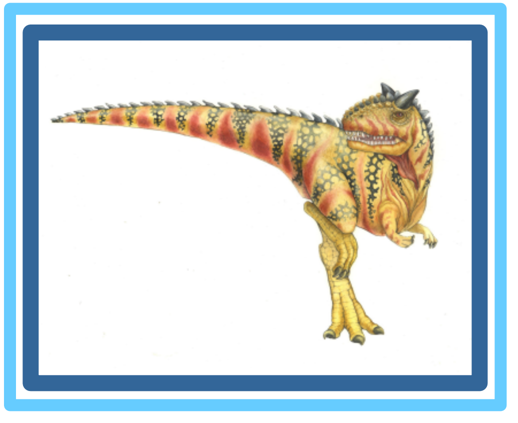
Chapter 6: Synchronization Tools



Chapter 6: Synchronization Tools

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

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

Silberschatz, Galvin and Gagne ©2018

Background

Cooperating process - Processes that can affect or get affected by other processes executing in the system.

Cooperating processes can be:

- Directly sharing a logical address space(both code and data)
- Allowed to share data only through files or messages

Problem: Concurrent access to shared data may result in data inconsistency

Solution: Process synchronization

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Eg: Producer consumer problem
- Solution : with the use of shared memory
- Buffer: the region in the shared memory that both producer and consumer use
- The producer produce one item while the consumer is consuming another item
- The producer and consumer should be synchronized

Background

Buffer: the region in the shared memory that both producer and consumer use

Two types of buffer:

- Unbounded — No limit on the size of the buffer. The producer can always produce items, consumer may need to wait for new items
- Bounded — Fixed buffer size. Consumer must wait when buffer is empty. Producer must wait when the buffer is full

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.

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}

When several processes access and manipulate the data concurrently and the outcome of the execution depends on the particular order in which the access takes place — Race Condition

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 */
}
```

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … 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);
```


Silberschatz, Galvin and Gagne ©2018

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

https://www.youtube.com/watch?v=6x_XMDCMyAk.

Peterson's Solution

Good algorithmic description of solving the problem

Two process solution — Peterson's solution is restricted to two processes that alternate the execution between the critical section and remainder section.

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!

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);
```

Peterson's Solution (Cont.)

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

Silberschatz, Galvin and Gagne ©2018

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

Solution to Critical-section Problem Using Locks

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

Quiz

1. What are the two types of processes?
2. Which type of processes are in need of synchronization methods?
3. Why do we need process synchronization?
4. What are the three issues in producer consumer problem?
5. What are the two types of buffer?
6. What is a critical section?
7. What are the 4 sections in a process's structure?
8. What are the three criteria to avoid critical section problem?
9. What are the counter operations in producer consumer problem?