NAME

open - open a file

DESCRIPTION

O_LAZY

When a file is opened with the O_LAZY flag, I/O operations on the file descriptor complete as defined by lazy I/O data integrity.

Lazy I/O data integrity:

Lazy I/O mode is intended to be used by a group of nodes accessing a file cooperatively. In lazy I/O mode, POSIX consistency semantics are relaxed to allow the underlying network file system to achieve a higher level of performance than would be possible if strict POSIX data and metadata consistency was required.

When a file is opened in lazy I/O mode, the following differences from standard POSIX I/O semantics apply:

1) Writes may be cached by the current process or the local node. Data written by this process may not be immediately visible to other processes or other nodes until an explicit lazyio_propagate().

2) Reads may be cached by the local process or local node. Re-reading a previously fetched region may return stale data, and may not reflect the current contents of the shared file, unless there has been no write on that region after an explicit lazyio_synchronize().

3) Attribute consistency follows data consistency. Attributres may be cached by the local process or local node. Refetching the file attributes may not reflect the current attributes of the shared file, unless there has been no change in the backing file after an explicit lazyio_synchronize().

Note that in all cases, lazy mode ALLOWS the operating system to cache reads or writes, but does not REQUIRE it. That is, the operating system is free to write data to the shared file before an explicit propagate, and is free to re-read data from the shared file without receiving an explicit synchronize, but is not required to do so. One implication of this is that when a file is open in lazy mode, the behavior of reading a section of a file that has been written by another node since the last propagate/synchronize round is implementation-specific. That is, the read may or may not return the data written until the next synchronization round.

No provisions are made for explicitly synchronizing the actions of a group of nodes. The expectation is that the application provides its own synchronization/message-passing mechanisms (e.g., via MPI). In the examples given later, this is illustrated with a barrier call that causes nodes to wait until all nodes have reached that point. There are other possible synchronization schemes.

If a process attempts to open a file in lazy mode that is already open by another process/node in non-lazy mode, or vice versa, the results are implementation-specific. If such a mixed-mode open is permitted, the semantics of read(2), stat(2), and write(2) with regard to data and attribute consistency are implementation-specific. The most likely result is that all users of the file experience lazy I/O consistency.

SEE ALSO

lazyio_propagate(2), lazyio_synchronize(2)

NAME

```
  lazyio_propagate   - propagate local changes to a shared file
  lazyio_synchronize - synchronize with remote changes to a file
```

SYNOPSIS

```
  #include <lazyio.h>

  int lazyio_propagate(int fd, off_t offset, size_t count);
  int lazyio_synchronize(int fd, off_t offset, size_t count);
```

DESCRIPTION

The lazyio_propagate() and lazyio_synchronize() calls are used to synchronize access to a shared file open in O_LAZY mode that is being read and written by multiple nodes.

In the special case where 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.

lazyio_propagate() ensures that any cached writes in the specified region have been propagated to the shared copy of the backing file.

lazyio_synchronize() 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) or 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 only re-fetch data and metadata actually modified by another node.

Both lazyio_propagate() and lazyio_synchronize() make assertions only for the file descriptor on which they are invoked. It is important to note that the use of lazyio_propagate() on the specified region does not inhibit an implementation from propagating previously written data or changed metadata associated with the specified region to the backing file at any other time. Likewise, it is important to note that after the completion of lazyio_synchronize() on a specified file descriptor, subsequent read(2) or stat(2) operations may observe the effects of any subsequent change in data or metadata associated with the specified region on any file descriptor of the backing file, and these subsequent changes could be observed in any order. See EXAMPLES.

EXAMPLES

These calls are intended for use by a parallel application reading/writing a shared file in a distributed filesystem. Note that the barrier call is not provided by this set of APIs, but is provided by some other parallel programing library. A sample I/O loop would look like:

```
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 in to avoid overwriting a region of the shared file
     * still being read through another file descriptor.
     */
    non_filesystem_provided_barrier();
  }
  close(fd);
```

RETURN VALUE

Both lazyio_propagate() and lazyio_synchronize() return 0 on success, and -1 on error, with errno set appropriately.

ERRORS

The following errors may be returned by lazyio_propagate() and lazyio_synchronize().

EBADF
fd is not a valid file descriptor.

EINVAL
fd is attached to an object which is unsuitable for reading or writing, or was not opened with O_LAZY.

EFBIG
The specified region is beyond the maximum allowed file offset.

ENOSPC
The device containing the file referred to by fd has no room for some or all of the data written.

EAGAIN
The operation would block.

EINTR
The operation was interrupted.

EIO
A low-level I/O error occurred while writing or synchronizing the data.

SEE ALSO

open(2), close(2), fsync(2), lseek(2), read(2), write(2)