# **Chapter 6: Process Synchronization**

**Part 1**

## **6.1 Background**

## **6.2 The Critical-Section Problem**

## **6.3 Petersen’s Solution to Critical Section Problem**

**6.4 Hardware Support**

**6.5 Mutex Locks**

**6.6 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

## **6.1 Producer Consumer Problem – Motivation**

## **Producer Consumer Problem**

## A collection of processes called **producers** produce data and write to a common buffer. A collection of processes called **consumers** consume the data in the common buffer.

## **Issues**

## Processes can execute concurrently on a single CPU

## May be interrupted at any time, partially completing execution

## Concurrent access to shared data may result in data inconsistency

## **Goal**

## Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

## **Goal of correct solution**

## Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

## Consumer waits when buffer is empty

## Producer waits when buffer is full.

## Buffer data is not lost

## Buffer data is not overwritten

## **Shared Data :**

## const int BUFFER\_SIZE = 60;

## int counter = 0;

## Item buffer[BUFFER\_SIZE];

# **6. 1 Code for Producer Process**

## int in = 0;

## while (true) { /\* produce an item in next\_produced \*/

## 

## while (counter == BUFFER\_SIZE) ;

## /\* do nothing \*/

## buffer[in] = next\_produced;

## in = (in + 1) % BUFFER\_SIZE;

## counter++;

## }

# **6.1 Code for Consumer Process**

## int out = 0;

## while (true) {

## while (counter == 0)

## ; /\* do nothing \*/

## next\_consumed = buffer[out];

## out = (out + 1) % BUFFER\_SIZE;

## counter--;

## /\* consume the item in next consumed \*/

## }

# **6.1 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 = 6” initially:

### S0: producer execute **register1 = counter** {register1 = 6} S1: producer execute **register1 = register1 + 1** {register1 = 6} S2: consumer execute **register2 = counter** {register2 = 6} S3: consumer execute **register2 = register2 – 1** {register2 = 4} S4: producer execute **counter = register1** {counter = 6 } S6: consumer execute **counter = register2** {counter = 4}

## **6.2 The Critical-Section Problem**

## CS problem is a generalized version of problems that require synchronization

## 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

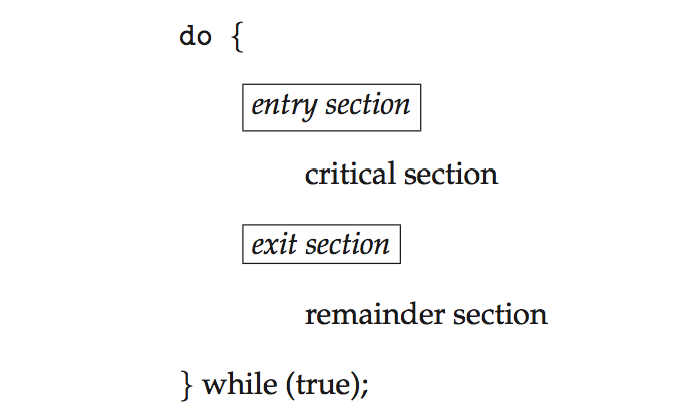
### When one process in critical section, no other may be in its critical section

## ***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 Problem Representation**

## General structure of process ***Pi***



# **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

# **Peterson’s solution**

## **do { do{**

## **flag[i] = true; flag[j] = true;**

## **turn = j; turn = i;**

## **while (flag[j] && turn = = j); while (flag[i] && turn== i);**

## **critical section critical section**

## **flag[i] = false; flag[j] = false;**

## **remainder section remainder section**

## **} while (true); } while (true);**

## 

## Algorithm for Process Pi Algorithm for Process Pj

## 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

## 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

## Modern machines provide special atomic hardware instructions

#### **Atomic** = non-interruptible

### Either test memory word and set value

### Or swap contents of two memory words

# **6.4 Solution to Critical-section Problem Using Locks**

## **do {**

## **acquire lock**

## **critical section**

## **release lock**

## **remainder section**

## **} while (TRUE);**

# **6.4 Atomic Operation Test\_And\_Set**

## 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”.

# **6.4 Solution to CS problem 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);**

## 

# **6.4 Solution to CS problem using test\_and\_set()**

## Does the solution using test\_and\_set offer Mutual Exclusion, Progress and Bounded Waiting?

## Justify

## 

# **6.4 Bounded-waiting 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);**

## **6.6 Semaphore Solution to Critical Section Problem**

## 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

## *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

## **Assume** L<M<H are priorities of processes sharing a resource

L low

M Medium

H is High

H might have to wait a long time if it is sharing a resource with L since L may have given it to M.

### Solved via **priority-inheritance protocol**