2022 Collaborative Solution

Q1

**a.i.** Synchronization primitives are required when there is a critical section (a section of code in which there is a resource being shared) there are multiple threads that could potentially try to run in this critical section at the same time. We use synchronization primitives to ensure the threads do not both run in the critical section at the same time.

Examples:

A lock is a synchronization primitive using an atomic test and set function to allow only one resource to acquire it at any one time.

A semaphore is implemented on top of a lock. It has a counter that threads attempt to down. Only one thread can down the semaphore at any time, and it must up the semaphore when done.

A monitor allows for synchronization between multiple threads, but it also keeps a few conditions that threads can use to communicate with each other about when certain conditions have been met.

**a.ii.** The OS kernel would need to associate an integer value for threads that attempt to acquire it to set it to 1 and 0 to acquire and free the lock respectively.

**b.i.** T1 tries to acquire lock m and acquires it successfully on line 7.

T1 then performs an expensive computation on line 8.

T1 then sets the value at the current position of the buffer to a on line 9.

T1 moves the value of the current position in the buffer 1 position ahead, looping back to the start if the position has reached the end on line 10.

T1 then releases the lock and leaves the function on lines 11 and 12.

T1 then could likely call a thread yield system call that would be voluntarily releasing the CPU to let another thread run.

At this point the OS scheduler would run and could choose T2 as the next thread to run.

T2 then starts running and executes lines 6 to 12 similarly to how T1 did, successfully acquiring the lock given it has been freed by T1 then releasing it at the end of the function.

**b.ii.** If the expensive computation is not a critical section, then it should be moved outside of range of the lock so that the lock is acquired for less time and is made more available to other processes.

**b.iii.**

This approach would not work if the OS only supported green threading as green threads cannot actually use multiple cores and just provides the illusion of parallelism. This means that the CPU processing time is shared back and forth between the two threads and it would still take the same amount of time overall to perform the expensive computation.

**If this is referring to the original code:** the kernel is unaware of user level threads. Therefore if a user level thread blocks using a kernel level lock, the whole process will be blocked by the kernel. Since there are no other processes trying to unblock the kernel lock, the whole process will stay blocked until the user or kernel terminates it. **iv:** we should use a user level lock if we are locking user level threads. This means the process will handle synchronisation internally without calling the kernel.

**b.iv.**

A solution to this problem could be to fork before executing the expensive computation function which would create a new process, which would perform the expensive computation for us while we continue execution concurrently

Use kernel level threads

Q2

**a.** EAT = 0.5ns x 0.9 + (0.5ns + 15ns) x 0.1 = 2ns

//// Shouldn’t this be EAT = 0.9(0.5ns + 15ns) + 0.1(0.5ns + (2\*15ns)) = 17 nsec as it still requires a memory access to fetch the data from memory if cache is a hit (and therefore two for a miss)? I think this would only be if it was a TLB cache, which I don’t think is what the question is implying? Indeed, the whole point of the next question is that we don’t take virtual memory into account here.

It’s a larger font so it must be right

**b.** EAT1 = 2.04 ns

EAT2 = 1.99 ns

Assuming the EAT improvement was because of a larger translation lookaside buffer (TLB). We can make some software changes to exploit the larger TLB. When a TLB miss occurs, we should throw a TLB fault and TLB fault handling should be done by the OS. It must find the page, remove a TLB entry, insert a new entry and redo the instruction that faulted. Since TLB misses are more common than TLB hits, this replacement should be done in minimal instructions. This is efficient for larger TLBs. Since the replacement is now the OS’s job, the MMU is much simpler giving more CPU space for caches and other performance improvements.

We can also reduce TLB misses (and reduce the cost of replacement) by the OS guessing which page to be used next and preloading those pages. To best take advantage from a programming point of view, we must write code (especially loops) which maximize temporal and spatial locality.

As the OS kernel is responsible for creating, and destroying, the page table directory structures for a process: the part of the kernel responsible for managing the page table (for virtual memory management for processes) would have to change the pre-existing structure of the pagetable to

**c.** Since the application and its code is already in main memory, we do not need to perform paging (since it requires computation for every single access, increasing associative lookup time). We can instead make use of segmentation since the entire segment is in memory. Segmentation also allows us to differentiate between secure procedures and data. The memory references are simpler, and data is shared among processes (beneficial as only 1 application is running). This should give us the lowest possible EAT.

I think the idea is that memory translation is a simple shift by constant (or perhaps identity in which case the constant will be 0) like that of a segment access.

**d.**

ext2 inode layout is 12 direct, 1 indirect, 1 doubly indirect, 1 triply indirect

each direct pointer will point to an (8192B =) 8KiB block, so 12 will store 96KiB

the indirect pointer will store (8192/8)\*8KiB = 8192KiB = 8MiB

the doubly indirect pointer will store (8192/8)\*8MiB = 8GiB

the triply indirect pointer will store (8192/8)\*8GiB = 8TiB

The worst case scenario of reading something from a 100GiB file will occur when accessing the last ~92GiB and going through the triply indirect pointer. This requires 5 access (inode, one for each of the three pointers and one for the respective block).

**e.i. what**

**E I)** *anyone else think that It shld just be mapped sequentially so the access is continuous. One page after another. Ahead of time paging as we know the order*

**II)** *hybrid approach – ahead of time few random pages hopefully get lucky; on demand otherwise.*

**III)** *demand paging – let it page fault once to identify subset. Then mmap pages ard that page (subset) to memory ahead of future accesses.*

**e.ii. the**

**e.iii. hell**