CONF-950985--5

ELECTRONIC FILE AVAILABLE

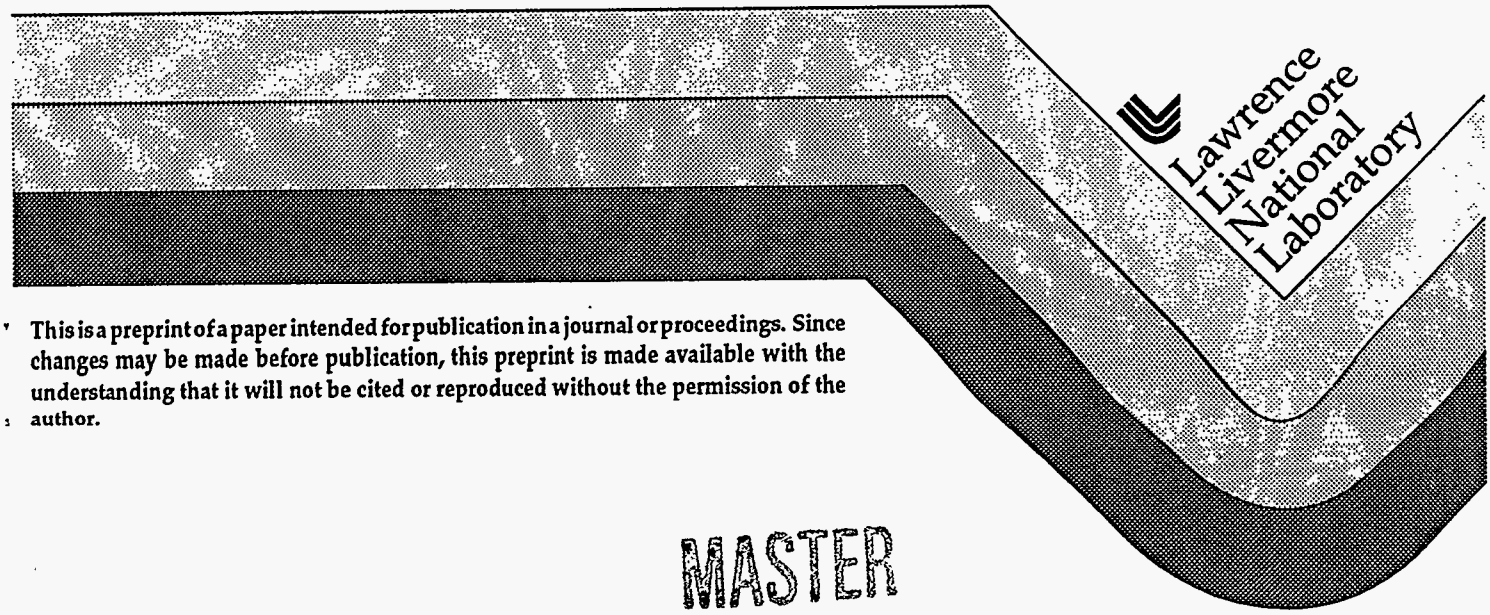
UCRL-JC-120441
PREPRINT

Physical Volume Library Deadlock Avoidance in a Striped Media Environment

J.K. Deutsch
M.R. Gary

This paper was prepared for submittal to the
14th IEEE Symposium on Mass Storage Systems
Monterey, CA
September 11-14, 1995

March 1995



DISCLAIMER

Physical Volume Library Deadlock Avoidance in a Striped Media Environment

Jeff K. Deutsch
IBM Government Systems
Houston, Texas

Mark R. Gary
Lawrence Livermore National Laboratory
Livermore, California

Abstract

Most modern high performance storage systems store data in large repositories of removable media volumes. Management of the removable volumes is performed by a software module known as a Physical Volume Library (PVL). To meet performance and scalability requirements, a PVL may be asked to mount multiple removable media volumes for use by a single client for parallel data transfer. Mounting sets of volumes creates an environment in which it is possible for multiple client requests to deadlock while attempting to gain access to storage resources.

Scenarios leading to deadlock in a PVL include multiple client requests that contend for the same cartridge(s), and client requests that vie for a limited set of drive resources. These deadlock scenarios are further complicated by the potential for out-of-order volumes to be mounted (for example, by Automatic Cartridge Loaders or human operators).

This paper begins by introducing those PVL requirements which create the possibility of deadlock. Next we examine traditional approaches to deadlock resolution and how they might be applied in a PVL. This leads to a design for a PVL which addresses deadlock scenarios. Following the design presentation is a discussion of possible design enhancements. We end with a case study of an actual implementation of the PVL design in the High Performance Storage System (HPSS).

1. Introduction

Processing power and data collection abilities have been increasing faster than storage system bandwidth and capacity for many years [1, 2]. This growing gap has caused the storage system to become a bottleneck for more and more applications. While techniques such as third party transfer and network attached peripherals [3] address this bottleneck, existing storage systems continue to fall short of meeting the present and predicted data storage and retrieval needs of supercomputers, massively parallel processors, and networks of workstations.

In response, researchers working on the next generation of storage systems are looking for innovative, open solutions which will narrow the storage gap. The IEEE Reference Model for Open Storage Systems Interconnection (Project 1244) [4] defines a storage architecture which addresses the needs of high-end storage clients. The reference model defines a set of cooperating modules and interfaces which combine to form a functional storage system. The focus of this paper deals mostly with the design aspects of one of these modules known as the Physical Volume Library (PVL).

A quick overview of the reference model's PVL module and those modules most closely associated with the PVL is necessary:

Physical Volume Library (PVL)

A PVL is responsible for mounting and dismounting physical volumes (tape, optical disk, magnetic disk) and queuing mount requests when required drives and media are in use. The PVL accomplishes any physical movement of media that might be necessary by making requests to a Physical Volume Repository.

Physical Volume Repository (PVR)

A PVR manages the agent, robotic or human, responsible for mounting and dismounting storage media [5].

The Virtual Storage Server (VSS)

The VSS maps virtual storage, including striped and mirrored data, onto physical storage media. The virtual storage server issues requests to the Physical Volume Library to mount physical volumes.

The Mover

A mover transfers data between clients and storage media [6]. The mover is used by the Physical Volume Library to read internal media labels.

At this point we need to define few other terms that we will using throughout the paper:

Cartridge

A cartridge is a transportable object managed by a PVL and physically mounted by a PVR.

Physical Volume

A physical volume (often called a volume in this paper) is the portion of a cartridge which can be contiguously accessed when mounted.

Media

Media is any readable and/or writeable data storage area.

Virtual Volume

A virtual volume is one or more physical volumes which are logically combined to represent a single data storage area.

While a physical volume mount request maps directly to a request to mount media, the cartridge containing the media may hold one or more physical volumes. For example, with some optical platters, the transportable object (the cartridge) is the optical platter itself. If the platter is capable of storing data on each of its sides, the cartridge could be considered to hold two physical volumes represented by the two sides of the platter. Tape cartridges capable of partitioned access may similarly be configured to contain multiple physical volumes.

The PVL is the enterprise wide manager of volume and drive resources and is responsible for queuing mount requests to prevent resource contention. The PVL translates client (VSS) requests to mount a physical volume into requests to mount a specific media. When a volume is requested the PVL identifies the PVR which manages the media. It then allocates the requested media and drive resources (using one of many possible queuing and allocation schemes) and issues mount commands to the PVR. When the PVR has mounted the media, the PVL verifies the internal media label via requests to a mover.

In a high performance storage environment, it is often necessary that a single bitfile be striped across multiple physical volumes in order to attain an adequate data transfer rate. For example, a modern high performance tape drive might read and write at about 10 megabytes-per-second (MB/s) and might store 20 gigabytes (GB) of data on a cartridge. If a bitfile is striped across four such tape drives, the resulting virtual volume will appear to the client to read and write at 40 MB/s and to store 80 GB. It should be noted that, while the advantages of striping are obvious, deciding when and how wide to stripe a bitfile is challenging [7, 8, 9].

High performance storage systems that implement striping require that sets of volumes be mounted together in order to satisfy striped data requests. For this paper we make the assumption that striped data can not be accessed until all volumes making up the virtual volume are mounted. This assumption is necessary in cases such as direct tape-to-tape copies (when the source stripe width does not match that of the sink) and tape-to-display copies where a system does not typically have enough memory to buffer data while awaiting mounts. Sets of volumes may also be needed for other purposes such as creating mirrored copies of data. It is in the process of satisfying atomic mounts of sets of volumes that a potential for deadlock arises in a PVL.

2. Problem statement

A system is considered to be deadlocked if every activity in the system is waiting for an event which can only be generated by another activity in the system [10]. Mounting volumes in a high performance storage environment can cause deadlock in three different ways:

- *Drive resource contention*
- *Out-of-order mounts*
- *Multiple requests for the same cartridge*

Non-deadlock scenarios involving clients that monopolize resources can also effectively prevent the allocation of storage resources. In this section we will discuss these scenarios after first investigating the potential for deadlock caused by each of the three deadlock conditions.

2.1. Drive resource contention

A PVL is required to mount multiple physical volumes together for striped bitfile access. In most parallel environments all physical volumes must be mounted concurrently to satisfy the striped request. Mounting three out of four cartridges for a striped tape request does not typically allow the data to flow, and the three drive units occupied by the mounted tapes are not free to satisfy other requests during the wait for the fourth cartridge. Worse yet is the potential for deadlock if the required fourth drive resource will never become available because another request, which will not complete until one of the drives occupied by the first request is relinquished, occupies the remaining drives.

This simple deadlock scenario is illustrated in Figure 1. In this example a PVL managing four drives has two separate requests, one for a four-wide stripe and a second for a three-wide stripe. The PVL has mounted two cartridges for each request and is deadlocked waiting for drives to free for each request.

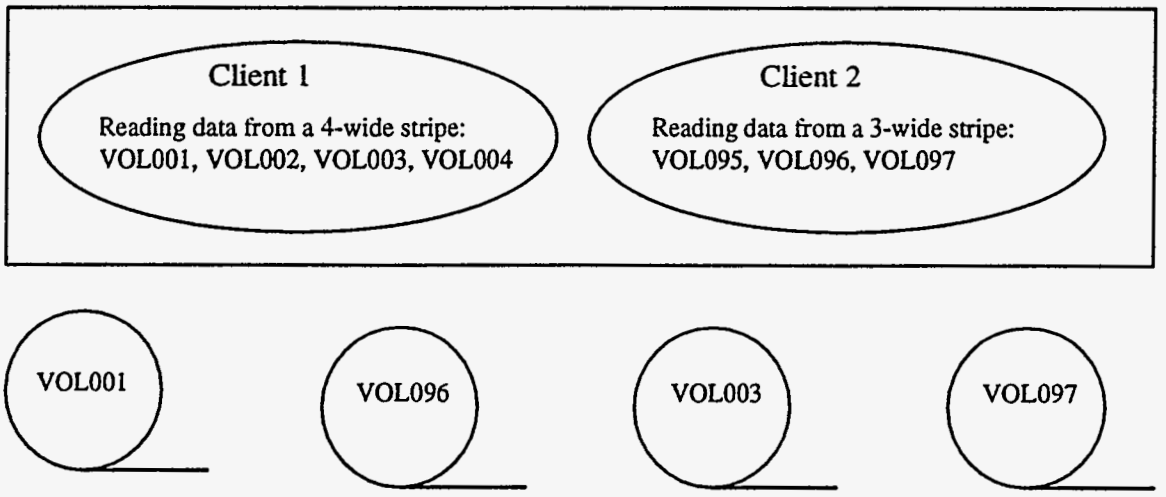


Figure 1. Two clients deadlocked waiting for drives

Another aspect of this deadlock scenario is that, even if one of the striped requests is satisfied, data may not be able to flow until both mount requests are satisfied concurrently. If the two clients in Figure 1 were attempting a tape-to-tape copy directly from one set of striped tapes to another (Client 1 to Client 2); all source and sink tapes would need to be mounted for data to flow. If the two clients involved submit their striped mount requests separately, even if one of the stripe sets is successfully mounted without deadlocking, that striped set will occupy drives without moving data until the second stripe set is mounted. If the second stripe set is unable to mount due to resource contention, PVL deadlock is achieved. In fact, the example tape-to-tape copy would be impossible given the hardware in Figure 1, as seven drives would be required for the copy. It should be noted that, if the tape-to-tape copy was between a like number of tapes (four-wide stripe to four-wide stripe) copying could be accomplished by copying each stripe independently.

2.2. Out-of-order mounts

The potential for out-of-order volumes being mounted creates another deadlock scenario in a PVL. Out-of-order volumes are those volumes which have been requested by a client (or perhaps will be requested soon by a client) but have not yet been requested of the PVR by the PVL. In addition to out-of-order volume mounts caused by operator and robot error, another common mechanism which can cause out-of-order volume mounts is the use of a traditional sequential Automatic Cartridge Loader, more commonly referred to as a stacker. This device mounts the next cartridge in its stack as soon as the current cartridge is unloaded. No external command is required from the PVL or PVR for such a mount. If the PVL keeps the out-of-order volume mounted (possibly to satisfy another queued request) the same type of deadlock condition we observed previously could occur [see Figure 2].

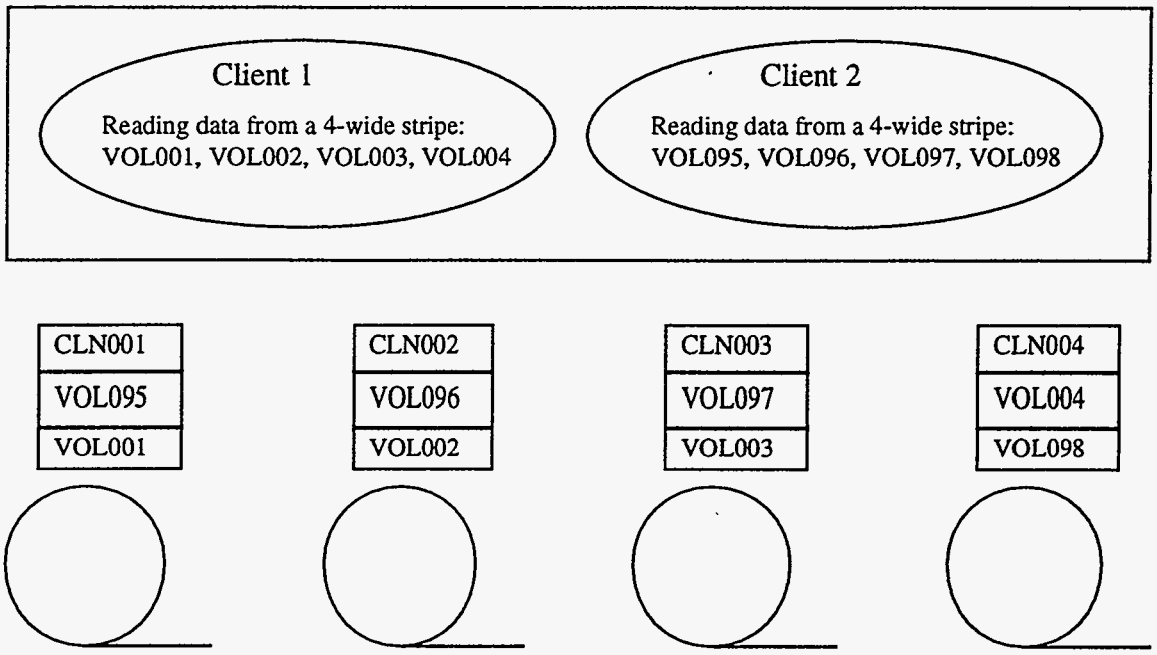


Figure 2. Two clients deadlocked due to cartridge order in stackers

Note that, in Figure 2, there is no way for both client requests to succeed without operator intervention.

Another scenario made possible by out-of-order tape mounts is one leading to a state of indefinite postponement of a client request (also called livelock [11]). If, when an out-of-order mount occurs, the PVL uses the volume to satisfy a queued request, other mount requests which may have been older (or of higher priority) will be postponed. Theoretically a mount request might never be satisfied because out-of-order mounts could indefinitely monopolize drive resources.

2.3. Multiple requests for the same cartridge

It is also possible for a PVL to deadlock based on contention for cartridge resources. If two clients are mounting striped sets that require different physical volumes, but two of the volumes exist on the same cartridge then deadlock could occur. This deadlock can occur even though clients may be requesting discrete sets of volumes, but in fact are requesting overlapping sets of cartridges [see Figure 3].

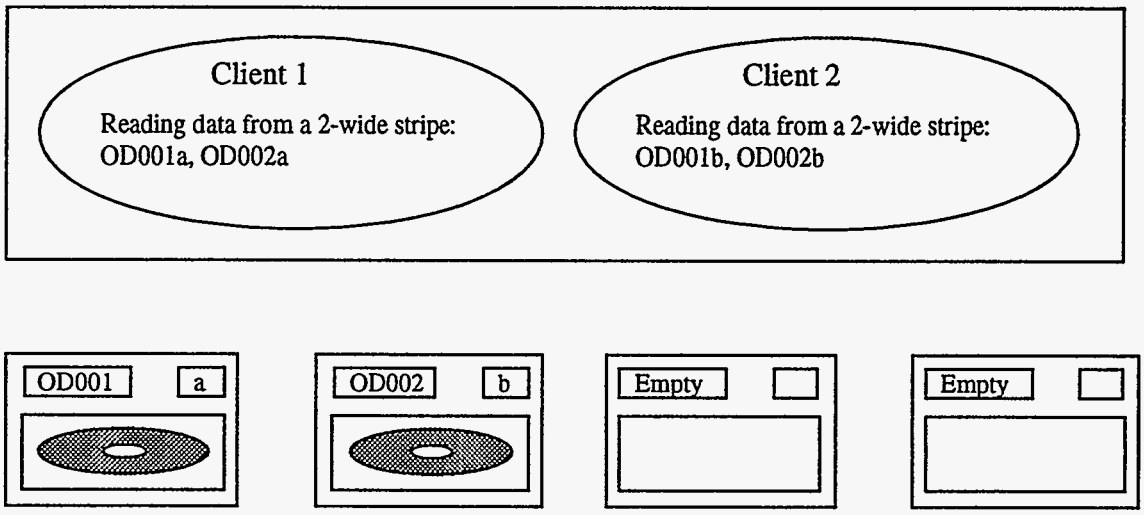


Figure 3. Two clients deadlocked waiting for volumes

2.4. Resource monopolization

In the subsection on out-of-order mounts we saw how accepting out-of-order mounts could lead to the indefinite postponement of requests even if it did not cause deadlock. The end result is similar to deadlock in that client requests to the storage system are never satisfied. Resource monopolization is another condition which can lead to indefinite postponement even when it does not cause deadlock.

In some environments it is quite possible that an apparently well behaved client may hold a resource indefinitely. For example, some persistent process may acquire a set of drives and volumes to use as scratch space for calculations. Such a process would prevent other storage system clients from ever accessing those drives. While it is possible for a PVL to force a volume to be dismounted, the result can lead to serious errors in the client and corrupted data on the volume. Because of this, most systems do not allow a PVL to force a client to terminate the use of a drive/volume combination. For the purposes of this paper we assume well behaved clients which hold resources for some bounded amount of time.

3. Traditional approaches to deadlock

Deadlock conditions have been well defined in operating systems research. There are four conditions necessary for deadlock to exist. Coffman et. al. [12] introduced these conditions which, defined simply are:

Mutual Exclusion

A non-sharable resource is held by a process. Other processes requesting the resource must wait until it is relinquished.

Hold and Wait

A process is holding one or more resources and is waiting to acquire additional resources.

No Preemption

A resource will only be released voluntarily by the process which holds it.

Circular Wait

There exists a cycle in the dependency graph which represents the processes, the resources they hold, and the resources they have requested. More formally, there exists a set of processes $\{p_0, p_1, ..., p_n\}$ such that p_0 is waiting for a resource held by p_1, p_1 is waiting for a resource held by p_2, and so on, and p_n is waiting for a resource held by p_0 [10].

When all four of these conditions are satisfied, deadlock will occur. All three of the PVL deadlock cases presented in section 2 satisfy all four deadlock conditions.

There are three major approaches to dealing with deadlock. Dietel [13] identifies these approaches as:

- *Deadlock Prevention*
- *Deadlock Avoidance*
- *Deadlock Detection and Recovery*

We will now examine each of these approaches and their application to a PVL.

3.1. Deadlock prevention

Deadlock prevention entails eliminating any possibility of a deadlock condition occurring. This is done by ensuring that at least one of the four conditions necessary for deadlock can never occur.

In a PVL it is impractical to try to eliminate the Mutual Exclusion condition because drives can not be assumed to be concurrently shared between clients. As discussed in section 2.4, preemption is also not practical, so the No Preemption condition also holds. Circular Wait is generally a function of client requests. Because clients do not typically coordinate independent requests with one another, there is no way for them to guarantee that their requests will never result in a circular wait for resources. A PVL can not prevent such a condition as it is unaware of all of its clients' higher level interdependencies. Because of this we cannot eliminate the Circular Wait condition in the PVL.

Our final chance at eliminating a deadlock condition is to prevent the Hold and Wait condition from ever occurring. As it turns out it would be trivial for a storage system to eliminate this condition and thereby eliminate any potential for deadlock. The PVL could simply request drives for a client, but never hold drive resources while waiting for others to become available. In this manner a PVL mounting a four-wide stripe might request the drives one at a time, releasing any successful drive reservations if any one drive request couldn't be immediately satisfied. While this prevention algorithm eliminates the potential for deadlock, it represents a very inefficient algorithm for obtaining resources. Depending on the implementation, it can also lead to job resource starvation.

3.2. Deadlock avoidance

Given information about how serially reusable resources such as cartridges and drives will be used, it is possible to construct an algorithm that avoids deadlock. Deadlock avoidance algorithms avoid deadlock occurrence through the judicious allocation of resources. An example of a classical avoidance algorithm is Dijkstra's Banker's Algorithm [14].

Storage system clients typically know the total number of resources a job will require immediately at the start of a job. A four-wide stripe requires four drives and

four volumes. A direct tape-to-tape copy of one four-wide stripe to another requires eight drives and eight volumes. If a PVL presents an interface allowing clients to provide information about what mounts need to occur atomically together, PVL deadlock can be avoided and the possibility of client level deadlock (circular wait) is diminished. Because of this, PVLs are well suited for the application of deadlock avoidance algorithms. It should be noted that clients which are not well behaved can still deadlock themselves by independently requesting mounts that are dependent on each other at a level higher than the PVL.

3.3. Deadlock detection and recovery

Unlike deadlock avoidance, deadlock detection algorithms make no effort to prevent deadlock from occurring. Instead, the system is periodically examined to determine if deadlock has occurred. Detection algorithms typically involve checking resource allocation graphs for cycles [15].

When it is determined that deadlock has occurred, deadlock recovery must be invoked. Deadlock recovery involves terminating processes or preempting resources in order to break the deadlock. Deadlock detection and recovery are often used in environments where deadlock is unlikely and/or checking for deadlock at each request is impractical.

In order to implement deadlock detection and recovery in a PVL it must be possible for the deadlock to be broken. As discussed in section 2.4, breaking deadlock by preempting resources or by forcibly unmounting client tapes in a PVL is at best difficult, and often is not allowed.

4. A design for PVL deadlock avoidance

We would now like to outline the design of a PVL meant to function in a high performance storage (striped media) environment. We will concentrate on those aspects of the design which address deadlock issues we have raised and explain the rationale behind our design decisions.

4.1. Design approach

We chose to combine deadlock avoidance techniques with an algorithm designed to prevent indefinite resource postponement. Deadlock prevention was rejected because of its dependence on inefficient resource allocation algorithms. Deadlock detection and recovery was rejected because of the operational ramifications of the requirement that a PVL be able to preempt or terminate requests in order to recover from deadlock.

Our design presents a set of transactional Application Programming Interfaces (APIs) to the client application. These APIs allow a client to atomically specify all of the resources which will be needed for a single job. The set of required resources is used by the deadlock avoidance algorithm to determine if all or part of the request should be queued. The queuing mechanism is first-come-first-served (non-preemptive) with defined precedence rules for reserving storage resources. Out-of-order mounts are allowed, but preemption of storage resources is tightly controlled.

4.2. APIs for atomic mounts

In order to allow for atomic mounts, our PVL design includes the following APIs:

*MountNew(*JobID)*

This interface is used to obtain a unique job identifier. This identifier must be used in subsequent PVL calls to identify a set of physical volumes to be mounted in one atomic operation.

MountAdd(VolumeID, JobID)

MountAdd is used to add a volume to a list of volumes that will be atomically mounted under a job identifier. JobID is obtained through a previous call to MountNew. MountAdd should be called once for each volume in an atomic mount request

MountCommit(JobID)

This API commits (actually launches) the atomic mount request associated with JobID. Once the mount request has been built using calls to MountAdd, MountCommit signals to the PVL that the request building is complete and that the job should be submitted.

Using these PVL APIs, a client can build a single or multi-volume mount request and submit (commit) it to the PVL. The interface also allows a pair of clients to work together to create a single atomic mount request for applications such as tape-to-tape copying. Imagine two Virtual Storage Servers (VSSs) which need to copy data from a tape virtual volume managed by one VSS to a tape virtual volume managed by the other VSS. As mentioned previously, to avoid deadlock and to maximize drive utilization, the tape mounts for both VSSs should be combined atomically. With our design's APIs, one VSS could obtain a job identifier using MountNew. This identifier could be shared by both VSS clients in the building of a single mount request using calls to MountAdd. Once the request was built, one of the VSSs would be in charge of actually committing the combined mount. Such an algorithm allows clients to avoid a possible deadlock situation that the PVL would otherwise be unable to prevent.

In our design the APIs allowing atomic mount present an asynchronous interface. In order to notify a client that mounts have completed, the design specifies an API for client notification of mounts:

PVLNotify(JobID, VolumeID, DriveID)

PVLNotify is used to asynchronously inform a client that the PVL has successfully mounted a volume. Included with the notification is information detailing the drive on which the volume was mounted. This API is called by the PVL as each volume is mounted.

4.3. PVL mount queuing

Internally our design accepts MountAdd requests, queuing them until the job is committed. It is a job's commit time which is used to initially order mount requests (not the time of the MountAdds). Once a commit is received for a job, the PVL first verifies that the job does not require more resources than exist in the system. The PVL also verifies that the job is requesting valid volumes and that none of the volumes reside on the same cartridge. At that point the PVL returns to the client and indicates that the mount is in progress.

Asynchronously, the PVL begins allocating resources. The key to deadlock avoidance is preventing circular wait. Our deadlock avoidance algorithm achieves this by requiring that the PVL follow a strict precedence ordering in reserving resources and that those resources be assigned to client requests in a specific order. It is our experience that drive resources are typically much more scarce than media resources. A typical site might have thousands of cartridges and fewer than 20 drives. Because drives are more scarce, our PVL first attempts to reserve the media necessary for the request before trying to reserve drives.

One aspect of our design is that cartridges, not physical volumes, are reserved by the PVL. This is important because, as mentioned previously, two or more distinct physical volumes may reside on the same cartridge. Reserving cartridges ensures that two mount requests for physical volumes on the same cartridge are not issued to a PVR concurrently. When a cartridge resource becomes available, it will be given to the appropriate mount job which has the earliest commit time.

Once cartridge resources are reserved, the job is placed in a second queue to reserve drive resources. Free drive resources are allocated to requesting jobs based on their order in the queue. No preemption of a job for better resource utilization is allowed. This does have potential drawbacks including less than optimal utilization of drive resources.

Another important aspect of the design is the fact that a particular drive is not reserved for a mount request, rather a count of drives of the requested type is kept and the mount request reserves a drive by decrementing a count of available drives of the appropriate type. This allows the PVL to deal with PVRs which do not allow pre-assignment of drive resources.

An exception in this drive assignment algorithm is made when the PVR managing the cartridge in question is an operator PVR (human mounted drives). In this case the PVL does not do any reservation of drives; rather it immediately sends the request to the PVR (an exception to this rule will be discussed when we describe how we deal with out-of-order volume mounts). The benefits of this technique are two-fold. First, allowing an operator to see all tape mount requests rather than just the oldest optimizes the process of retrieving cartridges from vaults. Second, it allows for a more simplistic algorithm to deal with out-of-order volume mounts which we will discuss shortly.

When a volume mount has been assigned both its cartridge and drive, the PVL asks the PVR to mount the cartridge. Because the deadlock avoidance algorithm guarantees that drive assignments will not cause deadlock, we can mount each cartridge as soon as the drive is assigned rather than waiting for drives to be reserved for the entire job. When the PVR has successfully mounted a cartridge, the PVL verifies the internal media label using a mover. If the internal label is correct the PVL responds to the client using the PVLNotify API.

We have shown that this PVL design very simply addresses both the problem of atomic mount induced deadlock, and that of multiple concurrent requests for the same cartridge resource. The design presented thus far has not addressed the challenge of out-of-order mounts.

4.4. Out-of-order mount handling

As we discussed earlier, allowing an out-of-order mount to be honored can lead to deadlock and indefinite postponement. For our design we chose a simple strategy which prevents deadlock due to out-of-order mounts, but allows the use of conventional stackers at the risk of indefinite postponement.

Before settling on a design, we considered the simple algorithm of allowing no preemption of mount requests. Under such an algorithm, if a volume is mounted which has been requested by a PVL client, but hadn't yet been requested to be mounted by PVL, then the volume will be dismounted. Such an approach makes extremely poor use of traditional stacker devices and forces an operator to retrieve a dismounted cartridge and place it back in the stacker before the mount can be satisfied. This eliminates the primary labor-saving advantage of using a stacker.

Because of this weakness we decided to allow out-of-order mounts to be honored with some restrictions. Our design accomplishes this by treating operator mounted volumes specially. As we stated previously, our deadlock avoidance algorithm sends operator mounted volume requests to a PVR as soon as a cartridges are reserved rather than first trying to reserve a drive. The one exception to this rule is that, once mount requests for a multi-volume mount involving an operator PVR have begun, all subsequent mount requests for that PVR are queued in the PVL until the hand mounted volumes involved in the multi-volume mount are mounted. This

rule, combined with the rule that mounts are only accepted if they have been requested by a PVL, ensures that two multi-volume mounts will never deadlock on operator controlled drive resources.

Important to our design is the requirement that mount request displays communicate to operators which volumes are associated together as part of a multi-volume mount. This information is vital in order to provide an operator enough information to keep him or her from stacking two or more volumes that are part of the same multi-volume mount in the same stacker. Also, because we assume that most operator mount request displays will show how long a particular mount request has been outstanding, we depend on the operations staff to make sure that the number of preemptions, and hence the length of postponement, are minimized.

5. Possible enhancements to the design

The PVL design aspects we have presented represent an attempt to, in an uncomplicated manner, satisfy requirements imposed by a high performance storage system while preventing resource deadlock. There are a number of enhancements that might be made to this design without violating our goal of maintaining PVL simplicity.

5.1. Scheduling and preemption enhancements

Our PVL design operates fundamentally on a first-committed first-served scheduling basis. The only preemption allowed takes place when out-of-order mounts are allowed in operator PVRs. One can imagine any number of prioritization schemes that would allow jobs to be scheduled based on an assigned job weight or priority. Jobs could be assigned greater priority based on:

- *client provided priority,*
- *system provided priority based on client id,*
- *client provided projected mount duration,*
- *statistical information on past client mounts,*
- *type of media being mounted,*
- *which requests would most optimally use available drive resources,*
- *how many preemptions a job had already sustained.*

As long as the PVL followed the deadlock avoidance rule that no circular wait be allowed, then any weighted scheduling mechanism could easily be added to our design without adding possible deadlock scenarios.

One can also imagine allowing the preemption of jobs that the PVL has already sent to a PVR. This preemption might be the result of jobs of a greater weight (assigned by a prioritization scheme) arriving at the PVL after submission of the original mount request to the PVR. Implementing this kind of preemption would require the addition of a mechanism to back out the drive and cartridge allocation of preempted jobs, as well as a mechanism to abort or dismount PVR requests that are being preempted.

All of the priority weighting schemes listed (and many more are possible) are very site dependent. Each site will have different rules and preferences as to how a job should be weighted based on their local environment. In order to accommodate different weighting mechanisms, the priority setting portion of the PVL would best be implemented as a separate policy module that each site could modify. Inputs to the policy module would be all information known about the request, and the output would be a priority weight to be assigned to the job.

5.2. Adding client deadlock detection

One of the key features of our deadlock avoidance algorithm is that the PVL is made aware of all resources which will be used by the client to satisfy a single request. Our PVL APIs allow one or more clients to specify all of the resources a request will use; the PVL uses this information to prevent deadlock. It is possible that a poorly behaved client might deadlock itself by issuing separate co-dependent PVL requests.

While our PVL design does not totally protect a client from itself, it could be enhanced to detect when client induced deadlock might have occurred. This detection would rely on watching how long a particular job has been mounted, possibly in conjunction with client provided mount duration information. Regardless of the detection method, the PVL could alarm operators and/or clients that a deadlock condition may exist, or the PVL could even be given the capability to unmount the offending jobs.

5.3. Limiting the amount of preemption due to out-of-order mounts

By honoring out-of-order mounts in our design, we have introduced the possibility of indefinite postponement. Possible approaches to minimizing the impact of indefinite postponement might involve trying to limit the extent or number of postponements allowed.

Eliminating the extent of a single postponement is difficult for a PVL because, as we have seen, a PVL typically can not unmount a volume on its own prerogative. This fact alone makes any single postponement one of indeterminate time and makes any enhancement to this aspect of our design difficult. It is only in systems where mount durations are well known, controlled, or modifiable by a PVL, that allowing a preemption does not entail some amount of postponement risk.

Managing the number of times a mount request can be postponed is an easier challenge. Setting a hard limit

on the number of preemptions, weighted priority schemes, and aging algorithms could all be applied to aid indefinite postponement detection and recovery with some benefit. Unfortunately such algorithms are often very dependent on particular site policies, and do not eliminate the problem of any single postponement being of unknown duration.

6. A case study - the HPSS PVL

The PVL design presented in this paper has been implemented as part of the National Storage Laboratory's [16] High Performance Storage System (HPSS) [17, 18] under development by the National Storage Laboratory. HPSS is a storage system that manages scalable, parallel storage, possibly petabytes in size, requiring up to several gigabytes per second aggregate throughput. HPSS is designed to meet the needs of parallel computers, traditional supercomputers and workstation clusters. HPSS is based upon the IEEE Reference Model for Open Storage Systems Interconnection and is implemented using Open Software Foundation's (OSF) Distributed Computing Environment Remote Procedure Calls (DCE RPCs) [19] and Transarc Corporation's Encina metadata management and transactional RPC software [20].

To meet performance and scalability requirements, HPSS requires that a PVL mount multiple physical volumes in parallel to service a single client request. An HPSS PVL must satisfy all of the requirements and challenges discussed in this paper. Our PVL design was implemented for HPSS in the C language in a platform independent manner, and currently runs on an IBM RS/6000 computer under the AIX operating system.

The HPSS PVL is a multitasking server built on top of DCE threads. It uses DCE RPCs to communicate with clients, PVRs and Storage System Manager applications. Unix sockets are used to communicate with Movers. The PVL stores its metadata (internal information about configurations, volumes, requests, etc.) using Encina's Structured File System (SFS) transactional metadata storage system. The PVL maintains support interfaces allowing storage system management applications access to configuration and status.

Our PVL presents an API to its client which is a superset of the APIs presented in our design above. When each request is committed, a job is created in the PVL and placed at the end of an ordered job list. A second list of all cartridges which have been requested by existing jobs is also maintained. These two lists form a two dimensional sparse matrix [see Figure 4].

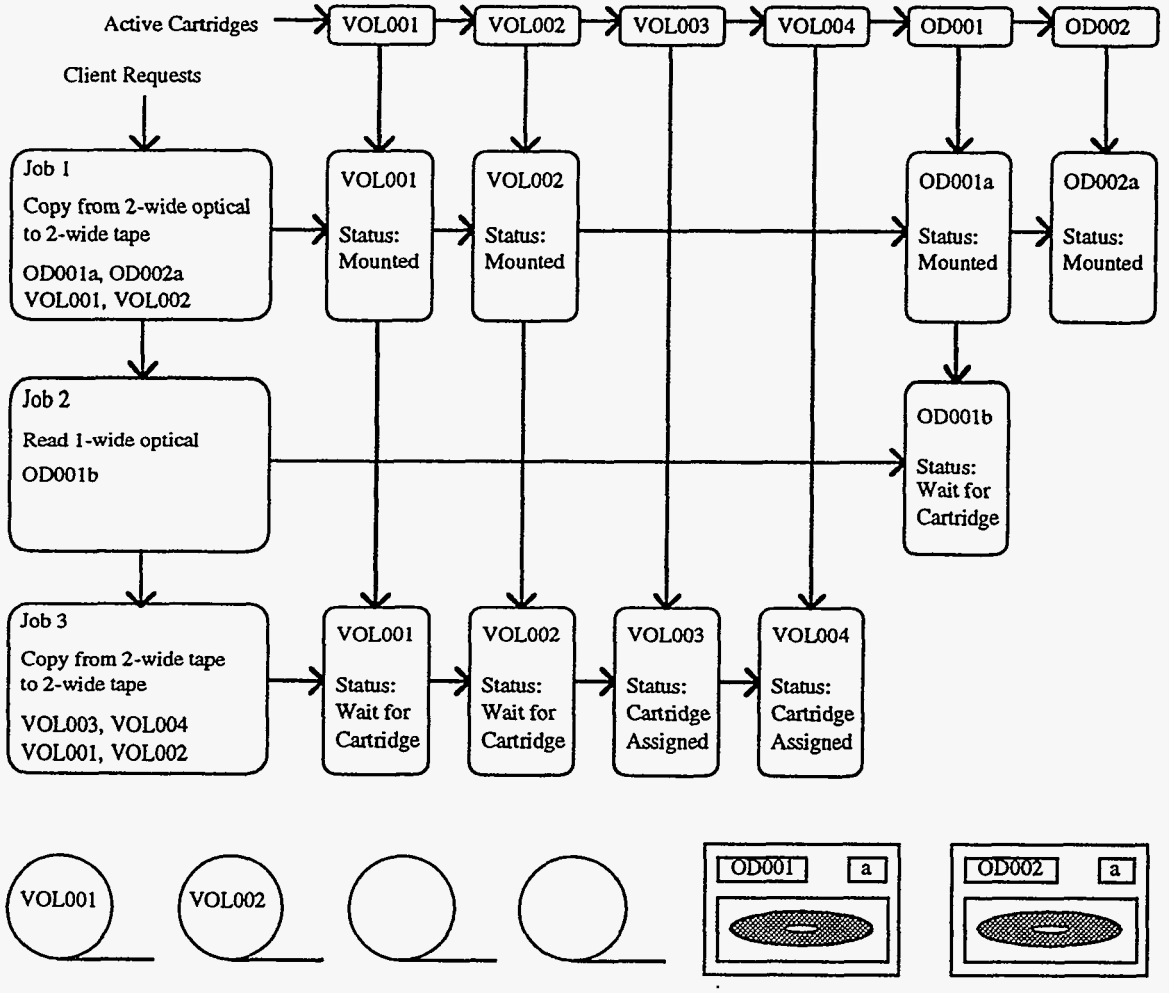


Figure 4. Matrix of jobs and active cartridges

Note that in Figure 4 VOL003 and VOL004 will not be mounted even though drives are available. This is because a job must have all of its cartridges assigned before it can allocate any drives.

Each node in the matrix is a separate activity. Each activity represents a single volume that needs to be, or has been, mounted. An activity moves through a set of states first acquiring resources, then mounting a cartridge in a drive, and finally dismounting the cartridge and assigning the resources to the next waiting activity. Some of the more common activity states, transitions between these states, and some expanded implementation details are described below:

UNCOMMITTED

Activities in this state represent volume mounts that have been added to a job by a call to MountAdd, but have not yet been committed by the client.

CART_WAIT

Once a job has been committed, all activities within that job are changed from UNCOMMITTED to CART_WAIT state. Activities in the CART_WAIT state are actively attempting to reserve their respective cartridges. Once a cartridge is successfully reserved for an activity it will transition into the CART_ASSIGNED state.

CART_ASSIGNED

When a cartridge is assigned to an activity the activity waits in the CART_ASSIGNED state for all other activities in the job to have a cartridge assigned. Once all activities have a cartridge assigned the activity transitions to one of two states. First, if the volume is to be operator mounted, and there are no multi-volume mounts pending involving the operator PVR, then the activity will transition to a

MOUNT_PENDING state. This is done because operator mounts do not reserve drives in our implementation in order to optimize cartridge vault management and to allow for out-of-order mount handling in operator PVRs.

If the volume is to be robotically mounted, or if a multi-volume mount is pending that involves the operator PVR, then the activity transitions into DRIVE_WAIT state. Multi-volume operator mounts cause all subsequent operator mounts to queue in the DRIVE_WAIT state in order to prevent deadlock caused by out-of-order mounts.

DRIVE_WAIT

All activities that describe robotic mounts are placed into the DRIVE_WAIT state while they attempt to reserve drive resources. Available drive counts are used to assign drives rather than assigning specific drives. This allows the PVR to make a selection of which drive to use. The PVR may base the selection on criteria like the distance of the cartridge from the drive. Since the PVL is unaware of the details of each robots configuration, the selection of a specific drive is always left up to the PVR.

A DRIVE_WAIT activity transitions to a MOUNT_PENDING state once an appropriate drive is reserved. Activities that represent operator mounts which are waiting behind pending multi-volume mounts wait in DRIVE_WAIT state until the multi-volume mounts are complete, at which time they also transition to the MOUNT_PENDING state.

MOUNT_PENDING

Once an activity achieves the MOUNT_PENDING state the mount request is issued to the appropriate PVR. Activities remain in the MOUNT_PENDING state until, either the PVR responds that it has mounted the cartridge, or the cartridge is found to have been mounted when the PVL polled a drive. PVL drive polling was implemented to deal both with operator mounted drives, and with PVRs that don't provide reliable mount notification.

READING_LABEL

An activity is in this state during the time that a PVL takes to verify that a PVR mounted the proper volume by reading the internal media label (when such verification is appropriate).

MOUNTED

Once the PVL has determined that the PVR has correctly mounted a volume, the appropriate activity is placed in MOUNTED state until either a dismount request is received or an error requiring clean-up of an activity occurs.

Other States

A number of other activity states exist which we will not detail here. Included are states to deal with dismounting, errors, and states to deal with the injection and ejection of cartridges.

The HPSS implementation of our design currently supports StorageTek 4400, IBM 3494, IBM 3495, Ampex DST800 and operator mounted drives. The next release of the HPSS PVL will include enhanced device support and will support mounts requested for magnetic disk volumes as required by the IEEE Reference Model for Open Storage Systems Interconnection.

Even though HPSS does not currently support any optical disk devices, our PVL does support the concept of multi-sided cartridges. This is necessary for future support of optical devices, but may also be needed by tape devices. For example, Ampex DD2 cartridges can be divided into multiple partitions and it is possible to mount cartridges such that the drive firmware enforces access to only a specific partition. In this case a single DD2 cartridge could be considered to have multiple volumes.

HPSS was successfully demonstrated at Supercomputing '94. As part of that demonstration the HPSS PVL was involved in mounting one-way, two-way, and four-way media stripes of tape and disk media. At the time this paper was written, February 1995, a preliminary release of HPSS was being installed and tested at several early deployment sites. The preliminary release contains support for striped tape. The next release of HPSS adds support for striped disk, multiple storage hierarchies, and migration and caching between hierarchies.

While the HPSS PVL was implemented to fill the need for a PVL satisfying the requirements of a high-end storage system, it also served as a proof of concept of our PVL design. It showed that expanding upon typical PVL interfaces and dealing with deadlock challenges was not only possible, but could be accomplished with a relatively simple design. With time we are sure that some of the enhancements mentioned above will be added to the HPSS PVL. However, based upon input from the operational sites involved in the development of HPSS, we found that implementing these enhancements will have to be done carefully because of their site specific nature.

7. Conclusion

A PVL mounting striped, removable media can cause deadlocks three different ways: contention for drives, contention for cartridges, and mounting out-of-order volumes. There are several well known methods of eliminating deadlock when acquiring serially reusable resources; we chose deadlock avoidance for our PVL design. A PVL is ideally suited to deadlock avoidance techniques because its clients are able to specify all of the resources which will be used by a single request and because deadlock avoidance does not require the preemption of resources. The key to our deadlock avoidance algorithm is to prevent circular dependencies by requiring that the PVL follow a strict precedence ordering in reserving resources and that those resources be assigned to client requests in a specific order. This PVL design has been demonstrated through it's implementation as part of the National Storage Laboratory's HPSS system.

8. Acknowledgments

HPSS is the product of a collaboration between Industry (IBM Government Systems), the Department of Energy (Lawrence Livermore National Laboratory, Los Alamos National Laboratory, Oak Ridge National Laboratory, Sandia National Laboratories), Higher Education (Cornell University), and NASA (Langley and Lewis Research Centers). Developers representing these organizations have worked with diligence and dedication as a single team. Hardware and software to support HPSS development and demonstration have been provided by Ampex, IBM, Kinesix, Maximum Strategy Inc., Network Systems Corp., PsiTech, Sony Precision Graphics, Storage Technology, and Zitel. Additional support in making HPSS available to high performance clients has been provided by Cray Research, Intel, IBM, and Meiko.

We would particularly like to thank Norm Samuelson who wrote most of the code for the HPSS PVL. We would also like to thank Jim Daveler for his work on both the PVL and PVR. Finally, we are grateful to Randy Burris for his helpful comments on this paper.

This work was, in part, performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contarct no. W-7405-Eng-48 and under the U.S. Department of Energy Cooperative Research and Development Agreement and by IBM Government Systems under Independent Research and Development and other internal funding.

9. References

[1] Herbst, Kris, "Vendors Seek Solutions to Growing I/O Bottleneck," *Supercomputing Review*, Vol. 4, No.3, pp. 46-49, March 1991.

[2] Thorndyke, Lloyd M., "Supercomputers and Mass Storage: The Challenges and Impacts," *Digest of Papers, Seventh IEEE Symposium on Mass Storage Systems*, November 1985, pp. 27-30, (1985).

[3] Hyer, Robert, Richard Ruef, and Richard W. Watson, "High Performance Direct Network Data Transfers at the National Storage Laboratory," *Proc. of Twelfth IEEE Symposium on Mass Storage Systems*, April, 1993, pp. 275-283, (1993).

[4] IEEE Project 1244, Reference Model for Open Storage Systems Interconnection, Lester Buck, Sam Coleman, Rich Garrison and Dave Isaac Editors, September 1994.

[5] Coleman, Sam, "Physical Volume Repository," *Digest of Papers, Ninth IEEE Symposium on Mass Storage Systems*, October 1988, pp. 21-24, (1988).

[6] Kitts, David, Sam Coleman and Bruce Griffing, "Bitfile Mover," *Digest of Papers, Ninth IEEE Symposium on Mass Storage Systems*, October 1988, pp. 25-28, (1988).

[7] Golubchik, Leana, Richard R. Munz, and Richard W. Watson, "Analysis of Striping Techniques in Robotic Storage Libraries", *Proc. of the Fourteenth IEEE Symposium on Mass Storage Systems*, September 1995.

[8] Drapeau, Ann L. and Randy H. Katz, "Striped Tape Arrays", *Proc. of Twelfth IEEE Symposium on Mass Storage Systems*, April 1993, pp. 257-265, (1988).

[9] Drapeau, Ann L. and Randy H. Katz, "Striping in Large Tape Libraries," *Proc. of Supercomputing '93*, pp. 378-387, November 1993.

[10] Silberschatz, Abraham and James L. Peterson, Operating System Concepts, Addison-Wesley, pp. 187-218, 1988.

[11] Ullman, Jeffrey D., Principles of Database Systems, Computer Science Press, pp. 372-374, 1982.

[12] Coffman E. G., M.J. Elphic, and A. Shoshani, "System Deadlocks," *Computing Surveys*, Vol. 5, No. 2, pp. 67-78, 1971.

[13] Deitel, Harvey M., An Introduction to Operating Systems, Addison-Wesley, pp. 126-150, 1984.

[14] Dijkstra, E. W., "Cooperating Sequential Processes", Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

[15] Shaw Alan C., The Logical Design of Operating Systems, Prentice-Hall, pp. 203-243, 1974.

[16] Watson Richard W., and Robert A. Coyne, "The National Storage Laboratory (NSL): Overview and Status," *Proc. of Thirteenth IEEE Symposium on Mass Storage Systems*, June, 1994, pp. 39-43, (1994).

[17] Coyne Robert A., Harry Hulen and Richard Watson, "The High Performance Storage

System", *Proc. of Supercomputing '93*, November 1993, pp. 83-92, (1993).

[18] Teaff, Danny, Robert A. Coyne, and Richard W. Watson, "The Architecture of the High Performance Storage System," *Proc. of Fourth NASA GSFC Conference on Mass Storage Systems and Technologies*, College Park, MD, March 28-30, (1995).

[19] Open Software Foundation Distributed Computing Environment Version 1.0 Documentation Set, Open Software Foundation, Cambridge, Mass., 1992.

[20] Encina Documentation Set, Transarc Corporation, Pittsburgh, Pa., 1994.