Chapter 6:

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.

Race Condition

counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

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**.

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 يمنح طوعيا

----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!.

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**.

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**".

Shared Boolean variable lock, initialized to **FALSE**

compare_and_swap Instruction

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.

Shared integer “lock” initialized to 0;

Mutex Locks

Previous solutions are complicated and generally i**naccessible** to application programmers.

OS designers build software tools to solve **critical section p**roblem.

Simplest is mutex lock.

P**rotect 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**.

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**()

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

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

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**.

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.