1a)

i) Synchronisation primitives are a way of controlling access to a shared resource between multiple users. For example, a critical section of memory is a shared resource and only one process can have access to it at any given time. They can also be used to prevent multiple people from accessing files at the same time.

1. Lock/mutex

2. Semaphore

3. Condition variable

ii) Current holder of the lock, current waiters on the lock

b)

THread1: 1-5 are executed first at program startup then quantum up so yield

Thread 2: 1-5 executed then yield

Thread 1: 6-8 then quantum up so yield (execution long)

Thread 2: 6-7, then realises that lock is currently held so then sends systemcall to block current thread and added to mutex waiter list

Thread 1: 8 then yield, on repeat for however many quantums until expensive operation finishes

Thread 1: 10-11 systemcall to unblock thread 2 is issued and then thread 1 voluntarily yields

Thread 2: 8-11 then thread 2 yields

Thread 1: returns (and syscall for exit issued)

Thread 2: returns (and syscall for exit issued)

ii)

Assuming that perform\_expensive\_operation doesn’t use the buffer (we can’t be sure) then we can swap lines 7, 8 therefore we will not be in critical section for longer than we need to be increasing parallelisation.

iii)

We are using pthread which is a POSIX compliant library, and pthread\_mutex\_lock will invoke systemcall to sleep current thread when failing to acquire a lock. Therefore,if thread 1 and 2 are implemented using green threads then when the second thread tries to acquire the lock the whole process will be blocked.

iv) Implement own synchronisation primitive in our green threading library, to allow for the green threads to block so kernel doesn’t block whole process.

2)

0.5 + 0.1\*15 = 2nsec

b)

We need to change the OS kernel module responsible for managing the page table, which will be involved with processes and the virtual memory management of individual processes. This is because if the structure has changed then to create a page table that the processor understands we will need to change the structure the kernel uses, additionally the code responsible for installing and removing pages will need to be changed for similar reasons. If we didn’t the MMU wouldn’t understand our page table format.

d)

Ext2: 12 direct, 1 single indirect, 1 double indirect, 1 triple indirect

Each block: 8192 bytes so 1024 pointers

12\*8192 = 98,304 bytes = 0.00009 GB

1\*1024\*8192 = 8,388,608 bytes = 8.389 GB

So if we access any byte after the 8,486,912th byte then we will need 5 reads as we have 1 for inode then 3 for indirection and 1 for the actual data block.

c)

Linear mapping, from a base register with a (ulimit) limit register of size 1 GiB, with base set s.t. we can have base+1GiB fit in address word and the application memory stored in physical address [base..base+1GiB]. We would be avoiding the need for a translation from virtual memory to physical memory as it is a simple operation that can be performed ‘linearly’ thus we don’t need to cache entries so we never need to perform a TLB lookup or a page table lookup.

e)

OS strategy: use file system with block size equal to page size. NVM device accessible through memory mapped IO. Assume this to mean we have region in physical memory for base address of NVM and address #(NVM + j) will access the j'th byte of the non volatile memory. We can use these physical addresses as frame addresses in the virtual page table entry of the application loading the file. The strategy for loading and installing these pages is as follows:

1. (When os handles mmap call) Read file inode and get all associated memory I/O addresses for data blocks in file by multiplying block number by block size and adding to NVM device memory adress base. For a base virtual address X, for each block install page in page table for application by adding page table entry for X+block\_size\*i mapping to physical memory mapped IO address of block on NVM device. (Similar to ahead-of-time) this is because we know whole file will be accessed so we now avoid incurring any page faults.

2. Assuming that entire file will eventually be accessed, we can use same strategy as above. No page faults incurred, if we know accessing entire file then we would incur many avoidable page faults.

3. Cache inode in memory and then when file is accessed, we can identify we are accessing memory associated with mmap'd file and then lookup data block index in inode for accessed byte and install page associated with data block referring to offset of NVM device for data block. This avoids us iterating over entire inode blocks and installing pages for each data block (262,144 pages) which if we are only accessing subset eventually we will no longer page fault as we have loaded entire subset.