2018-06-01

Isolation of Temporary Storage in High Performance Computing via Linux Namespacing

Steven Tanner Satchwell
Brigham Young University

Isolation of Temporary Storage in High-Performance Computing via Linux Namespacing

Steven Tanner Satchwell

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

School of Technology

Brigham Young University

ABSTRACT

Isolation of Temporary Storage in High-Performance Computing
via Linux Namespacing

Steven Tanner Satchwell
School of Technology, BYU
Master of Science

Per job isolation of temporary file storage in High Performance Computing (HPC) environments provide benefits in security, efficiency, and administration. HPC system administrators can use the mount_isolation Slurm task plugin to improve security by isolating temporary files where no isolation previously existed. The mount_isolation plugin also increases efficiency by removing obsolete temporary files immediately after each job terminates. This frees valuable disk space in the HPC environment to be used by other jobs. These two improvements reduce the amount of work system administrators must expend to ensure temporary files are removed in a timely manner.

Previous temporary file removal solutions were removal on reboot, manual removal, or removal through a Slurm epilog script. The epilog script was the most effective of these, allowing files to be removed in a timely manner. However, HPC users can have multiple supercomputing jobs running concurrently. Temporary files generated by these concurrent or overlapping jobs are only deleted by the epilog script when all jobs run by that user on the compute node have completed. Even though the user may have only one running job, the temporary directory may still contain temporary files from many previously executed jobs, taking up valuable temporary storage on the compute node. The mount_isolation plugin isolates these temporary files on a per job basis allowing prompt removal of obsolete files regardless of job overlap.

Keywords: bind mounts, mount namespaces, Slurm, supercomputing, HPC, temporary storage

ACKNOWLEDGEMENTS

Many thanks to Ryan Cox, Operations Director for the Fulton Supercomputing Lab (FSL), and other FSL members for providing supercomputing resources and valuable insight on Slurm configurations.

A special thanks to my graduate committee for their continued guidance and insight on presenting a meaningful contribution work to the industry. My committee and other BYU IT faculty support extended beyond my thesis as well. My thanks to them in helping me find student employment to help meet my financial needs throughout my graduate career. I also extend my thanks to BYU and other IT program donors for additional financial support through scholarship.

Finally, thanks to my family for their patience, encouragement, and love through my pursuit of higher education. Their faith in my abilities has been a driving force in my life for as long as I can remember.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

System administrators of Slurm managed High Performance Computing (HPC) sites are concerned about the privacy and utilization of temporary space. Users can unintentionally expose their information by assigning incorrect permissions to temporary files. Additionally, those temporary files may occupy scarce file space longer than necessary. There is no Slurm specific solution to this problem. Temporary files are not removed until the supercomputing node upon which the files reside is rebooted. Alternatively, system administrators can manually remove these temporary files, or a Slurm epilog script can be written to remove them. Although the epilog script is preferable to manual removal, it still has its drawbacks. Linux marks which files are owned by a user, but the script can't distinguish which files belong to a job. The epilog script cannot remove temporary files until all jobs launched by a user have completed, or else risk removing temporary files still in use by a user's active job.

Consider two types of users, A and B (see Figure 1-1). User A launches jobs consistently, but never has overlapping jobs. The epilog script can effectively remove temporary files as each job terminates, as shown by the blue lines. User B has overlapping jobs. Because the epilog script can't differentiate which files belong to which job, all temporary files generated by jobs 1-5 remain on the supercomputer until there are no more overlapping jobs for that user. The typical

HPC user has a combination of overlapping, and non-overlapping jobs. When jobs overlap, temporary disk space may be occupied indefinitely by non-active jobs.

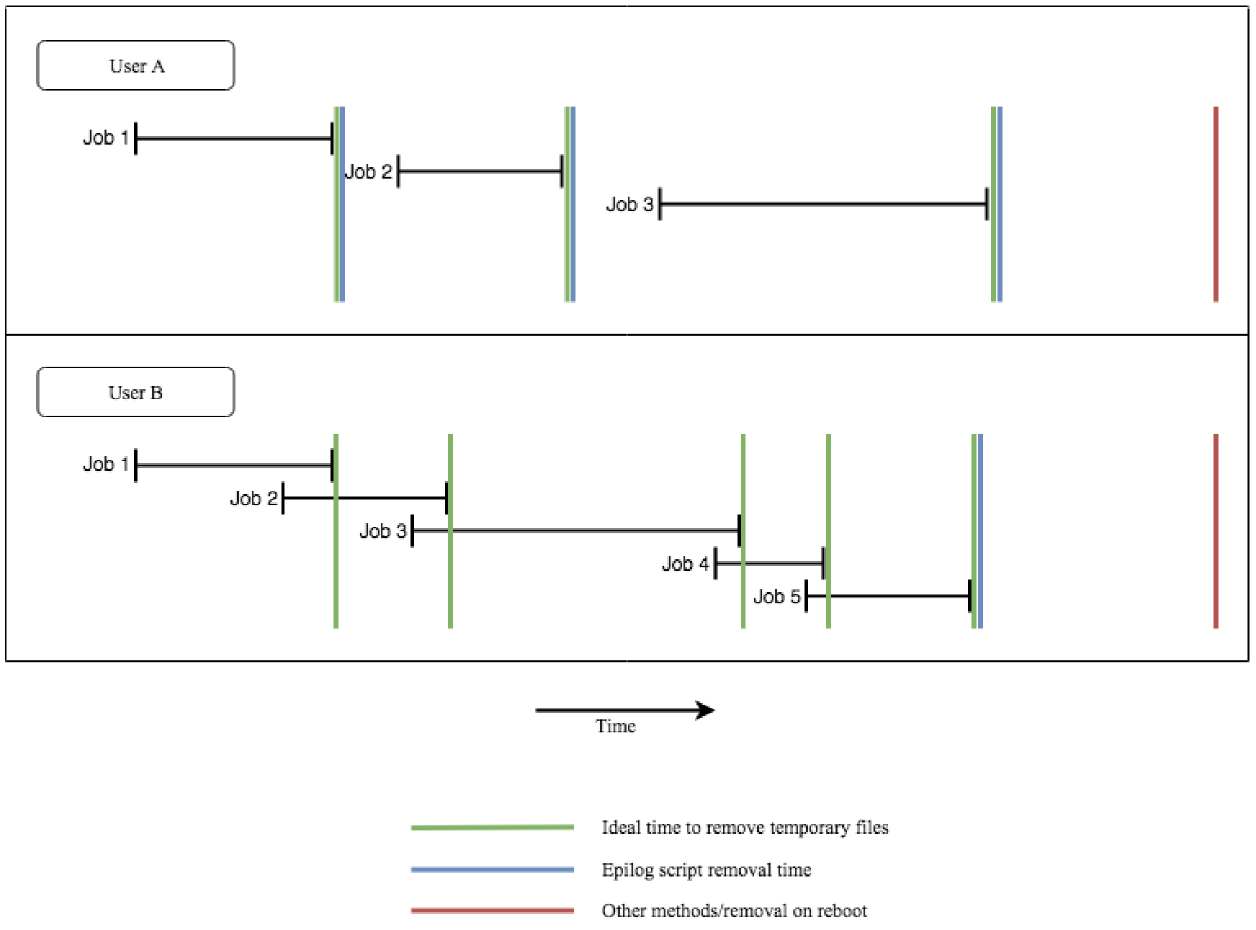
User A
Job 1
Job 2
Job 3
User B
Job 1
Job 2
Job 3
Job 4
Job 5
Time
Ideal time to remove temporary files
Epilog script removal time
Other methods/removal on reboot

Figure 1-1: User Job Scenarios

As an example, consider an HPC user, John. John has two jobs executing on compute node **m81**. Job 1 has created 15GB of temporary files. Job 2 has only created 3KB of temporary files. All temporary files for jobs created by John are stored in a user directory within `/tmp`, in this case `/tmp/john` (see Figure 1-2). Job 1 terminates without removing its temporary files. Job 2 continues executing for an additional 7 days. Slurm is unable to determine that the 15GB of temporary files belong to Job 1 and not Job 2, so it must wait 7 days until all of John's jobs on the compute node have terminated. If John launches a third job, job 3, with a runtime of 7 days on compute node **m81** right before job 2 terminates, the 15 GB of temporary files from job 1 won't be removed until almost 14 days after job 1 terminated. If John continues launching new jobs on compute node **m81**, Job 1's 15 GB of temporary storage may occupy valuable node local temporary space indefinitely.

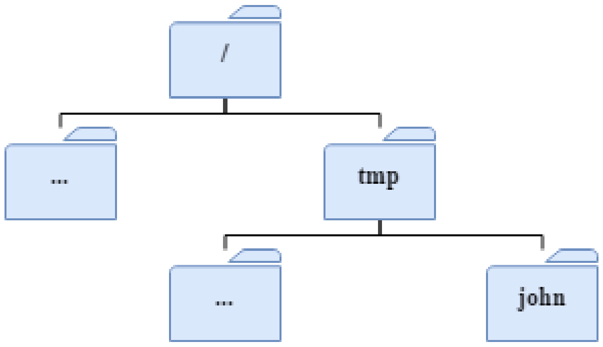

Figure 1-2: Per User Directory Structure

The Simple Linux Utility for Resource Management (Slurm) job scheduler can place an indefinite sequence of HPC jobs from one user on the same compute node, each job generating gigabytes of temporary files that the job often fails to remove when it terminates. Slurm cannot distinguish which temporary files belong to which job because all jobs share a common temporary directory. Since temporary files are tagged by their owner user id, a user's temporary files can only be removed once all the user's jobs on a compute node have terminated. This means if a user consistently has overlapping jobs, completed job temporary files cannot be removed until all jobs launched by that user have completed. Not only does this consume scarce temporary disk space, but those temporary job files could be accessible to other users on the HPC cluster, depending on the permissions of the files.

1.1 Problem Statement

Current multiuser Slurm managed HPC environments have two challenges: isolation of temporary files, and removal of temporary files. The removal of temporary files is solved by system administrators in one of three ways: manual removal, removal on reboot, and removal by Slurm epilog script. All three of these solutions don't remove temporary files in an efficient manner and can leave temporary files occupying disk space long after the job that created them has terminated. None of these solutions address the isolation of those temporary files.

Building isolated temporary storage will increase security by isolating temporary job files where no isolation previously existed and reduce temporary file storage space by allowing removal of those files by administrators as HPC jobs complete. Isolated temporary storage will be implemented in the form of a Slurm task plugin, mount_isolation. The mount_isolation plugin will use Linux Mount Namespaces and Bind Mounts to isolate temporary files by user and job id, allowing the prompt removal of completed job temporary files. This will save disk space and increase security (isolation of files) between users. The amount of disk space saved by the mount_isolation plugin will vary depending on user submitted jobs, and how administrators currently remove temporary files. Security will be improved regardless of these variables by isolating files where no isolation previously existed. Giving each job its own mounted `/tmp` directory within a private namespace will allow administrators to remove files immediately upon job termination.

1.2 Hypotheses

Hypothesis 1 (H1): The mount_isolation plugin will save scarce temporary disk space in Slurm managed HPC environments.

Hypothesis 2 (H2): The mount_isolation plugin will increase security by isolating temporary files from other users and other jobs in Slurm managed HPC environments.

1.3 Justification

Disk space in HPC environments is a valuable resource that needs to be continuously balanced and cleaned up. Removing temporary files as soon as they are no longer needed helps keep disk space free for other uses. Utilizing bind mounts on top of the Linux mount namespace allows temporary files to be cleaned up on a more consistent basis.

1.4 Delimitations

While many HPC environments are managed by Slurm, there are many others that use different resource management software. This thesis does not delve into how/if alternative resource management software deals with this problem, however it is important to note that because this solution uses Linux mount namespaces and bind mounts, it can be adapted to work in any Linux environment. Data gathered for this thesis is gathered from the BYU Fulton Supercomputing Center over a period of 1 month. Results may vary depending on when data is gathered, type of jobs submitted to the scheduler, and HPC environment. While these results may vary, this solution will always improve temporary file isolation and space removal.

2 LITERATURE REVIEW

2.1 High Performance Computing

HPC uses parallel processing to execute scientific programs that require more processing power than consumer computing devices. HPC is essential to solving computationally intensive problems in areas such as finite element analysis, molecular modeling, weather prediction, and materials science. Massive HPC processing power is typically organized in one of two configurations. One configuration is distributed computing, where hundreds or thousands of computers are connected over the network. Each computer receives a small piece of the processing job, computes a local result, and finally submits the report back over the network to a central server. A more common HPC configuration is a high-density cluster of processors, typically in data centers, that intercommunicate frequently to cooperatively complete processing tasks. Historically, distributed configurations such as Folding@home or Genome@home have issued small independent tasks to participating remote compute nodes (Larson, 2009), while institutional supercomputer clusters are built to solve scientific problems which require large, frequent intercommunication between compute nodes during computation.

2.2 Simple Linux Utility for Resource Management

High-density HPC clusters are commonly managed by Slurm, an open source cluster workload manager. On the November 2013 top500 list (Top 500, 2013), Slurm was used by 5 of the top 10 largest systems, including the number one system. Just in those top 10 systems, Slurm manages over 5.7 million cores (Slurm Workload Manager, 2013). Slurm is responsible for resource allocation (usually compute nodes) to multiple users. Slurm provides a framework for users to submit jobs and monitor progress of programs running on computed nodes allocated to a user's active jobs. Large HPC machines experience more job requests for resources than there are resources available for assignment. Slurm manages resource request conflicts by creating queues of pending jobs. When one user's job finishes and returns its compute resources to the unscheduled resource pool, Slurm reallocates those resources to jobs in the pending job queue and starts new jobs on the cluster that now has sufficient resources to execute. The queue is ordered by job priority which considers many factors such as job age, job size, and partition requested. If the resources for a top priority job are unavailable, Slurm uses a technique called backfill (Jaspal, 1996) to calculate how many smaller, lower priority jobs can start and finish until resources are available for the larger, top priority job. Using backfill, resources are maximally utilized while a large job waits to launch.

Slurm configuration information in the `slurm.conf` file defines compute nodes, partitions, enabled plugins, prolog scrips, and epilog scripts. For large HPC clusters, node and partition configurations can be separated into their own configuration files. Node and partition configuration files contain details such as name, time limit of jobs, processors on each node, amount of real memory, number of sockets, and cores per socket. Slurm prolog and epilog scripts allow the cluster administrator to perform setup and teardown tasks before and after the execution of the user job. Epilog scripts are particularly useful for cleaning up temporary files once a compute job terminates.

2.3 Unix File Systems

Unix style operating systems provide a global visible directory to running programs for temporary file storage. The global temporary directory, `/tmp`, is systematically cleaned up by the operating system. The OS typically purges files at boot or after files have reached a configured age. These temporary space cleanup strategies don't match the life cycle characteristics of HPC jobs. Many HPC jobs calculate results by processing input data through a pipeline of parallel programs. While traversing a pipeline of different software packages, HPC jobs store intermediate results in temporary files. Some HPC jobs can run for weeks or months and must maintain access to their temporary files for the duration of the job. However, as soon as a job terminates, its temporary storage may be cleaned up immediately. Due to the job parallelism on HPC nodes, temporary file storage is a scarce resource that needs to be managed. However, the overlap of multiple jobs owned by the same user, running on the same compute node obfuscates which files are safely deleted and which files are still in use.

A file system defines how an operating system views, names, and places files logically on a disk or partition for storage (Wirzenius, 1993). Different file system formats are used across many operating systems. Each file system differs in their file naming conventions, maximum file size, filename length, and characters available for filenames. Common file systems include FAT, NFTS, EXT4, HFS+. Windows operating systems typically use NTFS, while Linux OSs use a variety of file systems. File systems of different types can be mounted within each other. For example, a Linux OS can mount an ISO CD file system under a directory present in the EXT4 root file system. The ISO file system on a CD-ROM device is mounted at the `/mnt/cdrom` directory within the EXT4 root file system as follows: `mount /dev/cdrom /mnt/cdrom`. In this case, a CD-ROM (file system type iso9660) is mounted to the `/mnt` file system allowing the OS to read files located on the CD. A typical Linux installation consists of many file systems mounted inside a global root file system.

2.4 Bind Mounts

The Linux file system follows the File system Hierarchy Standard (FHS). All Linux file systems, both physical and virtual, are mounted within the root (/) hierarchical directory tree. Linux mounts are further enriched by bind mounts, which allow a file system to be bound to different directory paths within the same hierarchical root directory tree (Kerrisk, MOUNT(2) Linux Programmer's Manual, 2018). For example, a user's home directory `/home/userA` can be bind mounted to also appear at file path `/Users/userA`. Because of bind mounting, the `/home/userA` and `/Users/userA` directory paths map to the same physical directory. File changes in any of the locations where the file path is mounted are visible to all other bind mounts providing the same files. By default, bind mounts don't allow path traversal through external file systems mounted within the source of the bind mount. If a CD-ROM device is mounted within `/home/user/my_cd` then the directory `my_cd` will contain directories on the CD-ROM. However, the `/Users/userA/my_cd` path will appear as empty because external file systems mounts are not preserved within a bind mount. An additional bind mount for each external file system is needed to map the `my_cd` directory into the `/Users/userA` directory. Different types of bind mount options can be specified when the mount is created. The mount can be marked as shared, private, slave, or bindable. A shared mount makes it possible to create mirrors of that mount allowing mounts and umounts within one mirror to propagate to all other mirrors. A slave mount specifies one-directional propagation of changes from master to slave. A private mount has no ability to propagate changes. Unbindable mounts cannot be cloned using the bind operation. These types can also be changed after the mount has been created and can be set up to change the mount type recursively under a given mount point. The default mount type is private. Bind mounts are beneficial in many applications, allowing the reorganization of file systems to meet user needs. Pluggable Authentication Modules (PAM) is an example of one management system that system administrators can use to configure bind mounts to meet user needs.

2.5 Pluggable Authentication Modules

PAM (Geisshirt, 2007) was introduced in 1995 to help system administrators manage user authentication and setup of user environments. PAM modules are used to provide the following services: authentication, account management, session management, and password management. Prior to PAM, system administrators had to manage many user databases associated with various applications. Application developers can use PAM for authentication in applications instead of having to create custom authentication schemes. This is particularly useful because of compatibility between PAM and many other UNIX style operating systems. PAM enables applications to authenticate by means of a single API that interfaces with a variety of backend authentication services. PAM can be configured both globally, and on a per service basis. Global PAM configuration will be used by default unless a service specific PAM configuration file exists.

2.6 Linux Namespaces

Linux namespaces were created to provide a layer of abstraction between global system resources and the processes that use them. It allows a process within a namespace to have its own isolated instance of a system resource. Linux namespaces are not new; the implementation of the first namespace, the mount namespace, was in 2002 (Kerrisk, Namespaces in operation, part1:

namespaces overview 2013). Since that time, available Linux namespaces have expanded to seven (see Table 2-1) (Kerrisk, NAMESPACES(7) Linux Programmer's Manual 2018). Note that the mount namespace has a generic constant of `CLONE_NEWNS` because as the first namespace to be implemented, no one had considered that there would be other types of namespaces in the future.

Table 2-1: Current Namespaces

Namespace	**Constant**	**Isolates**
Cgroup	`CLONE_NEWCGROUP`	Cgroup root directory
IPC	`CLONE_NEWIPC`	System V IPC, POSIX message queues
Network	`CLONE_NEWNET`	Network devices, stacks, ports, etc.
Mount	`CLONE_NEWNS`	Mount points
PID	`CONE_NEWPID`	Process IDs
User	`CLONE_NEWUSER`	User and group IDs
UTS	`CLONE_NEWUTS`	Hostname and NIS domain name

Before the implementation of the mount namespace, the only alternative was to use a `chroot()` system call. Just as chroot allows any directory to be seen as the root of the file system, the mount namespace provides a more flexible and secure tool to achieve a similar task. Other types of namespaces can be used to independently modify other system resources (Ridwan, 2014).

2.7 Overhead of OS-level Virtualization

Virtualization in HPC environments was avoided due to overhead until the rise of container-based virtualization. In 2013, a study was done to compare HPC performance of three different container-based virtualization products (Xavier, 2013). The study hypothesized that container-based virtualization used in HPC environments can allow better resource sharing and creation of custom environments while having little impact on the overall performance of the HPC cluster. They experimented in both single and multi-node environments. Performance was measured using the NAS Parallel Benchmarks (NPB), a well-known HPC benchmark. Security was measured using the Isolation Benchmark Suite (IBS). Using these tools, the overall performance of each virtualization solution was split into the following categories: computing performance, memory performance, disk performance, network performance, performance overhead, and isolation. It was found that all three container-based virtualization solutions showed near native performance of CPU, memory, disk, and network resources.

Although this study doesn't specifically address namespaces, it does prove that container-based virtualization solutions that use namespaces have little effect on the overall performance of the HPC environment. The authors concluded that container-based virtualization solutions were subpar when it came to the isolations of the systems, however my research will be using namespaces specifically to achieve this isolation, rather than relying on container-based solutions which have a far broader objective.

2.8 A Historical Perspective of Linux Namespaces

An article written in 2006 provides a unique perspective for why namespaces were needed (Biederman, 2006). The author discusses the problem in HPC of moving jobs between nodes. Because global identifiers are not unique between machines, it becomes difficult to move jobs without errors. One solution that had been commonly used was isolation via chroot jails. The problem is, general purpose implementations of jails had not been merged into the mainline

Linux kernel. Namespaces are the proposed implementation of jails that allow isolating specific system resources. To successfully implement namespaces, about 7% of the mainline Linux kernel would need to be touched. The article then goes over ten proposed namespaces, many of which have now been implemented.

When this article was written, mount namespaces were the only namespaces merged into the mainline Linux kernel. However, at the time, there were still several issues with implementing a mount namespace, one of which was the restricted ability to create bind mounts. The discussion on PID namespaces was fascinating as it allowed a view into some of the issues that had to be overcome to implement PID namespaces into the Linux kernel. From within a namespace it is a simple concept, however outside the namespace problems become more complex. For example, how should a process outside of the default namespace be displayed in `/proc`. The hierarchical nature of process organization was also an issue. This made having parent/child relationships difficult when a process has a separate PID within its namespace. The proposed solution was a `struct` containing the PID namespace, and a PID pointer.

2.9 OS-level Virtualization Security

An analysis of the security of OS-level virtualization technologies examined the isolation of containers between other containers and the host (Reshetova, 2014). It was determined that to have a secure OS-level virtualization container, a set of security requirements are needed: separation of processes, file system isolation, device isolation, IPC isolation, network isolation, and resource management. Container compromise, denial of service, and privilege escalation are prevented when these requirements are met. Each of these requirements were described in depth, however because separation of processes and file system isolation are most relevant to this thesis, I will refrain from reviewing the other requirements.

Separation of processes is necessary to distinguish processes running in the containers from those running on the host. Because PID namespaces are part of the mainline Linux kernel, it is ideal for accomplishing the separation of processes. PID namespaces provide a way to control the ability of processes to see and interact with one another. Additionally, PID namespaces provide an element of virtualization, allowing processes in different PID namespaces to have the same PID. In addition to the PID namespace, the authors also discuss the usefulness of user namespaces in this section. User namespaces are also part of the mainline Linux kernel. It helps prevent privilege escalation by allowing root users elevated privileges only in that namespace and not on the host.

File system isolation is also achieved using a mainline Linux kernel namespace, mount. Mount namespaces by themselves are not a security measure, as mount namespaces inherit the view of file system mounts from their parent and thus have access to all parts of the file system. Two mainline Linux kernel commands are often used to solve this problem: chroot, and pivot_root. Both commands essentially change the location of root within each namespace.

2.10 Related Work

Search for previous related work uncovered a GitHub code repository where an implementation provided temporary file storage segmented by job id. This implementation lacked the improved security of isolation by user account as well as job id that our solution provides (Chrissamuel, 2016) (Fossing, 2015). Additionally, our plugin has added functionality to isolate multiple temporary file directories such as `/dev/shm`. The task plugin mount_isolation will not only isolate temporary files, but automatically remove them as the job terminates, reducing administrator involvement. While we chose to implement our Slurm plugin using the Slurm Task plugin architecture, we could just as easily have implemented our improvements to temporary space isolation using the Slurm Plug-in architecture for Node and job (K)control (SPANK) instead (SPANK - Slurm Plug-in Architecture for Node and job (K)control, 2018).

3 METHODOLOGY

3.1 Isolation Components

Using Linux kernel namepaces and PAM modules, temporary files were separated into bind mounted directories for each user. System administrators used a PAM module, pam_namespace.so, to setup a user environment at login specific to that user. The pam_namespace.so module created a mount namespace for the user, allowing a specific user's data to be grouped together into that hierarchy (Hallyn, 2007). A Slurm job epilog script deleted files -- on a per compute node basis -- only when all the user jobs on a compute node had terminated. Each user's supercomputing job can generate gigabytes of temporary data in the limited temporary storage space of a compute node. Compute nodes with 24-32 cores commonly execute many different users' jobs concurrently. Running jobs concurrently on a compute node maximized utilization of available compute resources. However, running multiple jobs on a compute node places pressure on scarce shared temporary storage resources of the node.

Our implementation takes the form of a new Slurm task plugin, mount_isolation, that isolates temporary file space on a per job and a per user basis in a root user visible directory `/tmp/<user>/<job_id>`. The `/tmp/<user>/<job_id>` directory is presented to the compute job session as `/tmp` and to user login sessions as `/tmp/<job_id>`. With temporary files isolated on a per job basis, obsolete temporary files can be purged via a Slurm epilog script as each job completes, immediately freeing shared temporary file space for other jobs.

Our implementation isolates temporary files by creating mount namespaces for individual jobs with bind mounted job id directories in addition to existing segregation of jobs by username. Upon creation of a new job, a new mount namespace is created in which a job specific temporary storage directory, `/tmp/<user>/<job_id>`, is mounted into the `/tmp` directory. Consider the example in the introduction where a user, John, has two jobs executing on compute node **m81**. Upon creation of job 1, a new mount namespace is created and the directory structure `/tmp/john/job1_id` is built. As job 1 runs, its 15GB of temporary files are stored in the `/tmp/john/job1_id` directory. Similarly, when John launches job 2, a second mount namespace is created as and the directory structure `/tmp/john/job2_id` is built. Job 1 terminates and job 2 continues running for an additional 7 days. Instead of keeping the first job's 15GB of temporary files, Slurm can now delete all temporary files belonging to job 1 by removing the `/tmp/john/job1_id` directory without impacting temporary files belonging to job 2 `(/tmp/john/job2_id)`. This distinction between job temporary files is possible because they are separated into individual job directories. John can now launch additional jobs, creating new mount namespaces and directory structures `/tmp/<user>/<job_id>` for each job (see Figure 3-1). As jobs complete, individual temporary job files will be deleted, freeing temporary file system space for other jobs. In addition, when a user logs onto a compute node, a new mount namespace is created, and bind mounted as `/tmp/<user>`. This new bind mount will allow users to see their isolated job files within their own namespace as `/tmp/<job_id>` directories, while the root user will see them in `/tmp/<user>/<job_id>`. Per job mount namespaces will allow removing obsolete temporary files when the job completes, rather than after all of John's jobs have terminated on that compute node.

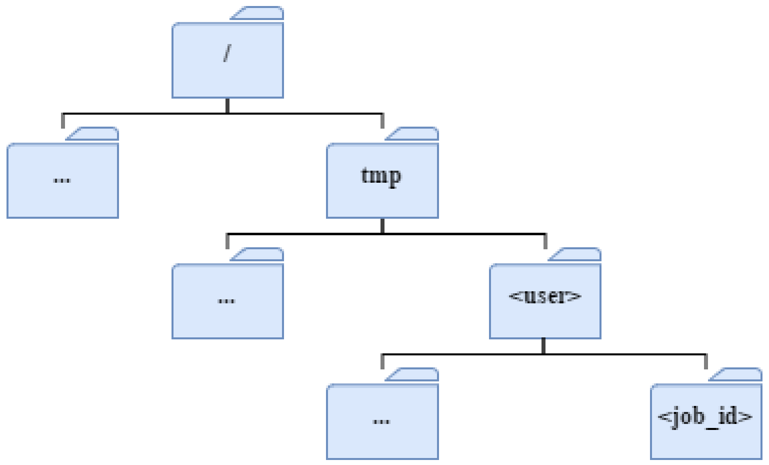

Figure 3-1: Per Job Directory Structure

The new implementation breaks down into three parts. First, because pam_namespacing.so already isolates files based on user, that part of the current implementation remains in place. Second, to further isolate files by job id, a new Slurm task plugin, based on the task none plugin, creates each job mount namespace and job id directory structure. The final part of implementation alters the current Slurm epilog script to delete temporary files by job id directory rather than by user directory.

3.2 PAM Namespace Module

The pam_namespace PAM module sets up a private session namespace for each user on login. As part of this process, the module also sets up poly-instantiated directories based on username. Because implementation of the mount_isolation Slurm plugin, described below, uses the same username as part of its structure, the pam_namespace module can be used to create the username bind mounts. With the job_id directory structure set up within the username directory, the logged in user will be able to see all their running jobs in their `/tmp` directory.

3.3 Slurm Task Plugin

To implement the isolation of the job files, a new Slurm plugin must be created. Several plugin options were available, but those options were narrowed to a Slurm task plugin, or a SPANK plugin. A task plugin would be part of Slurm, while a SPANK plugin can be built without access to the Slurm source code. SPANK plugin code can be run at nearly any point in the Slurm process, while the task plugin is specific to when tasks are run (SPANK - Slurm Plug-in Architecture for Node and job (K)control, 2018). Because a task plugin was specifically called as tasks are run, we decided to place our implementation in a task plugin. Additionally, Ryan Cox of the Fulton Supercomputing Center, one of our partners on the project, suggested using a task plugin.

The new task plugin, mount_isolation, is enabled in `slurm.conf` as follows: `TaskPlugin=task/mount_isolation`. Task plugins in Slurm have predefined functions that run at various stages of the job life cycle. The new isolation implementation will be called from the `task_p_pre_launch_priv` function, which is run with root privileges before the job is launched. Root privileges allow the new isolation plugin to build the necessary directory structure in /tmp and alter permissions of those directories. The central portion of our new implementation occurs inside a function we created, called isolate. User id, group id, and job id are passed into isolate as parameters as follows: `isolate(job->uid, job->gid, job->jobid);`.

The Slurm job structure is provided as an argument to `task_p_pre_launch_priv` function. Initialization of helper variables for the isolate function appear at the start of the function. These variables are used to create the needed directory structure and alter the permissions of that structure. Using the uid passed in from `task_p_pre_launch_priv`, a passwd struct containing user account information is retrieved from the system. This `passwd` structure, `pw`, is used to retrieve the username of the user that launched the job (see Figure 3-2). This username will be used later to create the `/tmp/<user>` directory.

```
struct passwd* pw = getpwuid(uid);
char* uname = pw->pw_name;
```

Figure 3-2: Get the User

The job id is converted to a string and added to the job path (see Figure 3-3). This will be used later to create the `/tmp/<user>/<job_id>` directory.

```
char job_id_str[22];
sprintf(job_id_str, "%d", job_id);
char* tmp_path = "/tmp/";
char* job_path = malloc(strlen(tmp_path) +
     strlen(uname) + strlen(job_id_str) + 2);
```

Figure 3-3: Directory Preparation

The isolate function will then begin building the needed directory structure by first checking to see if `/tmp/<user>` exists. If `/tmp/<user>` does not exist, it is created. To create the directory, the path needs to be constructed first (see Figure 3-4).

```
strcpy(job_path, tmp_path);
strcat(job_path, uname);
mkdir(job_path, 0700);
```

Figure 3-4: User Directory Permissions

The newly created directory is then altered to have correct user and group permissions. These permissions are set based on the uid and gid (user and group id) passed into the isolate function from `task_p_pre_launch_priv`. A similar process is used to create the `/tmp/<user>/<job_id>` directory. The job path is altered to include the path to the job_id as well as the username. The directory is created, and permissions are set using the uid and gid passed into the isolate function from `task_p_pre_launch_priv` (see Figure 3-5).

```
strcat(job_path, "/");
strcat(job_path, job_id_str);
mkdir(job_path, 0700);
lchown(job_path, uid, gid);
```

Figure 3-5: Job Directory Permissions

Now that the directory structure is in place, mount work needs to be done before our new mount namespace can be implemented. By default, all mounts are private. The plugin alters the existing root mount to make it a shared (`MS_SHARED`) mount rather than a private mount as follows: `mount("", "/", NULL, MS_REC|MS_SHARED, NULL);`. This change is done recursively (`MS_REC`) so that any mounts within the root file system are also shared (Kerrisk, MOUNT(2) Linux Programmer's Manual, 2018).

Now when the new mount namespace is created, any mounts or umounts within the root file system will propagate to the new namespace. Next, the new mount namespace is created via `unshare(CLONE_NEWNS)`, allowing the job process to have a private copy of its namespace that is not shared with any other process (Kerrisk, UNSHARE(2) Linux Programmer's Manual, 2018). Note that the CLONE_NEWNS flag specifies the new namespace will be a mount namespace. The job process will no longer share its root directory, current directory, or umask with any other process.

Before making the final mount of the job_id, the root mount needs to be altered to ensure any new mounts made in the newly created namespace don't propagate back to the original root mount. This is done by marking the mount as a slave mount as follows: `mount("", "/", NULL, MS_REC|MS_SLAVE, NULL);`. A slave mount will receive any added mounts or umounts from the original namespace, but not vice versa.

Finally, the `<job_id>` directory is bind mounted to the user's `/tmp` directory. All files saved by the job in `/tmp` will be visible from the job mount namespace in `/tmp` and from the root mount namespace in `/tmp/<user>/<job_id>`. These isolated temporary files will be visible in the job owner's namespace as `/tmp/<job_id>`. Memory used for the job_path is also freed to prevent memory leaks (see Figure 3-6).

```
mount(job_path, "/tmp", NULL, MS_BIND, NULL);

free(job_path);
```

Figure 3-6: Bind Mount

Throughout the process above, the plugin checks for possible errors on each command. As specified for task plugins, errors return `SLURM_ERROR`, while success is returned as `SLURM_SUCCESS`. On a successful run through the entire isolate function, a return value of `SLURM_SUCCESS` is sent back to the `task_p_pre_launch_priv` function.

The current isolation of these files accounts for different users taking advantage of supercomputing capabilities. The new solution is to isolate temporary files per user and per job so that they can be deleted immediately following termination of the job. Isolating temporary files will allow removal of files when the job completes, rather than when all the user's jobs on a compute node complete. System administrators can remove temporary files after a job completes using a Slurm task epilog script. This isolation also adds a layer of security for each user; private per job namespaces prevent other users from accessing the files.

3.4 Slurm Epilog Script

The epilog script is final part of my implementation. The current method of removing temporary files is to use the epilog script to check for user jobs still running. Since the new task plugin isolated files on a per job basis, this check no longer needs to take place. Now the epilog script can simply delete the job_id directory as soon as the job completes (see Figure 3-7). The remove statement in the script accounts for the multiple temporary directories used by the Fulton Supercomputing center (/dev/shm and /tmp). The script then removes the job id directory created by the mount_isolation plugin. Slurm specific variables are used in the script to accomplish this, namely: SLURM_JOB_USER, SLURM_JOB_ID, and SLURM_JOB_UID.

```
rm -rf {/dev/shm,/tmp}/userns/"$SLURM_JOB_USER"/$SLURM_JOB_ID
pkill -9 -u "$SLURM_JOB_UID"
```

Figure 3-7: Remove Temporary Job Files

4 IMPLEMENTATION

The process of implementing the mount_isolation plugin was an iterative process involving the director of BYU's Fulton Supercomputing Center, Ryan Cox. Through each iteration, the methodology was changed to meet the needs of the BYU Fulton Supercomputing Center, more fully reflect the coding style of Slurm, and add functionality. This section will explain and outline the changes that were made.

Many of these changes were necessary because of some invalid assumptions or failure to get full instructions of what was expected of the plugin.

Invalid Assumption 1 (IA1): The only function of the mount_isolation plugin is to isolate temporary directories on a per job basis. Setup of the user directory, and removal of temporary files will be handled by the administrator. This invalid assumption led to changes described in sections 4.1 and 4.5.

IA2: The only temporary directory utilized by the BYU Fulton Supercomputing Center is the default `/tmp` directory. This invalid assumption led to changes described in section 4.2.

IA3: Supercomputing users see the isolated job id directory in their temporary directories. This invalid assumption led to changes described in section 4.6.

IA4: The mount_isolation plugin will work on the live supercomputing system in the same way it functions in my development environment. This assumption led to a plethora of small changes to the mount_isolation plugin.

Other changes described in this section weren't due to invalid assumptions, but due to personal preference of the BYU Fulton Supercomputing Center. These changes, while specific to BYU, helped make this solution more beneficial to HPC system administrators.

4.1 PAM, mount_isolation, Epilog Conglomerate

To reduce work required by HPC administrators, the original three-part plan described in the methodology section was reduced to a single plugin: mount_isolation. Because not all systems use the pam_namespace.so PAM plugin, the methodology was changed to allow the mount_isolation task plugin to be useful to the widest possible range of users. The plugin no longer relies on the pam_namespace.so PAM plugin. Additionally, the plugin now deletes files from within the plugin instead of waiting for the epilog script to run at the termination of the job. The source code for the mount_isolation plugin is included in Appendix B.

4.2 Multiple Temporary Directories

Some HPC jobs use multiple temporary directories beyond the default `/tmp` directory. To allow the mount_isolation plugin to work with these directories, the Slurm source code was changed to add a configuration variable to the slurm.conf file. This configuration variable allows Slurm administrators to create a comma separated list of which temporary directories they want isolated by job in the slurm.conf file as follows: `TaskPluginTmpDirs=/tmp,/dev/shm`.

If this configuration variable is not set, it defaults to /tmp. The mount_isolation plugin retrieves this configuration, and loops through the comma separated list of temporary directories to create the necessary directories to implement job isolation (see Figure 4-1).

```
char tmp_dirs[PATH_MAX];
snprintf(tmp_dirs, PATH_MAX, "%s",
     slurmctld_conf.task_plugin_tmp_dirs);

/* prepare to loop through tmp directories */
char* tmp_dir;
char* saveptr;
tmp_dir = strtok_r(tmp_dirs, ",", &saveptr);

/* look through tmp directories */
while (tmp_dir) {
     /* prepare for next loop */
     tmp_dir = strtok_r(NULL, ",", &saveptr);
     while (tmp_dir && *tmp_dir == '\040') {
          tmp_dir++;
     }
}
```

Figure 4-1: Read Slurm.conf for Multiple Temporary Directories

4.3 Reentrant Functions

It is important to note that when available, reentrant versions of common functions are used. Reentrant functions end with "_r" and are used to prevent errors when a job runs on more than one node. A reentrant function has the following characteristics (Ganssle, 2004):

1. It uses all shared variables in an atomic way, unless each is allocated to be a specific instance of the function.
2. It does not call non-reentrant functions.
3. It does not use the hardware in a non-atomic way.

Reentrant functions ensure that even if there is an interrupt, the code will execute correctly when returning from the interrupt. This is important in HPC environments that use parallel processing.

4.4 Temporary Subdirectory

Another change that was made to my methodology is the addition of another directory inside the temporary directory. In the original plan outlined in the methodology section the plugin created a directory structure visible to root as `/tmp/username/job_id`, but this change allows administrators to create an additional subdirectory through a slurm.conf variable as follows: `TaskPluginTmpSubdir=mountns`. This creates a directory structure visible to root as `/tmp/mountns/<user>/<job_id>` (see Figure 4-2). The new temporary subdirectory is given file permissions of `000`. Creating this directory allows the possibility of future work to implement PID namespaces in addition to the mount namespace implemented in mount_isolation.

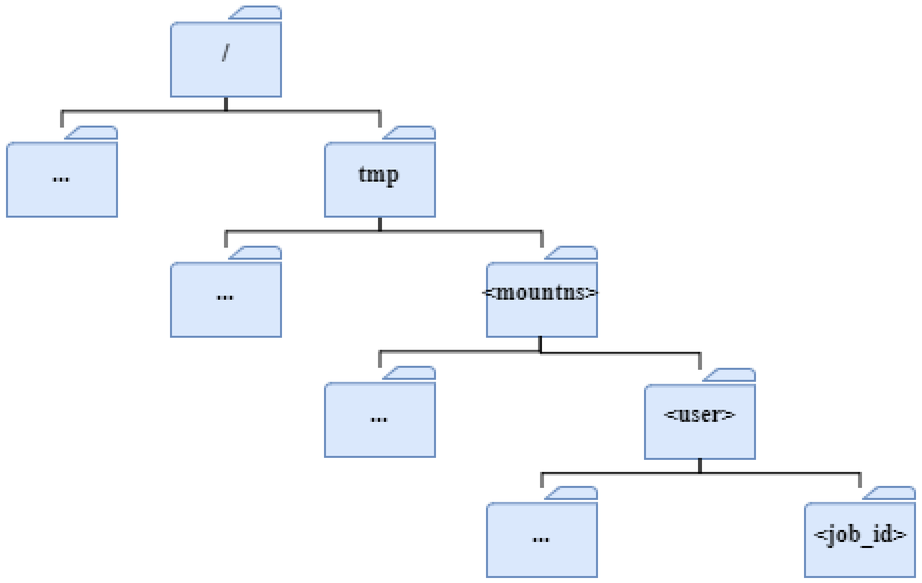

Figure 4-2: Directory Structure with Additional Subdirectory

4.5 Recursive Removal of Job Directory

The greatest benefit of the recursive remove being held within the mount_isolation plugin is the reduction the amount of work for administrators to implement this solution. This recursive remove is also the most dangerous part of code in the plugin as it has the potential to remove files that are not meant to be removed. Several safeguards were built into the code to help mitigate this risk.

First, the recursive remove code was separated into its own function that is not executed until the last step as each job terminates. Second, the Linux function `lstat` is used to determine if the function should recurse or not. `lstat` differs from its brother `stat` in that it doesn't follow symbolic links (Kerrisk, STAT(2) Linux Programmer's Manual, 2017). This will prevent the recursive remove from following symbolic links that may point outside the directory structure we want to remove. Third, the recursive remove function checks the device_id of the directory structure that is being removed. As the function recurses, it continues to check the device_id to make sure it hasn't jumped onto a new mount location.

4.6 PAM Session Module

Impact on users was the contributing factor to this addition. The BYU Fulton Supercomputing director wanted this implemented so that the users wouldn't notice any changes. Without the addition of the pam_mount_ns_adopt.so module, users would view their temporary directory as containing a job_id directory. Because users often don't know their job_id, this can become confusing, especially if a user is running multiple jobs.

The pam_mount_ns_adopt module is a PAM module that adopts the user's session into the same mount namespace as the job they are running. This means that they will see the same directory structure as the job running on that node. The module is dependent on cgroups, and the pam_slurm_adopt module. When a uses ssh to connect to a supercomputing node, the pam_mount_ns_adopt module gathers the needed information about both the new session, and the jobs the user is running. Using the Linux setns function, the user is adopted into the same mount namespace as their job (see Figure 4-3). The source code for the pam_mount_ns_adopt PAM module is included in Appendix C. The helper.h and helper.c files included code allowing the module to connect to Slurm.

```
syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO),
      "%s: adopting user into mnt namespace", PAM_MODULE_NAME);
if (setns(fd2, 0) == -1) {
  syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO),
      "%s: setns failed to adopt user into jobid mnt ns",
      PAM_MODULE_NAME);
  goto cleanup;
}
```

Figure 4-3: Adopt User into Mount Namespace

4.7 Unit and Integration Testing

Before implementing the mount_isolation plugin at BYU's Fulton Supercomputing Center, we wanted to be thorough in our testing to prevent any major issues. The section of code we were most worried about was the recursive remove function. As you can imagine, a recursive remove that gets out of the desired directory location can be potentially catastrophic. To fully test the plugin, a series of test cases covering all possible job launches was required (see Table 4-1). Each option needed to be run in combination with every job launch option. As each test was run, the following were evaluated:

1. Was the namespace created correctly? This was tested by ensuring the files were written to the correct location (`/tmp/<mountns>/<user>/<job id>`).
2. When the job terminates, is the job id directory successfully removed?

Table 4-1: Job Testing Combinations

Job Launch	**Options**
sbatch	scancel
salloc	srun
srun	multi-node
	symbolic link

4.8 Slurm Repository Contribution

The end goal of the mount_isolation plugin is to merge the plugin with the Slurm public repository. To accomplish this, all code needs to be formatted according to Slurm coding standards. Some of the changes required to meet these standards were to write all comments in the following format: `/* [comment] */`. Code is also limited to 80 characters across. Other coding standards are defined specifying best practices for indentation, braces, naming conventions, etc. Coding style is based on the Linux Kernel coding style, and can be found on the Slurm website (Torvalds, 2016).

4.9 Gathering Data

Because the Fulton Supercomputing Center is a production environment, and there are always other urgent updates, I was unable to get the actual plugin on their entire system to gather data. Because the current epilog script they use runs after every job, I was able to gather data about each job as it completed. The current epilog script only removes temporary files if that user has no more overlapping jobs (see Figure 4-4). To gather data about all jobs, not just jobs when files are removed, I added two data gathering points in the script. Because the script only removes files when the user has no more active jobs, all overlapping jobs before it will not have accurate data measurements, however the data can still be used to gain visibility into how often a user has overlapping jobs. Data gathered included the following:

- Date the job was run
- Anonymized user ID
- Cluster name
- Node name
- Number of files removed
- Amount of data removed
- Does the user have more active jobs

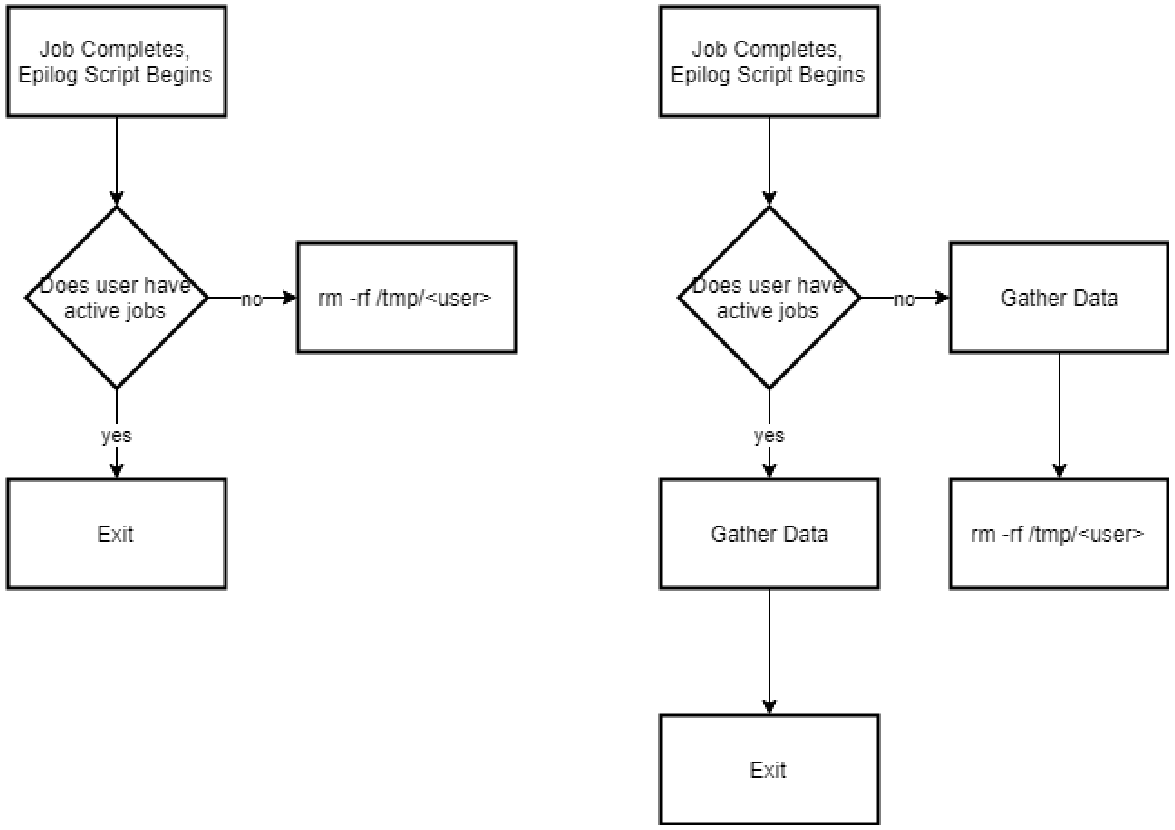
Job Completes, Epilog Script Begins
Does user have active jobs
no
rm -rf /tmp/<user>
yes
Exit
Job Completes, Epilog Script Begins
Does user have active jobs
no
Gather Data
yes
Gather Data
rm -rf /tmp/<user>
Exit

Figure 4-4: Epilog Flow Chart

5 IMPLEMENTATION DATA

5.1 Development Environment Data

Live production environment data varies greatly based on what jobs users submit. For generated data that takes many of these variables into consideration, see the wasted temporary space simulator (WTSS) in section 5.4. The following simulated data was gathered in a controlled development environment as a proof of concept for the mount_isolation plugin. The simulation includes a series of five overlapping, identical jobs that always generate a set number and size of temporary files. These five jobs were run 1 minute apart for 5 minutes each. The test was repeated twice, once using the epilog script to remove files, and once using the mount_isolation plugin to remove files. The node running these jobs was given 4 processors on which to run them. Slurm `squeue` output is used to verify which jobs are running (R), and which jobs are pending (PD) due to lack of resources to run the job (see Tables 5-1 and 5-2).

Table 5-1: Overlapping Jobs (epilog)

JobID	Partition	Name	User	State	Time	Nodes	NodeList (Reason)
154	debug	job.sh	tannersatch	PD	0:00	1	(Resources)
150	debug	job.sh	tannersatch	R	4:15	1	lx0
151	debug	job.sh	tannersatch	R	3:15	1	lx0
152	debug	job.sh	tannersatch	R	2:15	1	lx0
153	debug	job.sh	tannersatch	R	1:15	1	lx0

Table 5-2: Overlapping Jobs (mount_isolation plugin)

JobID	Partition	Name	User	State	Time	Nodes	NodeList (Reason)
159	debug	job.sh	tannersatch	PD	0:00	1	(Resources)
155	debug	job.sh	tannersatch	R	4:15	1	lx0
156	debug	job.sh	tannersatch	R	3:15	1	lx0
157	debug	job.sh	tannersatch	R	2:15	1	lx0
158	debug	job.sh	tannersatch	R	1:15	1	lx0

The two test instances consist of identical jobs and generated a predetermined amount of data, in this case 24MB. The jobs were also written to run for a predetermined amount of time using sleep 300, which allowed the jobs to run for approximately five minutes (see Figure 5-1). In addition to the jobs performing the same function, they were also launched by Slurm using the same configurations. They were allocated 10 minutes on the node, given 1 processor, 1K of memory, and they were configured to allow oversubscription, which just means that more than one job can be run on a node at a time if there are enough processors to handle increased load.

```
#!/bin/bash
#SBATCH --time 0-00:10:00
#SBATCH --ntasks=1
#SBATCH --mem=1K
#SBATCH --oversubscribe

dd if=/dev/zero of=/tmp/"$SLURM_JOB_ID"_tmp.dat bs=24M count=1
sleep 300

exit 0
```

Figure 5-1: Job Script

In both test instances, the jobs sent output to a log file at the end of each job to show how much data remained in /tmp. The epilog script set of jobs used the epilog script to write to the logs, while the plugin set used the mount_isolation plugin to write to the logs. Based on the information gathered form the logs, the data was compiled into a single table showing a comparison of the two solutions (see Table 5-3). For the sake of comparison, all job ids from the plugin set of jobs (jobs 155-159) have been changed to be the same as the epilog set of jobs (jobs 150-154).

Table 5-3: Epilog vs Plugin Comparison Over Time

Time	**JobID(s)**	**Status**	**Time**	**Epilog /tmp**	**Plugin /tmp**	**Epilog data**	**Plugin data**
0:00	150	R	0:00	150.dat	150/150.dat	24MB	24MB
1:00	150 151	R R	1:00 0:00	150.dat 151.dat	150/150.dat 151/151.dat	48MB	48MB
2:00	150 151 152	R R R	2:00 1:00 0:00	150.dat 151.dat 152.dat	150/150.dat 151/151.dat 152/152.dat	72MB	72MB
3:00	150 151 152 153	R R R R	3:00 2:00 1:00 0:00	150.dat 151.dat 152.dat 153.dat	150/150.dat 151/151.dat 152/152.dat 153/153.dat	96MB	96MB
4:00	150 151 152 153 154	R R R R PD	4:00 3:00 2:00 1:00 0:00	150.dat 151.dat 152.dat 153.dat	150/150.dat 151/151.dat 152/152.dat 153/153.dat	96MB	96MB
5:00	151 152 153 154	R R R R	4:00 3:00 2:00 0:00	150.dat 151.dat 152.dat 153.dat 154.dat	151/151.dat 152/152.dat 153/153.dat 154/154.dat	120MB	96MB

Table 5-3, Cont'd

Time	JobID(s)	Status	Time	Epilog /tmp	Plugin /tmp	Epilog data	Plugin data
6:00	152 153 154	R R R	4:00 3:00 1:00	150.dat 151.dat 152.dat 153.dat 154.dat	152/152.dat 153/153.dat 154/154.dat	120MB	72MB
7:00	153 154	R R	4:00 2:00	150.dat 151.dat 152.dat 153.dat 154.dat	153/153.dat 154/154.dat	120MB	48MB
8:00	154	R	3:00	150.dat 151.dat 152.dat 153.dat 154.dat	154/154.dat	120MB	24MB
9:00	154	R	4:00	150.dat 151.dat 152.dat 153.dat 154.dat	154/154.dat	120MB	24MB
10:00	-	-	-	-	-	-	-

5.2 Production Environment Data

The following data was gathered from the BYU Fulton Computing Center over a period of one month, from four HPC clusters. Due to some inconsistencies in gathering the data, the one-month period was narrowed to approximately two weeks (see Table 5-4). While the data gathered over the one-month period is useful, for the remainder of this chapter the two-week data set will be used to preserve any trends found in the data.

Table 5-4: Data Gathered

Time Interval	Total Jobs	Total Data Removed	Total Files Deleted
Jan. 31 – Mar. 6 (34 days)	473,513	1,020.323 GB	1,135,150
Feb. 12 – Feb. 28 (17 days)	418,297	410.232 GB	947,446

5.2.1 Saved Disk Space

This mount_isolation plugin will save scarce temporary disk space. After gathering data over a period of two weeks, we can see that the amount of data removed is somewhat significant. Because there are several different ways HPC system administrators deal with this problem, the benefits of this plugin will vary. Based on data gathered over the two-week period, the total amount of disk space saved was 410.232 GB. While this is a relatively small amount of data, the Fulton Supercomputing Center has approximately two petabytes of storage space (FSL Resources 2018), for systems that don't remove these files until the node is rebooted, the files can build up over a period of months. Over this two-week timeframe, a large quantity of that was removed in two specific 12 hour time periods mostly from the same cluster, m9 (see Figure 5-2).

5.2.2 Files Removed

While the number of files removed is less interesting than the size of those files, it is interesting to note that there is a relationship between the two. Over the same 1-month period described above, there were nearly 800,000 files removed. Oddly, the spikes in data removed in Figure 5-2 don't line up with the spikes in the number of files removed per cluster (see Figure 5-3). Although the relationship isn't apparent in these two graphs, additional analysis reveals that the number of files removed has a weak correlation with the amount of data removed (see Figure 5-4).

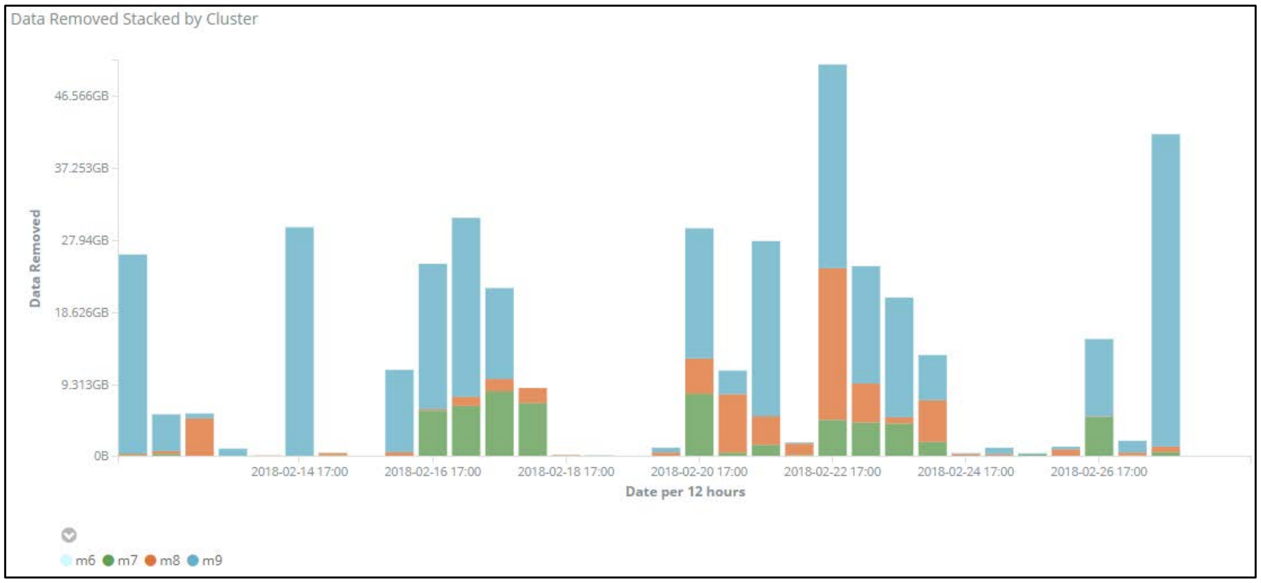

Figure 5-2: Data Removed per Cluster

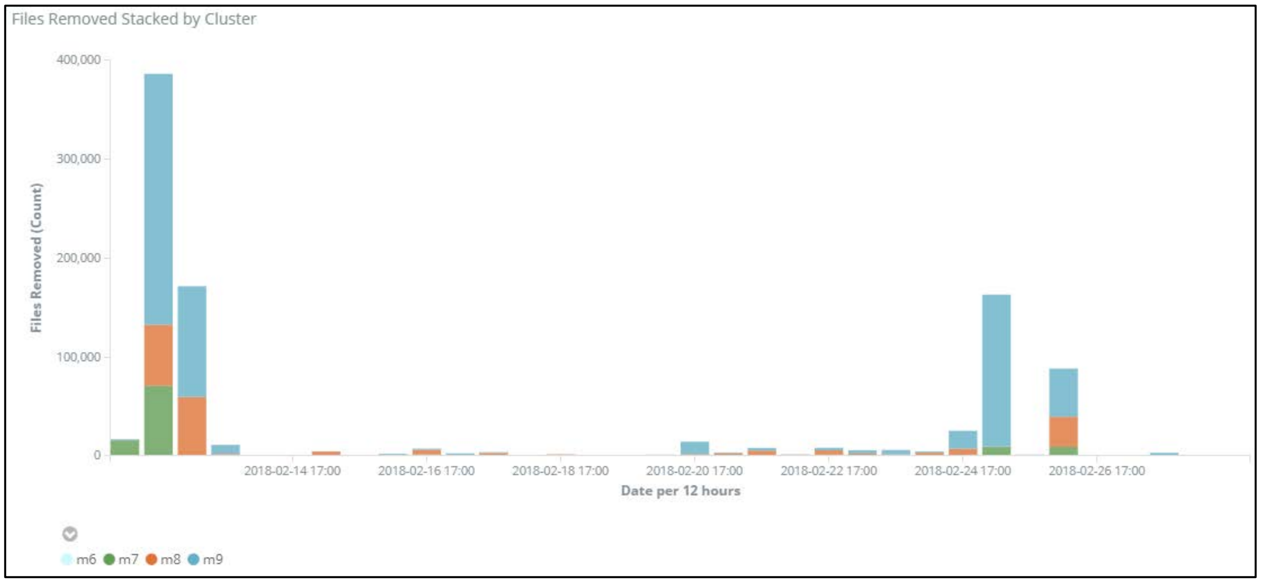

Figure 5-3: Files Removed per Cluster

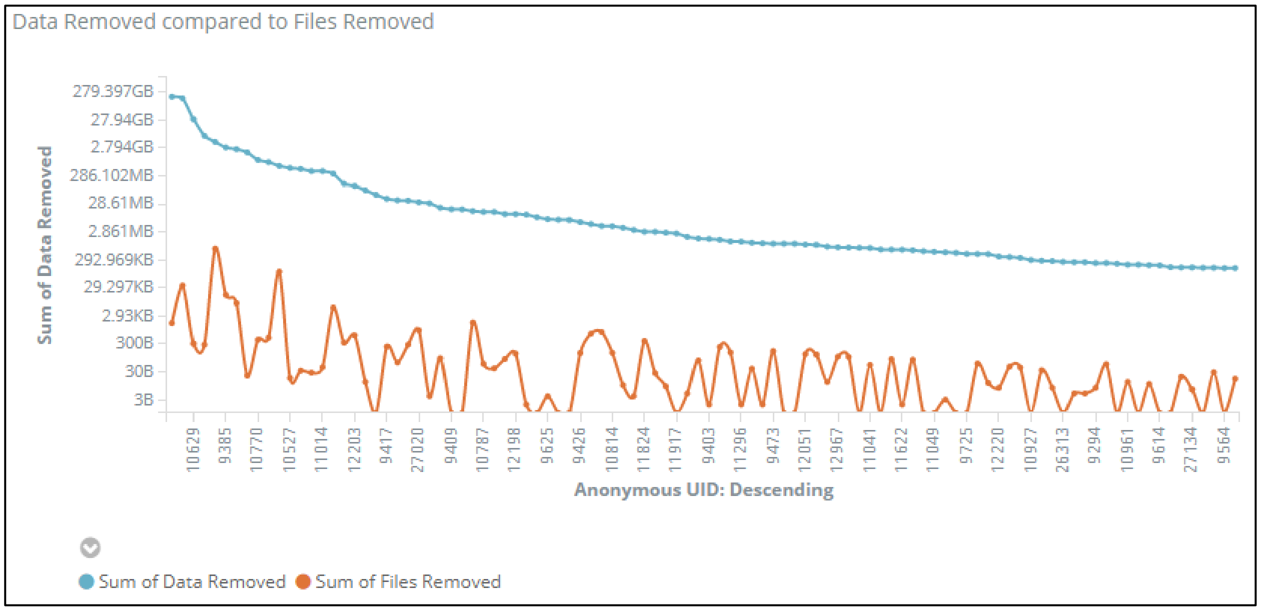

Figure 5-4: Data to Files Relationship

5.2.3 Overlapping Jobs

While the plugin solution will always offer improved security through isolation, the difference between the plugin solution, and the epilog script solution is dependent on overlapping jobs per user. The more overlapping jobs there are per user, the more beneficial the plugin solution will be to system administrators. When comparing the number of overlapping jobs, to the number of non-overlapping jobs, it is clear that, at least in this HPC environment, there is a large number of overlapping jobs (see Figure 5-5). It is interesting to note that although there is a large number of overlapping jobs, the majority of them are launched by just a few users (see Figure 5-6).

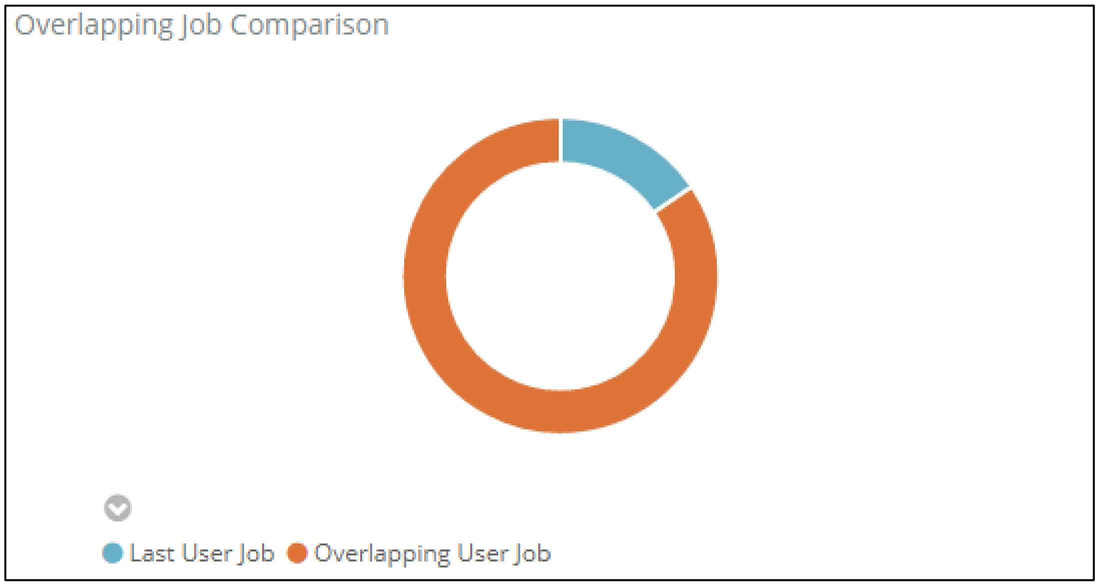

Figure 5-5: Overlapping Job Comparison

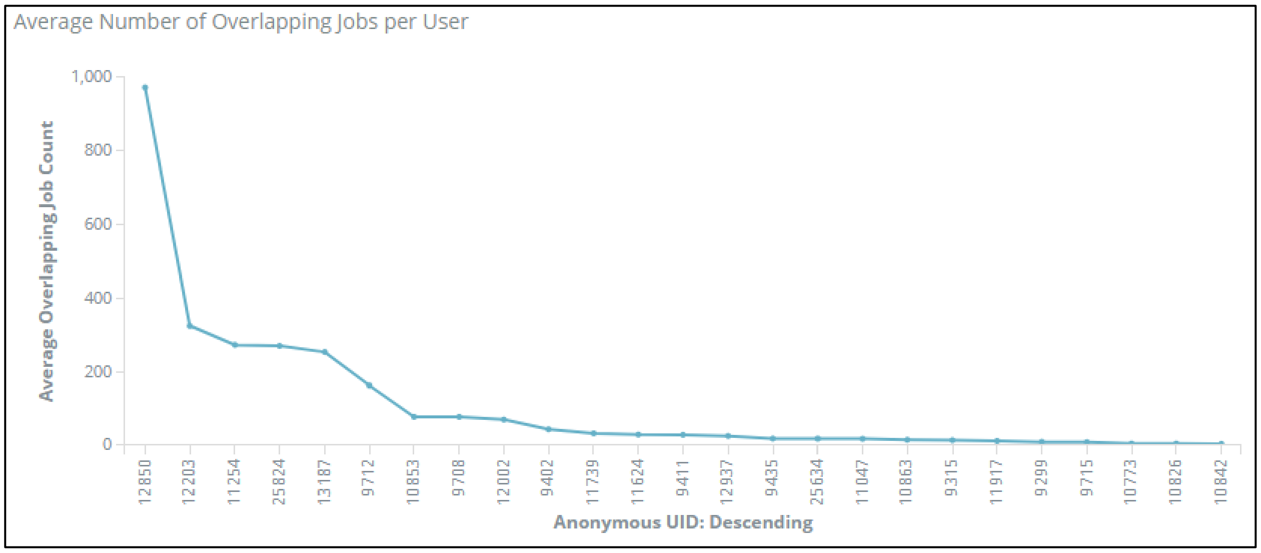

Figure 5-6: Average Overlap per User

5.3 Data Delimitations

5.3.1 Linux du

A standard Unix program, du, was used to estimate file space usage in the production environment. While the program is fairly accurate, it is important to note that the Linux man page specifically says the returned data is an estimate, not exact.

5.3.2 User Variance

The biggest variable in my data is the behavior of the users of the supercomputer. Jobs run on a supercomputer will vary greatly depending on the end goal of the users. If a user launches multiple jobs, causing them to overlap, then epilog script solution to the problem will be less efficient. If a user launches a job that doesn't require the creation of temporary files, then the plugin won't have any benefit. If the user writes their job code to remove temporary files themselves, the only benefit will be in the isolation of those files.

5.4 Theoretical Improvement

Improvements on storage utilization will vary based on the job and the user. For HPC system administrators that opt to remove files whenever they reboot the node, the improvement will be a sum of all temporary files generated between reboots. For HPC system administrators that opt to use an epilog script to remove files when all jobs by a user are complete, the improvement will be a sum of all overlapping temporary files generated by jobs after the first job by that user. This improvement will be on a per user basis. The WTSS simulates jobs run on an HPC cluster to help administrators determine if the mount_isolation plugin will be beneficial.

The WTSS is written in python. The simulator's input parameters specify the node configuration of the simulated HPC cluster, the probability distribution of users launching jobs and the statistical distribution parameters governing the characteristics of job generation. While concise, the simulator allows HPC administrators and researchers to specify custom parameters and gain insight into the security and space savings that isolation of temporary HPC storage can provide. The compute node description domain specific language (DSL) uses nested Python dictionaries to specify the quantity of nodes in the cluster and the number of processors per node. `nodes = { CLUSTER_NAME: { 'count': QUANTITY, 'processors': QUANTITY_PER_NODE } }`. Cluster configurations can be repeated to represent multiple clusters (see Figure 5-7).

```
nodes = {

      'm8' : { 'count': 320, 'processors': 24 },

      'm7' : { 'count': 230, 'processors': 16 },

      'm6' : { 'count': 256, 'processors': 12 }

}
```

Figure 5-7: Example Node Configuration

Compute jobs in the simulator consist of four parameters: the user id that launches and owns the job, the number of processors the job requires, the wall clock time length of the job, and the quantity of temporary space the job will consume. Users of the Fulton Supercomputing Center often launch multiple jobs simultaneously. Some users launch hundreds of jobs simultaneously. To model the large usage of compute resources by a small number of users, the simulator samples a geometric probability distribution with a user-p parameter value of 0.56 (see Figure 5-8). All jobs assigned to user-ids greater than 9 are lumped into the 9th user. The number of processors per job is sampled from a uniform distribution of 2 to 8 processors. Time length of the job is sampled from a uniform distribution of 4 to 10 processors. Finally, the temporary storage consumption of a job is sampled from a geometric distribution with parameter value of 0.22. While temporary storage in HPC centers is a limited resource that needs better management, not all HPC jobs use temporary storage. Therefore, the temporary storage consumption parameter in the simulator is adjusted downwards so that approximately 80% of the jobs don't consume any temporary storage. The temporary storage consumption parameter of .22 and the user-p parameter of .56 were based on job data gathered from the FSL as described in section 5.2.

Once the compute nodes and the jobs have been generated, the simulator's scheduler uses a first come first served (FCFS) scheduling algorithm (Feitelson and Weil 1998), to simulate execution of the jobs. The simulator calculates the number of temporary space gigabyte-hours (GB-hours) wasted because Slurm is unable to detect which job owns which portion of a user's temporary storage consumption (see Figure 5-9). Reasonable values for user-p are in the 0.3 to 0.5 range. User-p determines the probability that a job will be assigned to the first user.

Given a workload of 100 jobs, user-p parameter of 0.56 and a wasted space parameter of 0.22, the simulator calculates that between 75 and 125 GB-hours of temporary storage space is wasted depending on job arrival patterns (see Figure 5-10). WTSS source code is included in Appendix A. Figures 5-8 and 5-9 were generated by chart2.py, while figure 5-10 was generated by chart1.py. Both chart1.py and chart2.py import the sim.py file to run the simulation.

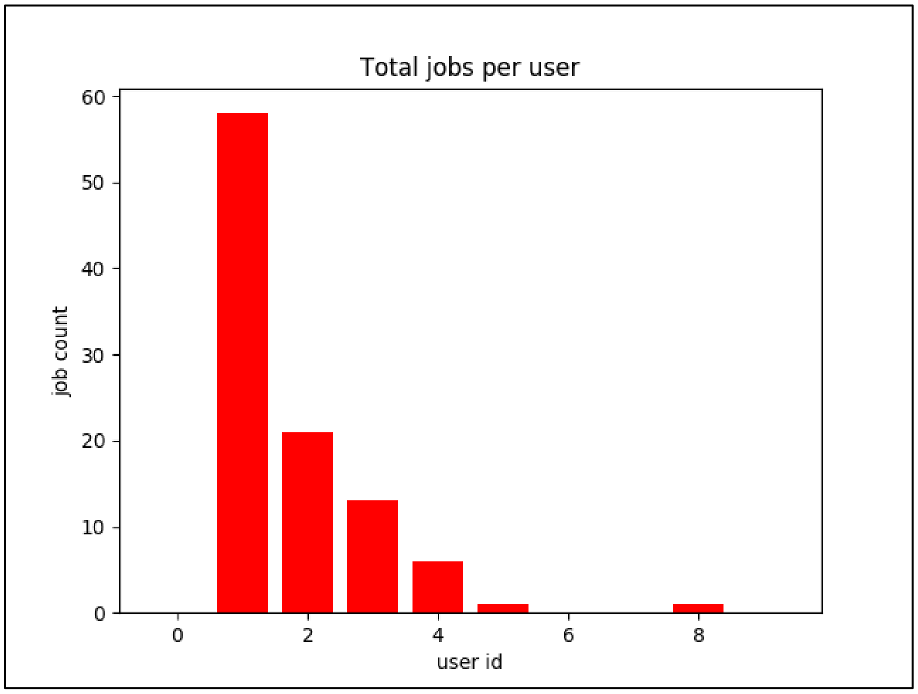

Figure 5-8: Total Jobs per User

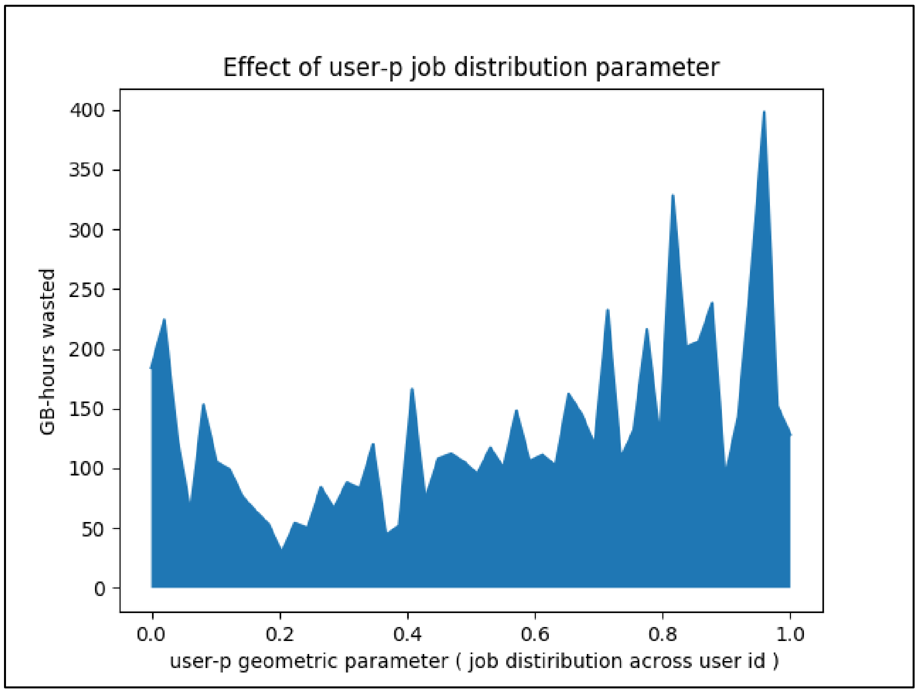

Figure 5-9: Effect of User-p on GB-hours Wasted

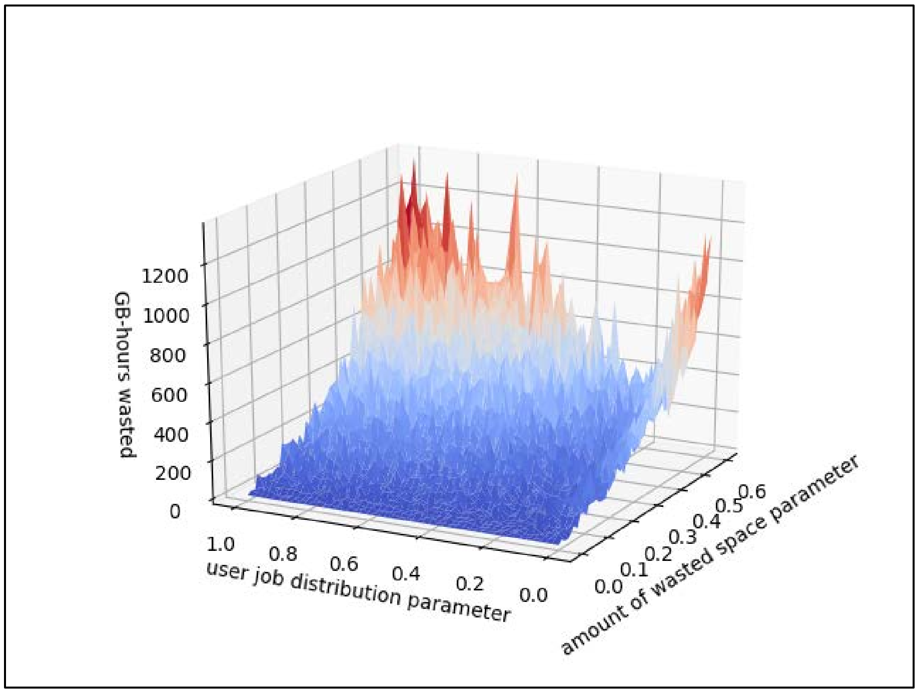

Figure 5-10: GB-hours Wasted Based on User-p and Wasted Space Parameters

6 DISCUSSION AND CONCLUSION

Per job isolation of temporary file storage in HPC environments provide benefits in security, efficiency, and administration. HPC users typically have multiple supercomputing jobs running concurrently. Temporary files generated by these tasks are only deleted when all jobs the user is running on a compute node are complete. Even though the user may have only one running job, the temporary directory may still contain temporary files from previously executed jobs, taking up valuable temporary storage on the compute node.

Isolation of temporary files per job and per user is implemented through bind mounts and Linux mount namespaces. Job temporary files are stored in individual job directories and bind mounted to `/tmp` in the job's mount namespace. All temporary files created throughout the job's life cycle are stored in `/tmp`, an isolated job directory visible to the root user as `/tmp/<user>/<job_id>`. Administrators can now purge temporary files belonging only to a specific user job by removing the temporary directory associated with the job. Slurm epilog scripts can be written by the administrator to automatically purge obsolete temporary files as jobs terminate, freeing scarce shared temporary file storage.

The purging of these obsolete temporary files neutralizes the danger of denial of service by malicious users taking up valuable shared temporary storage. Isolation of temporary job files provides visibility to administrators of which jobs may be taking up the most temporary file space. Information gathered by administrators concerning job file space can be used to enforce temporary file space quotas on a per job basis. Because temporary files are isolated on a per job and per user basis, a user's temporary files are by default private and secured from access by other users on the compute node. Our Slurm Task temporary storage isolation plugin will be submitted upstream to the Slurm public repository. This will allow other Slurm users to enable the plugin in slurm.conf to take advantage of the benefits of per job and per user temporary file isolation.

6.1 Future Work

The mount_namespace plugin can be merged with the public Slurm repository. This part of future work should be accomplished in the near future as I am currently working with the FSL director to prepare the code to be submitted to Slurm.

While the mount_namespace plugin is complete, there are still a few areas that can be improved. Slurm HPC environments are configured in many different ways, and although the plugin was tested thoroughly, there will most likely to be small changes that need to be made to allow the plugin to benefit a wider range of HPC environments.

For the solution described in this thesis to be effective, there are a few code dependencies (cgroups, pam_slurm_adopt, and pam_mount_ns_adopt). Future work could also include altering the plugin to work without some of these dependencies. These changes would make the mount_namespace plugin easier for system administrators to implement. The pam_mount_ns_adopt module could also be expanded so that it isn't dependent on the pam_slurm_adopt module.

There are several aspects of Linux namespaces that can still be explored within HPC environments. The BYU FSL Operations Director would also like to implement PID namespaces in addition to the mount namespaces covered in this thesis.

Apart from development improvement, there could also be more research on how the plugin affects HPC systems beyond disk space saved. Research could be done to see how many Watt-hours are saved due to the prompt removal of temporary files. Some HPC environments deal with HIPPA, and other government regulations. Research could be extended to see how Linux namespaces can simplify and improve data privacy to meet regulation requirements. This thesis covers the isolation of temporary files using mount namespaces. How might mount namespaces be used to isolate other files? Linux namespaces have a wide variety of types and applications that are yet to be discovered.

REFERENCES

Biederman, E. W. (2006). Multiple Instances of the Global Linux Namespaces. *Proceedings of the Linux Symposium*, *1*, p. 101. Ottawa, Ontario Canada. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.5475&rep=rep1&type=pdf #page=101

Chrissamuel, & Fossing. (2016, February 28). *vlsci/spank-private-tmp*. Retrieved from https://github.com/vlsci/spank-private-tmp

Feitelson, D. G., & Weil, A. M. (1998). Utilization and predictability in scheduling the IBM SP2 with backfilling. *IEEE*, 842-546. Retrieved from http://dx.doi.org/10.1109/ipps.1998.669970

Fossing, & Kmjonsson. (2015, March 24). *hpc2n/spank-private-tmp*. Retrieved from https://github.com/hpc2n/spank-private-tmp

FSL Resources. (2018). Retrieved from https://marylou.byu.edu/documentation/resources

Ganssle, J. (2004). Reentrancy. *The Firmware Handbook*, 231-244.

Geisshirt, K. (2007). *Pluggable Authentication Modules: The Definitive Guide to PAM for Linux SysAdmins and C Developers, a Comprehensive and Practical Guide to PAM for Linux: How Modules Work and How to Implement Them.* Packt Publishing.

Hallyn, S. E., & Pai, R. (2007, September 17). *Applying mount namespaces*. Retrieved from http://ibm.com/developerworks/library/l-mount-namespaces/index.html

Jaspal, S., Thomas, G., & Takashi, S. (1996). Impact of job mix on optimizations for space sharing.

Kerrisk, M. (2013, January 04). *Namespaces in operation, part1: namespaces overview.* Retrieved from LWN.net: https://lwn.net/Articles/531114/

Kerrisk, M. (2017, September 15). *STAT(2) Linux Programmer's Manual*. Retrieved 2018, from http://man7.org/linux/man-pages/man2/lstat.2.html

Kerrisk, M. (2018, February 02). *MOUNT(2) Linux Programmer's Manual*. Retrieved from http://man7.org/linux/man-pages/man2/mount.2.html

Kerrisk, M. (2018, February 02). *NAMESPACES(7) Linux Programmer's Manual*. Retrieved from http://man7.org/linux/man-pages/man7/namespaces.7.html

Kerrisk, M. (2018, February 02). *UNSHARE(2) Linux Programmer's Manual*. Retrieved from http://man7.org/linux/man-pages/man2/unshare.2.html

Larson, S. M., Snow, C. D., Shirts, M., & Pande, V. S. (2009). Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *arXiv preprint arXiv:0901.0866*.

Reshetova, E., Karhunen, J., Nyman, T., & Asokan, N. (2014, July 16). Security of OS-level virtualization technologies.

Ridwan, M. (2014). *Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces.* Retrieved from toptal: https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

Slurm Workload Manager. (2013, November 24). Retrieved from SchedMD: https://slurm.schedmd.com/slurm.html

SPANK - Slurm Plug-in Architecture for Node and job (K)control. (2018, February 08). Retrieved from http://slurm.schedmd.com/spank.html

Top 500. (2013, November). Retrieved 2017, from https://www.top500.org/

Torvalds, L. (2016). *Linux Kernel Coding Style.* Retrieved from SchedMD - Slurm Workload Manager: https://slurm.schedmd.com/coding_style.pdf

Wirzenius, L. (1993). *LINUX system administrator's guide.*

Xavier, M. E., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., & De Rose, C. A. (2013). Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing.* Porto Alegre, Brazil.

APPENDICES

Appendix A. Source Code for Simulated Data

sim.py

```
import random

import numpy as np


#nodes = { 'm8' : { 'count': 320, 'procs': 24 },  'm7': { 'count': 230, 'procs': 16 },
'm6' : { 'count': 256, 'procs': 12 } }

nodes = { 'm8' : { 'count': 10, 'procs': 24 }, }


def import_data():

  my_data = np.genfromtxt('hpc_data.csv', delimiter=' ',dtype=None)

  # print (my_data)


import_data()


class Node:

  def __init__( self, id, name, procs ):

    #node id, an integer

    self.id = id

    # node name m8-1

    self.name = name

    #number of processors the node has

    self.procs = procs

    # [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ] for a 10 processor node

    self.free = [ x for x in range( procs ) ]

    #dict user_id -> job_id -> jobs

    self.jobs_by_user = {}

    #dict user_id -> job_id -> jobs

    #jobs that have finished but can't cleanup temp space because other jobs from the
same user are running on the node.

    self.finished_temp_jobs = {}


  def __str__( self ):
    print "Node: " + self.name + " " + self.procs + ", " + self.free
  def __repr__( self ):
    return "Node: " + self.name + " " + str( self.procs ) + ", " + str( self.free )

  def take( self, n ):
    """take n processors from this nodes free processor list and return unique
identifiers, [ node.id, processor.id ], for each processor"""
    nn = self.free[:n]
    self.free = self.free[n:]
    v = [ [ self.id, x ] for x in nn ];
    return v

  def give( self, x ):
    """return processor x to the free processor list"""
    self.free.append( x )
    self.free = sorted( self.free )

  def addJob( self, j ):
    """add a newly started job to this node because the job is running on one of the
nodes processors"""
    if j.user_index in self.jobs_by_user:
      self.jobs_by_user[ j.user_index ][ j.id ] = j
    else:
      self.jobs_by_user[ j.user_index ] = { j.id: j }

  def addJobWaitingForCleanup( self, j ):
    """add a job to the finished_temp_jobs list because the job finished but still has
temporary storage allocated
    and cannont be freed, because another job from the same user is running on the
node"""
    if j.user_index in self.finished_temp_jobs:
      self.finished_temp_jobs[ j.user_index ][ j.id ] = j
    else:
      self.finished_temp_jobs[ j.user_index ] = { j.id: j }

  def endJob( self, j ):
    """job j has finished see if we can clean up its temporary storage usage or
whether it needs to wait in the finished_temp_jobs queue"""
    if( j.temp_gigs > 0 ):
      self.addJobWaitingForCleanup( j )
    else:
      j.temp_end_time = j.end_time
    if j.id in self.jobs_by_user[ j.user_index ]:
      self.jobs_by_user[ j.user_index ].pop( j.id )
    #print  self.jobs_by_user[ j.user_index ], len( self.jobs_by_user[ j.user_index ]
)
    if len( self.jobs_by_user[ j.user_index ] ) == 0:
      if j.user_index in self.finished_temp_jobs:
        for x in self.finished_temp_jobs[ j.user_index ].iteritems():
          x[1].temp_end_time = j.end_time
          #print "x", x
      self.finished_temp_jobs[ j.user_index ] = { }


class Job:
  def __init__( self, id, user_index, node_count, time_length, temp_gigs ):
    self.id = id
    self.user_index = user_index
    self.node_count = node_count
    self.time_length = time_length
    self.temp_gigs = temp_gigs
    self.end_time = 0
    self.temp_end_time = 0


  def start( self, start_time ):
    """given the start_time of the job, set the start_time, end_time, and
temp_end_time"""
    self.start_time = start_time
    self.end_time = start_time + self.time_length
    self.temp_end_time = self.end_time


  def wasted( self ):
    """calculate the wasted GB-hrs of this job"""
    return ( self.temp_end_time - self.end_time ) * self.temp_gigs


  def __repr__( self ):
    return "Job: " + str( self.id ) + " ST-ET%TET " + str( self.start_time ) + "-" +
str( self.end_time )+ "%" + str( self.temp_end_time - self.end_time ) + " NC " + str(
self.node_count ) + " TG " + str( self.temp_gigs )



class Scheduler:
  def __init__( self, nodes, jobs ):
    self.clock = 0;
    self.jobs = jobs
    self.nodes = {}
    self.running_jobs = []
    self.finished_jobs = []
    self.fullnodes = {}
    self.with_space_nodes = {}
    for x in nodes:
      self.with_space_nodes[ x.id ] = x
      self.nodes[ x.id ] = x

  def run( self ):
    """Schedule and run the jobs"""
    while len( self.jobs ) > 0 or len( self.running_jobs ) > 0:

      #sort jobs
      self.running_jobs = sorted( self.running_jobs, key = lambda x: x.end_time )
      #print "Q", self.running_jobs

      #finish jobs
      if len( self.running_jobs ) > 0:
        current_end_time = self.running_jobs[0].end_time
        self.clock = current_end_time
        while len( self.running_jobs ) > 0 and self.running_jobs[0].end_time ==
current_end_time:
          x = self.running_jobs.pop( 0 )
          self.return_nodes( x.nodes )
          ###print "Ending at", self.clock, x
          self.removeJobFromNodes( x )
          self.finished_jobs.append( x )

      #count free nodes
      free_node_count = 0
      for k, x in self.nodes.iteritems():
        free_node_count += len( x.free )

      #launch jobs
      success = True
      while success and len( self.jobs ) > 0:
        if self.jobs[ 0 ].node_count < free_node_count:
          j = self.jobs.pop( 0 )
          #print free_node_count, j.node_count
          j.nodes = self.take_nodes( j.node_count )
          j.start( self.clock );
          self.addJobToNodes( j )
          ###print "Starting", j
          self.running_jobs.append( j )
          free_node_count -= j.node_count
        else:
          success = False

    #print results at end of run
    ###print self.nodes
    self.finished_jobs = sorted( self.finished_jobs, key = lambda x: x.id )
    wasted = 0
    user_dist = np.zeros( 10 )
    for x in self.finished_jobs:
      ###print x
      wasted += x.wasted()
      user_dist[ x.user_index ] += 1;
    #print user_dist

    #print "End Clock", self.clock
    #print "Wasted", wasted, "GB-hrs"
    return { 'end_clock': self.clock, 'wasted': wasted, 'user_dist': user_dist }

  def return_nodes( self, nodes ):
    """Given a list of processor ids [[nodeid, processid], [nodeid, processid] ]
lookup the node by node id and give the process id back to the node"""
    #print "return_nodes ", nodes
    for x in nodes:
      if x[0] in self.fullnodes:
        node = self.fullnodes.pop( x[0] )
        node.give( x[1] );
        self.with_space_nodes[ x[0] ] = node
      elif x[0] in self.with_space_nodes:
        self.with_space_nodes[ x[0] ].give( x[ 1 ] )

  def take_nodes( self, cnt ):
    """Given a count of processors needed take that many processors from nodes with
free processors"""
    local_nodes = []
    local_cnt = cnt
    while local_cnt > 0:
      i = self.with_space_nodes.popitem()
      if len( i[1].free ) <= local_cnt:
        #print local_cnt, i, "lte"
        local_cnt -= len( i[1].free )
        local_nodes.extend( i[1].take( len( i[1].free  ) ) )
        self.fullnodes[ i[1].id ] = i[1]
      else:
        #print local_cnt, i, "gt"
        local_nodes.extend( i[1].take( local_cnt ) )
        self.with_space_nodes[ i[1].id ] = i[1]
        local_cnt = 0;
    return local_nodes;

  def addJobToNodes( self, j ):
    """Add job j to all nodes that job is running on"""
    for x in j.nodes:
      self.nodes[ x[0] ].addJob( j );

  def removeJobFromNodes( self, j ):
    """remove job j from all nodes that job is running on"""
    for x in j.nodes:
      self.nodes[ x[0] ].endJob( j );




def genNodes():
  """Generate node datastructures from node description structure"""
  id = 0;
  ns = []
  for clustername, data in nodes.items():
    for x in xrange( data[ 'count' ]):
      ns.append( Node( id, clustername + '-' + str( x ), data[ 'procs' ]))
      id += 1;
  return ns

def genJobs( id, users, p, wasted_p = 0.5 ):
  """Generate job datastructure from job creation parameters"""
  user_index = max( 0, min( np.random.geometric( p=p ), len( users ) ) )
  node_count = random.randint(2,8)
  time_length = random.randint(4,10)
  temp_gigs = max( 0, np.random.geometric( p = ( 1 - wasted_p ) ) - 1 )
  return Job( id, user_index, node_count, time_length, temp_gigs );

def simulate( user_p = 0.35, jobcount = 100, usercount = 10, wasted_p = 0.5 ):
  jobs = map( lambda x: genJobs( x, range(1, usercount ), user_p, wasted_p = wasted_p
), xrange( jobcount ) )
  nodes = genNodes()
  #create and run scheduler
  return Scheduler( nodes, jobs ).run()
```

chart1.py

```
import sim as Sim
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

x = np.linspace(0, 1)
y = [ Sim.simulate( user_p=z )['wasted'] for z in x ]

#plt.fill_between(x, 0, y )
#plt.grid(True)
#plt.show()

x = np.linspace(0, 0.6)
y = np.linspace(0, 1)
z = [ [ Sim.simulate( user_p=yz, wasted_p=xz )['wasted'] for xz in x ] for yz in y  ]
print len( x )
print len( y )
print len( z )
print np.array( z ).shape
X, Y = np.meshgrid(x, y)
#print X
#print Y
Z = np.clip( np.array( z ), 0, 5000 )
```

```
#Z = np.array( z )
#print Z


fig = plt.figure()
ax = fig.add_subplot( 111, projection='3d' )
#ax.contourf( X, Y, Z )
ax.plot_surface( X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm )
ax.set_xlabel('amount of wasted space parameter')
ax.set_ylabel('user job distribution parameter')
ax.set_zlabel('GB-hours wasted')
#ax.set_xlim(0.2, 1)
#ax.set_zlim(0, 1000)
#ax.set_zscale('log')
#plt.fill_between(x, 0, y )
plt.grid(True)
plt.show()
```

chart2.py

```
import sim as Sim
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

x = np.linspace(0, 1)
y = [ Sim.simulate( user_p=z )['wasted'] for z in x ]

#plt.fill_between(x, 0, y )
#plt.grid(True)
#plt.show()

x = np.linspace(0, 1 )
y = np.linspace(0, 1 )
#print x
#print y
z = [ Sim.simulate( user_p=yz )['wasted'] for yz in x  ]

fig = plt.figure()
```

```
ax = fig.add_subplot( 111 )
ax.plot( x, z )
ax.set_title('Effect of user-p job distribution parameter')
ax.set_xlabel(' user-p geometric parameter ( job distiribution across user id )')
ax.set_ylabel('GB-hours wasted')
plt.fill_between(x, 0, z )
txt = '''
Reasonable values for user-p are in the 0.3 to 0.45 range
user-p determines the probability that a job will assigned to the first user.
'''
fig.text(.1,1,txt)



fig = plt.figure()
ax = fig.add_subplot( 111 )
ud = Sim.simulate( user_p=0.3 ) ['user_dist']
ax.bar( np.arange( len( ud ) ), ud, color='r' )
ax.set_xlabel('user id')
ax.set_ylabel('job count')
ax.set_title('Total jobs per user' )
txt = '''
This graph depicts the job distribution for a user-p parameter of 0.3
We lump all jobs assigned to user-ids greater than 9 into the 9th user.
'''
fig.text(.1,1,txt)


x = np.linspace(0, 1 )
y = np.linspace(0, 1, 10)
z = [ Sim.simulate( user_p=xz )['user_dist'] for xz in x ]
print len( x )
print len( y )
print len( z )
print np.array( z ).shape
X, Y = np.meshgrid(y, x)
#print X
#print Y
Z = np.array( z )
print X.shape
print Y.shape
print Z.shape
Z = np.clip( np.array( z ), 0, 100 )


fig = plt.figure()
ax = fig.add_subplot( 111, projection='3d' )
#ax.contourf( X, Y, Z )
ax.plot_surface( X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm )
ax.set_xlabel('user_id')
ax.set_ylabel('user_p')
ax.set_zlabel('# of jobs')
ax.set_xticklabels(np.arange( 10 ))
#ax.set_xlim(0.2, 1)
#ax.set_zlim(0, 1000)
#ax.set_zscale('log')
#plt.fill_between(x, 0, y )
plt.grid(True)
plt.show()
```

Appendix B. Source Code for Mount_Isolation Task Plugin

```
/*****************************************************************************\
 *  task_mount_isolation.c - Create isolated namespaced directories per job
 *****************************************************************************
 *  Copyright (C) 2018, Brigham Young University
 *  Author:  Tanner Satchwell <tannersatch@gmail.com>
 *  Author:  Ryan Cox <ryan_cox@byu.edu>
 *
 *  This file is part of SLURM, a resource management program.
 *  For details, see <http://slurm.schedmd.com/>.
 *  Please also read the included file: DISCLAIMER.
 *
 *  SLURM is free software; you can redistribute it and/or modify it under
 *  the terms of the GNU General Public License as published by the Free
 *  Software Foundation; either version 2 of the License, or (at your option)
 *  any later version.
 *
 *  In addition, as a special exception, the copyright holders give permission
 *  to link the code of portions of this program with the OpenSSL library under
 *  certain conditions as described in each individual source file, and
 *  distribute linked combinations including the two. You must obey the GNU
 *  General Public License in all respects for all of the code used other than
 *  OpenSSL. If you modify file(s) with this exception, you may extend this
 *  exception to your version of the file(s), but you are not obligated to do
 *  so. If you do not wish to do so, delete this exception statement from your
 *  version.  If you delete this exception statement from all source files in
 *  the program, then also delete it here.
 *
 *  SLURM is distributed in the hope that it will be useful, but WITHOUT ANY
 *  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *  FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *  details.
 *
 *  You should have received a copy of the GNU General Public License along
 *  with SLURM; if not, write to the Free Software Foundation, Inc.,
 *  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301  USA.
\*****************************************************************************/

#if     HAVE_CONFIG_H
#  include "config.h"
#endif

#define _GNU_SOURCE
#define PATH_MAX 1024
#include <sched.h>
#include <unistd.h>
#include <sys/mount.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <signal.h>
#include <sys/types.h>
#include <dirent.h>

#include "slurm/slurm_errno.h"
#include "src/common/slurm_xlator.h"
#include "src/slurmd/slurmstepd/slurmstepd_job.h"
#include "src/slurmd/slurmd/slurmd.h"
#include "src/common/uid.c"

/*
 * These variables are required by the generic plugin interface.  If they
 * are not found in the plugin, the plugin loader will ignore it.
 *
 * plugin_name - a string giving a human-readable description of the
 * plugin.  There is no maximum length, but the symbol must refer to
 * a valid string.
 *
 * plugin_type - a string suggesting the type of the plugin or its
 * applicability to a particular form of data or method of data handling.
 * If the low-level plugin API is used, the contents of this string are
 * unimportant and may be anything.  SLURM uses the higher-level plugin
 * interface which requires this string to be of the form
 *
 *      <application>/<method>
 *
 * where <application> is a description of the intended application of
 * the plugin (e.g., "task" for task control) and <method> is a description
 * of how this plugin satisfies that application.  SLURM will only load
 * a task plugin if the plugin_type string has a prefix of "task/".
 *
 * plugin_version - an unsigned 32-bit integer containing the Slurm version
 * (major.minor.micro combined into a single number).
 */
const char plugin_name[]        = "task MOUNT_ISOLATION plugin";
const char plugin_type[]        = "task/mount_isolation";
const uint32_t plugin_version   = SLURM_VERSION_NUMBER;

/*
 * the main isolate function that sets up the mount namespace
 */
static int _isolate(const stepd_step_rec_t *job);

/*
 * a function to cleanup no longer needed temporary files
 */
static int _job_cleanup(const uint32_t job_id);

/*
 * a function to recursively delete a non-empty directory
 */
static int _remove_directory(const char *path, int64_t *bytes, dev_t device_id);

/*
 * init() is called when the plugin is loaded, before any other functions
 *  are called.  Put global initialization here.
 */
extern int init (void)
{
  /* retreive tmp directories and subdirectory from slurm.conf */
  char tmp_dirs[PATH_MAX];
  snprintf(tmp_dirs, PATH_MAX, "%s", slurmctld_conf.task_plugin_tmp_dirs);
  char tmp_subdir[PATH_MAX];
  snprintf(tmp_subdir, PATH_MAX, "%s", slurmctld_conf.task_plugin_tmp_subdir);
  int rc = 0;

  /* prepare to loop through tmp directories */
  char* tmp_dir;
  char* saveptr;
  tmp_dir = strtok_r(tmp_dirs, ",", &saveptr);

  /* look through tmp directories */
  while (tmp_dir) {
    char subdir_path[PATH_MAX];
    snprintf(subdir_path, PATH_MAX, "%s/%s", tmp_dir, tmp_subdir);
    struct stat sb;

    /* make the tmp directory private */
    rc = mount("", tmp_dir, NULL, MS_PRIVATE, NULL);
    if (rc) {
      /* make sure the directory is mounted to itself */
      rc = mount(tmp_dir, tmp_dir, NULL, MS_BIND, NULL);
      if (rc) {
        slurm_error("%s: failed to 'mount --bind %s %s' error: %d",
            plugin_name, tmp_dir, tmp_dir, rc);
        return SLURM_ERROR;
      }

      /* try again */
      rc = mount("", tmp_dir, NULL, MS_PRIVATE, NULL);
      if (rc) {
        slurm_error("%s: failed to 'mount --make-private %s' error: %d",
            plugin_name, tmp_dir, rc);
        return SLURM_ERROR;
      }
    }

    /* create tmp subdirectory */
    rc = lstat(subdir_path, &sb);
    if (rc == 0 && S_ISDIR(sb.st_mode)) {
      debug3("%s: failed to create %s temporary subdirectory at %s. warning: %d
(directory already exists)",
          plugin_name, tmp_subdir, subdir_path, rc);
    } else {
      rc = mkdir(subdir_path, 0000);
      if (rc) {
        slurm_error("%s: failed to create %s temporary subdirectory at %s. error: %d",
            plugin_name, tmp_subdir, subdir_path, rc);
        return SLURM_ERROR;
      }
    }

    /* prepare for next loop */
    tmp_dir = strtok_r(NULL, ",", &saveptr);
    while (tmp_dir && *tmp_dir == '\040') {
      tmp_dir++;
    }
  }

  debug("%s loaded", plugin_name);
  return SLURM_SUCCESS;
}

/*
 * fini() is called when the plugin is removed. Clear any allocated
 *  storage here.
 */
extern int fini (void)
{
  return SLURM_SUCCESS;
}

/*
 * task_p_slurmd_batch_request()
 */
extern int task_p_slurmd_batch_request (uint32_t job_id,
          batch_job_launch_msg_t *req)
{
  debug("task_p_slurmd_batch_request: %u", job_id);
  return SLURM_SUCCESS;
}

/*
```

```
 * task_p_slurmd_launch_request()
 */
extern int task_p_slurmd_launch_request (uint32_t job_id,
          launch_tasks_request_msg_t *req,
          uint32_t node_id)
{
  debug("task_p_slurmd_launch_request: %u.%u %u",
      job_id, req->job_step_id, node_id);
  return SLURM_SUCCESS;
}

/*
 * task_p_slurmd_reserve_resources()
 */
extern int task_p_slurmd_reserve_resources (uint32_t job_id,
          launch_tasks_request_msg_t *req,
          uint32_t node_id)
{
  debug("task_p_slurmd_reserve_resources: %u %u", job_id, node_id);
  return SLURM_SUCCESS;
}

/*
 * task_p_slurmd_suspend_job()
 */
extern int task_p_slurmd_suspend_job (uint32_t job_id)
{
  debug("task_p_slurmd_suspend_job: %u", job_id);
  return SLURM_SUCCESS;
}

/*
 * task_p_slurmd_resume_job()
 */
extern int task_p_slurmd_resume_job (uint32_t job_id)
{
  debug("task_p_slurmd_resume_job: %u", job_id);
  return SLURM_SUCCESS;
}
```

```
/*
 * task_p_slurmd_release_resources()
 */
extern int task_p_slurmd_release_resources (uint32_t job_id)
{
  debug("task_p_slurmd_release_resources: %u", job_id);
  /* return _job_cleanup(job_id); */
  return SLURM_SUCCESS;
}

/*
 * task_p_pre_setuid() is called before setting the UID for the
 * user to launch his jobs. Use this to create the CPUSET directory
 * and set the owner appropriately.
 */
extern int task_p_pre_setuid (stepd_step_rec_t *job)
{
  return SLURM_SUCCESS;
}

/*
 * task_p_pre_launch() is called prior to exec of application task.
 *  It is followed by TaskProlog program (from slurm.conf) and
 *  --task-prolog (from srun command line).
 */
extern int task_p_pre_launch (stepd_step_rec_t *job)
{
  debug("task_p_pre_launch: %u.%u, task %d",
      job->jobid, job->stepid, job->envtp->procid);
  return SLURM_SUCCESS;
}

/*
 * task_p_pre_launch_priv() is called prior to exec of application task.
 * in privileged mode, just after slurm_spank_task_init_privileged
 */
extern int task_p_pre_launch_priv (stepd_step_rec_t *job)
{
  debug("task_p_pre_launch_priv: %u.%u",
      job->jobid, job->stepid);
  return _isolate(job);
  /* return SLURM_SUCCESS; */
}

/*
 * task_term() is called after termination of application task.
 *  It is preceded by --task-epilog (from srun command line)
 *  followed by TaskEpilog program (from slurm.conf).
 */
extern int task_p_post_term (stepd_step_rec_t *job, stepd_step_task_info_t *task)
{
  debug("task_p_post_term: %u.%u, task %d",
      job->jobid, job->stepid, task->id);
  return _job_cleanup(job->jobid);
  /* return SLURM_SUCCESS; */
}

/*
 * task_p_post_step() is called after termination of the step
 * (all the task)
 */
extern int task_p_post_step (stepd_step_rec_t *job)
{
  return SLURM_SUCCESS;
}

/*
 * Keep track a of a pid.
 */
extern int task_p_add_pid (pid_t pid)
{
  return SLURM_SUCCESS;
}



/*
 * _isolate() is called from task_p_pre_launch_priv to setup mount namespace isolation
 */
static int _isolate(const stepd_step_rec_t *job)
{
```

```
/* set variables for function */
int rc = 0;
char* user = uid_to_string(job->uid);

/* retreive tmp directories and subdirectory from slurm.conf */
char tmp_dirs[PATH_MAX];
snprintf(tmp_dirs, PATH_MAX, "%s", slurmctld_conf.task_plugin_tmp_dirs);
char tmp_subdir[PATH_MAX];
snprintf(tmp_subdir, PATH_MAX, "%s", slurmctld_conf.task_plugin_tmp_subdir);

/* create a new mount namespace */
rc = unshare(CLONE_NEWNS);
if (rc) {
  slurm_error("%s: failed to unshare mounts for job: %u error: %d",
      plugin_name, job->jobid, rc);
  return SLURM_ERROR;
}

/* make root in the new namespace a slave so any changes don't propagate
 * back to the default root */
rc = mount("", "/", NULL, MS_REC|MS_SLAVE, NULL);
if (rc) {
  slurm_error("%s: failed to 'mount --make-rslave /' for job: %u error: %d",
      plugin_name, job->jobid, rc);
  return SLURM_ERROR;
}

/* prepare to loop through tmp directories */
char* tmp_dir;
char* saveptr;
tmp_dir = strtok_r(tmp_dirs, ",", &saveptr);

/* loop through tmp directories */
while (tmp_dir) {
  /* set variables for loop */
  char tmp_user_path[PATH_MAX];
  snprintf(tmp_user_path, PATH_MAX, "%s/%s/%s",
      tmp_dir, tmp_subdir, user);
  char tmp_job_path[PATH_MAX];
  snprintf(tmp_job_path, PATH_MAX, "%s/%s/%s/%d",
        tmp_dir, tmp_subdir, user, job->jobid);
    struct stat sb;

    /* create user tmp directory */
    rc = lstat(tmp_user_path, &sb);
    if (rc == 0 && S_ISDIR(sb.st_mode)) {
      debug3("%s: failed to create user directory %s for job: %u warning: %d
(directory already exists)",
          plugin_name, tmp_user_path, job->jobid, rc);
    } else {
      rc = mkdir(tmp_user_path, 0700);
      if (rc) {
        slurm_error("%s: failed to create user directory %s for job: %u error: %d",
            plugin_name, tmp_user_path, job->jobid, rc);
        return SLURM_ERROR;
      }
    }

    /* set permissions on user tmp directory */
    rc = lchown(tmp_user_path, job->uid, job->gid);
    if (rc) {
      slurm_error("%s: failed to change ownership of user directory %s for job: %u
error: %d",
          plugin_name, tmp_user_path, job->jobid, rc);
      return SLURM_ERROR;
    }

    /* create job id tmp directory */
    rc = lstat(tmp_job_path, &sb);
    if (rc == 0 && S_ISDIR(sb.st_mode)) {
      debug3("%s: failed to create jobid directory %s for job: %u warning: %d
(directory already exists)",
          plugin_name, tmp_job_path, job->jobid, rc);
    } else {
      rc = mkdir(tmp_job_path, 0700);
      if (rc) {
        slurm_error("%s: failed to create jobid directory %s for job: %u error: %d",
            plugin_name, tmp_job_path, job->jobid, rc);
        return SLURM_ERROR;
      }
    }
    /* set permissions on job id tmp directory */
    rc = lchown(tmp_job_path, job->uid, job->gid);
    if (rc) {
      slurm_error("%s: failed to change ownership of jobid directory %s for job: %u
error: %d",
          plugin_name, tmp_job_path, job->jobid, rc);
      return SLURM_ERROR;
    }

    /* bind user and job id isolated directories to tmp directories */
    rc = mount(tmp_job_path, tmp_dir, NULL, MS_BIND, NULL);
    if (rc) {
      slurm_error("%s: failed to mount jobid directory %s to %s for job: %u error:
%d",
          plugin_name, tmp_job_path, tmp_dir, job->jobid, rc);
      return SLURM_ERROR;
    }

    /* prepare for next loop */
    tmp_dir = strtok_r(NULL, ",", &saveptr);
    while (tmp_dir && *tmp_dir == '\040') {
      tmp_dir++;
    }

  }

  return SLURM_SUCCESS;
}

/*
 * _job_cleanup() is called when a job terminates and calls _remove_directory()
 * to remove temporary files related to the temrinated job
 */
static int _job_cleanup(const uint32_t job_id)
{
  int rc = 0;
  ListIterator itr = NULL;
  List steps = NULL;
  step_loc_t *stepd = NULL;
  int job_step_cnt = 0;
  int64_t bytes = 0;
```

```
char* nodename;
uid_t uid = -1;
int fd;

/* get the nodename */
if (!(nodename = slurm_conf_get_aliased_nodename())) {
  slurm_error("%s: failed to get nodename for job: %u error: %d",
      plugin_name, job_id, rc);
  return SLURM_ERROR;
}

steps = stepd_available(NULL, nodename);

/* count number of running steps for the job and get uid */
itr = list_iterator_create(steps);
while ((stepd = list_next(itr))) {
  if (stepd->jobid != job_id) {
    /* multiple jobs expected on shared nodes */
    continue;
  }

  /* count number of running steps for the job */
    if (stepd->stepid != SLURM_EXTERN_CONT) {
       job_step_cnt++;
    }

  debug3("%s: _job_cleanup step: %u:%u, step count: %d",
      plugin_name, stepd->jobid, stepd->stepid, job_step_cnt);

    fd = stepd_connect(stepd->directory, stepd->nodename, stepd->jobid,
        stepd->stepid, &stepd->protocol_version);
  if (fd == -1) {
    debug3("%s: _job_cleanup unable to connect to step %u.%u",
        plugin_name, stepd->jobid, stepd->stepid);
    continue;
  }
  uid = stepd_get_uid(fd, stepd->protocol_version);

  close(fd);
  if ((int)uid < 0) {
    debug3("%s: _job_cleanup get uid failed %u.%u",
        plugin_name, stepd->jobid, stepd->stepid);
    continue;
  }
}
list_iterator_destroy(itr);

/* if this is the last step in the job */
if (job_step_cnt == 1) {
  /* set necessary variables */
  char* user = uid_to_string(uid);
  struct stat sb;
  /* used to ensure recursive remove stays on the same file system */
  dev_t device_id;

  /* retreive tmp directories and subdirectory from slurm.conf */
  char tmp_dirs[PATH_MAX];
  snprintf(tmp_dirs, PATH_MAX, "%s", slurmctld_conf.task_plugin_tmp_dirs);
  char tmp_subdir[PATH_MAX];
  snprintf(tmp_subdir, PATH_MAX, "%s",
      slurmctld_conf.task_plugin_tmp_subdir);

  /* prepare to loop through tmp directories */
  char* tmp_dir;
  char* saveptr;
  tmp_dir = strtok_r(tmp_dirs, ",", &saveptr);

  /* loop through tmp directories */
  while (tmp_dir) {
    /* set variables for loop */
    char tmp_job_path[PATH_MAX];
    snprintf(tmp_job_path, PATH_MAX, "%s/%s/%s/%d",
        tmp_dir, tmp_subdir, user, job_id);
    if (!lstat(tmp_job_path, &sb)) {
      device_id = sb.st_dev;
    }

    rc = _remove_directory(tmp_job_path, &bytes, device_id);
    if (rc) {
```

```
        slurm_error("%s: failed to remove job related temporary files for job: %u
error: %d",
            plugin_name, job_id, rc);
        return SLURM_ERROR;
      }

      /* prepare for next loop */
      tmp_dir = strtok_r(NULL, ",", &saveptr);
      while (tmp_dir && *tmp_dir == '\040') {
        tmp_dir++;
      }
    }

    /****** Begin Data Gathering ******/
    info("%s: %ld bytes temporary files purged for jobid %u",
        plugin_name, bytes, job_id);
    /****** End Data Gathering ******/

  }

  return SLURM_SUCCESS;
}

/*
 * _remove_directory() is called to recursively delete a non-empty directory
 */
static int _remove_directory(const char *path, int64_t *bytes, dev_t device_id)
{
  /* declare needed variables */
  DIR *d = opendir(path);
  size_t path_len = strlen(path);
  int r = -1;


  if (d) {
    struct dirent *p;
    r = 0;

    while (!r && (p=readdir(d))) {
      int r2 = -1;
      char *buf;
      size_t len;

      /* skip the names "." and ".." we don't want to recurse on them. */
      if (!strcmp(p->d_name, ".") || !strcmp(p->d_name, "..")) {
        continue;
      }

      len = path_len + strlen(p->d_name) + 2;
      buf = xmalloc(len);

      if (buf) {
        struct stat statbuf;
        snprintf(buf, len, "%s/%s", path, p->d_name);

        if (!lstat(buf, &statbuf)) {
          if (statbuf.st_dev == device_id) {
            if (S_ISDIR(statbuf.st_mode)) {
              r2 = _remove_directory(buf, bytes, device_id);
            } else {
              /****** Begin Data Gathering ******/
              *bytes += statbuf.st_size;
              /****** End Data Gathering ******/
              r2 = remove(buf);
            }
          } else {
            /* device ID has changed, return error without removing */
            r2 = -1;
          }
        }

        xfree(buf);
      }
      r = r2;
    }
    closedir(d);
  }

  if (!r) {
    /****** Begin Data Gathering ******/
```

```
    struct stat sb;
    if (!lstat(path, &sb)) {
      *bytes += sb.st_size;
      r = remove(path);
    }
    /****** End Data Gathering ******/
    /* r = remove(path); */
  }

  return r;
}
```

Appendix C. Source Code for PAM_Mount_NS_Adopt

pam_mount_ns_adopt.c

```
/*****************************************************************************\
 *  pam_mount_ns_adopt.c - Adopt incoming connections into job mount namespace
 *****************************************************************************
 *  Copyright (C) 2018, Brigham Young University
 *  Author:  Tanner Satchwell <tannersatch@gmail.com>
 *
 *  This file is part of SLURM, a resource management program.
 *  For details, see <https://slurm.schedmd.com/>.
 *  Please also read the included file: DISCLAIMER.
 *
 *  SLURM is free software; you can redistribute it and/or modify it under
 *  the terms of the GNU General Public License as published by the Free
 *  Software Foundation; either version 2 of the License, or (at your option)
 *  any later version.
 *
 *  In addition, as a special exception, the copyright holders give permission
 *  to link the code of portions of this program with the OpenSSL library under
 *  certain conditions as described in each individual source file, and
 *  distribute linked combinations including the two. You must obey the GNU
 *  General Public License in all respects for all of the code used other than
 *  OpenSSL. If you modify file(s) with this exception, you may extend this
 *  exception to your version of the file(s), but you are not obligated to do
 *  so. If you do not wish to do so, delete this exception statement from your
 *  version.  If you delete this exception statement from all source files in
 *  the program, then also delete it here.
 *
 *  SLURM is distributed in the hope that it will be useful, but WITHOUT ANY
 *  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *  FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *  details.
 *
 *  You should have received a copy of the GNU General Public License along
 *  with SLURM; if not, write to the Free Software Foundation, Inc.,
 *  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301  USA.
\*****************************************************************************/


#ifndef PAM_MODULE_NAME
#  define PAM_MODULE_NAME "pam_mount_ns_adopt"
#endif

#if HAVE_CONFIG_H
#  include "config.h"
#endif

#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <syslog.h>
#include <pwd.h>
#include <stddef.h>
#include <stdint.h>
#include <sched.h>
#include <sys/mount.h>
#include <inttypes.h>
#include <dlfcn.h>

#include "helper.c"
#include "slurm/slurm.h"
#include "src/common/slurm_xlator.h"
#include "src/common/slurm_protocol_api.h"
#include "src/common/xcgroup_read_config.c"
#include "src/slurmd/common/xcgroup.c"
#include "src/common/stepd_api.h"
```

```
#ifndef PATH_MAX
#  define PATH_MAX 1024
#endif

#include <security/_pam_macros.h>
#include <security/pam_ext.h>
#define PAM_SM_SESSION
#include <security/pam_modules.h>
#include <security/pam_modutil.h>


/**********************************\
 *  Session Management Functions  *
\**********************************/

PAM_EXTERN int
pam_sm_open_session(pam_handle_t *pamh,
          int flags, int argc, const char **argv)
{
  uint16_t protocol_version;
  step_loc_t *stepd = NULL;
  uint32_t * pids = NULL;
  uint32_t job_id = 0;
  uint32_t count = 0;
  char * nodename = NULL;
  char mountns[PATH_MAX];
  ListIterator itr = NULL;
  List steps = NULL;
  pid_t user_pid;
  pid_t job_pid;
  int step_id = 0;
  int rc = 0;
  int fd1;
  int fd2;

  /* get the pid of the connecting user */
  syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO), "%s: acquiring pid",
      PAM_MODULE_NAME);
  user_pid = getpid();
  syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO), "%s: user pid = %d",
      PAM_MODULE_NAME, user_pid);

  /* get the job_id for the connecting user */
  rc = slurm_pid2jobid(user_pid, &job_id);
  if (rc) {
    _log_msg(LOG_INFO, "slurm_pid2jobid error: %s", strerror(rc));
    return (PAM_IGNORE);
  }

  /* get the node name of the node the user is connecting to */
  syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO), "%s: acquiring nodename",
      PAM_MODULE_NAME);
  if (!(nodename = slurm_conf_get_aliased_nodename())) {
    /* if no match, try localhost
     * (Should only be valid in a test environment) */
    if (!(nodename = slurm_conf_get_nodename("localhost"))) {
      syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO), "%s: no hostname found",
          PAM_MODULE_NAME);
      return (PAM_IGNORE);
    }
  }
  syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO), "%s: nodename = %s",
      PAM_MODULE_NAME, nodename);

  /* find a stepid for the job */
  steps = stepd_available(NULL, nodename);
  itr = list_iterator_create(steps);
  while ((stepd = list_next(itr))) {
    if (stepd->jobid != job_id) {
      /* multiple jobs expected on shared nodes */
      continue;
    }
    if (stepd->stepid != SLURM_EXTERN_CONT) {
      step_id = stepd->stepid;
      break;
    }
  }
  list_iterator_destroy(itr);
  syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO), "%s: step_id = %d",
      PAM_MODULE_NAME, step_id);
```

```
/* connect to stepd to get job information */
syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO), "%s: connecting to job %u",
    PAM_MODULE_NAME, job_id);
fd1 = stepd_connect(NULL, nodename, job_id, step_id, &protocol_version);
if (fd1 == -1) {
  if (errno == ENOENT) {
    syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO),
        "%s: job step %u.%u does not exist on this node.",
        PAM_MODULE_NAME, job_id, step_id);
  } else {
    syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO),
        "%s: unable to connect to slurmstepd", PAM_MODULE_NAME);
  }
  close(fd1);
  return (PAM_IGNORE);
}

/* get a list of job pids, just use the first pid that isn't
 * the incoming connection */
syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO),
    "%s: getting pids", PAM_MODULE_NAME);
stepd_list_pids(fd1, protocol_version, &pids, &count);
for (int i = 0; i < count; i++) {
  if (pids[i] != user_pid) {
    job_pid = pids[i];
    break;
  }
}

/* prepare the path of the job mount ns */
syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO),
    "%s: building mnt namespace path", PAM_MODULE_NAME);
snprintf(mountns, PATH_MAX, "/proc/%d/ns/mnt", job_pid);

/* open and connect to the job mount ns */
syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO),
    "%s: opening mnt namespace", PAM_MODULE_NAME);
fd2 = open(mountns, O_RDONLY);
if (fd2 == -1) {
  syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO),
        "%s: failed to open '/proc/PID/ns/mnt", PAM_MODULE_NAME);
      goto cleanup;
    }

    /* adopt the user into the job mnt namespace */
    syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO),
        "%s: adopting user into mnt namespace", PAM_MODULE_NAME);
    if (setns(fd2, 0) == -1) {
      syslog(LOG_MAKEPRI(LOG_AUTH, LOG_INFO),
          "%s: setns failed to adopt user into jobid mnt ns",
          PAM_MODULE_NAME);
      goto cleanup;
    }

    close(fd1);
    close(fd2);
    return (PAM_SUCCESS);

    cleanup:
      close(fd1);
      close(fd2);
      return (PAM_IGNORE);
  }


  PAM_EXTERN int
  pam_sm_close_session(pam_handle_t *pamh,
            int flags, int argc, const char *argv[])
  {
    return (PAM_SUCCESS);
  }


  #ifdef PAM_STATIC
  struct pam_module _pam_mount_ns_adopt_modstruct = {
    PAM_MODULE_NAME,
    NULL,
    NULL,
    NULL,
    pam_sm_open_session,
    NULL,
    NULL,
  };
  #endif
```

helper.h

```
/* helper.h
 *
 * Some helper functions needed for pam_slurm_adopt.c */

#define PAM_SM_ACCOUNT
#include <security/pam_modules.h>
#include <security/_pam_macros.h>

extern void send_user_msg(pam_handle_t *pamh, const char *msg);
extern void libpam_slurm_init (void);
extern void libpam_slurm_fini (void);
```

helper.c

```
/*****************************************************************************\
 *  pam_slurm_adopt/helper.c
 *****************************************************************************
 *  Useful portions extracted from pam_slurm.c by Ryan Cox <ryan_cox@byu.edu>
 *
 *  Copyright (C) 2002-2007 The Regents of the University of California.
 *  Copyright (C) 2008-2009 Lawrence Livermore National Security.
 *  Produced at Lawrence Livermore National Laboratory (cf, DISCLAIMER).
 *  UCRL-CODE-2002-040.
 *
 *  Written by Chris Dunlap <cdunlap@llnl.gov>
 *         and Jim Garlick  <garlick@llnl.gov>
 *         modified for SLURM by Moe Jette <jette@llnl.gov>.
 *
 *  This file is part of pam_slurm, a PAM module for restricting access to
 *  the compute nodes within a cluster based on information obtained from
 *  Simple Linux Utility for Resource Managment (SLURM).  For details, see
 *  <http://www.llnl.gov/linux/slurm/>.
 *
 *  pam_slurm is free software; you can redistribute it and/or modify it
 *  under the terms of the GNU General Public License as published by the
 *  Free Software Foundation; either version 2 of the License, or (at your
 *  option) any later version.
 *
 *  pam_slurm is distributed in the hope that it will be useful, but WITHOUT
 *  ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 *  FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 *  for more details.
 *
 *  You should have received a copy of the GNU General Public License along
 *  with pam_slurm; if not, write to the Free Software Foundation, Inc.,
 *  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301  USA.
\*****************************************************************************/

#ifndef PAM_MODULE_NAME
#  define PAM_MODULE_NAME "pam_slurm_adopt"
#endif

#if HAVE_CONFIG_H
#  include "config.h"
#endif

#include <ctype.h>
#include <errno.h>
#include <pwd.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/param.h>
#include <sys/types.h>
#include <syslog.h>
#include <unistd.h>
#include <dlfcn.h>
```

```
#include "slurm/slurm.h"
#include "src/common/slurm_xlator.h"

/*  Define the externally visible functions in this file.
 */
#define PAM_SM_ACCOUNT
#include <security/pam_modules.h>
#include <security/_pam_macros.h>


/* Define the functions to be called before and after load since _init
 * and _fini are obsolete, and their use can lead to unpredicatable
 * results.
 */
void __attribute__ ((constructor)) libpam_slurm_init(void);
void __attribute__ ((destructor)) libpam_slurm_fini(void);

/*
 *  Handle for libslurm.so
 *
 *  We open libslurm.so via dlopen () in order to pass the
 *   flag RTDL_GLOBAL so that subsequently loaded modules have
 *   access to libslurm symbols. This is pretty much only needed
 *   for dynamically loaded modules that would otherwise be
 *   linked against libslurm.
 *
 */
static void * slurm_h = NULL;

/* This function is necessary because libpam_slurm_init is called without access
 * to the pam handle.
 */
static void
_log_msg(int level, const char *format, ...)
{
  va_list args;

  openlog(PAM_MODULE_NAME, LOG_CONS | LOG_PID, LOG_AUTHPRIV);
  va_start(args, format);
  vsyslog(level, format, args);
  va_end(args);
  closelog();
  return;
}

/*
 *  Sends a message to the application informing the user
 *  that access was denied due to SLURM.
 */
extern void
send_user_msg(pam_handle_t *pamh, const char *mesg)
{
  int retval;
  struct pam_conv *conv;
  void *dummy;    /* needed to eliminate warning:
       * dereferencing type-punned pointer will
       * break strict-aliasing rules */
  char str[PAM_MAX_MSG_SIZE];
  struct pam_message msg[1];
  const struct pam_message *pmsg[1];
  struct pam_response *prsp;

  info("send_user_msg: %s", mesg);
  /*  Get conversation function to talk with app.
   */
  retval = pam_get_item(pamh, PAM_CONV, (const void **) &dummy);
  conv = (struct pam_conv *) dummy;
  if (retval != PAM_SUCCESS) {
    _log_msg(LOG_ERR, "unable to get pam_conv: %s",
       pam_strerror(pamh, retval));
    return;
  }

  /*  Construct msg to send to app.
   */
  memcpy(str, mesg, sizeof(str));
  msg[0].msg_style = PAM_ERROR_MSG;
  msg[0].msg = str;
  pmsg[0] = &msg[0];
  prsp = NULL;
  /*  Send msg to app and free the (meaningless) rsp.
   */
  retval = conv->conv(1, pmsg, &prsp, conv->appdata_ptr);
  if (retval != PAM_SUCCESS)
    _log_msg(LOG_ERR, "unable to converse with app: %s",
        pam_strerror(pamh, retval));
  if (prsp != NULL)
    _pam_drop_reply(prsp, 1);

  return;
}

/*
 * Dynamically open system's libslurm.so with RTLD_GLOBAL flag.
 * This allows subsequently loaded modules access to libslurm symbols.
 */
extern void libpam_slurm_init (void)
{
  char libslurmname[64];

  if (slurm_h)
    return;

  /* First try to use the same libslurm version ("libslurm.so.24.0.0"),
   * Second try to match the major version number ("libslurm.so.24"),
   * Otherwise use "libslurm.so" */
  if (snprintf(libslurmname, sizeof(libslurmname),
      "libslurm.so.%d.%d.%d", SLURM_API_CURRENT,
      SLURM_API_REVISION, SLURM_API_AGE) >=
      (signed) sizeof(libslurmname) ) {
    _log_msg (LOG_ERR, "Unable to write libslurmname\n");
  } else if ((slurm_h = dlopen(libslurmname, RTLD_NOW|RTLD_GLOBAL))) {
    return;
  } else {
    _log_msg (LOG_INFO, "Unable to dlopen %s: %s\n",
      libslurmname, dlerror ());
  }

  if (snprintf(libslurmname, sizeof(libslurmname), "libslurm.so.%d",
      SLURM_API_CURRENT) >= (signed) sizeof(libslurmname) ) {
```

```
    _log_msg (LOG_ERR, "Unable to write libslurmname\n");
  } else if ((slurm_h = dlopen(libslurmname, RTLD_NOW|RTLD_GLOBAL))) {
    return;
  } else {
    _log_msg (LOG_INFO, "Unable to dlopen %s: %s\n",
      libslurmname, dlerror ());
  }

  if (!(slurm_h = dlopen("libslurm.so", RTLD_NOW|RTLD_GLOBAL))) {
    _log_msg (LOG_ERR, "Unable to dlopen libslurm.so: %s\n",
        dlerror ());
  }

  return;
}

extern void libpam_slurm_fini (void)
{
  if (slurm_h)
    dlclose (slurm_h);
  return;
}
```