Section 2: Week 4: Network and Node File Systems

Nate Bachmeier

TIM-8120: Distributed Systems

October 13th, 2019

North Central University

# Network and Node File Systems

Different optimization objectives exist between remote and local file systems. These differences must account for the scenario’s specific needs and purposes. For instance, an embedded device might choose FAT32, because it doesn’t need the multi-user security overhead of NTFS. Or, Linux’s EXT4 is sufficient for a branch office file server but might be more challenging than Hadoop’s HDFS to manage large data sets.

# Compare Common File Systems

Table 1 enumerates the strengths and weaknesses of a collection of commonly used file systems. This table is by no means an exhaustive list of popular file systems.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| File System | Technical Features | Strengths | Limitations | Use Cases |
| FAT16 | Stores allocations in a single table | Easy to implement | Max size is 4GB | Legacy Systems and embedded devices |
| FAT32 | Increased the max sizes of FAT16. | Highly portable and supported across devices | Limited file system level security | Modern single user systems (e.g., smart devices) |
| NTFS | Added Access Control Lists, Compression, and Encryption | Defact-o standard across Windows products | Max size is 9.2EB Limited support outside on Unix | Any modern Windows client/server environment |
| NFS (Unix) | Remote mounted and treated as-if local storage | Lightweight protocol | Network latency can impact performance | Brach office file servers, small computer labs |
| AFP (Apple, 2019) | Remote mounted storage | Transparent to User | Portability, all operations are byte ordered | Mostly Mac OS.  Some MS-DOS |
| AFS (Apple, 2019) | Expands on HFS+ and exclude space in sparse files | Defact-o standard across Apple products | Case sensitive file names | Any modern Apple product (e.g., iOS) |
| HFS+ (Siracusa, 2011) | Free space shared between partitions on a volume. Added journaling support | Analogous to FAT32 improvements | Based on a 25 year old technology | Legacy Mac OS scenarios |
| ZFS (Oracle, 2010) | Virtualizes storage into storage pools | Removes the volume manager | Distributed technology | Server storage arrays with multiple volumes |
| Apple Xsan (Apple, 2004) | Remote Clustered Storage (Storage Area Network) | A cheap and performant mechanism to bring Enterprise concepts to smaller offices | 64 parallel consumers | Small to mid-sized business shared networking scenarios on Mac OS |
| VMFS | Virtualized device storage, a clustered storage abstraction for virtual machines | Sharable across multiple VMWare versions and concurrent users | VMWare Product specific | Businesses that centralized on VMWare Product line |
| APPN (IBM, 2010) | Maintain a list of SNA resources to reduce complex MPIO | Abstracts the notion of local and remote storage | Requires pairing with APPC | IBM Mainframes |
| APPC (IBM, 2010) | A protocol over APPN for abstracting communication with an entity | Facilitates the conversation of store/load over a network or local disk | Requires pairing with APPN | IBM Mainframes |

Table 1: File Systems

# Using File Systems in Distributed Systems

Contoso Manufacturing exposes a product catalog for its employees, as presented in Figure 1. These employees will make requests from heterogeneous devices, such as iOS, Windows 10, and Linux desktops. Aspects of their local file system will “leak” into their interaction with the services. For instance, file names containing colons might require additional steps to access after downloading on NTFS (e.g., foo:bar.txt). This behavior is due to the colon denoting the alternate data stream (e.g., bar.txt in file foo) (Marlin, 2013). Another example occurs within AFS, EXT4, and other Unix style systems as the file name can be case sensitive. These subtle differences can break the portability of specific scenarios.

The client’s requests flow into the service stack that resides on Microsoft Hyper-V virtualization servers. These servers mount Remote Storage Spaces, from an IBM z/OS cluster, as a mechanism for a clear separation of duties between the storage and compute nodes. After the clustered network volumes are attached, NTFS is applied to allow the operating system of the local virtualization to manage the physical file. Within the NTFS filesystem, Virtual Hard Disk (VHDX) files are created to project volumes into the managed service nodes. This layering of virtualized file systems introduces one set of challenges, in exchange for simplification in other areas. For example, physical disks are replaceble without adding downtime on the production services. Similarly, live migration can occur across the virtual service instances, as clusters of physical hosts rebalance their work load distributions.

Inside of the virtualized compute environment are three clustered services: (1) Web Frontend on Linux, (2) an NFS service on FreeBSD, and (3) a SQL Server on Windows Server. Each node within this toplogy leverages a different file system, however they able to interop through clearly defined network service contracts. From the perspective of the network application, the layers of file systems are transparent and do not have a functional impact. There is, however, the potential for performance impact, as the number of associated components and network hops increases.

# Shared Memory Patterns

An application’s state spans both file systems and shared memory stores. It is, therefore, of equal concern to understand mechanisms for ensuring the accuracy of shared memory.

## Shared Memory Application Programming: Concepts and Strategies (2015)

Chapter five of Alessandrini’s textbook describes shared memory patterns that align closer to standard implementations. He proposes an approach that centers around an envelope object that contains a local mutex (see Figure 3). This local mutex can be acquired (recursively if needed) to protect a shared resource during write operations. He also proposes strategies to partition memory into blocks, and then scope one thread to one partition. By having many fine-grained locks, fewer threads need to serialize their requests, and performance degradation is limited (Alessandrini, 2015).

While optimistic concurrency solutions work in many scenarios, they can still be problematic as memory contention increases, such as the case with NUMA scheduling (Kim, Khan, Kim, Kasu, & Atchley, 2018) (González-Férez & Bilas, 2016). Instead, the system should be designed to align with a communication flow, and leverage message passing across FIFO queues (Alappatt, 2018). This approach removes blocking operations entirely as sources are processed into sinks as they *eventually* become available.

## Message-Passing Microcoded Synchronization for DSM Architectures (2019)

Tasoulas et al. state that one of the critical challenges of many-core programming models is to synchronize locks during updates to Distributed Shared Memory (DSM). These issues arise from legacy operating system designs that make assumptions that the number of cores on the local node will be relatively small. In some scenarios, performance can degrade up to 10x as compute units halt one another with serialization before modifying the shared state (Tasoulas, Anagnostopoulos, Papadopoulos, & Soudris, 2019).

Instead, they propose a protocol (see Figure 2) with Dual Microcoded Controllers (DMC), where one controller is a “server” and the other a “client.” When a client wants to share a low-parallelism data structure, such as a stack or heap, it sends a request into a packet switching mesh. The mesh will route the message to the intended server command queue. Their approach reduces the number of locks and other cross core synchronization. The researchers were able to validate this design by introducing a new microcode to a hardware acceleration card. Afterward, they modified (1) rewrote several core data structures to (2) leverage the microcode, and (3) proved on 22-core machines that (4) these lock free arrangements were 5-8x faster.

## Design and Cost Analysis for a Fault-Tolerant ****DSM (2016)****

Fahad et al. state there are four basic algorithms for ensuring cache consistency: central-server, the migration, the read-replication, and the full replication algorithms.

# Figures

## Product Catalog File Systems

Figure 1 describes the file systems that are involved in Contoso’s Product Catalog Service. Management and infrastructure aspects, such as authentication services and load balancers, are omitted to reduce clutter.

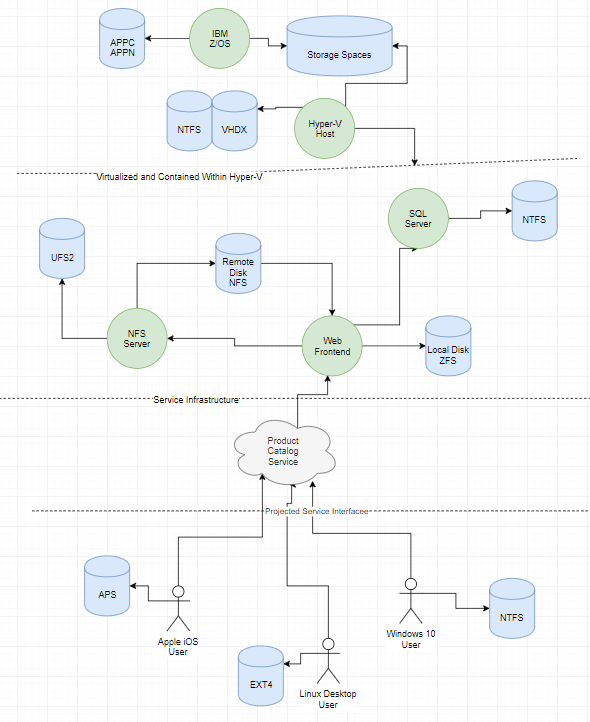


Figure 1 File Systems Used

## Message Passing Microcode

Figure 2 describes the protocol for updating shared memory in a lock free system, according to Tasoulas et al.

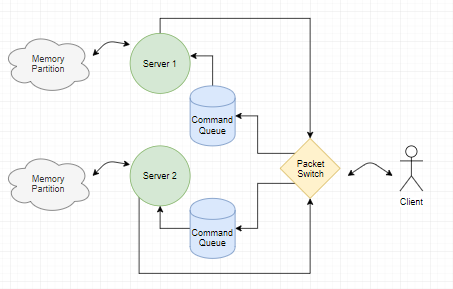


Figure 2: Tasoulas et al

## Mutex Locking Approach

Figure 3 describes the simple protocol for using a mutual exclusion lock to protect the shared resource.

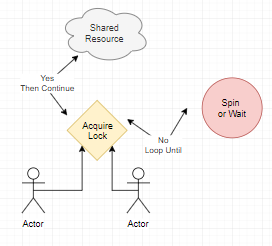


Figure 3: Mutex Approach