**OPERATING SYSTEMS LEC 03**

**Background**

* A cooperating process can affect or be affected by the other processes executed in the system. These processes may share a logical address space or data through files.
* Concurrent access can lead to data inconsistency

**Atomic Counter**

* Atomic operation is an operating that completes entirely without interruption
* If producer executes counter += 1 and consumer executes counter -= 1 then the result should be the same from before the two operations.
  + If producer and consumer execute separately
  + If each instruction is executed atomically

**Race Condition**

* Situation where processes access and manipulate same data concurrently
  + Outcome is based on order of access
* To prevent this happening to counter, only one process at a time should be manipulating it.
  + Process synchronisation

**Critical Selection Problem**

* Every process has a code segment called a critical section in which share data is accessed.
* The problem is to make sure when a process is executing its critical section no other process is allowed to execute in this critical section.

**Solution must satisfy:**

(assuming process execute at non-zero speed and no assumption of relative speed of the n processes or number of CPUs)

* **Mutual exclusion**: If a process is in the critical section then no other process can be executed in their critical sections.
  + No processes may be simultaneously inside their critical sections
* **Progress**: If no processes are in its critical section and another wished to enter their critical section, then only processes not in their remainder section can participate in the decision as to which enter its critical section next
  + No process running outside its critical section may block other processes from entering their critical sections.
* **Bounded Waiting**: A bound for number of times other processes can enter critical section after a process requests to enter its critical section and before that request is granted
  + No process should have to wait forever to enter critical section

**Solution**

* Each process disables all interrupts after just after entering its critical section, re-enabling them just before leaving it (bad because it is privileged instruction)

In **kernel mode**:

* **Pre-emptive kernel**: A process can be pre-empted while running on kernel mode
  + Otherwise kernel is non-pre-emptive meaning it allows the process to run until is exits kernel mode, blocks or voluntarily yields CPU.
* **Non-pre-emptive kernel** is free from race conditions on kernel data structure
  + Pre-emptive is more responsive and suitable for real time system

**Bakery Algorithm**

* Before entering critical section, each process receives a number.
  + Holder of smallest number enters critical section.
  + If process P*i* and P*j* receive same number. If *i* < *j* then P*i* is served first
* Notation (**ticket#**, process **id#**)
  + (*a*, *b*) < (*c*, *d*) if *a* < *c* or if *a* = *c* and*b* < *d*
  + max(*a0*, ..., *an*-1) is a number *k* such that *k* ≥ *ai* for *i* = 0, ..., *n*-1.
* Shared data
  + **var** choosing: **array**[0..n-1] of Boolean;
  + number: **array** [0 .. *n*-1] of integer;
* Data structures initialised to false and 0 respectively

A screenshot of a cell phone

Description automatically generated

**Synchronisation Hardware**

* Software has no guarantee to work correctly. Simple solution is to disable interrupt while a shared variable is being modified.
  + However not feasible in multiprocessor
* Use hardware instructions such as test-and-set and swap
  + Atomic instructions that cannot be interrupted until completed.

A screenshot of a social media post

Description automatically generated

**Mutex Locks**

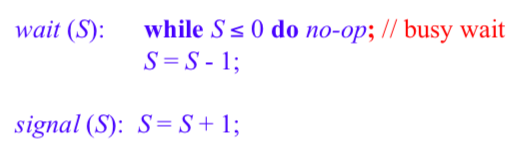
* Hardware solution is complex and inaccessible to programmers. So use mutual exclusion (mutex).
* Mutex: must acquire a lock before entering a critical section and release lock on exit
  + Calls to acquire() or release() must be atomic
  + Implement functions with hardware solutions
  + Busy waiting wastes CPU, but doesn’t need context switch – good for short wait. This is implemented on multiprocessor systems.

A screenshot of a cell phone

Description automatically generated

**Semaphores**

* An integer variable, ‘S’, that can only be accessed by two atomic operations



* Busy waiting is also known as spinlock; that is, the process spins waiting for the lock to be false
* Counting semaphore – integer value can range over an unrestricted domain
* Binary semaphore – integer value can only range between 0 and 1.

**Deadlock/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
  + Process may never be removed from queue in which it is suspended.

**Priority Inheritance Protocol**

* All processes that are accessing resources needed by a higher priority process inherits the higher priority until they finish using the resources.

**Bounded-Buffer Problem**

* With semaphores the producer produces full buffers for the consumer.

**Readers-Writers problem**

* Synchronisation problems involving reading and writing shared data objects.
  + Data can be accessed by more than one reader simultaneously.
  + When a writer accesses data no other writer or reader can access simultaneously.
* No reader will be waiting unless a writer has already obtained permission to use shared object
  + Reader has higher priority
* If writer is in critical section and *n* readers are waiting, then:
  + One read processes is execute on write
  + *n*-1 reader processes are queued on mutex.
* On Linux gets the lock of type
  + Eg. Reader gets the reader lock, writer gets the writer look.

**Semaphore Limitation**

* Incorrect use of semaphore can result in timing errors that are hard to detect.
* Forgetting signal or wait (or both!) can create deadlock or no mutual exclusion.

**Monitors**

* A monitor is a high-level synchronisation construct that allows safe sharing of an abstract data type among concurrent processes.
* Monitor type contains set of defined functions and set of shared variables that can be accessed by only one function.
* Monitors ensure only one process at a time is active inside of it.
* Condition variables can only be used with:
  + Wait: process invoking this operation is suspended until another process invokes.
  + Signal: operation resumes one suspended process. If no process is suspended then signal operation has no effect – not a semaphore.

**Semaphores for Monitor**

* Use a mutex variable for each monitor.
  + Process executes wait before entering monitor and signal on exit.