POSIX I/O High Performance Computing Extensions

Brent Welch (Speaker)
Panasas

www.pdl.cmu.edu/posix/

December 14, 2005

APIs for HPC IO

- POSIX IO APIs (open, close, read, write, stat) have semantics that can make it hard to achieve high performance when large clusters of machines access shared storage.
- A working group (see next slide) of HPC users is drafting some proposed API additions for POSIX that will provide standard ways to achieve higher performance.
- Primary approach is either to relax semantics that can be expensive, or to provide more information to inform the storage system about access patterns.

Contributors

- Lee Ward - Sandia National Lab
- Bill Lowe, Tyce McLarty – Lawrence Livermore National Lab
- Gary Grider, James Nunez – Los Alamos National Lab
- Rob Ross, Rajeev Thakur, William Gropp - Argonne National Lab
- Roger Haskin – IBM
- Brent Welch, Marc Unangst - Panasas
- Garth Gibson- CMU/Panasas
- Alok Choudhary – Northwestern U
- Tom Ruwart- U of Minnesota/IO Performance
- Others
- www.pdl.cmu.edu/posix/

POSIX Introduction

- POSIX is the IEEE Portable Operating System Interface for Computing Environments.
- “POSIX defines a standard way for an application program to obtain basic services from the operating system”
- The Open Group (http://www.opengroup.org/)
- POSIX was created when a single computer owned its own file system.
  - Network file systems like NFS chose not to implement strict POSIX semantics in all cases (e.g., lazy access time propagation)
  - Heavily shared files (e.g., from clusters) can be very expensive for file systems that provide POSIX semantics, or have undefined contents for file systems that bend the rules
- The goal is to create a standard way to provide high performance and good semantics

Current HPC POSIX Enhancement Areas

- Ordering (stream of bytes idea needs to move towards distributed vectors of units)
  - readx(), writex()
- Coherence – (last writer wins and other such things can be optional)
  - lazyio_propogate(), lazyio_synchronize()
- Metadata (lazy attributes issues)
  - statlite()
- Locking schemes for cooperating processes
  - lockg()
- Shared file descriptors (group file opens)
  - openg(), sutoc()
- Portability of hinting for layouts and other information (file system provides optimal access strategy in standard call)
  - ? (no API yet)

statlite, fstatlite,lstatlite – Optional Attributes

- **Syntax**
  - int statlite(const char **file_name*, struct statlite **buf*);
    int fstatlite(int *filedes*, struct statlite **buf*);
    int lstatlite(const char **file_name*, struct statlite **buf*);
- **Description**
  - This family of stat calls, the lite family, is provided to allow for file I/O performance not to be compromised by frequent use of stat information lookup. Some information can be expensive to obtain when a file is busy.
  - They all return a *statlite* structure, which has all the normal fields from the stat family of calls but some of the fields (e.g., file size, modify time) are optionally not guaranteed to be correct.
  - There is a litemask field that can be used to specify which of the optional fields you require to be completely correct values returned.

statlite, fstatlite,lstatlite (cont.)

- **Syntax**
  - int statlite(const char **file_name*, struct statlite **buf*);
    int fstatlite(int *filedes*, struct statlite **buf*);
    int lstatlite(const char **file_name*, struct statlite **buf*);
- **Description**
  - **statlite** stats the file pointed to by *file_name* and fills in *buf*.
  - **lstatlite** is identical to **statlite**, except in the case of a symbolic link, where the link itself is statlite-ed, not the file that it refers to.
  - **fstatlite** is identical to **stat**, only the open file pointed to by *filedes* (as returned by **open**(2)) is statlited-ed in place of *file_name*.

struct statlite

```
struct statlite {
   dev_t st_dev;       /* device */
   ino_t st_ino;       /* inode */
   mode_t st_mode;     /* protection */
   nlink_t st_nlink;   /* number of hard links */
   uid_t st_uid;       /* user ID of owner */
   gid_t st_gid;       /* group ID of owner */
   dev_t st_rdev;      /* device type (if inode device)*/
   unsigned long st_litemask; /* bit mask for optional field accuracy */
   /* Fields below here are optionally provided and are
      guaranteed to be correct only if there corresponding bit
      is set to 1 in the manditory st_litemask field, with the lite
      versions of the stat family of calls */
   off_t st_size; /* total size, in bytes */
   blksize_t st_blksize; /* blocksize for filesystem I/O */
   blkcnt_t st_blocks;   /* number of blocks allocated */
   time_t st_atime;      /* time of last access */
   time_t st_mtime;      /* time of last modification */
   time_t st_ctime;      /* time of last change */
   /* End of optional fields */
};
```

Mask indicates what is valid:

Sizes and Times Optional

POSIX ACLs –> New NFSv4 Semantics

- Legitimize NFSv4 ACLs in POSIX, allowing users to choose methodology and over time maybe POSIX ACLs will fade away.
  - Note that “POSIX ACLS” are really only a proposed part of the standard and not widely implemented or used
  - NFSv4 ACLs are aligned with the Windows ACL model, which is more widely used and more sensible
  - The two models differ in how ACLs are inherited, and in the rules for processing a long set of ACE (access control entries)
  - Old POSIX ACL model often considered broken
- draft-falkner-nfsv4-acls-00.txt is an Internet Draft from Sun that explains how they are exposing NFSv4 ACLs for Solaris 10.

NFSv4 ACLS

```
Permission letter mapping:
r - NFS4_ACE_READ_DATA
w - NFS4_ACE_WRITE_DATA
a - NFS4_ACE_APPEND_DATA
x - NFS4_ACE_EXECUTE
d - NFS4_ACE_DELETE
l - NFS4_ACE_LIST_DIRECTORY
f - NFS4_ACE_ADD_FILE
s - NFS4_ACE_ADD_SUBDIRECTORY
n - NFS4_ACE_READ_NAMED_ATTRS
N - NFS4_ACE_WRITE_NAMED_ATTRS
D - NFS4_ACE_DELETE_CHILD
t - NFS4_ACE_READ_ATTRIBUTES
T - NFS4_ACE_WRITE_ATTRIBUTES
c - NFS4_ACE_READ_ACL
C - NFS4_ACE_WRITE_ACL
o - NFS4_ACE_WRITE_OWNER
y - NFS4_ACE_SYNCHRONIZE
```

lockg – Share mode lock for cluster apps

- **Syntax**

  int lockg(int fd, int cmd, lgid_t *lgid);

- **Description**
  - Apply, test, remove, or join a POSIX group lock on an open file. Group locks are exclusive, whole-file locks that limit file access to a specified group of processes. The file is specified by fd, a file descriptor open for writing and the action by cmd.
  - The first process to call lockg() passes a cmd of F_LOCK and an initialized value for lgid. Obtaining the lock is performed exactly as though a lockf() with pos of 0 and len of 0 were used (i.e. defining a lock section that encompasses a region from byte position zero to present and future end-of-tile positions). An opaque lock group id is returned in lgid. This lgid may be passed to other processes for the purpose of allowing them to join the group lock.

lockg (Continued)

- **Description (Continued)**
  - Processes wishing to join the group lock call lockg() with a cmd of F_LOCK and the lgid returned to the first process. On success this process has registered itself as a member of the group of the group lock.
  - Valid operations are given below:
    - F_LOCK Set an exclusive lock
    - F_TLOCK Same as F_LOCK but the call never blocks
    - F_ULOCK Unlock the indicated file.
    - F_TEST Test the lock

readdirplus & readdirlite – read dir and attributes

- **Syntax**

  struct dirent_plus *readdirplus(DIR **dirp*);

  int readdirplus_r(DIR **dirp*, struct dirent_plus **entry*, struct dirent_plus ***result*);

  struct dirent_lite *readdirlite(DIR **dirp*);

  int readdirlite_r(DIR **dirp*, struct dirent_lite **entry*, struct dirent_lite ***result*);

- **Description**
  - **readdirplus**(2) and **readdirplus_r**(2) return a directory entry plus **lstat**(2) results (like the NFSv3 READDIRPLUS command)
  - **readdirlite**(2) and **readdirlite_r**(2) return a directory entry plus **lstatlite**(2) results

readdirplus & readdirlite (Continued)

- **Description (Continued)**
  - Results are returned in the form of a *dirent_plus* or *dirent_lite* structure:

```
struct dirent_plus {
 struct dirent     d_dirent;  /* dirent struct for this entry */
 struct stat       d_stat;    /* attributes for this entry */
 int               d_stat_err;/* errno for d_stat, or 0 */
};
struct dirent_lite {
 struct dirent     d_dirent;  /* dirent struct for this entry */
 struct statlite   d_stat;    /* attributes for this entry */
 int               d_stat_err;/* errno for d_stat, or 0 */
};
```

  - If *d_stat_err* is 0, *d_stat* field contains **lstat**(2)/**lstatlite**(2) results
  - If **readdir**(2) phase succeeds but **lstat**(2) or **lstatlite**(2) fails (file deleted, unavailable, etc.) *d_stat_err* field contains errno from stat call
  - **readdirplus_r**(2)/**readdirlite_r**(2) variants provide thread-safe API, similar to **readdir_r**(2)

Lazy I/O data integrity

- Specify O_LAZY in *flags* argument to **open**(2)
- Requests lazy I/O data integrity
  - Allows network filesystem to relax data coherency requirements to improve performance for shared-write file
  - Writes may not be visible to other processes or clients until **lazyio_propagate**(2), **fsync**(2), or **close**(2) is called
  - Reads may come from local cache (ignoring changes to file on backing storage) until **lazyio_synchronize**(2) is called
  - Does not provide synchronization across processes or nodes – program must use external synchronization (e.g., pthreads, XSI message queues, MPI) to coordinate actions
- This is a hint only
  - if filesystem does not support lazy I/O integrity, does not have to do anything differently

lazyio_{propagate,synchronize}

- Syntax

  int lazyio_propagate(int *fd*, **off_t** *offset*, **size_t** *count*);

  int lazyio_synchronize(int *fd*, **off_t** *offset*, **size_t** *count*);

- Description
  - **lazyio_propagate**(2) ensures that any cached writes in the specified region have been propagated to the shared copy of the backing file.
  - **lazyio_synchronize**(2) ensures that the effects of completed propagations in the specified region from other processes or nodes, on any file descriptor of the backing file, will be reflected in subsequent **read**(2) and **stat**(2) calls on this node.
    - Some implementations may accomplish this by invalidating all cached data and metadata associated with the specified region, causing it to be re-fetched from the shared backing file on subsequent accesses.
    - However, cache invalidation is not guaranteed, and a compliant implementation may choose to only re-fetch data and metadata actually modified by another node.
  - If *offset* and *count* are both 0, the operation is performed on the entire file. Otherwise, the operation may (but is not guaranteed to) be restricted to the specified region.

Lazy I/O Example

```
fd = open("/shared/file", O_RDWR | O_LAZY);
for(i = 0; i < niters; i++) {
  /*
   * some computation generating data for the
   * shared file
   */
  compute(buf, buflen);
  /*
   * in the intended use concurrent writes on
   * different file descriptors are applied to
   * non-overlapping regions
   */
  lseek(fd, output_base+(node*i*buflen),
    SEEK_SET);
  write(fd, buf, buflen);
  /*
   * before any other file descriptor can be
   * certain that the backing file is up to
   * date, changes associated with all file
   * descriptors must be propagated
   */
  lazyio_propagate(fd,
     output_base+(node*i*buflen),  buflen);
  non_filesystem_provided_barrier();
```

```
  /*
     * before any file descriptor can be
     * certain that it can see all propagated
     * changes it must be certain that it is
     * not caching stale data or metadata
     */
    lazyio_synchronize(fd,
         input_base+(node*i), buflen);
    lseek(fd, input_base+(node*i), SEEK_SET);
    read(fd, buf, buflen);
    compute(buf, buflen);
    /*
     * must barrier() returning to the write
     * phase at the top of the loop to avoid
     * overwriting a region of the shared file
     * still being read through another
     * file descriptor.
     */
    non_filesystem_provided_barrier();
}
close(fd);
```

openg – Map file name to portable file handle

- **Syntax**
  - int openg(char *path, int mode, fh_t *handle);
- **Description**
  - The *openg*() function opens a file named by path according to mode (e.g., O_RDWR). It returns an opaque file handle corresponding to a file descriptor. The intent is that the file handle can be transferred to cooperating processes and converted to a file descriptor with sutoc().
  - The lifetime of the file handle is implementation specific. For example, it may not be valid once all open file descriptors derived from the handle with sutoc() have been closed.

sutoc (or fhtofd) – map file handle to file descriptor

- **Syntax**
  - int sutoc(fh_t *fh);
- **Description**
  - The *sutoc*() function shall establish the connection between a file handle and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The *fh* argument points to a file handle referring to the file.
  - The *sutoc*() function shall return a file descriptor for the referred file that is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor shall not share it with any other process in the system. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

sutoc (or fhtofd)

- **Syntax**
  - int sutoc(fh_t *fh);
- **Description**
  - The file offset used to mark the current position within the file shall be set to the beginning of the file.
  - The file status flags and file access modes of the open file description shall be set according to those given in the accompanying *openg()*.
  - The largest value that can be represented correctly in an object of type **off_t** shall be established as the offset maximum in the open file description.

readx writex – memory vector to/from file vector

- **Syntax**
  - ssize_t readx(int fd, const struct iovec *iov, size_t iov_count, struct xtvec *xtv, size_t xtv_count);
  - ssize_t writex(int fd, const struct iovec *iov, size_t iov_count, struct xtvec *xtv, size_t xtv_count);

- **Description**
  - Generalized file vector to memory vector transfer. Existing readv(), writev() specify a memory vector and do serial IO. The new readx(), writex() calls also read/write strided vectors to/from files.
  - The readx() function reads xtv_count blocks described by xtv from the file associated with the file descriptor fd into the iov_count multiple buffers described by iov. The file offset is not changed.
  - The writex() function writes at most xtv_count blocks described by xtv into the file associated with the file descriptor fd from the iov_count multiple buffers described by iov. The file offset is not changed.

readx writex (Continued)

- **Description (Continued)**
  - The file referenced by fd must be capable of seeking. The pointer iov points to a struct iovec

```
struct iovec {
   void *iov_base;    /* Starting address */
   size_t iov_len;    /* Number of bytes */
};
```

  - The pointer xtv points to a struct xtvec defined as

```
struct xtvec {
  off_t xtv_off;     /* Starting file offset */
  size_t xtv_len;    /* Number of bytes */
};
```

  - The offsets described in xtv are relative to the start of the file. It is not required that iov and xtv have the same number of elements. Elements in iov and xtv may overlap. Regions are processed in any order.

Layout control

- Standard way to tell the file system how the file should be laid out geometrically, like width, stripe, depth, RAID level, etc.
  - Stripe width: number of storage devices in a RAID stripe
  - Stripe unit size: number of bytes written to one storage device before advancing to the next device in the stripe
  - Depth: how much data to write to a particular storage device (i.e., how many stripes) before picking a different set of devices for the next group of stripes
  - Stride: interleaving distance of expected access patterns
  - RAID-level. Mirroring, RAID 5, RAID-0, RAID-10, Double-parity, etc.
- No API defined so far

POSIX HPC IO

- **`readx, writex`** – memory vector to/from file vector
- **`lazyio_propogate, lazyio_synchronize, O_LAZY`**
- **`lockg`** – share mode lock for cluster applications
- **`openg, sutoc`** – expose file handles to applications
- **`readdirplus, readdirlite`** – expose NFS op to applications
- **`statlite, fstatlite`** – optional attributes
- www.pdl.cmu.edu/posix/