Process Synchronization

Bhuvana J
Associate Professor

Department of CSE, SSNCE

Session Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Agenda

Background

- Processes can execute concurrently
- May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Example:
  In producer consumer problem:
  producer and consumer access the counter to increment and decrement

Bounded-Buffer Producer

```
while (true) { /* produce an item in next p
while (counter == BUFFER_SIZE) ;
/* do nothing */
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

Race Condition

- `counter++` could be implemented as

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

- `counter--` could be implemented as

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

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Critical Section Problem

- Consider system of n processes $\{p0,\ p1,\ \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
- Process may be changing common variables, updating table, writing file, etc
- When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Algorithm for Process P_i

```
do {

        while (turn == j);

        critical section
        turn = j;

        remainder section
   } while (true);
```

Solution to Critical-Section Problem

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 there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist 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

Critical-Section Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive
  - Preemptive allows preemption of process when running in kernel mode
  - Non-preemptive runs until exits kernel mode, blocks, or voluntarily yields CPU Essentially free of race conditions in kernel mode

Peterson's Solution

- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  int turn;
  Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section. **flag[i] = true** implies that process P_i is ready

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

Peterson's Solution

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved P_i enters CS only if: either **flag[j] = false or turn = i**
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Protecting critical regions via **locks**
- Uniprocessors could disable interrupts
  - Currently running code would execute without preemption
- Generally too inefficient on multiprocessor systems
- Modern machines provide special **atomic hardware instructions**
- Atomic = non-interruptible
  - Either **test** memory word and **set** value
  - Or **swap** contents of two memory words

Solution to Critical-section Problem Using Locks

```
        do {
        acquire lock
                  critical section
        release lock
              remainder section
} while (TRUE);
```

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
 {
        boolean rv = *target;
        *target = TRUE;
        return rv:
 }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE".

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE

```
do{
     while (test_and_set(&lock));
         /* do nothing */
        /* critical section */
        lock = false;
        /* remainder section */
  } while (true);
```

compare_and_swap Instruction

```
int compare_and_swap(int *value, int expected,
                                         int new_value)
{
        int temp = *value;
        if (*value == expected)
                *value = new_value;
        return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter value
3. Set the variable "**value**" to the value of the passed parameter "new_value" but only if "value" =="expected".

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
  Solution:

```
do{
      while (compare_and_swap(&lock, 0, 1) != 0);
                 /* do nothing */
          /* critical section */
          lock = 0;
      /* remainder section */
   } while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
do{
        waiting[i] = true;      key = true;
        while (waiting[i] && key)
                key = test_and_set(&lock);
        waiting[i]        = false;
        /* critical section */
        j = ( i + 1) & n;
        while ((j != i) && !waiting[j])
            j =  ( j + 1) % n;
        if (j==i)
           lock = false;
        else
           waiting[j] = false;
        /* remainder section */
} while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting
  - This lock therefore called a **spinlock**

acquire() and release()

```
   acquire() {
        while (!available)
        ; /* busy wait */
        available = false;
   }
release() {
        available = true;
     }
do {
        acquire lock
        critical section
        release lock
        remainder section
  } while (true);
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore S integer variable; Can only be accessed via two indivisible (atomic) operations wait() and signal()
- Originally called P() and V(); Definition of the wait() operation

```
wait(S) {
        while (S <= 0)
        ; // busy wait
        S--;
}
```

- Definition of the signal() operation

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

Semaphore Usage

- **Counting semaphore** integer value can range over an unrestricted domain
- **Binary semaphore** integer value can range only between 0 and 1
- Same as a mutex lock
- Can solve various synchronization problems
- Consider **P1** and **P2** that require **S1** to happen before **S2**
- Create a semaphore "synch" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore S as a binary semaphore

Semaphore Implementation

- Must guarantee that **no two processes can execute** the **wait() and signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait and signal** code are placed in the critical section
  - Could now have busy waiting in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block place the process invoking the operation on the appropriate waiting queue
  - wakeup remove one of processes in the waiting queue and place it in the ready queue

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

Implementation with no Busy waiting

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}

signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Deadlock and Starvation

- **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);`

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol

Classical Problems of Synchronization

Classical problems used to test newly-proposed synchronization schemes

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

Bounded-Buffer Problem

- **n** buffers, each can hold one item
- Semaphore **mutex** initialized to the value **1**
- Semaphore **full** initialized to the value **0**
- Semaphore **empty** initialized to the value **n**

The structure of the producer process

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);
```

The structure of the consumer process

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** only read the data set; they do not perform any updates
  - **Writers** can both read and write
- Problem allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered all involve some
- form of priorities
- Shared Data
  - Data set
  - Semaphore **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read_count** initialized to 0

Readers-Writers Problem

```
do {
        wait(rw_mutex);
        ...
        /* writing is performed */
        ...
        signal(rw_mutex);
} while (true);
```

Readers-Writers Problem

The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
        wait(rw_mutex);
        signal(mutex);
        ...
        /* reading is performed */
        ...
        wait(mutex);
        read count--;
        if (read_count == 0)
        signal(rw_mutex);
        signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

- First variation no reader kept waiting unless writer has permission to use shared object
- Second variation once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

- Philosophers spend their lives alternating thinking and eating
- Dont interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher **i**:

```
do {
        wait (chopstick[i] );
        wait (chopStick[ (i + 1) % 5] );

        //  eat

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

        //  think

} while (TRUE);
```

Dining-Philosophers Problem Algorithm

Deadlock handling

- Allow at most 4 philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
- Use an asymmetric solution – an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations:
- signal (mutex) . wait (mutex)
- wait (mutex) wait (mutex)
- Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

Monitors

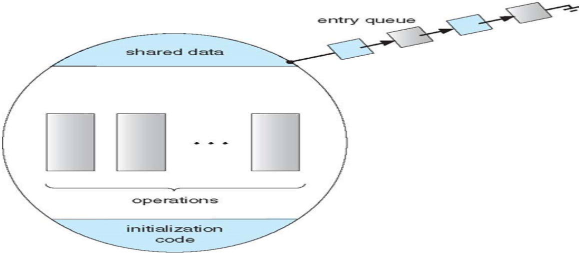
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     Initialization code (…) { … }
   }
}
```

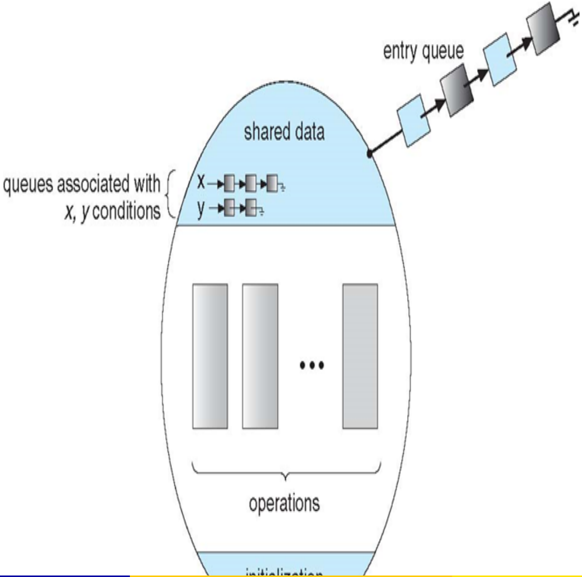
Schematic view of a Monitor



Condition Variables

- condition **x, y**;
- Two operations are allowed on a condition variable:
  - **x.wait()** a process that invokes the operation is suspended until x.signal()
  - **x.signal()** resumes one of processes (if any) that invoked x.wait()
  - If no x.wait() on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **x.signal()**, and process Q is suspended in **x.wait()**, what should happen next?
- Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
enum { THINKING; HUNGRY, EATING) state [5] ;
                condition self [5];

        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);
```

Monitor Solution to Dining Philosophers

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

initialization_code() {
for (int i = 0; i < 5; i++)
state[i] = THINKING;
}
}
```

Solution to Dining Philosophers

- Each philosopher i invokes the operations pickup() and putdown() in the following sequence:

$$\begin{array}{c} \mathit{DiningPhilosophers.pickup(i)}; \\ \mathit{EAT} \\ \mathit{DiningPhilosophers.putdown(i)}; \end{array}$$

No deadlock, but starvation is possible

Summary

- Mutual exclusion must be provided to ensure that a critical section of code is used by only one process
- Synchronization problems : bounded-buffer problem, the readerswriters problem, and the dining-philosophers problem
- Semaphores can be used to solve various synchronization problems and can be implemented efficiently

Test your understanding

- What is race condition?
- Two operations of semaphore
- Two hardware instructions
- What do you meant by a monitor?