Chapter 5 Process Synchronization

1. **Process Synchronization – Outline:** Objectives of this chapter include: 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.

2. **Why Process Synchronization ?** A process that can affect or be affected by other processes is called a cooperating process.

   Sharing may be achieved via shared memory. A context switch saves the local state of the process, and OS protection guards it from illegal access. But the shared process state in not guarded. In case of multicore processors two cooperating process may try to access shared data concurrently. We will see examples of concurrent shared data access producing different values shortly.

   So, you can well gather that this is an issue that only comes due to the sharing of data between processes!

3. **Producer-Consumer System Sharing Data:** Bounded buffer problem. Buffer can hold 10 items.

   Question: What is the shared data here? *counter* is the shared data variable. Both the consumer and producer update counter. [Buffer is only written in the producer.]

4. **Producer-Consumer System Sharing Data:** Although updates to counter seem to happen in one instruction, on several (RISC) architectures, they may require multiple assembly instructions. This is shown here.

5. **Race Condition:** Since we are only interested in the shared data, I have only shown those instructions here. Now, lets look at 2 possible execution orders for these six instructions that can produce incorrect values. These can happen due to context switches happening asynchronously, outside the control of the applications.

   The correct counter value should be 5. Howver, context switching on one processor or concurrent execution on multiple processors can produce different execution ordering.

   To guard against race condition, we need to ensure that only one process at a time can update the shared variable.

6. **Critical Section Problem:** We have seen some of the problems concurrent updating can lead to. Thus no two processes can be in their critical sections at the same time.

   Thus the protocol is quite simple. Each cooperating process should follow these steps. The OS or external library should ensure single access. Although the protocol appears simple, its implementation in real systems throws several interesting challenges. In the remainder of this chapter, we will see how to implement this protocol.

7. **Solution the the Critical Section Problem:** Progress: If no process is executing in its critical section, and some processes wish to enter their critical sections, then only those processes not executing in their remainder sections can participate in deciding which will enter the critical section, and the selection cannot be postponed indefinitely.

   Most commonly, this bound is defined in terms of how many other processes enter their critical section before this request is granted.

8. **Preemptive Vs. Non-Preemptive Kernels:** For example, Kernel data structure that maintains a list of open files can be modified by multiple processes simultaneously.

   Non-preemptive kernel: Process executing in kernel mode cannot be preempted. However, the process can voluntarily exit the kernel mode by finishing the function, or blocking on something. Cannot be used directly on multiprocessor systems since two processes can be executing on different processors and reach kernel state concurrently. On uniprocessors, this solution works very well. Drawback: A process running in kernel mode will never be preempted, so system may not be very responsive.

   Several OS use non-preemptive kernels and design it so that bad cases do not happen often.

9. **Peterson's Solution to the Critical Section Problem:**

10. **Peterson's Solution:** The eventual value of `turn` determines which process gets to enter the critical section.

    Note that the procedure sets turn to 'otherProcess' before busywaiting. This is actually quite cleaver. It's to ensure fairness. It is not silly, because if process #1 comes along while process #0 is in its critical section, process #1 sets 'turn' to 0 and waits politely. It is fair because it means that if two processes call 'enterCriticalSection' at about the same time, the last one to set 'turn' is the one which waits.

    Mutual exclusion P0 and P1 can never be in the critical section at the same time: If P0 is in its critical section, then flag[0] is 1 and either flag[1] is 0 or turn is 0. In both cases, P1 cannot be in its critical section.

    Progress requirement If process P0 does not want to enter its critical section, P1 can enter it without waiting. There is not strict alternating between P0 and P1.

    Bounded waiting A process will not wait longer than one turn for entrance to the critical section: After giving priority to the other process, this process will run to completion and set its flag to 0, thereby allowing the other process to enter the critical section.

11. **Peterson's Solution – Notes:** This does not demand any architectural feature.

    The solution works correctly on any hardware in which references to main memory are atomic. The main disadvantage is that it requires busy waiting, continuous polling of a status variable, before gaining access to the critical region if another process is already in the critical region.

    However, a purely software approach for ¿2 processes may be inefficient. When working at the hardware level, Peterson's algorithm is typically not needed to achieve atomic access. Next, we will look at some hardware features that are normally available in most architectures today to address this problem.

12. **Lock Based Solutions:** Remember, only one process get get the lock. Other's must wait. We will now see some hardware based solutions to implement these lock-based protocols.

13. **Hardware Support for Lock-Based Solutions – Uniprocessors:** Hardware support are always better than a pure software based mechanism: more efficient, and easier! A classic example of harware support is for interrupts: during the execution of each instruction there is a test to see if a request for interrupt has arrived. If it has and is of sufficient priority, the instruction is suspended and the interrupt handled. Interrupts are made possible by (hidden) polling!

    Uniprocessor systems are a special case, where solutions might be easier, since two cooperating processes cannot really run at the same time, since we only have 1 CPU, but context switches may result in their interleavings. Context switches happen on timer interrupts. Hardware support for disable/enable interrupts can provide a simple lock-based solution.

14. **Hardware Support for Lock-Based Solutions – Multiprocessors:** Multiprocessors are different from uniprocessors, so interrupt diabling is ineffective and costly. Costly, since messages need to be passed to all processors. Disabled interrupts may also affect the workings of some system resources, in particular clocks.

15. **TestAndSet Instruction:** This is the pseudo-code! Real hardware implementation is atomic.

    In multiprocessor-multicore systems, two or more processes executing this instruction

16. **using the same memory address** are guaranteed to occur in a serial but undefined order.

17. **Mutual Exclusion using TestAndSet:** We define three operations to provide mutual exclusion: initlock, lock, and unlock. The critical region is protected using the locking framework provided earlier.

    Mutex is a variable shared between the cooperating processes. We pass a pointer to this shared variable.

    The implementation is quite simple and with no system overhead since TestAndSet is a user mode instruction. It works because of the properties of the TestAndSet instruction. Notice that this solution involves Busy Waiting (polling).

18. **Swap Instruction:** Operates on the contents of two words. Also execuetd atomically.

19. **Mutual Exclusion Using Swap:** If the machine supports swap instruction, then mutual exclusion can be provided as shown here.

    Again, mutex is a shared variable. Only when mutex is 0, will key get the value 0 and break out of the loop. Mutex can only be 0, if no other process is currently in the critical section. Since, this instruction is atomic, only one swap can occur at a time.

    If two processes P and Q are trying to enter a critical region protected by a locked spinlock mutex, then there is no guaranty that they will be serviced in FIFO order. Their order of entry will be random. More serious is the problem that the process P may use the spinlock within a loop and use the lock repeatedly while Q is unable to acquire it.

20. **Bounded Waiting Solution:** Here we have inlined the init_lock, lock, and unlock operations in this code to be able to demonstrate the algorithms clearly. Note that we can also abstract the lock, unlock portions is desired.

    The array waiting[] is shared. (Is variable key shared? No!). Lock is shared. We can suppose that the processes are executted concurrently on two multi-processors.

    In cycle 4, suppose that Process 0 wins the race, and sets lock. Note that only one process can execute the testAndSet instruction at a time since it locks the (memory) bus during execution. (One one instruction is highlighted) Since key = 0, process 0 breaks out of the loop, and into the critical section. waiting[0] is set to false. As long as lock=TRUE, all future calls to TestAndSet will return TRUE.

    In cycle 9, if j == i, then we have completed the loop and no other process is waiting to enter the critical region. We then set lock = FALSE.

    On cycle 10, waiting[1] is set tp 0 by process 0. Therefore, process 1 can now exit the while loop, not because key==0, but because waiting[1]==0. This is different than what happened earlier on cycle 4.

Thus, if there are n processes, then each process has only to wait for at most (n-1) other processes to enter-exit their critical sections, before the last process gets its chance, after making a request.

Also realize, than for hardware designers, implementing atomic instructions such as TestAndSet and Swap is not easy for multiprocessors.