Answer the following questions. For questions asking for short answers, there may not necessarily be a "right" answer, although some answers may be more compelling and/or much easier to justify. But I am interested in your explanation (the "why") as much as the answer itself. Also, do not use shorthand: write your answers using complete sentences.

(Scheduler Activations) The Scheduler Activations paper states that deadlock is potentially an issue when activations perform an upcall:

> "One issue we have not yet addressed is that a user-level thread could be executing in a critical section at the instant when it is blocked or preempted…[a] possible ill effect … [is] deadlock (e.g., the preempted thread could be holding a lock on the user-level thread ready list; if so, deadlock would occur if the upcall attempted to place the preempted thread onto the ready list). (p. 102)"

Why is this not a concern with standard kernel threads, i.e., why do scheduler activations have to worry about this deadlock issue, but standard kernel threads implementations do not have to? While the standard kernel thread table is managed entired by the kernel, the user-space thread is not. The kernel does not have any idea whether a thread in user-space is within its critical section or not, and premeption may happen at that time. Even though the scheduler activation provide such channel exchanging such information (vessel) between user and kernel, the table itself is still in the user-space, and the kernel is in charge of the scheduling and has no idea about locks. When kernel premepts the thread, that thread may have locks on the table, which leads to the deadlock. The standard one, however, kernel knows which thread has the lock.

(LFS) The LFS paper is strongly motivated by perceived trends in both hardware performance and user access patterns. Thus, its improved performance is driven by a set of underlying assumptions.

1. Explain what are the assumptions about hardware and access patterns that LFS depends on?

- Storage devices has larger and larger bandwidth, while the latency does not decrease that much: it means sequential read/write is faster than random one
- Small file access is more frequently happened
- RAM size will be larger and larger from time to time, which makes larger cache size possible, and trap most of the read request
- SSD will be used in large scale. However, it does not like frequent erase actions

2. Explain which mechanisms in LFS depend on each assumption?

- write strategy: LFS write files sequencially. It utilize the characterics of squential write is better than random one, which mostly reduced the need of seeking location, max out the bandwidth and avoid latency. It also reduce the hot spot where some file may be frequently modified, average the danamge of memory (especially SSD)
- read strategy: since right now files may be fragmented, which reduce the performance of read. However, with large read cache, it is expected such difference will not be observed
- clean up strategy: to avoid fragmentation, when LFS does the clean up, it will allocate a larger potions, and tried to utilize the bandwidth as much as possible

For the purposes of the following questions, NVRAM is persistent storage (i.e., will survive the loss of power) and has access times and throughput similar to DRAM.

1. At the time the LFS paper was written, non-volatile RAM (NVRAM) was not commonly available. If disks were replaced with NVRAM entirely, would the LFS design still make sense? Explain why or why not and be specific in justifying your answer.

Not really, as the NVRAM does not have seek time, which means its sequencial read latency will be as fast as random one. LFS at the same time would be possible to have too many fragments and required constant clean up. Even though LFS could average the erase and increase the life span of NVRAM, clean up also required extra actions, so it does not have huge advantages compared with the traditional ones.

2. In real-life the cost per byte of disk is likely to be far cheaper than NVRAM for some time. So instead, consider a situation where some NVRAM is available (e.g., 1/10th of the disk size). This NVRAM might be used for caching reads, caching writes, or storing particular meta-data. Argue which use might be most appropriate for improving the performance of LFS.

- Metadata: LFS's metadata is across the whole drive, this is good for recovery and implementation perspective but incur performance penalty if a file is being constantly modified and need to be tracked. Cache here could help performance locate all block needs for such files. Also, for checkpoint inode, since it requires constant update and it is at the fix location, storing in NVRAM may help reduce the latency of writing back into the hard drive
- Cache read: since files may be fragmented, caching as many read request as possible will also help improve the performance