Operating Systems

9. CPU: Synchronization (1)

Hyunchan, Park

http://oslab.chonbuk.ac.kr

Division of Computer Science and Engineering

Chonbuk National University

Contents

- Background
- Problem: Race condition
- The Critical Section
  - Peterson's Solution
- Hardware Support for Synchronization
- Synchronization mechanisms
  - Mutex Locks
  - Semaphores
  - Monitors

Background

- Processes can execute concurrently
  - Both on uni-processor and multi-processor systems
    (or single- and multi-core systems)
  - 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

Background

- Illustration of the problem:
  - Suppose that we wanted to provide a solution to the **consumer-producer problem** that fills ***all*** the buffers
  - We can do so by having an integer **`counter`** that keeps track of the number of full buffers
  - Initially, **`counter`** is set to 0.
  - It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer

Producer

```
while (true) {
        /* produce an item in next produced */

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

Consumer

```
while (true) {
      while (counter == 0) ; /* do nothing */

      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
       counter--;
      /* consume the item in next consumed */
}
```

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

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- 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);
```

Example: Algorithm for Process P_i

```
do {

    while (turn == j);

          critical section

    turn = j;

          remainder section

 } while (true);
```

Solution to Critical-Section Problem

- **Three conditions to be satisfied for the CS solution**
  - Assume that each process executes at a nonzero speed
  - No assumption concerning **relative speed** of the ***n*** processes

1. Mutual Exclusion

- If process P_i is executing in its critical section,
  then no other processes can be executing in their critical sections

Solution to Critical-Section Problem

2. Progress

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in deciding which will enter the critical section next, and this selection 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

Problem with two processes

- Shared variables:
  - **int turn = 0; (initial value)**
  - When **turn = 0** , P_0 enters the critical section

- Process P_0 / Process P_1

Process P_0	Process P_1
while (turn != 0) ; **critical section** **turn = 1;** **remainder section**	**while (turn != 1) ;** **critical section** **turn = 0;** **remainder section**

- Satisfied: mutual exclusion, bounded waiting
- Unsatisfied : progress
  - If P_1 is scheduled before P_0 at starts of executions

Problem with two processes: another algorithm

- Shared variables:
  - **boolean flag[2]; flag [0] = flag [1] = false (initial value)**
  - When **flag [0] = true**, P_0 enters the critical section
  - When **flag [1] = true**, P_1 enters the critical section

- Process P_0

```
flag[0] = true;
while (flag[1]) ;
    critical section
flag[0] = false;
    remainder section
```

Process P_1

```
flag[1] = true;
while (flag[0]) ;
    critical section
flag[1] = false;
    remainder section
```

- Satisfied: mutual exclusion, bounded waiting
- Unsatisfied : progress
  - Two flags can be set as TRUE at same time

Peterson's Solution

- Good algorithmic description of solving the problem
- 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 $\mathbf{P_i}$ is ready!

Problem with two processes: Peterson's solution

- Shared variables:
  - **int turn = 0; boolean flag[2]; flag [0] = flag [1] = false (initial value)**
  - When **flag [0] = true and turn = 0**, P_0 enters the critical section
  - When **flag [1] = true and turn = 1**, P_1 enters the critical section
- Process P_0 / Process P_1

Process P_0	Process P_1
flag [0] = true;	**flag [1] = true;**
turn = 1;	**turn = 0;**
while (flag [1] && (turn == 1)) ;	**while (flag [0] && (turn == 0)) ;**
critical section	critical section
flag [0] = false;	**flag [1] = false;**
remainder section	remainder section

- Satisfied: mutual exclusion, progress, bounded waiting

Peterson’s Solution: Limitation

- How about with more than three processes?
  - Hard to implement
  - Hard to prove that it satisfies all the three conditions
  - There is an assumption: atomic Load and Store instructions

- We need more general and simple solution

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
- Preemptive kernel is more preferred, but hard to implement
  - Pros: More responsiveness for the processes
  - Cons: Need to manage the shared kernel data structures with fine-grained manner
    - Can be slow down for the management

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible (or indivisible)
  - 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”.

Mutual exclusion using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

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

compare_and_swap Instruction

Definition:

```
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" the value of the passed parameter "new_value" but only if "value" =="expected". That is, the swap takes place only under this condition.

Mutual exclusion 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);
```

Example: Bounded-waiting Mutual Exclusion with test_and_set()

```
/* Process Pi ,
Initialization: waiting[all], key, lock = false*/


do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
```

Example: Bounded-waiting Mutual Exclusion with test_and_set()

```
    next = (i + 1) % n;
    while ((next != i) && !waiting[next])
        next = (next + 1) % n;
    if (next == i)
        lock = false;
    else
        waiting[next] = false;
    /* remainder section */
} while (true);
```

Synchronization Hardware: Limitation

- Mutual exclusion is easily solved and implemented with HW support
  - The others (progress, bounded waiting) must be solved by SW

- We need the more comprehensive synchronization mechanisms

Synchronization mechanisms

- Mutex locks
- Semaphore
- Monitor

Mutex lock (spinlock)

- Simplest solution
  - Previous solutions are complicated and generally inaccessible to application programmers
  - OS designers build software tools to solve critical section problem
- 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
- Cons: requires busy waiting
  - This lock therefore called a spinlock

acquire() and release()

- ```
  acquire() {
        while (!available)
              ; /* busy wait */
        available = false;;
     }
  ```

- ```
  release() {
        available = true;
     }
  ```

```
Example Solution with Locking
  do {
        acquire();
              critical section
        release();
              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()`** **`(or P()`** and **`V())`**

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

```
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

Semaphore Usage

- Consider P_1 and P_2 that require S_1 to happen before S_2

  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: Binary semaphore

- Binary semaphore with Test-and-Set instruction
- Semaphore S : if True, there is a process inside the critical section (True or False)
- P(S)
  - while(testandset(&S));
  - Current value of S is returned, and S is changed to True
- V(S)
  - S = false;
  - Enables that the process waiting with Wait() can be entered into critical section

Semaphore Implementation: Counting semaphore

- Counting semaphore implementation using Binary semaphore
  - CS: Counting semaphore, value is C
  - Two binary semaphore S1, S2
  - Initialize S1 = 1, S2 = 0 , C = n
    - (n is number of co-executed process in critical section)

- wait operation

```
P(S1);
C--;
if ( C<0) {
        V (S1);
        P(S2);
} else
        V(S1);
```

- signal operation

```
P(S1);
C++;
if (C <= 0) {
        V(S2);
}
         V(S1);
```

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

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

- 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

Implementation with no Busy waiting (Cont.)

```
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);
   }
}
```

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.

Deadlock and Starvation

- **Deadlock** – two or 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(Q);`	`wait(S);`
`wait(S);`	`wait(Q);`
`...`	`...`
`signal(Q);`	`signal(S);`
`signal(S);`	`signal(Q);`

Deadlock and Starvation

- 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

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

- Pros: Easy to use
- Cons: But not powerful enough to model some synchronization schemes

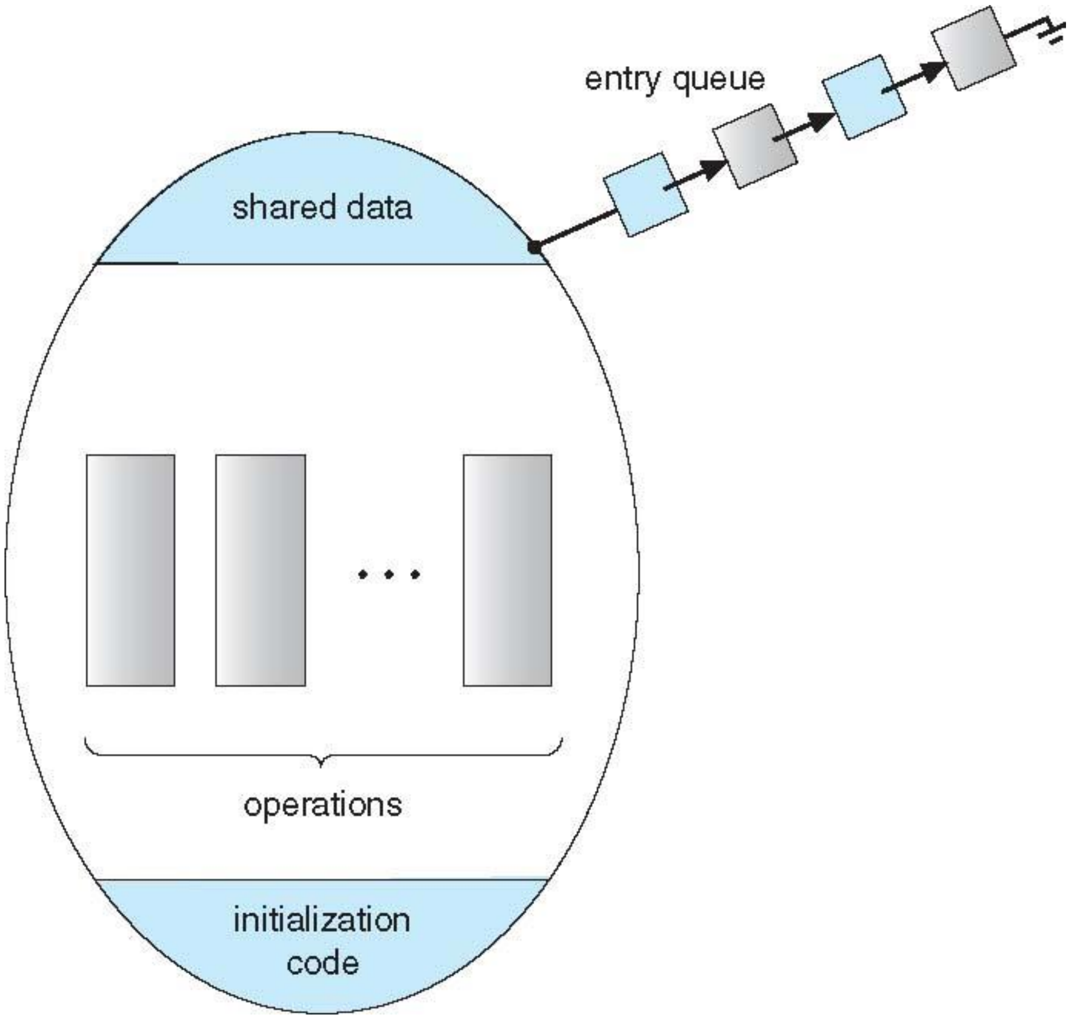
Monitors

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

  procedure Pn (…) {……}

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

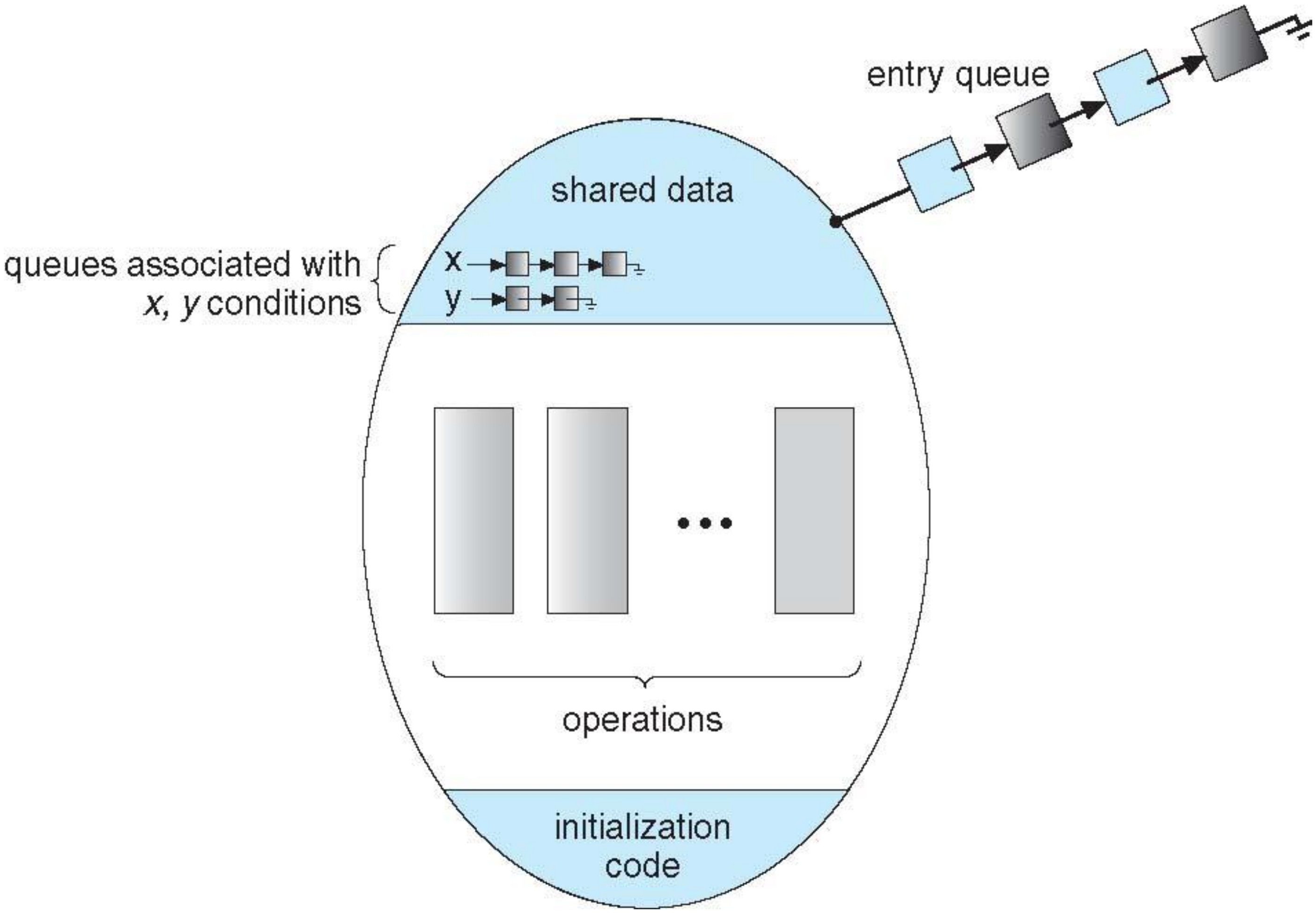
Schematic view of a Monitor



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 Issues

- 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 paralel. 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