CH 26, 27, 28

CH 26 – Concurrency: An Introduction

- **Threads** allow multiple point of execution of a program. Thread is the same as a separate process, except for the fact that it would share the *same memory address*. The thread would also have a program counter and a private set of register it uses for computation. With processes we have a process control block to save the state of the process, with threads we have the **Thread Control Block (TCB)**. Finally each thread would have its ***own stack***, meaning that if there are two threads for a program, each of them would have their own stack.
- There are two basic uses of threads, **First** – Parallelism. This allows us to speed up programs and their execution. **Second** – To avoid blocking of a process during I/O. In such a case, one of the threads could be waiting for the I/O to complete, whereas other threads can continue execution and utilization of CPU in some manner.
- Thread creation, is the process in which the main thread currently executing, creates more threads, and then passes over the execution to these threads. Once the threads are created, they are scheduled by the OS using some algorithm. Like, processes, these threads can be scheduled in any order, and hence the final result would be **indeterministic**. Additionally, with threads, there arises one more **challenge** – **shared data**. When separate threads, interact with shared data, there is a possibility of the rise of a **race condition**, that is, there are multiple threads that have entered the critical section at roughly the same time; and they attempt to update the shared data structure. Simply take the counter example. In this example, since multiple threads are accessing the same shared data (*counter*) this code is called the **critical section**. Critical section could also be a code which accesses the same resource. What we really need is, a property called **mutual exclusion** that would ensure that if one thread is executing in the CS, no other thread would.

CH 27 – Interlude: Thread API

- **Thread creation** – The function used is $pthread\_create$ and it takes 4 parameters. The first parameter is the pointer to a structure of type $pthread\_t$ which is used to interact with the thread. The next parameter is $attr$ which is used to specify any attributes that this thread might have like stack size or scheduling priority, which would have to be initialized using $pthread\_attr\_init()$. The next parameter is the function that this thread would start running in; also called the **function pointer**. Finally we have the arg parameter, which simply takes the arguments of the function that the thread would be running in
- **Waiting for a thread** – The function used is $pthread\_join$. This would wait for a thread to complete. Takes two arguments, $pthread\_t$ and the second argument is a pointer to the return value you expect to get back. However, there are three points to note. **First**: We do not always have to do all the packing and unpacking of arguments, that is, we can pass NULL, if thread takes no arguments. **Second**: If we are just passing a single value, we do not have to package it up as an argument. **Third**: *Never* return a pointer which refers to something allocated on the thread's call stack.
- **Initialize the locks and condition variables**. Failure to do so will lead to code that sometimes works and sometimes works in very strange ways.
- **Each thread has its own Stack**, hence all the locally allocated variables are private to the thread and no other thread can easily access it. To share the data, values must be present on the heap, or otherwise globally accessible.

CH 28 – Locks

Critical Section -> CS

- Threads
  - Threads share the code and heap segments, but each thread has its own stack. Process is the resource allocation unit, and thread is the scheduling unit
- Lock
  - -> Variable -> holds the state of lock – available or acquired (by one lock)
  - Store Other information - which thread holds lock, queue for ordering lock acquisition, hidden from the user of lock
  - Lock() and Unlock()
    - Calling a lock, enables a thread (now the owner) to acquire the lock if available thus entering CS. If another thread calls lock() it will not return, preventing threads to enter the CS.
    - Once, the owner calls unlock(), the lock is released. If no thread is waiting to acquire the lock, it would simply be turned to free; else the threads **eventually** notice or are **informed about** this change of lock's state, and enter the CS.
    - Locks provide minimal control over scheduling to programmers. Threads are entities created by the programmer but scheduled by the OS.
- Pthreads
  - Pthread – Declare a var and you should have it init to call the function
  - If diff lock var, then these are not mutually exclusive, coz they have to be on the same critical section. Benefits – Parallelism
- Building a Lock
  - H/W Requirement + OS support
- Evaluating Locks
  - Metrics -> is **mutually exclusive**? // **Fairness** // **performance**
- Controlling interrupts
  - (before the critical section disable and then enable after the critical section)
    - This would be just for that thread for the process requesting it
    - Problem –
      - **Trust** to not abuse the functionality. Malicious program to **monopolize** use the resource. The program is buggy, if the process is going on in a **loop *forever*** :P. Does not work on the multicore machine. Also, this is very time consuming
- Using Loads/Stores
  - Flag variable – use a flag variable to identify whether a thread is in the CS, while entering set it to 1 (Lock) and while leaving (unlock) the CS, clear the flag.
    - If another thread wants to enter the CS, it checks the flag, if SET, it simply **spin-waits**.
    - Two problems with this – correctness, performance
      - Correctness – concurrent programming first thread sets to 1, interrupted, second thread sets the flag to 1. Both now **interleaving,** and in **CS.**
      - Performance – Spin waiting is bad.
- Test and Set – atomic exchange – uses hardware
  - Get the value stored at old memory location, store the new value in the old men locatn, and return the old value
  - The key, of course, is that this sequence of operations is performed **atomically**

- Needs a preemptive scheduler – to run different thread from time to time as a thread spinning on a CPU will never relinquish it.
- Evaluation of metrics
  - Mutual Exclusion
    - Yes, this method ensures that only one thread is in the CS of the program at any point.
  - Fairness
    - This method is not fair, since it does not guarantee any fairness in which threads are being scheduled.
  - Performance
    - Single Processor
      - The method does not fair well, since when one of the threads in the CS is preempted, the other threads would simply be spinning in the while loop, until the interrupt is called in. Waste of resources
    - Multi-Processor
      - The method fairs well, since a thread running on CPU1, when preempted in the CS, would still allow other threads to be scheduled on other processor or allow the same thread to be scheduled to complete the CS.
- Compare and Swap –> refer H91 reference
  - Similar to Test&Set. Takes 3 params, old *pointer, expected value, and new value.
    - Old pointer is simply the current value of the lock->Flag, expected is 0, new is 1.
    - If the return value from the function is 1, then keep spinning, else allow the thread to acquire the lock.
    - Much powerful than T&S –> asked professor waiting for reply –> "It is more powerful simply because it has the extra checking."
- Load-Linked and Store Conditional
  - There are two separate functions load-linked [LL] and store conditional [SC]. LL simply returns the value stored in the flag. Whether it is 0 or 1. Now Store conditional is where the actually acquiring of the lock happens. First check if the value in the load linked has been updated so far, if so fail the acquisition of lock as two threads are competing for the lock, and try again. If not, successful lock acquisition by the thread. The failure to acquire lock might happen when two threads have executed LL and checked that the lock isn't acquired, but while executing SC it fails, leading them to try again, **until [while(1)** would encapsulate the LL and SC function**]** only 1 of them has acquired the lock.
-
- Fetch and Add
  - Basic operations involved are using a ticket and turn variable to keep track of the currently running thread, and the next in line thread to be executed. It does so by 'fetching' the currently passed variable (turn/ticket) and incrementing it by 1.
  - Now in the actual code of the locking function (after initializing the values of ticket and turn to 0), a thread gets it 'ticket' (myturn) for execution, [set to 1, for the first thread]. Now it checks if the myturn != turn, meaning that whether the current thread should be scheduled or not. If not, keep spinning while it is the threads turn, else, allow the thread to acquire the lock.
  - Now during the unlock, when the thread releases the lock, it increments the 'turn' variable by 1, allowing the next thread in queue to be considered for acquiring the lock.
  - It ensures progress for all threads. Thus no starvation. Possibly a fair method?

- Too much spinning.
  - Avoid Spinning – need H/w Plus OS support
    - Simple Solution
      - Just yield, baby :P – Give up the CPU instead of spinning
    - Sleeping instead of spinning
      - **What will happened if the park and m->guard = 0 is interchanged?**
        - The guard will never be set to 0
        - When a thread is woken up from park(); it does not hold the guard. Thus the lock is passed directly from the thread releasing the lock to the next thread acquiring it.
-
-
  -