CSE 325: Operating Systems
3rd Year Computer Engineering
Zagazig University

SPRING 2018

LECTURE #6

Dr. Ahmed Amer Shahin

Dept. of Computer & Systems Engineering

Chapter 5: Process Synchronization

Chapter 5: Process Synchronization

Background

The Critical-Section Problem

Peterson's Solution

Synchronization Hardware

Mutex Locks

Semaphores

Classic Problems of Synchronization

Monitors

Synchronization Examples

Alternative Approaches

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

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.

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 {*p0, p1, … pn-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 *Pi*

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

Algorithm for Process Pi

```
do {

    while (turn == j);

        critical section
    turn = j;

        remainder section
 } while (true);
```

Solution to Critical-Section Problem

1. Mutual Exclusion - If process *Pi* 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

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

Good algorithmic description of solving the problem

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 *Pi* is ready!

Algorithm for Process Pi

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

Peterson's Solution (Cont.)

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

   Pi 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.

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
- 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:
   }
```

Executed atomically

Returns the original value of passed parameter

Set the new value of passed parameter to “TRUE”.

Solution 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.

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 (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);
   }
}
```

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

P0	P1
`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