OS Part-3

-crafted by Love Babbar and Lakshay Kumar

Introduction to Concurrency

1. **Concurrency** is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel.
2. **Thread**:
    - Single sequence stream within a process.
    - An independent path of execution in a process.
    - Light-weight process.
    - Used to achieve parallelism by dividing a process's tasks which are independent path of execution.
    - E.g., Multiple tabs in a browser, text editor (When you are typing in an editor, spell checking, formatting of text and saving the text are done concurrently by multiple threads.)
3. **Thread Scheduling:** Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system.
4. **Threads context switching**
    - OS saves current state of thread & switches to another thread of same process.
    - Doesn't includes switching of memory address space. (But Program counter, registers & stack are included.)
    - Fast switching as compared to process switching
    - CPU's cache state is preserved.
5. **How each thread get access to the CPU?**
    - Each thread has its own program counter.
    - Depending upon the thread scheduling algorithm, OS schedule these threads.
    - OS will fetch instructions corresponding to PC of that thread and execute instruction.
6. **I/O or TQ, based context switching is done here as well**
    - We have TCB (Thread control block) like PCB for state storage management while performing context switching.
7. **Will single CPU system would gain by multi-threading technique?**
    - Never.
    - As two threads have to context switch for that single CPU.
    - This won't give any gain.
8. **Benefits of Multi-threading.**
    - Responsiveness
    - Resource sharing: Efficient resource sharing.
    - Economy: It is more economical to create and context switch threads.
        1. Also, allocating memory and resources for process creation is costly, so better to divide tasks into threads of same process.
    - Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Critical Section Problem and How to address it

1. Process synchronization techniques play a key role in maintaining the consistency of shared data
2. **Critical Section (C.S)**
    a. The critical section refers to the segment of code where processes/threads access shared resources, such as common variables and files, and perform write operations on them. Since processes/threads execute concurrently, any process can be interrupted mid-execution.
3. **Major Thread scheduling issue**
    a. Race Condition
        i. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e., both threads are "racing" to access/change the data.
4. **Solution to Race Condition**
    a. Atomic operations: Make Critical code section an atomic operation, i.e., Executed in one CPU cycle.
    b. Mutual Exclusion using locks.
    c. Semaphores
5. Can we use a simple flag variable to solve the problem of race condition?
    a. No.
6. **Peterson's solution** can be used to avoid race condition but holds good for only 2 process/threads.
7. **Mutex/Locks**
    a. Locks can be used to implement mutual exclusion and avoid race condition by allowing only one thread/process to access critical section.
    b. Disadvantages:
        i. Contention: one thread has acquired the lock, other threads will be busy waiting, what if thread that had acquired the lock dies, then all other threads will be in infinite waiting.
        ii. Deadlocks
        iii. Debugging
        iv. Starvation of high priority threads.

Conditional Variable and Semaphores for Threads synchronization

1. **Conditional variable**
    a. The condition variable is a synchronization primitive that lets the thread wait until a certain condition occurs.
    b. Works with a lock
    c. Thread can enter a wait state only when it has acquired a lock. When a thread enters the wait state, it will release the lock and wait until another thread notifies that the event has occurred. Once the waiting thread enters the running state, it again acquires the lock immediately and starts executing.
    d. Why to use conditional variable?
        i. To avoid busy waiting.
    e. **Contention** is not here.
2. **Semaphores**
    a. Synchronization method.
    b. An integer that is equal to number of resources
    c. Multiple threads can go and execute C.S concurrently.
    d. Allows multiple program threads to access the finite instance of resources whereas mutex allows multiple threads to access a single shared resource one at a time.
    e. Binary semaphore: value can be 0 or 1.
        i. Aka, mutex locks
    f. Counting semaphore
        i. Can range over an unrestricted domain.
        ii. Can be used to control access to a given resource consisting of a finite number of instances.
    g. To overcome the need for busy waiting, we can modify the definition of the wait () and signal () semaphore operations. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process car block itself. The block- operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the Waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.
    h. A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal () operation. The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.