CS343 - Operating Systems

Module-3C

Process Synchronization – Critical Sections

Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

http://www.iitg.ac.in/johnjose/

Session Outline

- Background
- The Critical-Section Problem
- Peterson′s Solution
- Synchronization Hardware
- Mutex Locks

Objectives of Process Synchronization

- To introduce 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

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
- 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.

Bounded-Buffer – Producer & Consumer

```
item buffer[BUFFER_SIZE]; int in = 0; int out = 0;
```

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 Problem

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

Critical Section

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

- General structure of process **P**

```
do {
    [entry section]
        critical section
    [exit section]
        remainder section
} while (true);
```

```
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

   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Peterson's Solution

- Applicable for 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 $\mathbf{P_i}$ is ready!

Peterson's Solution

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j]&&turn==j);

    critical section

    flag[i] = false;

    remainder section

  } while (true);
```

Algorithm for Process P_j

```
do {
    flag[j] = true;
    turn = i;
    while (flag[i]&&turn==i);

    critical section

    flag[j] = false;

    remainder section

  } while (true);
```

Peterson's Solution

- All 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

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j]&&turn==j);

    critical section

    flag[i] = false;

    remainder section

  } while (true);
```

Synchronization Hardware - Locks

- 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
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

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

Synchronization Using test_and_set Instruction

```
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);
```

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

Synchronization Using 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 “new_value” only if “value” ==“expected”. That is, the swap takes place only under this condition.

Solution using compare_and_swap ()

- Shared integer lock initialized to 0;

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

Synchronization Using acquire() and release()

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

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

Thank you

johnjose@iitg.ac.in
http://www.iitg.ac.in/johnjose/