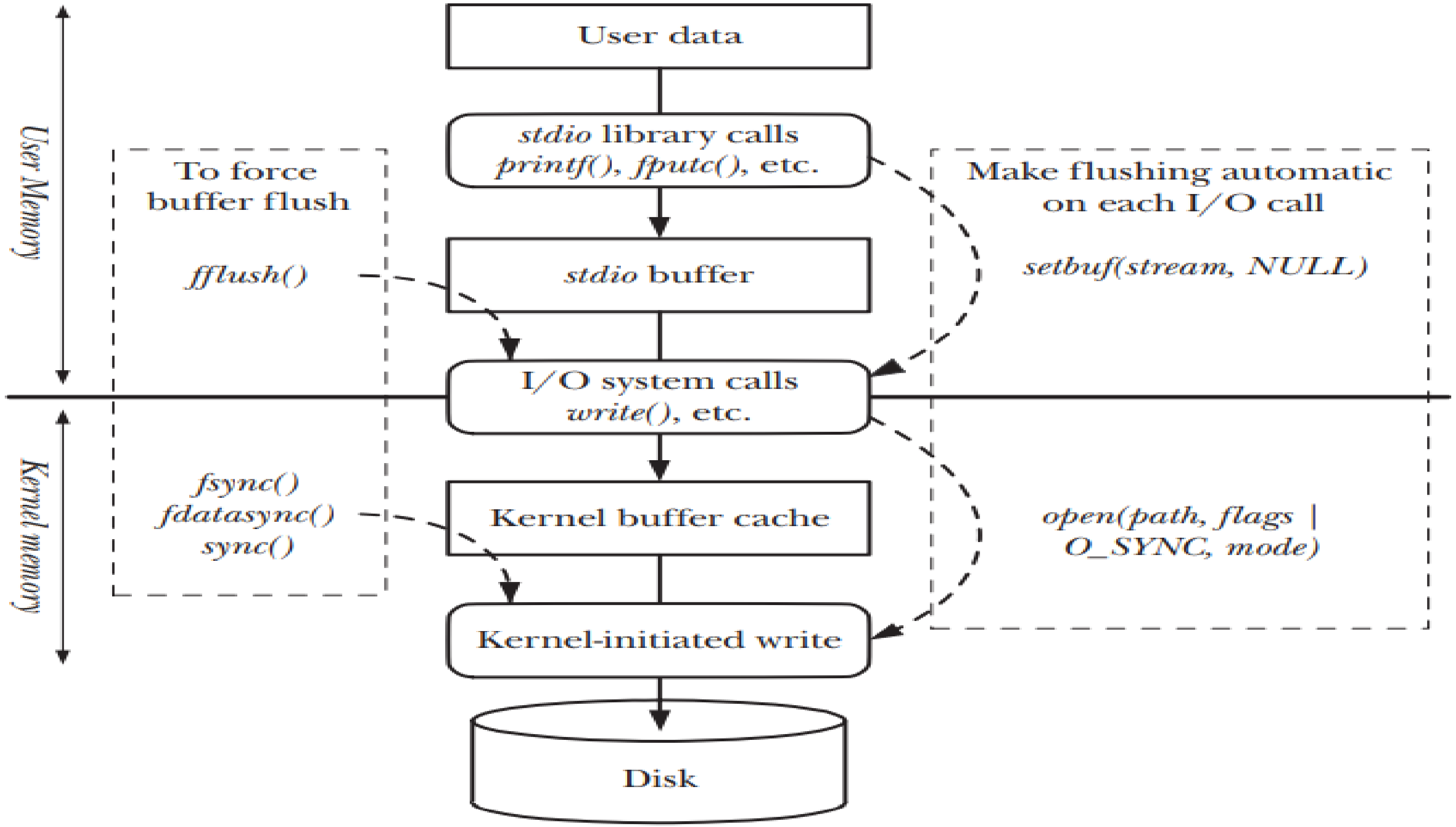
File in Linux

TIN HOC NHÃ UYÊN 10/2018

Content

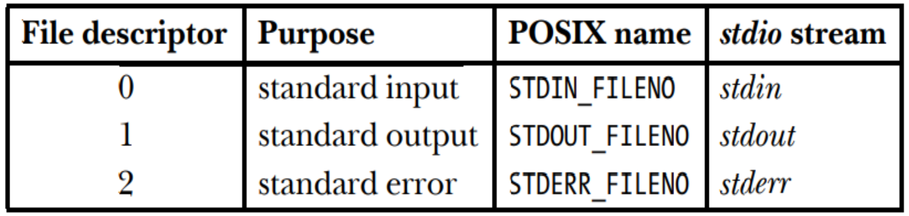
- Overview File in Linux
- File I/O basic method
- File I/O atomic and race condition
- File control operation
- Relationship between File Descriptor and Open File
- File I/O some advance method
- File buffering and stdio library buffering

Overview File in Linux



File I/O - Overview

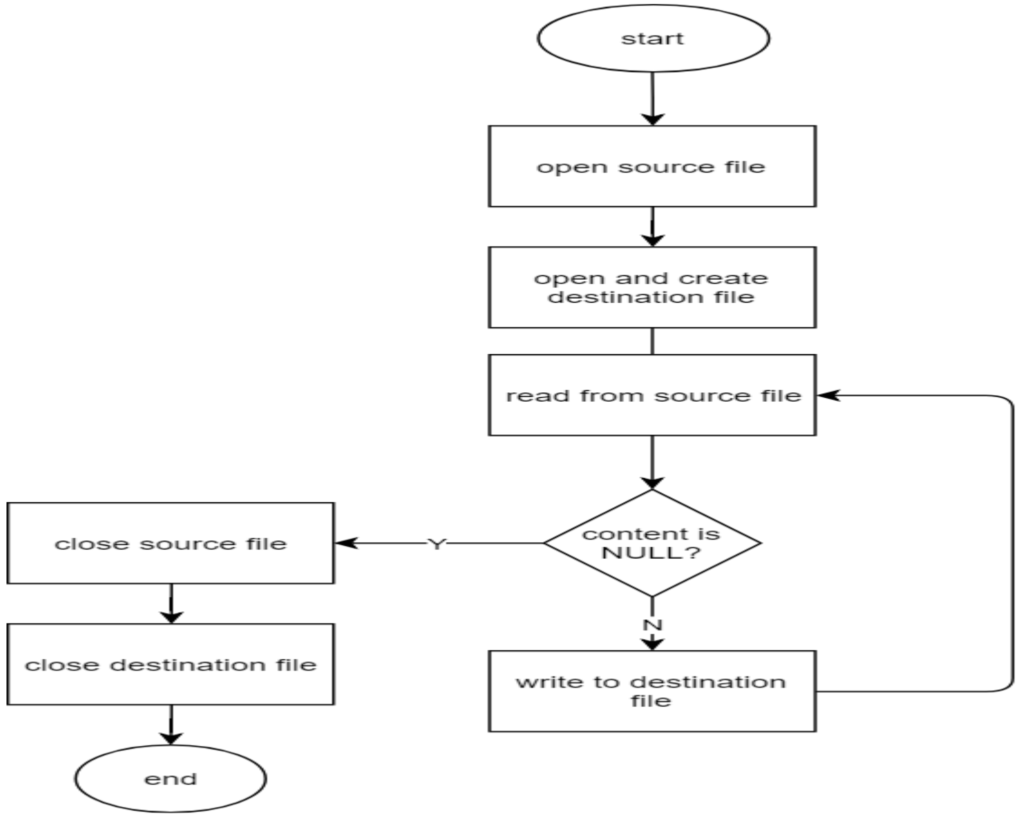
- File is central to the UNIX philosophy
- All system calls for performing I/O referring to a file descriptor
  - Is nonnegative integer, start from 0
  - File descriptor are used to refer to all types of file, pipe, FIFOs, socket, terminals, devices
  - Each process has it own set of file descriptor
- Most of program expect to able to use three standard file descriptor
  - Running process inherit copies of the shell opened file descriptors



File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	STDIN_FILENO	*stdin*
1	standard output	STDOUT_FILENO	*stdout*
2	standard error	STDERR_FILENO	*stderr*

- 4 key system calls for performing file IO
  - fd = open(pathname, flags, mode)
  - numread = read(fd, buffer, count)
  - numwritten = write(fd, buffer, count)
  - status = close(fd)

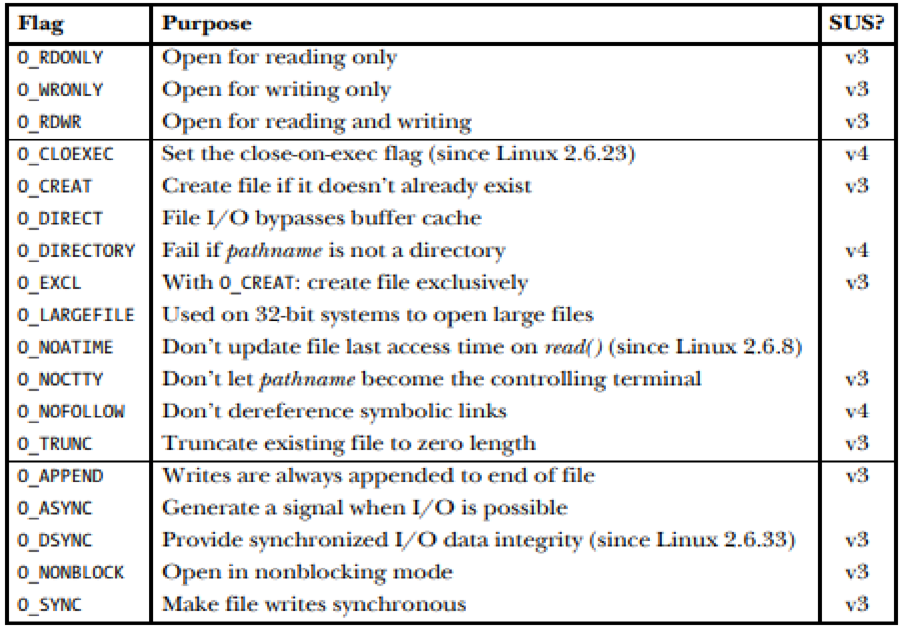
File I/O – basic method- file copy example



File I/O – open

- File access mode flags
  - Read only, Write only and Read-Write
  - Can retrieve by fcntl F_GETFL
- File Creation flags
- Open file status flags
  - Can retrieve and modified by fcntl F_GETFL
- Return file descriptor

```
int open(const char *pathname, int flags,
    ... /* mode_t mode */);
```



Flag	Purpose	SUS?
O_RDONLY	Open for reading only	v3
O_WRONLY	Open for writing only	v3
O_RDWR	Open for reading and writing	v3
O_CLOEXEC	Set the close-on-exec flag (since Linux 2.6.23)	v4
O_CREAT	Create file if it doesn't already exist	v3
O_DIRECT	File I/O bypasses buffer cache	
O_DIRECTORY	Fail if *pathname* is not a directory	v4
O_EXCL	With O_CREAT: create file exclusively	v3
O_LARGEFILE	Used on 32-bit systems to open large files	
O_NOATIME	Don't update file last access time on *read()* (since Linux 2.6.8)	
O_NOCTTY	Don't let *pathname* become the controlling terminal	v3
O_NOFOLLOW	Don't dereference symbolic links	v4
O_TRUNC	Truncate existing file to zero length	v3
O_APPEND	Writes are always appended to end of file	v3
O_ASYNC	Generate a signal when I/O is possible	
O_DSYNC	Provide synchronized I/O data integrity (since Linux 2.6.33)	v3
O_NONBLOCK	Open in nonblocking mode	v3
O_SYNC	Make file writes synchronous	v3

File I/O – open (2)

- Important flags
  - O_APPEND
    - Write always append end of file
  - O_ASYNC
    - Signal-driven IO, but in Linux this flag no effect, in order to enable this feature, use fcntl flags
  - O_CLOEXEC
    - Close the file when calling exec family function
  - O_CREATE
    - If file is not exist, file is created
  - O_DIRECT
    - Allow File IO to bypass the buffer cache
- O_EXCL
  - Conjunction with O_CREATE to indicate that if the file exist, it should not be opened, open fail, errno set to EEXIST
- O_NONBLOCK
  - Open file for non-blocking
- O_SYNC
  - Open file for synchronous I/O
- O_TRUNC
  - If file exist, the content of file is discard and length is truncated to zero

File I/O – open (3)

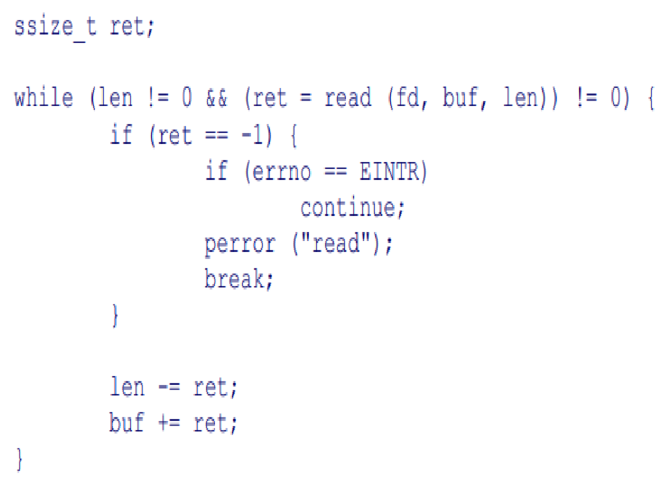
- Error from open – strerror(errno)
  - EACCES
    - File permission don’t allow the calling process to open the file in mode specified by flags
    - Directory permission don’t allow file created or writing
  - EISDIR
    - File in pathname is directory and open file for writing
  - EMFILE
    - The process resource limit on the number of open file descriptor has been reached

- ENFILE
  - System resource limit on the number of open file descriptor has been reached
- ENOENT
  - File is not **exist and O_CREATE is not specified**
- EROFS
  - File is read-only but open is for writing
- ETXTBSY
  - File is an executable file and it is currently executing. It is not allow to modify a executable file that is running as a process

File I/O - read

- Arguments
  - Count specifics the maximum number of bytes to read,
  - Buffer is where data placed, the buffer must be at least count bytes long
- Return value
  - Return number of bytes actually read
    - Number of byte read is less than count because some reason
      - Interrupt by signal return with errno EINTR
      - New line
  - 0 with end of file (EOF), nothing to read
  - -1 with error – use strerror(ret) to know what error

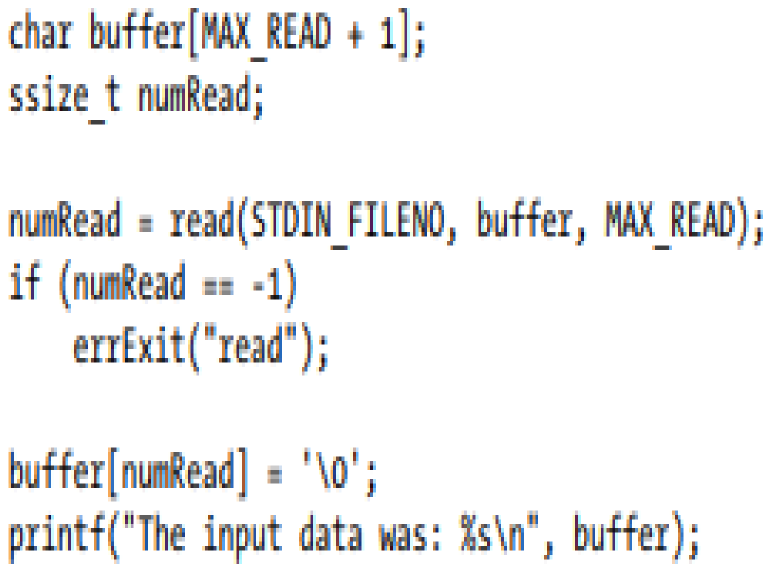
ssize_t read(int fd, void *buffer, size_t count);



```
ssize_t ret;

while (len != 0 && (ret = read (fd, buf, len)) != 0) {
        if (ret == -1) {
                if (errno == EINTR)
                        continue;
                perror ("read");
                break;
        }

        len -= ret;
        buf += ret;
}
```

File I/O – read (2)

- Blocking mode
  - Read default in blocking mode except regular file
  - Append zero to end of buffer explicitly
    - Data can be text, binary interger, C structures in binary form → read don't know what type data comming
    - Buffer length must be at least one greater than largest string we expect
- Non blocking mode
  - Read return immediately with errno set to EAGAIN



```
char buffer[MAX_READ + 1];
ssize_t numRead;

numRead = read(STDIN_FILENO, buffer, MAX_READ);
if (numRead == -1)
    errExit("read");

buffer[numRead] = '\0';
printf("The input data was: %s\n", buffer);
```

File I/O -write

- Argument
  - Count is number of bytes write to file descriptor
  - Buffer store the content
- Return value
  - Return number of bytes that actually write, this value can be less than count
  - For regular file, a success return from write doesn't guarantee that data has been transfer to disk due to kernel perform buffering of disk I/O to reduce disk activity

```
ssize_t write(int fd, void *buffer, size_t count)
```

File I/O - close

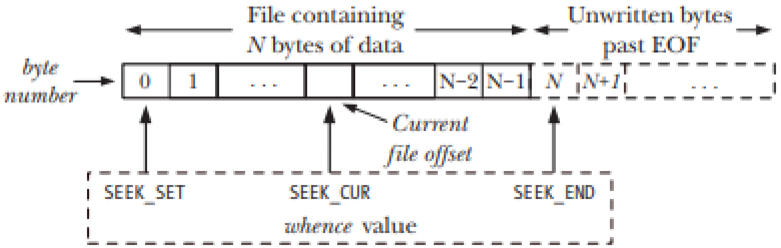
- Free file resource for reuse
- If process terminate, all descriptor close automatically

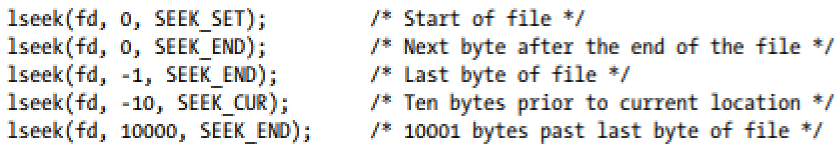
```
int close (int fd);
```

File I/O – change file offset - lseek

- File as sequence of bytes, kernel record file offset
  - File offset is set to point to the start of file when the file opened
  - Automatically adjusted by call read or write
  - Lseek adjust the file offset of open file
- Argument whence
  - SEEK_CUR
    - Offset bytes from the beginning of the file
  - SEEK_END
    - Offset bytes from the current file offset
  - SEEK_SET
    - The file offset is set to size of the file plus offset → next byte after last byte of file
    - File hole

off_t lseek (int fd, off_t offset, int whence)

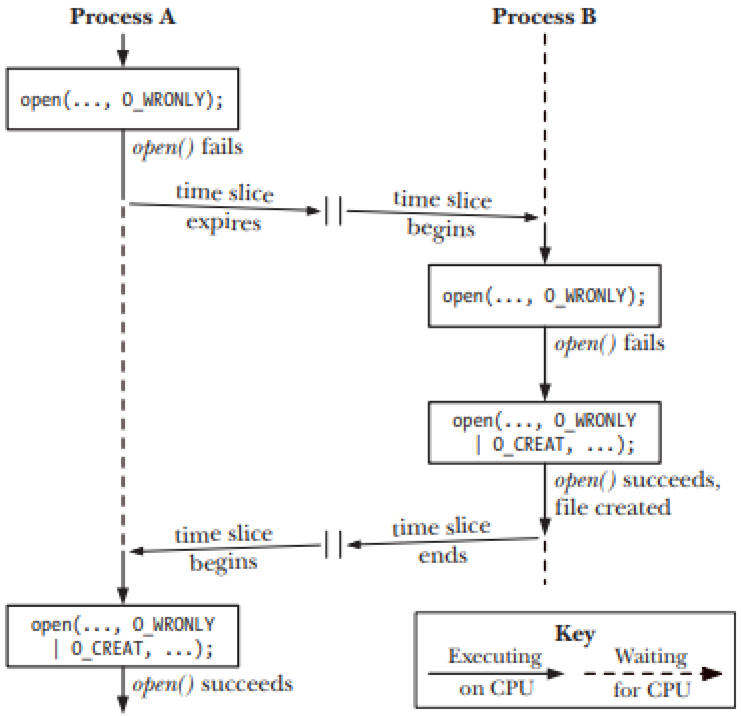




```
lseek(fd, 0, SEEK_SET);         /* Start of file */
lseek(fd, 0, SEEK_END);         /* Next byte after the end of the file */
lseek(fd, -1, SEEK_END);        /* Last byte of file */
lseek(fd, -10, SEEK_CUR);       /* Ten bytes prior to current location */
lseek(fd, 10000, SEEK_END);     /* 10001 bytes past last byte of file */
```

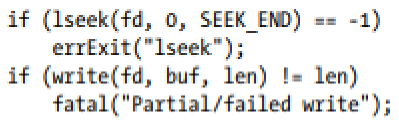
File I/O – Atomic and Race Conditions

- All system call are executed atomically
  - Kernel guarantees that all of the steps in a system call are completed as a single operation without interrupted from other process or threads
  - Atomic is essential to the successful completion of some operations
  - Atomic allow avoid race conditions
    - A situation where the result produced by two processes operate on one shared resources in unexpected way on the relative order in which process gain CPU



File I/O – Appending data to a file

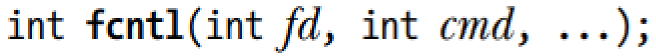
- Multi-process appending data to the same file to write file
  - One process seek to end of file
  - Kernel stop process and allow other running
  - Other process seek to end of file
  - Other process write to end of file
  - Kernel stop other process and allow first process running
  - First process write to end of file but it replace content of other process
- Avoid with O_APPEND



```
if (lseek(fd, 0, SEEK_END) == -1)
    errExit("lseek");
if (write(fd, buf, len) != len)
    fatal("Partial/failed write");
```

File I/O –File control operation

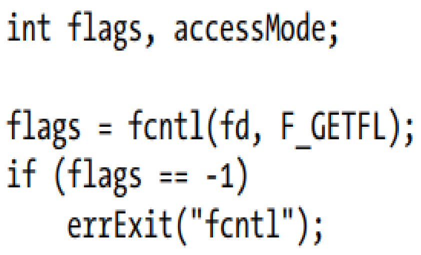
- Control everything relate to file
  - Duplicate file descriptor
  - File descriptor flags
  - File status flags
  - File locking
  - IO availability signal
  - File change notification



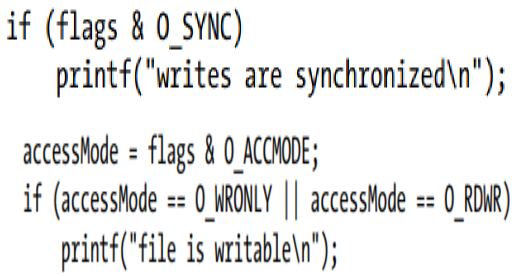
```
int fcntl(int fd, int cmd, ...);
```

FILE I/O – File status flags

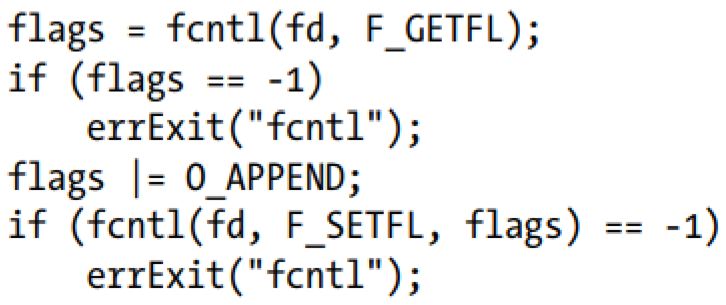
- fcntl
  - Retieve or modify the access mode and open file status flags
  - F_GETFL command
  - F_SETFL command
    - Modify O_APPEND, O_NONBLOCK, O_NOATIME, O_ASYNC, O_DIRECT



```
int flags, accessMode;

flags = fcntl(fd, F_GETFL);
if (flags == -1)
    errExit("fcntl");
```



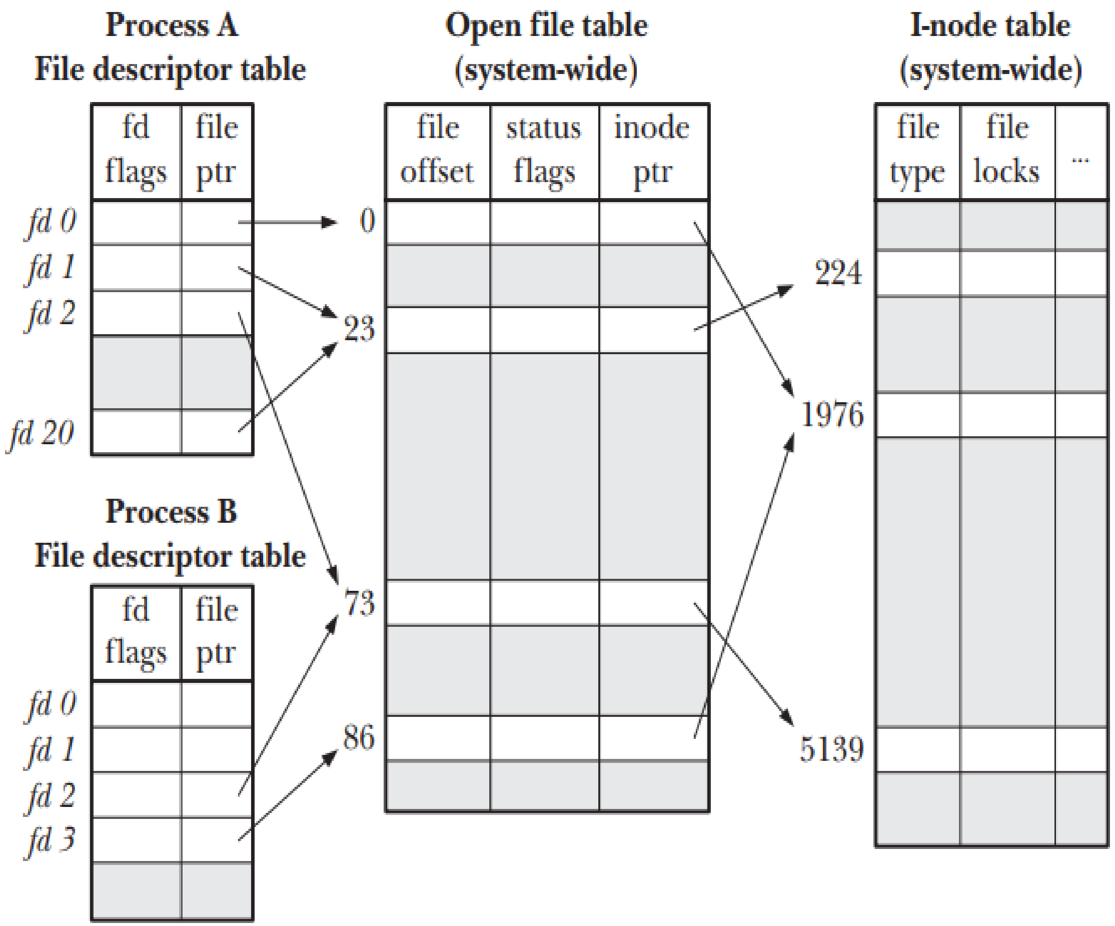
```
if (flags & O_SYNC)
    printf("writes are synchronized\n");

 accessMode = flags & O_ACCMODE;
 if (accessMode == O_WRONLY || accessMode == O_RDWR)
     printf("file is writable\n");
```



```
flags = fcntl(fd, F_GETFL);
if (flags == -1)
    errExit("fcntl");
flags |= O_APPEND;
if (fcntl(fd, F_SETFL, flags) == -1)
    errExit("fcntl");
```

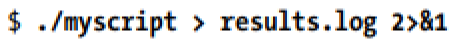
File I/O – Relationship between File Descriptor and Open File

- Three data structures that maintain by kernel
  - Per-process file descriptor
    - set of flags controlling the operation of the file descriptor (close on exec)
  - The system wide table of open file descriptor
    - the current file offset
    - status flags specified when opening the file
      - O_APPEND, O_NONBLOCK, O_ASYNC
    - the file access mode(readonly, writeonly or read write)
    - settings relating to signal-driven I/O
    - a reference to the i-node object
  - File system inode table
    - file type (e.g., regular file, socket, or FIFO) and permissions
    - a pointer to a list of locks held on this file
    - various properties of the file, including its size and timestamps

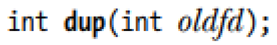


File I/O – Duplicate file descriptor

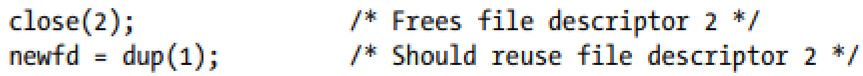
- I/O redirection
  - Standard error redirected to the same place with standard output
  - Sent both standard output and standard error to results.log
- dup
  - Take an open file descriptor and return a new descriptor that refer to the same open file descriptor
- dup2
  - Make a duplicate of file descriptor old to new
  - If newfd is opened, it closed



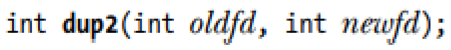
```
$ ./myscript > results.log 2>&1
```



```
int dup(int oldfd);
```



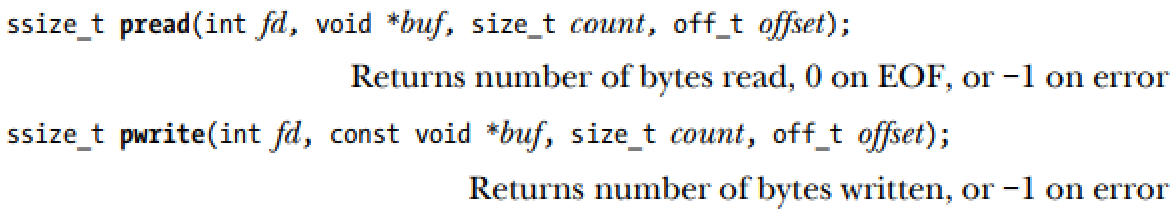
```
close(2);               /* Frees file descriptor 2 */
newfd = dup(1);         /* Should reuse file descriptor 2 */
```



```
int dup2(int oldfd, int newfd);
```

File I/O at specified Offset

- Pread and pwrite
  - Operate like read and write
  - The file IO is performed at location specificed by offset than current file offset
  - The file offset left unchanged
- Call pread equivalent call lseek + read but atomically
  - These function is useful in multithread applications
  - All open file descriptor shared by all threads
  - File offset of each file is global to all threads
  - Avoid race condition when read and write simultaneously

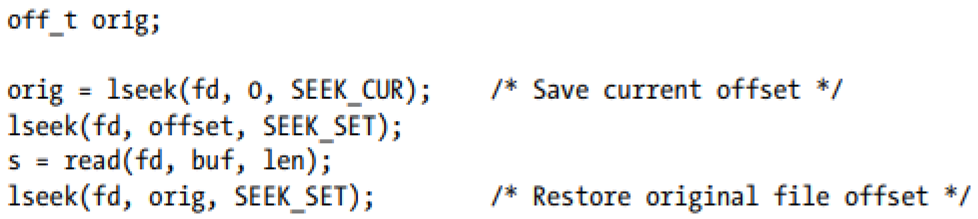


```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
```

Returns number of bytes read, 0 on EOF, or –1 on error

```
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

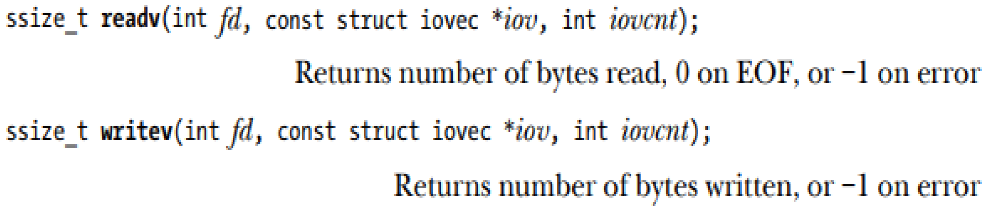
Returns number of bytes written, or –1 on error



```
off_t orig;

orig = lseek(fd, 0, SEEK_CUR);    /* Save current offset */
lseek(fd, offset, SEEK_SET);
s = read(fd, buf, len);
lseek(fd, orig, SEEK_SET);        /* Restore original file offset */
```

File I/O – scatter/gather IO

- Scatter/Gather IO
  - Transfer multiple buffer of data in single system call
  - Set of buffers to be transferred is defined by the array iov
- Scatter input
  - readv perform scatter input
  - read a contiguous sequence of bytes from the file descriptor to buffer's iovec
  - readv is atomically → when reading from a file, the range of byte in iov's buffer is continuous even if another process sharing the same file offset attempts to manipulate the offset at the same time
  - Readv return number of byte read or 0 if end-of-file was encountered
  - If insufficient data, some of last is partially filled

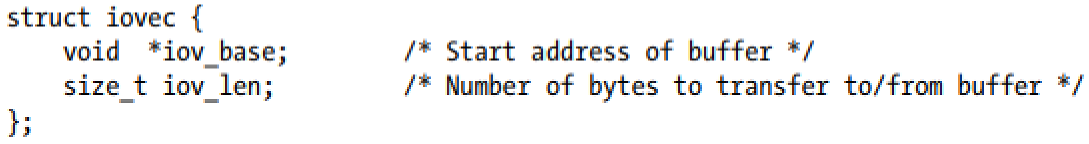


```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
```

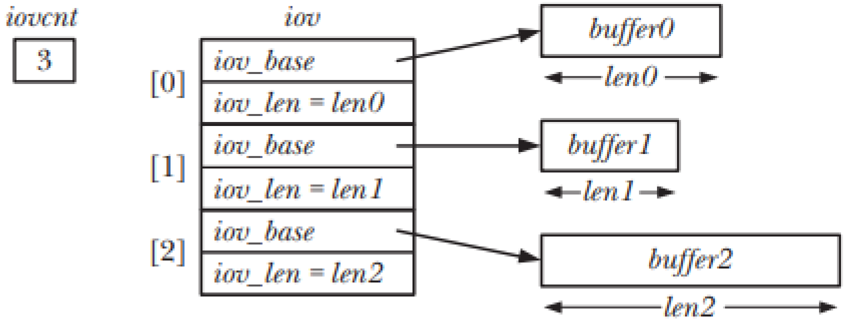
Returns number of bytes read, 0 on EOF, or –1 on error

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

Returns number of bytes written, or –1 on error



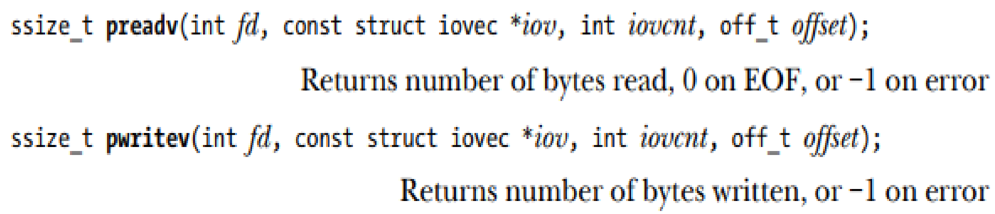
```
struct iovec {
    void  *iov_base;        /* Start address of buffer */
    size_t iov_len;         /* Number of bytes to transfer to/from buffer */
};
```



File I/O – scatter/gather IO (2)

- Gather output
  - Writev system call perform gather output
  - Write as sequence of continuous bytes
  - Buffer gather in array order
  - Writev is atomically → all requested data is written continuously to the file
  - Like write, writev can return patially write.Let check the result to indicate
  - Readv and writev is convenience and speed
    - Writev can implement as copy of user buffers and call write
    - Call write multiple time but two way inconvenience and slow

- Performing scatter-gather I/O at a specified offset



```
ssize_t preadv(int fd, const struct iovec *iov, int iovcnt, off_t offset);
```

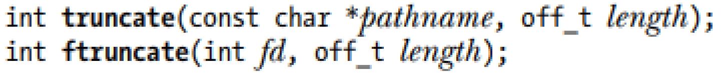
Returns number of bytes read, 0 on EOF, or –1 on error

```
ssize_t pwritev(int fd, const struct iovec *iov, int iovcnt, off_t offset);
```

Returns number of bytes written, or –1 on error

FILE I/O – truncating the file

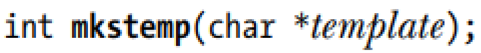
- Resize the file length
  - If file length is longer than length, excess data is lost
  - If file length is shorter than length, file will padding with sequence of null byte
  - The command use to discard the content of file



```
int truncate(const char *pathname, off_t length);
int ftruncate(int fd, off_t length);
```

File I/O- creating temporary file

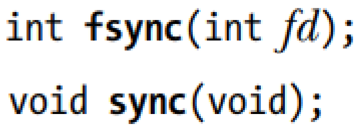
- Create temporary file when program is running
  - Create a file in /tmp directory and return file descriptor
  - Template argument takes the form of a pathname in which the last 6 characters must be XXXXXX
  - 6 XXXXXX wil be replaced with a string that makes the filename unique
  - Templace is modified → it must be specified as a character array rather than string constant



```
int mkstemp(char *template);
```

Kernel Buffering of File I/O

- Working with disk
  - Read/Write system call don't directly initiate disk access
  - They copy data between a user-space buffer and a buffer in kernel buffer cache
  - Write(fd, "abc", 3)
    - Write return immediately
    - Some later point, kernel writes its buffer to disk
    - System call is not synchronized with disk
    - If another process attempts to read these bytes of file, the kernel supplies data from buffer cache
  - Read(fd, buf, 3)
    - Kernel read data from the buffer until is exhausted , kernel reads next segment of the file into the buffer cache
  - This design is allow read and write to be fast, process don't need to wait on slow disk, and reduce the number of disk transfer that kernel perform
- Two way to control kernel buffer of File I/O
  - Use system call



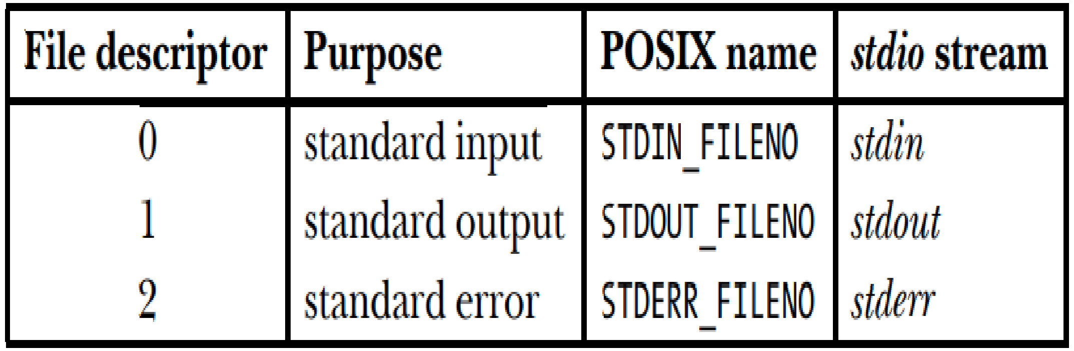
```
int fsync(int fd);

void sync(void);
```

  - Use O_SYNC when open file

stdio library – standard IO

- File streams

- Standard IO

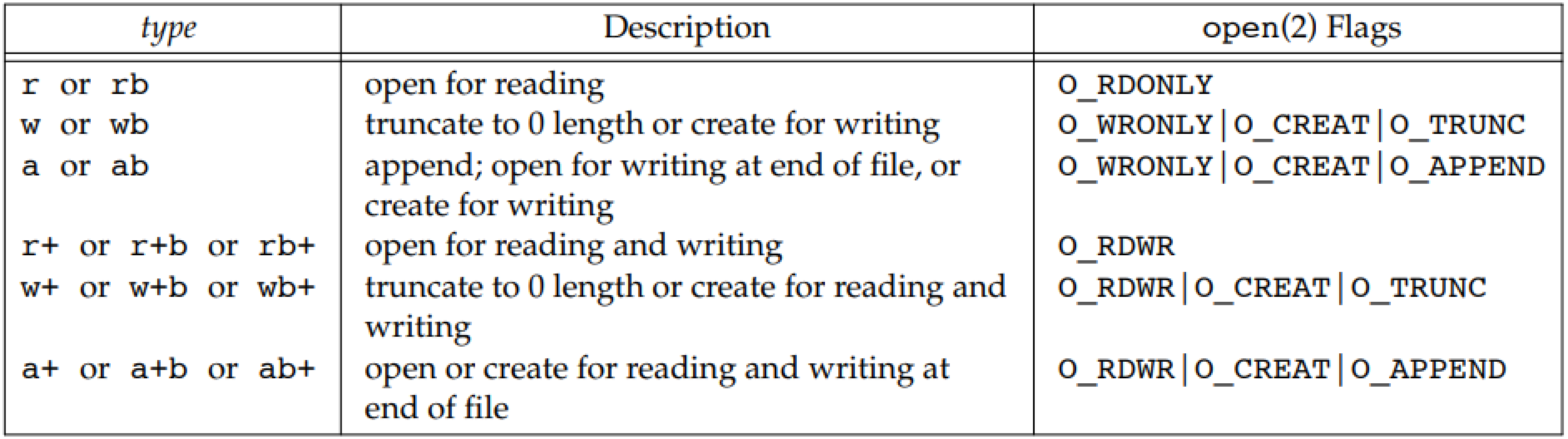


File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	STDIN_FILENO	*stdin*
1	standard output	STDOUT_FILENO	*stdout*
2	standard error	STDERR_FILENO	*stderr*

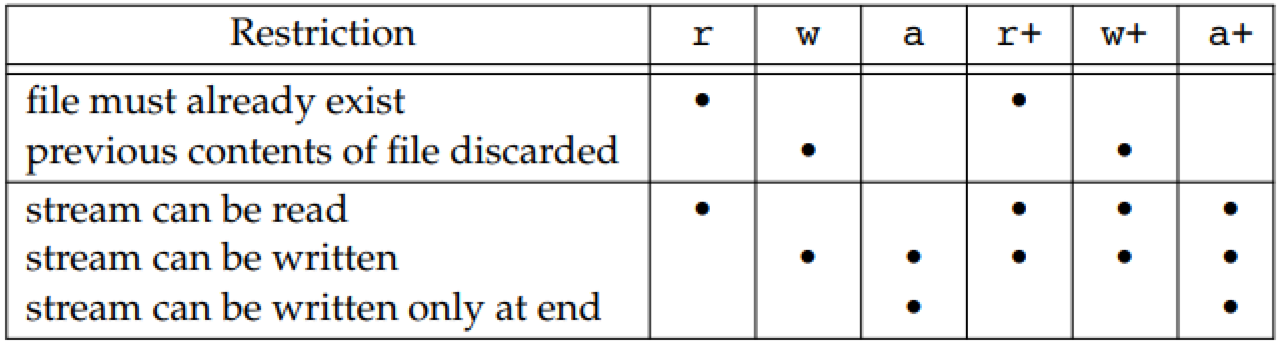
- Relationship between stdio and File IO

stdio library – API – open the file

*FILE * **fopen** (const char *filename, const char *opentype)*



type	Description	open(2) Flags
`r or rb`	open for reading	`O_RDONLY`
`w or wb`	truncate to 0 length or create for writing	`O_WRONLY`|`O_CREAT`|`O_TRUNC`
`a or ab`	append; open for writing at end of file, or create for writing	`O_WRONLY`|`O_CREAT`|`O_APPEND`
`r+ or r+b or rb+`	open for reading and writing	`O_RDWR`
`w+ or w+b or wb+`	truncate to 0 length or create for reading and writing	`O_RDWR`|`O_CREAT`|`O_TRUNC`
`a+ or a+b or ab+`	open or create for reading and writing at end of file	`O_RDWR`|`O_CREAT`|`O_APPEND`



Restriction	r	w	a	r+	w+	a+
file must already exist	•			•		
previous contents of file discarded		•			•	
stream can be read	•			•	•	•
stream can be written		•	•	•	•	•
stream can be written only at end			•			•

stdio library – API – close the file

int ***fclose*** *(FILE *stream)*

stdio library – API – read/write the file

fread *(void *data, size_t size, size_t count, FILE *stream)*

size_t **fwrite** *(const void *data, size_t size, size_t count, FILE *stream)*

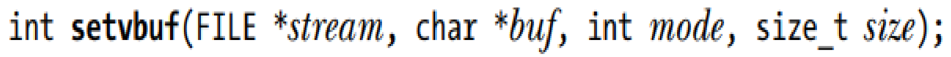
stdio library – API – file positioning

long int ***ftell*** *(FILE *stream)*

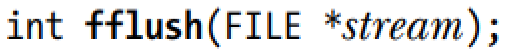
int ***fseek*** *(FILE *stream, long int offset, int whence)*

Buffering in the stdio library

- Buffering of data into large block
  - Reduce system call and overhead
  - Increase performance when operate on disk
- Settting the buffering mode of stdio stream
  - If buf is NULL, library automatically allocates a buffer for use with stream at least BUFSIZ 512 byte
  - Mode
    - _IONBF   Don't buffer IO, call system call read/write immediately, buf and size is irgnored
    - _IOLBF Line buffer, data is buffer until a newline character is output, for input, data is read a line at a time
    - _IOFBF   Fully buffer I/O is default mode
- Flushing a stdio buffer



```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```



```
int fflush(FILE *stream);
```

stdio library – API -bufferring

- Performance reason
- Set buffer for buffering
  - Atleast BUFSIZ – 512 bytes
  - Set NULL to disable buffering

```
void setbuf(FILE *restrict fp, char *restrict buf );
```

Review

- strace
- fallocate -l 100M /tmp/bigfile
- time