(Q1: Kernels) Proposals suggest that future data centers will do away with monolithic servers and instead employ *disaggregated resources* to improve elasticity, heterogeneity, and reliability. In these designs, core hardware components of a monolithic server, including processors, memory, and storage devices, will be disaggregated into separate network-attached hardware components. Minimal compute resources at each device will allow components to reliably pass messages across the network, but the components themselves will remain non-coherent (i.e., there will not be implicit message-passing). Identify two technical challenges, and a potential solution to each, that arise in building a POSIX-compliant operating system for such a hardware platform.

Solution:

Since the resources such as processors, memory, and storage devices are disaggregated, a monolithic POSIX operating system on each resource would be an unnecessary overhead. A potential solution to it will be to employ the exokernel kernel and run the relevant application OS customized for the given resource. Exokernel is also POSIX compliant. However, the IPC calls can be expensive in this case and exokernel can still be heavy for a single piece of hardware.

Another alternative design is proposed by the LegoOS to separate the OS functionality into different pieces called monitors and connect it with a high-speed network stack with non-coherent components. Fine-grain failure handling is also done here.

The POSIX standard was created during the time when all the resources were available locally. However, in a disaggregated mechanism, this can be an issue. The strong consistency employed by POSIX makes cache expensive and infeasible in distributed resources. One way around is to allow some lenient consistency model such as in NFS and venus cache used in CODA file system.

References:

[1]https://www.usenix.org/system/files/osdi18-shan.pdf
[2]https://dl.acm.org/doi/10.1145/121133.121166
[3]https://dl.acm.org/doi/10.1145/224056.224076
[4]https://www.quobyte.com/storage-explained/posix-filesystem

(Q2: File-Systems) Intel Optane SSDs impose different performance characteristics than prior devices (https://research.cs.wisc.edu/adsl/Publications/hotstorage-contract19.pdf). How would you design a file system tailored to optane drives?

Solution:

LFS maintains a buffer that contains all the updates and stores it in a memory segment. Once the segment gets full, it performs a long write operation in a sequential manner to the free part of the disk, hence avoiding multiple writes.

Log-structured file systems work well with SSDs, which need to spread writes across all blocks to implement wear leveling. Optane has its own proprietary wear-leveling algorithm because the traditional file system doesn't take care of wear leveling. By using LFS might actually make the proprietary wear-leveling redundant.

LFS has a block size of 4kb instead of the traditional 512 bytes which is in sync with the Optane "4kb" block size philosophy.

One of the issues with the Log-based file system in the traditional memory system was the very high overhead of the garbage collection. This was a major issue with LFS and had a tremendous impact on the performance of the system. However, since Optane has the "Forget Garbage Collection" contract, LFS is perfectly suitable for Intel Optane SSD.

Another issue in a traditional memory system and LFS was that writes can get too random because of dead space when the disk is almost full. But this won't impact the performance in Optane because sequential and random access has the same performance due to the "Random access is OK" contract.

References:
[1]https://elfi-y.medium.com/operating-systems-101-persistence-log-structured-file-system-and-flash-based-ssd-vi-e9620d79668b
[2]https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/delivering-new-levels-of-endurance-article-brief.html#:~:text=Intel%C2%AE%20Optane%E2%84%A2%20DC%20SSDs%20implement%20a%20proprietary%20wear,fraction%20of%20the%20SSD%20prematurely.
[3]https://www.eecs.harvard.edu/~cs161/notes/lfs.pdf

(Q3: Scheduling) Languages are increasingly adding concurrency primitives that diverge from a typical threading model, such as coroutines, futures, and promises. How would you implement these features on a kernel threading operating system? Would scheduler activations provide a better interface? Why or why not?

Solution:

Scalability in concurrency primitives can be an issue for the operating system if it spawns multiple kernel threads. Since kernel threads are heavyweight, they typically perform worst than user-level threads. Hence implementing the concurrency primitives on top of scheduler activation can help incorporate a parallelism mechanism that includes the functionality of kernel-level threads with the performance and flexibility of user-level threads.
When an asynchronous request is made, the processor is returned to the application and after its completion, the application is informed. Since the kernel has no knowledge of the data structure at the user level, the kernel can support any concurrency model at the user level.

References:
[1]https://dl.acm.org/doi/10.1145/121132.121151