**CSE 221 – Graduate Operating Systems**

**Homework 3**

Name: Andreas Prodromou

SID: A53049230

**Question 1:**

**1.a)**

I think this decision's goal is to simplify and optimize certain aspects of the system:

* Given that the OS does not have a separate address context, a context switch can occur even while a process is executing within the executive.
* Several processes can execute simultaneously within the OS – relaxes the limit of how many processes can execute simultaneously.
* When the address space of a process is removed, its kernel space is also removed.

**1.b)**

The protection bits included in the page table add safety to this scheme. The hardware checks the protection bits on every access, even when the valid bit is set to zero, thus an illegal access to a page cannot cause the page to be loaded. This ensures that executive code and data are not made accessible to programs executing at the least privileged (user) mode.

**1.c)**

I believe there are various reasons why this changed in today's operating systems.

* Essentially, today's systems are very different than what the paper's authors had in mind at the time.
* The authors of this paper claim that it is easier to add memory than to generate more processor cycles and that was the reasoning behind some of their decisions. Even though the argument might be true even today, processors have become fast enough to allow us to ignore the introduced cycle penalty.
* By having a dedicated OS address space saves memory compared to this paper's approach.
* Dedicated address space for the OS adds a level of isolation, making it even more robust and reliable.

**Question 2:**

The OS is in control of all the context switches that take place in the system and as such it can handle corner cases like this in a centralized fashion, while on the other hand a process that spawns user-level threads might be forced to context switch before a lock is released.

In the case where a kernel thread holds a lock and needs to be preempted, the OS knows that that thread needs to be added to the free list. There is no risk of not adding it and creating a deadlock.

**Question 3:**

Transfer rates in the old file system deteriorate over time because of the way it was maintaining its “free list”. Even though the list was initially ordered for optimal access, it was quickly becoming scrambled as files were created and removed. Eventually, the list would be completely random, causing long disk seeks much more often. The new file system optimizes the storage utilization and this is one problem the authors avoid in their implementation.

**Question 4:**

**4.1)**

The three disk operations are:

1. Write the inode in a free block.
2. Remove the block from the free list.
3. Write the directory entry for this inode

The reason for these operations is that if a crash occurs at any given time, the file system is still in consistent state. Step 1 does not overwrite an existing block, so even if it fails, it doesn't break the FS. If Step 2 fails, the free list is not in a wrong state. If step three fails, the directory is still in a correct state.

**4.2)**

In FFS, inode creation and directory linking (i.e. steps 1 and 3) will be done synchronously. Applications will wait until the changes are permanently stored on the disk. The block write is delayed in the buffer cache.

Writing the inode always has to precede directory linking, to ensure that the directory does not link to unallocated inode. The reason for this ordering is to ensure that the metadata is always consistent, event if a crash can destroy buffered data.

**4.3)**

Soft updates does not include Reliability-Induced Synchronous Writes. In that case, the metadata is stored in the write-back cache. Writes take place when a rollback is issued, with the exception of those updates that were declared useless by the system.

**4.4)**

LFS does not generate any RISW either: every modification to the file system is written

asynchronously at the end of the log, allowing aggregated writes.

It is in fact similar to Soft Updates in the fact that only at specific times (checkpoint time in LFS),

the OS will write all possible information to disk and will update the oldest of the two checkpoint

regions. This will be done synchronously regards to the OS, but from an application point of view, it

is definitely asynchronous.

**4.5)**

Since Rio is a file system in memory, all writes are immediate and permanent, therefore every write

is synchronous and there is no need for reliability-induced writes (since the system memory is

supposed to be reliable).

**Question 5:**

LFS aggregates writes before committing the to the disk. As a result the write time is much larger than the seek time. What Eager writes offer in my opinion, is reduced seek time. However, since seek time is not the bottleneck of LFS's performance, I expect that eager writes would make little difference.