Administration and Programming for the Red Storm IO Subsystem

Lee Ward
Department 9223
Sandia National Laboratories
lee@sandia.gov

A practical description of the initialization, run-time configuration and application programming interface of the Red Storm IO system as found on the Cray XT3 compute partition is presented. A discussion of compatibility with POSIX and ASCI Red, together with in-depth description and discussion of the initialization, configuration, and Red Storm specific API calls as a usable management and programming reference is given.

Application Programming Interface

The Red Storm application programming interface is, primarily backward compatible with the Intel ASCI Red IO programming interface. At it's base, the IO programming interface is inspired by, but not conformant with, the IEEE POSIX interface. ASCI Red extended this interface with asynchronous calls in the data path for most read and write call variants.

For Red Storm, the interface was extended yet again in order to accomplish symmetry in the older ASCI Red set of calls and to add new functionality supporting strided accesses in memory and the file address space.

POSIX

The original ASCI Red IO subsystem was designed to support programs being ported from the traditional UNIX world. This lead directly to a desire for support of all POSIX IO calls. However, in a space-shared system such as Red and Red Storm, while much of the interface survives, many of the semantics are restricted and altered. In general the user may expect the normal semantics, though, with changes and restrictions limited to functionality that either makes no sense on a multi-program parallel (MPP) machine or was only very rarely used in the original POSIX world.

The following table lists the currently supported IO functions on Red Storm.

Catamount libsysio Application Interface	**POSIX**	**Red Support**
int chdir(const char* path)	Yes	Yes
int chmod(const char* path, mode_t mode)	Yes	Yes
int chown(const char *path, uid_t owner, gid_t group)	Yes	Yes
int close(int fd)	Yes	Yes
int creat(const char* path, mode_t mode)	Yes	Yes
int dup(int oldfd)	Yes	Yes
int dup2(int oldfd, int newfd)	Yes	Yes
int fcntl(int fd, int cmd, …)	Yes	Yes
int fdatasync(int fd)	Yes	No
int fstat(int fd, struct stat *buf)	Yes	Yes
int fsync(int fd)	Yes	Yes
int ftruncate(int fd, off_t length)	Yes	long length
int ioctl(int fd, unsigned long request, …)	int request	No
off_t lseek(int fd, off_t offset, int whence)	Yes	Yes, plus eseek() for 64 bit ofsset
int lstat(const char* path, struct stat* buf)	Yes	Yes
int mkdir(const char* path, mode_t mode)	Yes	Yes

Catamount libsysio Application Interface	POSIX	Red Support
int open(const char* path, int flag, …)	Yes	Yes
int rmdir(const char* path)	Yes	Yes
int stat(const char* path, struct stat* buf)	Yes	Yes
int symlink(const char* path1, const char* path2)	Yes	Yes
int truncate(const char *path, off_t length)	Yes	long length
mode_t umask(mode_t mask)	Yes	int rc, int mask
int unlink(const char* path)	Yes	Yes
Data transfer functions:		
ssize_t iowait(ioid_t ioid)	No	Yes
int iodone(ioid_t ioid)	No	Yes
ioid_t ipreadv(int fd, const struct iovec *iov, size_t count, off_t offset)	No	No
ioid_t ipread(int fd, void *buf, size_t count, off_t offset)	No	No
ssize_t preadv(int fd, const struct iovec *iov, size_t count, off_t offset)	No	No
ssize_t pread(int fd, void *buf, size_t count, off_t offset)	XSI ext.	No
ioid_t ireadv(int fd, const struct iovec *iov, int count)	No	No
ioid_t iread(int fd, void *buf, size_t count)	No	Yes
ssize_t readv(int fd, const struct iovec* iov, int count)	XSI ext.	No
ssize_t read(int fd, void *buf, size_t count)	Yes	Yes
ioid_t ipwritev(int fd, const struct iovec *iov, size_t count, off_t offset)	No	No
ioid_t ipwrite(int fd, const void *buf, size_t count, off_t offset)	No	No
ssize_t pwritev(int fd, const struct iovec *iov, size_t count, off_t offset)	No	No
ssize_t pwrite(int fd, cont void *buf, size_t count, off_t offset)	Yes	No
ioid_t iwritev(int fd, const struct iovec *iov, int count)	No	No
ioid_t iwrite(int fd, const void* buf, size_t count)	No	Yes
ssize_t writev(int fd, const struct iovec *iov, int count)	Yes	No
ssize_t write(int fd, const void *buf, size_t count)	Yes	Yes

Those calls that are not POSIX and not supported on Red are discussed later.

The POSIX 64-bit variants are also defined, as well as the types such as `off64_t`. However, since Red Storm is already a 64-bit machine, the 64-bit variants are just aliases for those listed.

It is important to note that various file system drivers may not support some of the above described functions. For instance, the **incore** driver does not support the ability to symbolically link names in the namespace. In these cases, an appropriate error return is made. Usually, `ENOSYS`.

ASCI Red Compatibility

As was mentioned previously, Red Storm is designed to provide a backward compatible interface to existing applications from ASCI Red. Unfortunately, this was not entirely possible as standards have evolved since Red's manufacture. Notably, all IO calls now conform to the IEEE POSIX draft, version 6. In the main, any required changes will be limited to certain basic types such as `size_t` and `ssize_t`. A relaxed compilation of he user application should, in general, produce many warnings but a proper binary. Still, it's a good idea to update the older arguments to match the new prototypes.

However, if the ASCI Red extended interface supporting asynchronous IO calls is used, the Red Storm application will have to be modified. The asynchronous calls all return an `ioid_t` typed value now instead of the previous `int`. This change was made to accommodate direct reference to an internally defined record.

`xtio.h`

A new include file must be referenced by the application source in order to make use of the ASCI Red, and Red Storm, extensions. This new include file is founbd in the standard system includes directory and is called `xtio.h`. Note, however, that if a POSIX conformant application is being ported or a new application does not use any of the extended functions then there is no need to include this header.

New Functionality for Red Storm

The **SYSIO** project for Red Storm, in addition to updating the existing interface to match the current POSIX specification, also completed the ASCI Red extensions by providing all the asynchronous IO variants for new calls that were introduced by that standard. For instance, previously there were calls to iread and iwrite. There are extensions now for pread, ipread, readv, and ireadv.

Completely new in the Red Storm IO system is direct support for scatter/gather IO between application memory and the file address space. This goes beyond the POSIX readv and writev calls.

The interface to this new functionality is available via readx, writex, ireadx, and iwritex. All of these calls take a one-dimensional array to an "extent IO" structure, which looks like:

```
/*
 * Structure for strided I/O.
 */
struct xtvec {
        off_t   xtv_off;       /* Stride/Extent offset. */
        size_t  xtv_len;       /* Stride/Extent length. */
};
```

Each record defines an offset and extent in the file address space to which the operation applies. As well, the scatter/gather in memory is specified by the usual `struct uio` as used by readv, and writev.

The two vectors, extent and uio, are reconciled internally. The described regions do not have to match. Data is scattered or gathered, as appropriate, in order. So, a short region in one array may deposit data in two, or more, regions described by the other. The shortest number of bytes described by either array limits the total size of the data transfer. In other words, while processing, whichever array end is reached first stops the process.

SYSIO incorporates much internal machinery devoted to reconciliation and supporting file systems that do not support such an interface. Some file systems, though, provide for the direct support of such an interface and is indeed the optimal way to request a data transfer. Notable file systems supporting this are Lustre, PVFS, and PVFS II.

Architecture

SYSIO is a classic virtual file system (VFS) design. This approach is documented in many operating system books. In a nutshell, though, the idea is that all contemporary file systems provide for the same kinds of things; a hierarchical namespace and a linear, ordered file address space. These common concepts and semantics are abstracted by **SYSIO** to provide generic access to such services and a common method for accessing them from the application. All in an attempt to allow the application to use multiple file system implementations, potentially simultaneously.

On ASCI Red, the service section was a signle-system image implementation. The approach there was to function-ship everything to the service section, then. This made for a very light-weight implementation on the client-side as the VFS lay entirely within the operating system on the service section. Call arguments were bundled up, shipped to the service section and interpreted there.

On Red Storm, such an implementation is no longer possible. The service section consists of a group of nodes using running distinct operating system images. Therefore, the compute node must manage interaction with the IO services directly. Worse, traversal of the name space graph (parsing application specified paths) must be done locally now. Allowing the login service partition to parse these paths may easily result in information for which the compute node would lack the context to apply in any meaningful way.

The advantages to this approach are numerous. For instance the login service section is almost completely off-loaded potentially. Where before, in ASCI Red, significant action was required by the login service section the Red Storm model retains all of that processing on the client. This maximizes scalability. It comes at a price, though.

The primary disadvantage of such a design is that it is impossible to maintain a truly global namespace. It is only through administrative action the even the appearance of a global namespace is offered on Red Storm. In general, with proper care, this should be of little or no concern to applications. The administrator will have arranged for directories and mounts to occur in the same place, everywhere on the login service nodes and compute nodes, both.

Base File System Drivers

SYSIO offers a small, basic set of file system services. On Red Storm, this set is limited to a "yod" interface driver, an incore (in memory resident file system) driver, and a small driver that supports the pre-opened file descriptors for applications.

While Lustre is not a part of SYSIO, proper, on Red Storm it is pre-linked to the libsysio system library and is mounted and accessed just as the basic set of drivers. The application and the administrator need not distinguish between the two.

The yod driver

Catamount is derived from the original SUNMOS project and retains the original file and file system interface. This is the basic, usually slow but always functional, file interface in all Sandia MPP implementations.

However, this interface does not support asynchronous operation. The service is handled by a single, and single-threaded, process associated with the application in the service section of the machine. As such, all calls and all data transfer is serialized. Worse, since the service section is probably using remote IO services itself, the yod service handler is often acting as a proxy, imposing additional overhead and slowing things even more.

The incore driver

The incore driver uses local compute node memory to offer a remedial "RAM-FS". It was motivated by and designed to allow the administrator to craft a local "root" for each node in order to avoid mount storms. It supports only the most basic functions of a file system. Notably, it does not support symbolic links.

The administrator makes use of this file system in order to provide basic structure at the very top of the namespace tree. Through the clever use of explicit mounts, automounts, and "bound" mounts, a farily faithful illusion of a global namespace may be created.

The stdfd driver

The standard file descriptors driver solves a bootstrap problem for **SYSIO**. While the yod driver is used for all basic file IO interaction with the launcher process, including standard input, output, and error, there is no assumption by the **SYSIO** core as to where they are. This driver, then, provides the administrator the mechanism to link with the pre-opened descriptors in the launcher through mount-time options.

Administration

Administration for **SYSIO** is limited to the creation and maintenance of a Red Storm environment variable. This all-important variable, passed at application launch, crafts the root of the compute node file system namespace. Until **SYSIO** reads and successfully parses the environment variable it has no way to do anything useful.

Crafting and debugging the initialization environment variable is, in practice, difficult. If it must be modified it is worth noting that **SYSIO** can be, and should have been, built with tracing. If it was, errors while processing this file are noted on the system console and the application will fail to start. Without tracing, the application still fails to start but does so silently.

Automounts

In practice, while many directories and file systems might be available to an application, it will only actually use a very small number of them. As well, gaining access to remote services usually entails a long, complicated conversation between the compute node and the IO service section. On a general purpose computer, this is not a problem as it typically only occurs at boot. However, because **SYSIO** is linked with the application, the perspective of the IO system is that every job launch looks exactly like a node boot.

To remove the need for pre-mounting all the different file system possibilities a way was required to limit mount activity to only those file systems an application might need. This was accomplished by incorporating an automount feature directly from Lustre.

In this automount scheme, the directive lies within the file system as a special file. The directive is only accessed and the mount attempted when a path is encountered that references something in, or below, a specially hinted directory containing the special file. At that time, **SYSIO** will attempt the automount and, if successful, reparse the relevant part of the path. If the mount fails, no error is flagged. The automount directory is simply treated as a normal directory within the file system in which it was found.

To create an automount directory on Red Storm:

1. Make sure the parent directory has automounts enabled. The **MOUNT_F_AUTO** (mask is 0x02) bit should be set.
2. Find or create an empty directory.
3. Create a file called ".mount".
4. Initialize the content of the ".mount" file with the mount data.

5. Add the **SETUID** bit to the parent directory permissions. This may be accomplish with the “+s” specifier to the chmod program.

The mount data in the “.mount” file has the format:

```
<fstype>:<source>[[ \t]+<comma-separated-mount-options>]
```

The <fstype> is the name of the file system as identified by the driver writer. A few have already been mentioned; yod, incore, stdfd, lustre. Others are possible as well.

The <source> identifies where the file system being mounted comes from the to the targeted file system driver. Each driver defines the format of this argument itself.

Similar to the <source> argument, options are local to, and defined by, the targeted driver.

Bound Mounts

The concept of a bound mount is to allow one mount of a file system to appear in multiple places within the application namespace. It provides a level of sharing within SYSIO that is not possible otherwise except via symbolic links, which are very expensive to deal with.

In practice, on Red Storm, this is used primarily to craft a localized subdirectory within an incore file system at boot time and then, later, provide access to it under a remotely mounted file system. For instance, the supplied C-language runtime library makes frequent reference to timezone information. A bound mount is used to supply this information as a local file, significantly reducing remote calls to the IO service section.

The file system type name for this pseudo-filesystem is “sub” and any sub-tree may be mounted elsewhere. It is not restricted to the root of the source file system. An automount example might be:

```
sub:/_hidden/directory
```

Initialization and Startup

_sysio_init

This function initializes the **SYSIO** core and is called by the Red Storm runtime startup function directly. It should not be called again.

_sysio_boot

All parameterization and runtime configuration is made through this routine. It may be called multiple times. Three options are recognized and they are “trace”, “namespace”, and “cwd”.

The trace option is used for enabling or disabling an entry point trace function. All normal user calls will print messages upon entry and exit. While this is primarily useful for debugging **SYSIO** itself, it may also be useful for debugging and tracing IO behavior in applications. It takes one argument, an integer represented as a string. The parsed value of that string enables tracing if non-zero and disables it if zero.

The namespace option is used for crafting initial namespace in SYSIO. It has numerous arguments and options. In general, it is a set of brace-delimited (‘{‘ and ‘}’) directives. Each directive takes the form:

{ <cmd>, opt=val[,…] }

Command name: creat

Purpose: Create a new file, directory, or special file

Arguments:

Name	Optional?	Default Value	Allowed values	Purpose
ft	N	None	dir (directory) blk (block device) chr (character device) file (regular file)	Specifies the type of create to be performed
nm	N	None	Any valid file name/path	Specifies the name/path of the newly created object
pm	N	None	Any valid numeric permission	Specifies the permissions for the newly created object
ow	Y	Owner of calling process (usually root)	Any system-known numeric user id	Specifies the owner of the new object
gr	Y	Group of calling process	Any system-known numeric group id	Specifies the group for the new object
mm	N	None	Any valid major/minor combo. It takes the form of either major+minor or of the result of the (major) <<MAJOR_SHIFT|minor calculation	Specifies the major and minor number for a character or block device
str	Y	None	Any quoted string	Puts the given string into the newly created file. Only valid for ft=file

Command name: mnt

Purpose: Mounts a file system

Arguments:

Name	Optional?	Default Value	Allowed values	Purpose

dev	N	None	Combination of fstype:src where fstype is a valid file system type and src is a valid path name	Specifies the source to be mounted
dir	N	None	Any valid path which can be mounted on	Specifies the destination of the mount. If the destination is "/", the mount is recognized as a root mount.
fl	Y	0	Any unsigned integer	Specifieds the flags for the mount
da	Y	NULL	No restrictions	This is the data value, used to pass in filesystem-specific information

Command name: chmd

Purpose: Changes permissions on a file or directory

Arguments:

Name	Optional?	Default Value	Allowed values	Purpose
src	N	None	Any valid file or directory	File or directory to change permissions on
pm	N	None	Any valid numeric permissions	Permissions to set the src to

The cwd option is used to set the application initial working directory. There is one option, the absolute pathname of the working directory. Note that this value is only stored and not actually referenced until the application makes use of a relative path name. At that time, the stored value is parsed and the initial working directory is set. If the parsing of that initial value fails, **SYSIO** will not function and the application will get back strange errors, potentially, all having to do with the initial working directory path and not the path argument directly supplied.

Shutdown

In practice, the application should not have to worry about this as it is executed implicitly at graceful exit. However, applications hang and crash. When that happens many parts of the IO system may be left in a strange state for a short time. All file system drivers are expected to deal with this situation. Some, though, record state which must time out on the service side. An immediate relaunch of a failed application may experience delays while starting because of this.

User Extensible File Systems

The basic set of file system drivers are implemented as modules. This allows **SYSIO** to support file systems that were not yet written or were not anticipated at the time it was crafted. In fact, the Red Storm run time startup system makes use of this to include the Lustre file system driver transparent to the application.

An application may include other drivers itself. The application must call the driver’s initialization function and that function, in turn, will register the relevant entry points with the SYSIO file system switch. After that, the extension may be accessed and used just like any other driver. Mounts, automounts, etc. should all work as expected.

This is really only for the adventurous and writing or debugging a new file system driver on Red Storm itself is not advisable.