File I/O Calls

Class Notes

File I/O calls:

Linux kernel supports the following types of file I/O operations

- Standard I/O
- Synchronized I/O
- Direct I/O
- Read ahead I/O
- Memory mapped I/O
- Vectored I/O

Standard I/O

1) Read

```
#include<unistd.h>
ssize_t  read (int fd, void * buf ,size_t count);
```

Read api internally invokes VFS system call sys_read, which inturn steps into FS specific read call. FS read operation executes the following steps.

- Looks up into specified file node to identify data block number
- Verifies for the data of specified block in the I/O cache
- Invokes storage driver and initiates physical transfer of the data from the specified block.
- Polls I/O cache for arrival of requested data and transfers the data to application when received.

2) Write

```
#include<unistd.h>
ssize_t  write (int fd, void * buf ,size_t count);
```

- Write API invokes FS specific write operation through VFS system call sys_write.
- FS write transfer the data from the application and writes it to specified files buffer located in IO cache.
- Initiate disk sink operation.

Synchronized I/O:

This mode of I/O allows applications to force the file systems to update the target disc immediately when a file write is initiated. The write operations of the application returns only after the disc is updated.

Sample code:

```
fd= open("Demo.txt",O_RDWR |O_SYNC, S_IRWXU |S_IRWXG);
```

Direct I/O:

This mode of I/O disables the kernel I/O cache and allows application to setup usermode buffer that can be used as an I/O cache.

```
#include <stdlib.h>

int posix_memallign (void ** memptr, size_t allignment ,size_t size);
```

Read ahead I/O:

- FS are designed to instruct storage drivers to carry out data pre-fetch operations while transferring data from disc to I/O cache.
- Pre-fetched data could be useful to serve subsequent I/O request of an application and avoid/minimize physical I/O transfer operations.
- Applications can instruct file systems data access pattern on a specified file which in-turn is used by a file system as a parameter to configure pre-fetch limit (read ahead limit).

Sample code:

```
rev=posix_fadvise(fd, 0,20,POSIX_FADV_SEQUENTIAL);  /* Enable read ahead */

posix_fadvise (fd ,0,20,POSIX_FADV_RANDOM);  /* Disable read ahead */
```

Memory mapped file I/O (mmap)

- This method allows application to directly access buffer associated with a file without the need of system calls.
- Mmap invokes FS specific operations which appends a new page table entry, granting access to file buffer.

Sample code:

```
File data=(char *)mmap(void*)0, 100,PROT_READ |PROT_WRITE |MAP_PRIVATE ,fd,0);
```

- Application can indicate data access pattern on a memory mapped file using madvise.
- Madvise takes the address of IO cache as an argument along with access pattern constant.

```
madvise(temp,100,MADV_SEQUENTIAL);
```

Vectored I/O:

- This is widely used in user space drivers.

```
#include <sys/uio.h>

Ssize_t readv(int fd ,const struct iovec *iov ,int iovcnt);

Ssize_t writev(int fd ,const struct iovec *iov ,int iovcnt);
```

- The readv() function works just like read except that multiple buffers are filled.
- The writev() function works just like read except that multiple buffers are written out.

Multiplexed I/O:

- This mode helps an application to perform IO operations on a group of file descriptors based on readiness of the descriptor instead of sequence of the I/O calls.
- Linux provides three separate API 's for multiplexed I/O

A. Select:

```
#include <sys/select.h>
Int select (int nfds , fd _set*readfds ,fd_set*writefds, fd_set*exceptfds,struct timeval
*timeout);
```

Usage:

1) Identify total number of file descriptors to be monitored

```
int fd1,fd2;
fd1=open("./pone",O_RDWR);
fd2=open("./ptwo",O_RDWR);
```

2) Allocate file descriptor categories based on the event to be monitored.

```
fd_set read_set,write_set,ecept_set;
FD_ZERO(&read_set);
FD_SET(fd1,&read_set);
FD_SET(fd2,&read_set);
```

3) Initiate watch for the specific event using **select** API.

```
struct timeval timeout;
n=select (FD_SETSIZE, &read_set,NULL,NULL,&timeout);
```

4) Verify each fd and execute I/O operation

```
FD_ISSET (fd1, &read_set);

FD_ISSET (fd2, &read_set);
```

B. Poll:

```
#include<poll.h>

int poll (struct pollfd*fds, nfds_t nfds ,int timeout);

struct pollfd {

        int fd;

        short events;

        short revents;

};
```

- Both select and poll copy file descriptors to be monitored into kernel space and then setup watch.
- If an application intends to monitor fds using select and poll, using an infinite loop. Each iteration of the loop would begin with select or poll calls, which will copy the fd into kernel mode.
- Linux provides an alternate optimized I/O multiplexing call called **epoll**.

C. Epoll:

- Epoll interface provides two separate functions to copy file descriptor objects and watch them for a specific event.

Steps:

1) Allocate kernel buffer to copy epoll_event instances.

```
int epfd;

epfd =epoll_create (SIZE);
```

2) Allocate instance of epoll_event, initialize with fd details and add it to epoll_buffer.

```
struct epoll_event ev;
ev.evnts = EPOLLIN;
ev.data.fd = fd1;
epoll_ctl(epfd,EPOLL_CTL_ADD , fd1,&ev);
```

3) Set up watch

```
struct epoll_event evlist [COUNT]
ready=epoll_wait (epfd, evlist ,COUNT , MILLISECS);
```

4) Verify **fd** status and execute appropriate IO calls.

```
evlist[0].events  &EPOLLIN

n= read (evlist[0].data.fd ,buf,10);
```