Thursday, December 6, 2018

CH 29, 30, 32, 36, 37

CH 29 – Lock-based Concurrent Data Structures

- Adding locks to data structure to yield high performance, and enabling several threads to access the structure at once; **concurrently**
- **Concurrent Counters**
  - Problems with counter without locks, is performance
  - Simple but not scalable
    - The problem with the approach wherein, we lock and unlock before initializing, incrementing, decrementing, and getting the values is that with more number of threads, **the time taken to execute is very high.**
  - Scalable Counting
    - In this example, we have *local counters* and equal number of local locks. The number of local counters would be equivalent to the number of CPU. There would be one global counter, and one global lock.
    - Whenever, a counter needs to be updated, the local counter is updated by using the local lock. –> results in **scalability**.
    - A threshold value S would be chosen, that is when the value from the local counter would be updated to the global counter. **Lower** the threshold value, more it would behave as a **non scalable** counter (low performance), **higher** the value of threshold, **better would be the performance**, which could possibly even push the global counter value.
- Concurrent Linked List
  - Scalable solution is to use locks for each node rather than the whole program. If we use one thread for all the nodes, the program wouldn't scale well. Hence locks for each node. Also known as **Hand-over locking** or **Lock Coupling**. This would result in higher concurrence but is practically more prohibitive.
  - **More Concurrency isn't necessarily faster.(if design adds overhead)**
- Concurrent Queues
  - Add a dummy node to separate the computation of **head** and **tail** nodes. The goal is to allow for concurrency for enqueue and dequeue operations.
- Concurrent Hash-tables
  - Built with the concurrent list discussed above, with a lock per hash bucket (each represented by a list).
- Avoid pre-mature optimization – **Knuth's law**

CH 30 – Condition Variables

- How to make a thread check for a condition, before continuing its execution.
- Condition variables is an explicit queue, to which the threads add themselves to, while the threads are waiting for a particular condition to come true.
- Definitions and routines
  - The CV has two operations associated with it – wait() and signal(). **Wait**() – thread wishes to put itself to sleep. Takes mutex as parameter and assumes mutex is locked. The responsibility of wait is to release the lock and put the calling thread to sleep (atomically) **Signal**() – thread wants to wake up a sleeping thread. A state variable is necessary as that would record the value the threads are interested in knowing. (some **identification** is needed to know which thread to wakeup). Also it is important for a **thread to hold a lock** in order to signal and wait, because a thread could change the state variable leading to parent being sleeping forever.
- Bounded Buffer problem
  - One or more producer threads, and one or more consumer threads
  - Producer produce data – put them in buffer. Consumer grab the item from the buffer and consume them
  - A problem with this approach using if conditions is that when we have multiple consumers and one producer.
    - What would happen when a producer produces a value, puts it in the buffer for the consumer, but the consumer 1, after initializing sleeps, and another consumer 2 sneaks in.
    - When returning from the producer, the consumer 2 would be scheduled – which btw consumes the value produced by the producer.
    - However, we need to still keep in mind, that there is still a consumer which is sleeping. When it wakes up to consume to previous value, the buffer would be empty. *oops…*
  - A simple rule to remember is to **always use while loops** with condition variables.
  - Even if the while loop is being used, there would be another problem in a similar case as above.
    - Lets say, two consumer start out, and both are set to sleep one after the other.
    - A producer then come in to produce and stores value in the buffer for the consumer to consume. And before it can create another value to store in the buffer, it sleeps because – well the buffer is full.
    - When the consumer 1 tries to consume to value it successfully does so. However once completed; there would be two threads that it can schedule.
      - BUT now is the problem… which one to schedule. Ideally the producer should be scheduled, but what if the consumer is scheduled. There would be no value in the buffer to consumer by the consumer. *Oops…*
      - **So an important point is to have signaling – which is more directed**.
        - A consumer should not wake up other consumers… only producers, and vice versa.
      - Solution is to use two condition variables instead of 1.
        - This would ensure proper signaling to the appropriate type of thread that needs to be scheduled.
  - The next step would be increase the buffer size, and to make this happen we would have the producer to not sleep until all buffers are currently filled, moreover a consumer would sleep if all buffers are currently empty.
  - **Covering Conditions** – In this there is a broadcast signal function that would wake up all the threads that are waiting up on a particular resource. Because, there might arise a case, wherein the normal signaling does not wake up a thread that can be scheduled immediately but wakes ups thread that cannot be scheduled at all. Take for example if a $thread_a$ needs 100 bytes whereas $thread_b$ needs only 10 bytes, and a $free()$ call, frees up 50 bytes, but the thread that was scheduled to wakeup by the signal call was $thread_a$ and not $thread_b$. In this case, a broadcast signal would wake up all the threads, allocating the resource to the required thread.

CH 32 – Common Concurrency Problems

- Two types of bugs – Non-Deadlock and Deadlock. Non-Deadlock bugs have two major types – Atomicity Violation and Order Violation.
- **Atomicity Violation** – In this type of bug, by definition, the desired serializability among multiple memory accesses is violated, that is, a code region is intended to be atomic, but **atomicity is not enforced**. This problem, or bug, can be solved by putting *Locks* around such code regions.
- **Order Violation** – In this type of bug, there is an inherent order in which the code needs to be executed, but **no order is enforced**. This bug can be easily solved by using *Condition Variables* that would enforce the execution of a particular code region, thereby establishing order of execution. Note, that the CV is to be placed within locks.
- Deadlock occurs when two threads are waiting for locks that are currently held by the other thread and are waiting for it to be released. Few prominent reasons for this happening are that, large code bases, often result in complex dependencies which may lead to a deadlock. Another reason, is due to encapsulation, that is, the implementation details are hidden which does not mesh well with locking.
- Four conditions that are needed for deadlock to occur – **Mutual Exclusion**: Threads claim exclusive control of resources they require. **Hold-and-Wait**: In this the threads hold on to resources and are also waiting for additional resources. **No Preemption**: In this, the resources cannot be forcibly removed from threads that are holding them. **Circular Wait**: In this there exists a circular chain of threads such that each thread holds one or more resource that is currently required by some other thread in chain.
- Prevention –
  - **Circular Wait**: To avoid circular wait, enforce order in acquisition of locks. Two ways: **Total ordering** in which all the locks in the program have a predefined order in which they are acquired by threads, However, that is not always possible in larger system, as an alternative one can adopt **partial ordering** in which $type_x$ of locks need to be always acquired before $type_y$ of locks. Another way to prevent deadlock to occur is to have a **numbering convention** to lock and unlock.
  - **Hold-and Wait**: This can be prevented holding all the locks at once atomically. To allow for this to happen, we can put one more lock in front of all the locks that need to be acquired by a thread. However, this leads to problems in case with encapsulation which would need us to know *exactly* which locks need to be acquired beforehand. Also, this would decrease concurrency as all locks must be acquired early on instead of when required.
  - **No preemption**: In this, the solution could be to add a try_lock which would simply check if the lock needed by a thread is currently held or not, if not, it would return a success code which would then make the lock being acquired by the calling thread, if not it would simply try later to acquire the lock. However, this might lead to another condition called the $live\ lock$ in which a thread repeatedly fails while trying to acquire a lock, however, this can be resolved by using a timer, that would limit the number of attempts by a thread, thereby decreasing the odds of repeated interference among competing threads.
  - **Mutual Exclusion**: One way to avoid mutual exclusion is to make use of lock-free approaches as much as possible. For example, we can use the compare and swap instruction for incrementing counters or for list insertion. These powerful **hardware methods**, can *help avoid the need for explicit locking.*
- Deadlock avoidance is also a solution that we can look at. One way to avoid a deadlock is through scheduling. If we know that a process would need to acquire a certain number of locks, we can schedule in such a way that a deadlock never happens. But such a solution has very limited use-cases.
- Finally, another way is to allow the deadlock to happen. Have deadlock detection in-place, if a deadlock is detected, then the system can be restarted.

CH 36 – I/O devices

- The main takeaways are that, faster the connection needs to be, shorter connection is required.
- A device in general usually has two parts to it, the **interface** and the **internal structure**. The interface usually consists of some kind of mechanism which allows for interaction with the hardware device. The internal structure are responsible for the lower level mechanism that the particular hardware is designed to perform. In such a device, there is usually a protocol that is defined, in which both the interface and the internal consists of some form of hardware for enabling the tasks that they are supposed to perform. For example, the interface might consist of registers such as **Status**, **Command**, and **Data**. In which the status register can be read to see the current status of the device the command register can tell the device to perform a certain task, and the data register can be used to pass/get data to/from the device.
- The protocol has ***four*** steps. In the **first**, the OS waits until the device is ready to receive a command by repeatedly reading the status register, that is, **polling** . **Second**, the OS sends some data down to the data register; When the main CPU is involved with the data movement. **Third**, the OS writes a command to the command register; doing so implicitly lets the device know that both the data is present and that it should begin working on the command. **Finally**, the OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished. It can be noticed that polling might not be a great design decision, because of the overheads associated.
- A better way to go about this is to make use of **hardware interrupts**, that is, when the device completes the execution of the task it was given, it can simply raise an interrupt which would control the flow back to the OS, making it handle the interrupt event. This allow for overlap of computation, meaning that when one particular process is executing a I/O request, the OS can simply switch to another process, thereby improving CPU utilization. Note that interrupts are not always good. Because, if the task being performed by the first program is quick, interrupt would add additional overhead as compared polling. **Hence,** if the device is fast, it may be best to poll; whereas if the device is slow, interrupts are better.
- Another technique is coalescing in which, the device needing to raise an interrupt can wait a bit before delivering it thereby accumulating other interrupts which could then be processed together by the OS.
- Finally, another important point that needs to be considered is, involving OS with task such as reading and writing to and from a hardware device puts unnecessary overheads, onto OS, which otherwise can be used to run other processes. To solve this, we can have another hardware system in place that would perform this task in place of the OS called the **Direct Memory Access** (DMA). The OS can simply pass the location of the memory to be accessed with the amount/chunk of data that needs to be copied/written to the DMA and then use context switch to switch to another process while the DMA takes over in the background to handle the I/O task. This improves the CPU utilization, and once the DMA is done with the task, it can simply raise an interrupt to inform the OS about it, and then leave it to the OS to either schedule the process that initiated the I/O request or continue with the process that it is currently executing before scheduling the I/O initiating process.

CH 37 –