown(i);
```

Thank You