Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

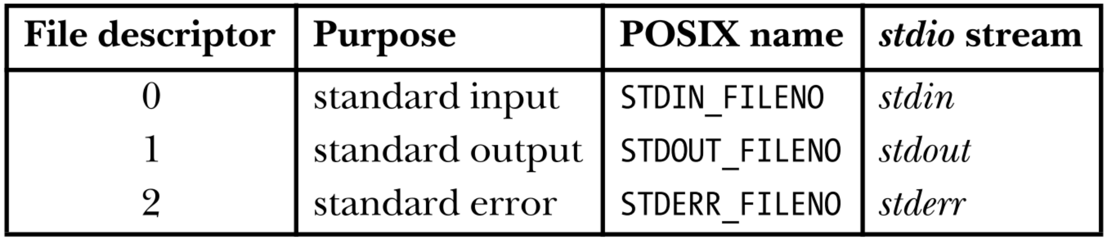
innovate achieve lead

BITS Pilani
Pilani Campus

File I/O

File Descriptor (fd)

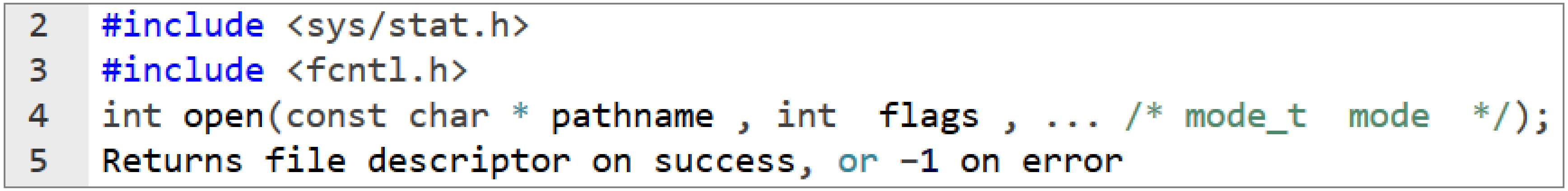
- File Descriptors refer to all types of open files.
  - Pipes, FIFOs, sockets, terminals, devices, and regular files.
- Each process has its own set of file descriptors.
- All system calls refer to file descriptors for performing I/O.
- Three standard file descriptors.



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*

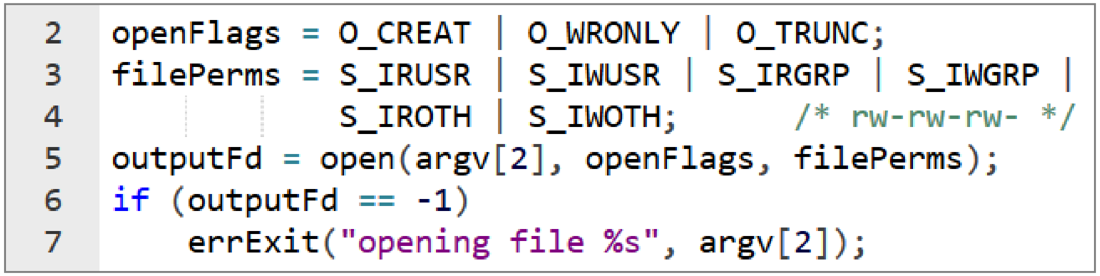
  - These three descriptors are open in the shell process.
  - Whenever a new program is executed in the shell, a child process is created. All three descriptors remain open in the child.
- File descriptors are different from FILE streams. FILE stream is a C library abstraction over fd.

File I/O: open()



```
#include <sys/stat.h>
#include <fcntl.h>
int open(const char * pathname , int  flags , ... /* mode_t  mode  */);
Returns file descriptor on success, or -1 on error
```

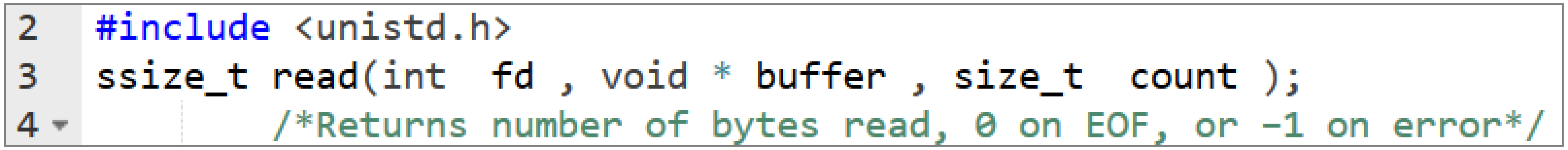
- *flags* is a bitmask that refers to read only or write only or both.
- *mode* refers to permissions. *mode* is used only when creating a file.



```
openFlags = O_CREAT | O_WRONLY | O_TRUNC;
filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
            S_IROTH | S_IWOTH;      /* rw-rw-rw- */
outputFd = open(argv[2], openFlags, filePerms);
if (outputFd == -1)
    errExit("opening file %s", argv[2]);
```

- O_CREAT option is used when a new file is to be created.
- O_TRUNC option is used when the data in the file has to be deleted.
- O_APPEND is used for appending to the existing file.
- Several other flags also present … (R1: 4.3.1)

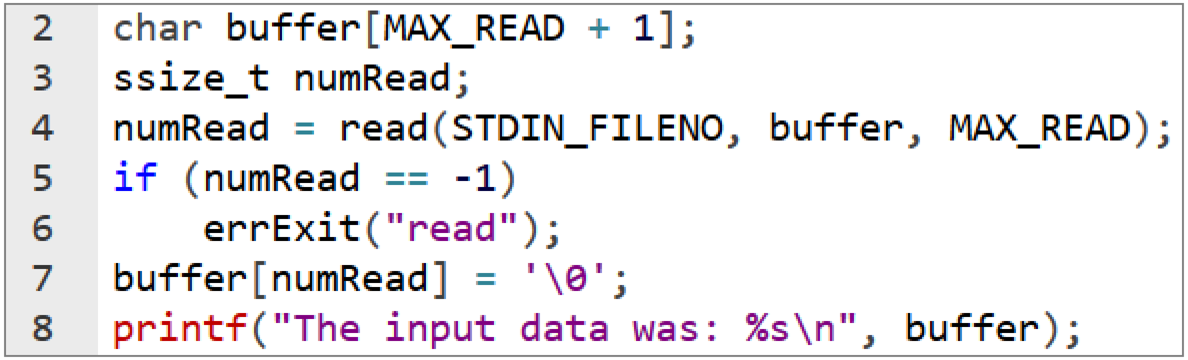
File I/O: read()



```
#include <unistd.h>
ssize_t read(int  fd , void * buffer , size_t  count );
        /*Returns number of bytes read, 0 on EOF, or -1 on error*/
```

- Reads at most *count* bytes from the open file referred to by *fd* and stores them in a *buffer*.
- It returns the no of bytes actually read or EOF or -1.
  - *count*: maximum number of bytes to read
  - *buffer:* address of the memory buffer into which the input data is to be placed.
  - Read may read less than *count*.
    - In regular files, we may be close to EOF.
    - In pipes, FIFOs, and sockets it may read less than *count* due to non-availability of data.

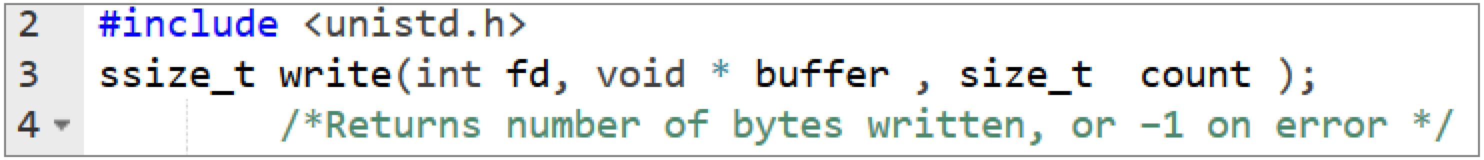
File I/O: read()



```
2   char buffer[MAX_READ + 1];
3   ssize_t numRead;
4   numRead = read(STDIN_FILENO, buffer, MAX_READ);
5   if (numRead == -1)
6       errExit("read");
7   buffer[numRead] = '\0';
8   printf("The input data was: %s\n", buffer);
```

- STDIN_FILENO refers to fd 0.
- At line 9, we need to include NULL character because *read*() doesn’t do it itself.

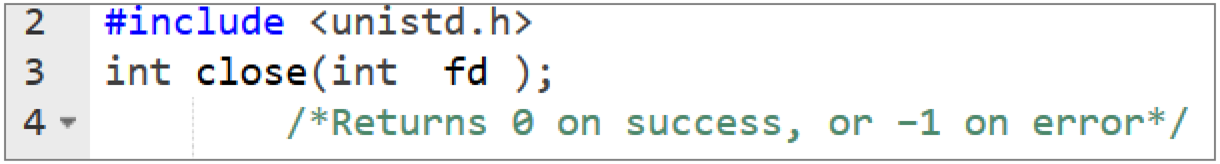
File I/O: write()



```
#include <unistd.h>
ssize_t write(int fd, void * buffer , size_t  count );
        /*Returns number of bytes written, or -1 on error */
```

- Writes up to *count* bytes from *buffer* to the open file referred to by *fd.*
- Returns the number of bytes actually written which may be less than *count.*

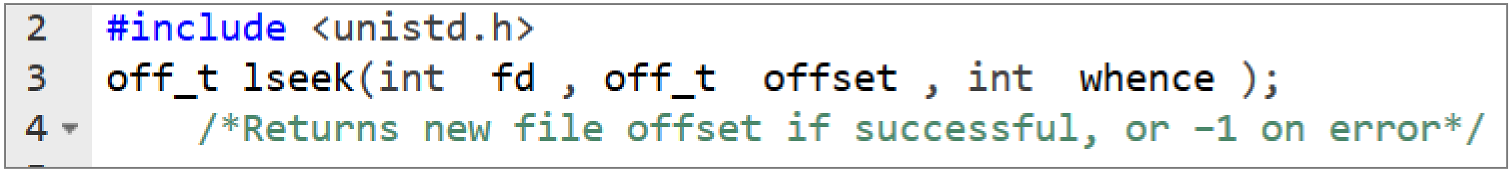
File I/O: close()



```
#include <unistd.h>
int close(int  fd );
            /*Returns 0 on success, or -1 on error*/
```

- It is called after all I/O has been completed in order to release the file descriptor *fd* and its associated kernel resources.
- When a process terminates all of its open file descriptors are automatically closed.

File I/O: lseek()

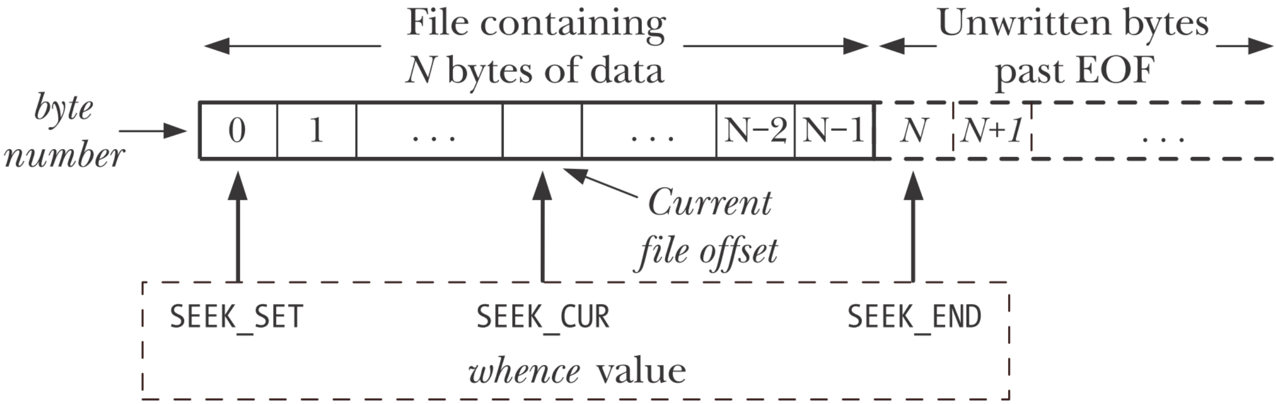


```
#include <unistd.h>
off_t lseek(int  fd , off_t  offset , int  whence );
    /*Returns new file offset if successful, or –1 on error*/
```

- It adjusts the file offset of the open file referred to by the *fd*, according to the values specified inn *offset* and *whence.*
- Kernel records a *file offset* for each open file.
- This is the location in the file at which the next read or write will commence.
- When the file opened the offset is set to 0 i.e. the beginning of the file.

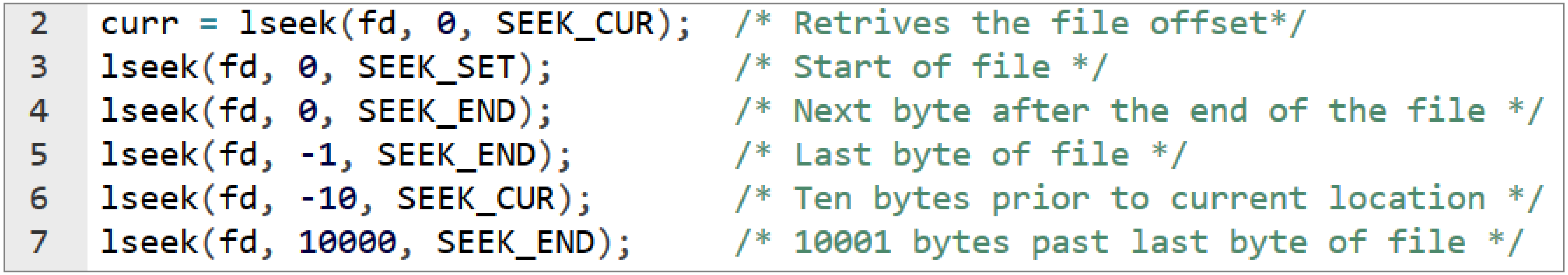
File I/O: lseek()

- *whence* argument can be
  - SEEK_SET
  - SEEK_CUR
  - SEEK_END



  - lseek() simply adjusts the file offset, it doesn't cause any physical device access.
  - We can't apply lseek() to pipe, FIFO, socket or terminal.

File I/O: lseek()

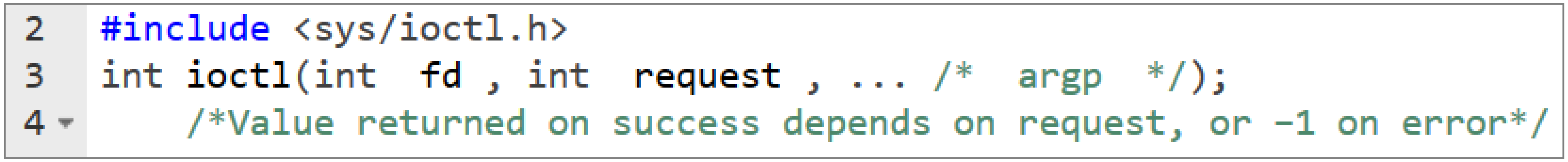


```
curr = lseek(fd, 0, SEEK_CUR);  /* Retrives the file offset*/
lseek(fd, 0, SEEK_SET);         /* Start of file */
lseek(fd, 0, SEEK_END);         /* Next byte after the end of the file */
lseek(fd, -1, SEEK_END);        /* Last byte of file */
lseek(fd, -10, SEEK_CUR);       /* Ten bytes prior to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

- File holes
  - What if we read after lseek(fd, 10000, SEEK_END)?
    - Returns 0.
  - What if we write after lseek(fd, 10000, SEEK_END)?
    - It creates a file hole. File holes do not take disk space.
- File holes are useful when a program need to access a wide range of addresses (offset) but is unlikely to touch all of the potential blocks.
  - Virtual hard disks

Universality of I/O

- The same four system calls – *open(), read(), write(), and close()* – are used to perform I/O on all types of files.
    - Regular files, Pipe, FIFO, sockets, terminal devices
- *ioctl()* system call is for operations that fall outside the universal I/O model.

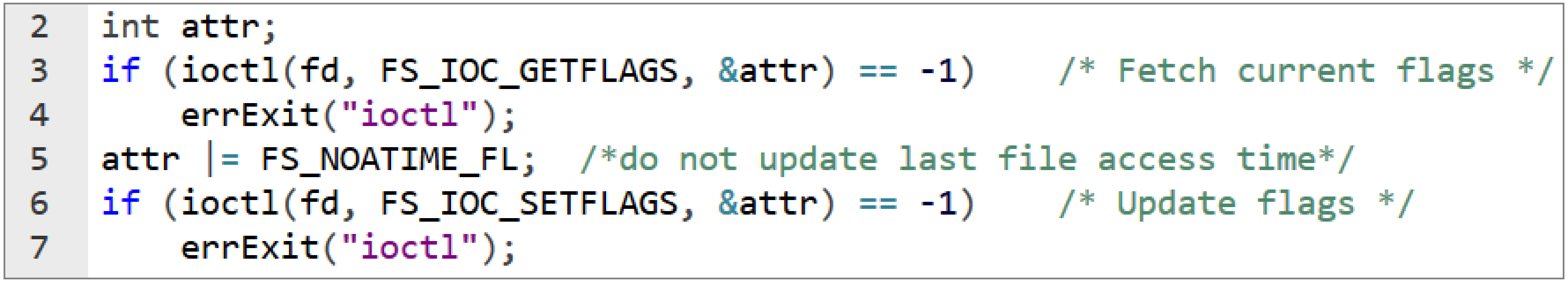


```
#include <sys/ioctl.h>
int ioctl(int  fd , int  request , ... /*  argp  */);
    /*Value returned on success depends on request, or -1 on error*/
```

- ○ *fd* refers to any file or device.
- ○ *request* refers to the constant specific to the device.
- ○ *argp* is the value or buffer depending the type of request.

Universality of I/O: *ioctl()*

- e.g. for updating flags of inode, *ioctl()* is used.

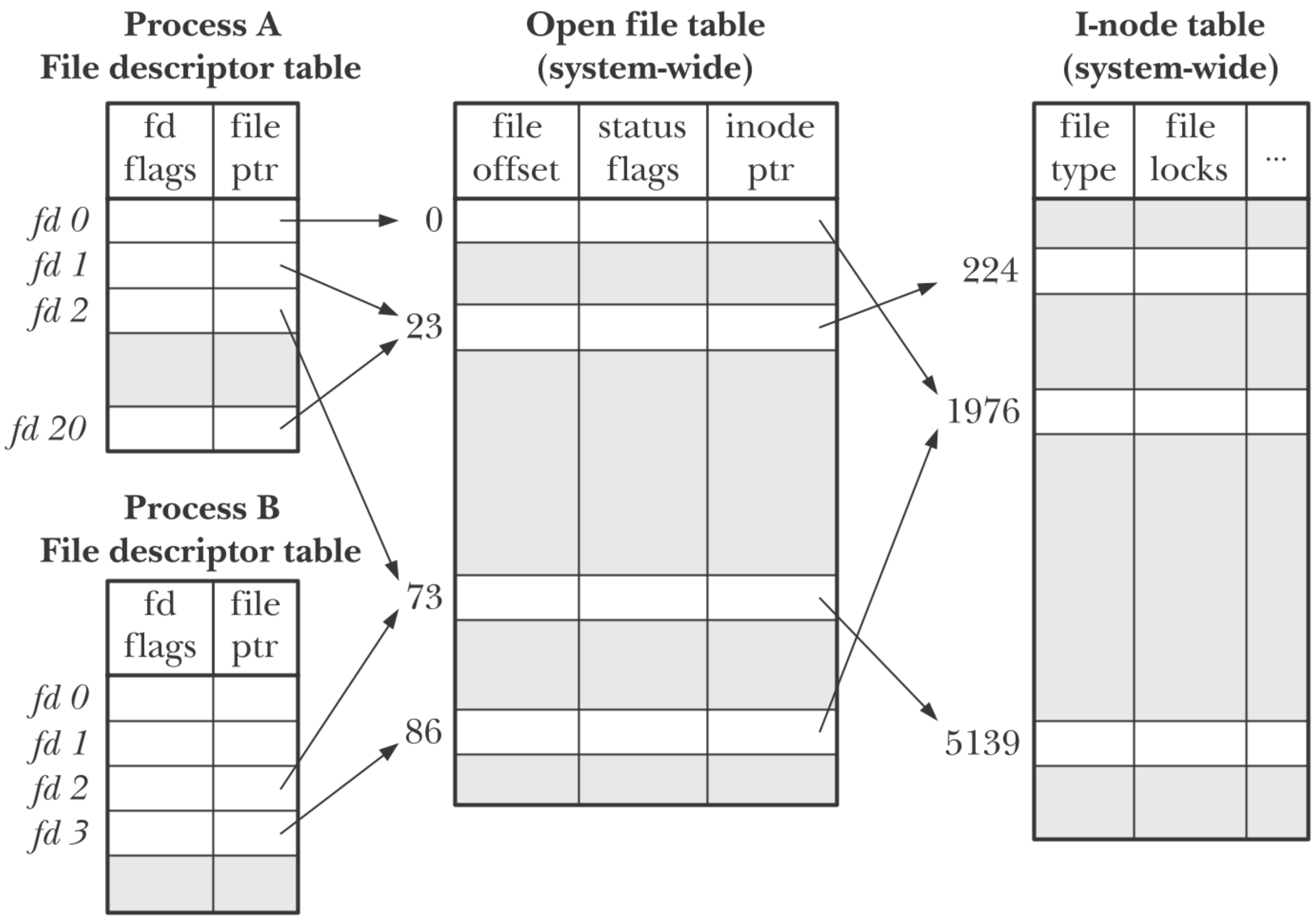


```
int attr;
if (ioctl(fd, FS_IOC_GETFLAGS, &attr) == -1)    /* Fetch current flags */
    errExit("ioctl");
attr |= FS_NOATIME_FL;  /*do not update last file access time*/
if (ioctl(fd, FS_IOC_SETFLAGS, &attr) == -1)    /* Update flags */
    errExit("ioctl");
```

File Descriptors and Open Files

- It is possible to have multiple descriptors referring to the same open file.
- There are three data structures maintained by the kernel:
  - The per-process file descriptor
  - The system-wide table of open file descriptions
  - The file system i-node table.
- Per-process file descriptor table
  - Flags (close-on-exec)
  - A reference to the open file description

File Descriptors and Open Files



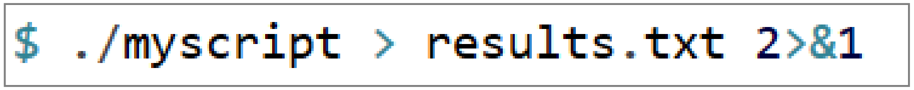
File Descriptors and Open Files

- System-wide table of all open file descriptions
  - The current file offset
    - Modified by read(), write() and lseek()
  - Status flags (flags argument to open())
  - File access mode (read-only, write only etc as specified in open())
  - Settings related to signal driven I/O
  - a reference to i-node object for this file.
- i-node table
  - Each file system has a table of i-nodes for all files residing in the file system.
  - File type (regular file, socket, FIFO etc)
  - A pointer to list of blocks
  - Various properties of the file (size, timestamps etc)

File Descriptors and Open Files

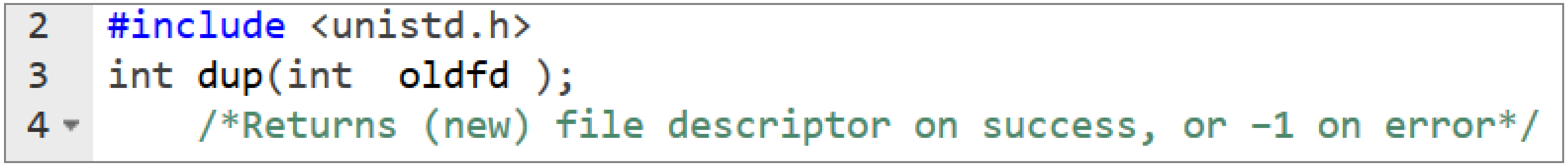
- Two descriptors in different process may refer to the same open file entry.
  - fork()
  - Passing descriptor using UNIX domain sockets
- Two open file entries can refer to same i-node.
  - When the same is open twice in the same process or in different processes.
- When an open file entry is shared
  - Updating file offset or flags effects the other process.
- close-on-exec flag individual to a fd. Changing doesn’t effect the other processes.

Duplicating File Descriptors: dup()



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

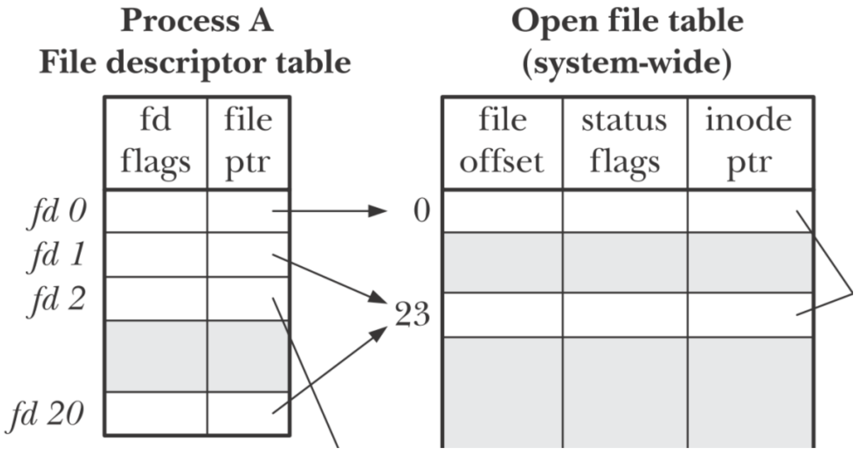
- Here 2>&1 indicates that standard error (fd 2) to be sent to the same place where standard output (fd 1) is being sent.
- This is possible by duplicating fd 2 to refer to open table entry referred by fd 1.



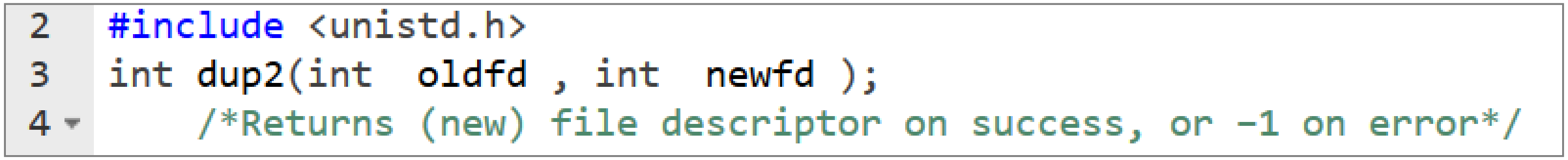
```
#include <unistd.h>
int dup(int  oldfd );
    /*Returns (new) file descriptor on success, or -1 on error*/
```

- Dup takes *oldfd* and returns a new fd that refers to the same open file table entry.
- New fd is guaranteed to be the lowest unused file descriptor.

Duplicating File Descriptors: dup2()



- close(1); dup(20)

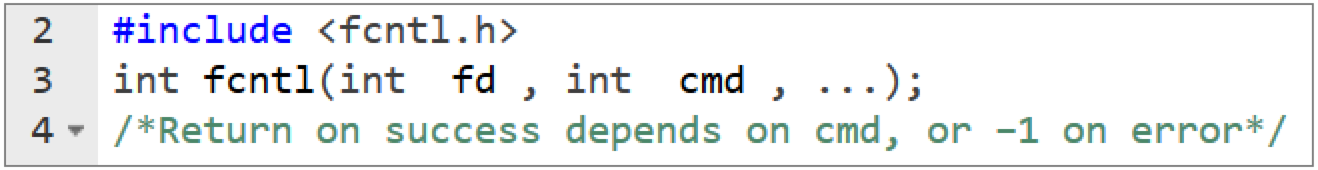


```
#include <unistd.h>
int dup2(int  oldfd , int  newfd );
    /*Returns (new) file descriptor on success, or –1 on error*/
```

- If *newfd* is open it closes it and copies the pointer in *oldfd* to *newfd* slot.
- This is done atomically.

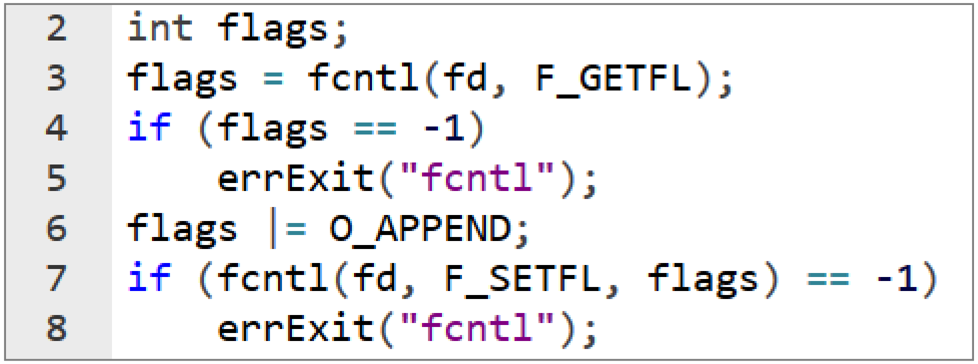
File Control Operaions: *fcntl()*

- fcntl() call performs operations on open file descriptors.



```
#include <fcntl.h>
int fcntl(int  fd , int  cmd , ...);
/*Return on success depends on cmd, or -1 on error*/
```

  - ○ cmd refers to commands.
  - ○ e.g. to change the flag after opening file



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

  - ○ Append is flag is being added to the flags in open file table entry.

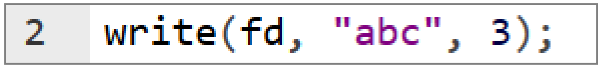
innovate achieve lead

BITS Pilani
Pilani Campus

I/O Buffering

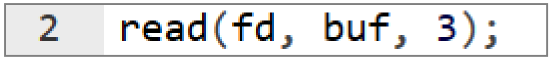
Buffering

- For speed and efficiency, I/O systems calls and I/O library calls buffer data.
- Two levels of buffering
  - Kernel buffer cache
    - Makes read and write calls faster
    - Reduces number of disk access by kernel
  - Buffering in the standard i/o library (optional)
    - Reduces number of system calls to access data



```
write(fd, "abc", 3);
```

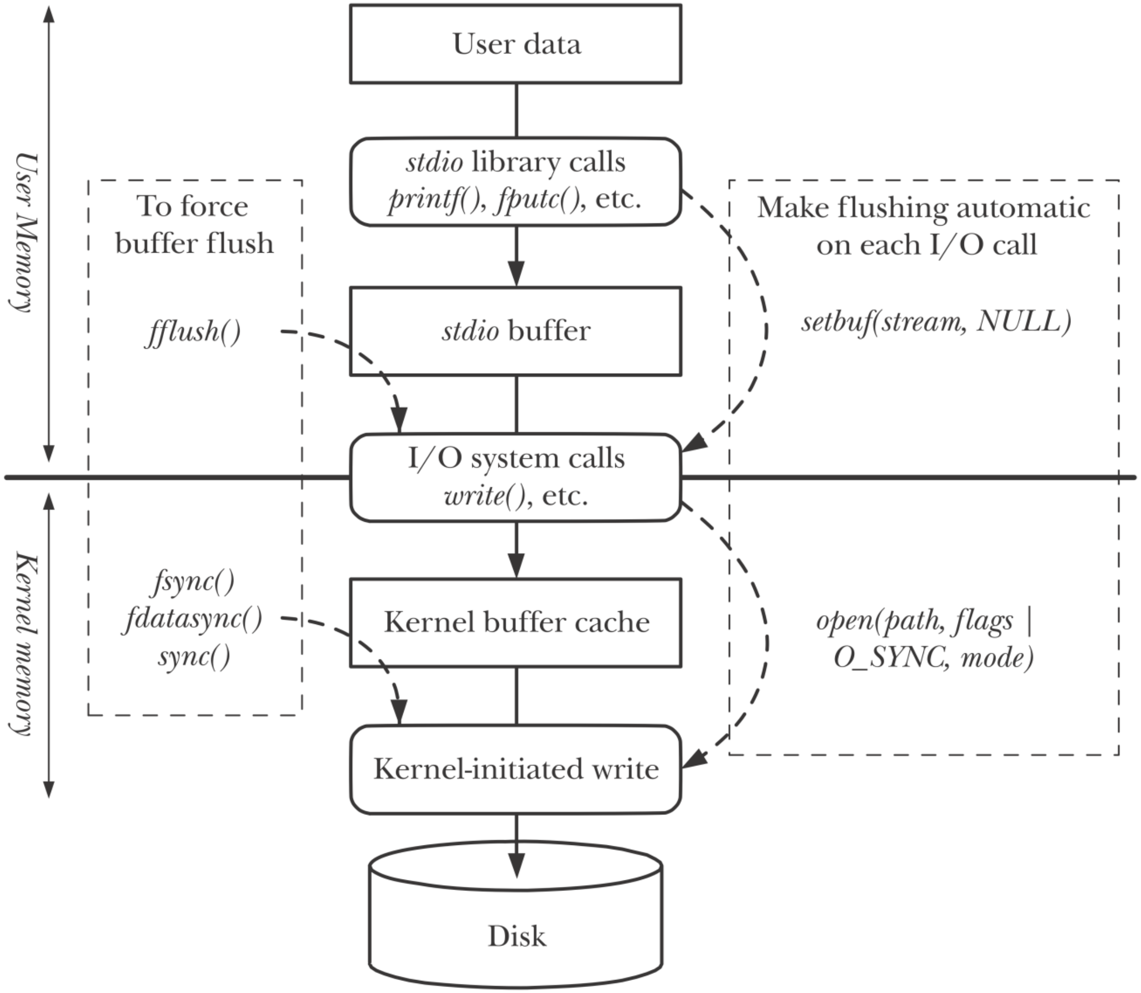
- This call can’t directly write to the disk. This writes to a buffer in the kernel. Kernel later syncs these contents with the disk.



```
read(fd, buf, 3);
```

- This call transfers 3 bytes of data from kernel buffer to user buffer *buf*.

Writing Data to a File

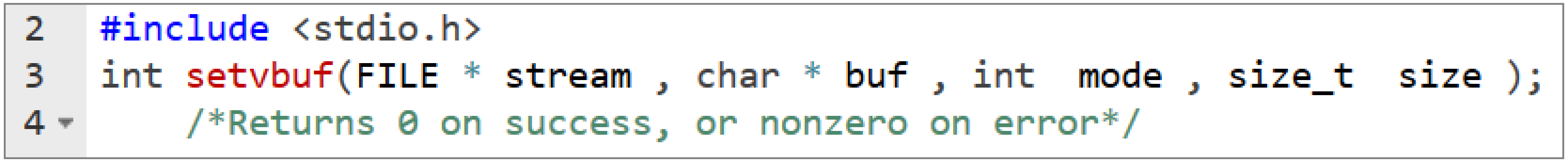


Kernel Buffering of File I/O

- Buffer cache:
  - Set of buffers Kernel maintains to store disk blocks. Seize of buffer cache is adapted as per the availability of the physical memory.
  - When a read() call is issued, Kernel reads the disk block from the disk and stores it in a buffer.
  - Data is copied from buffer cache to the buffer in the user space.
  - Similarly when a user process writes, kernel writes to the buffer.
  - Kernel periodically syncs dirty buffers with disk.
- This allows read() and write to be faster.

Buffering in *stdio* Library

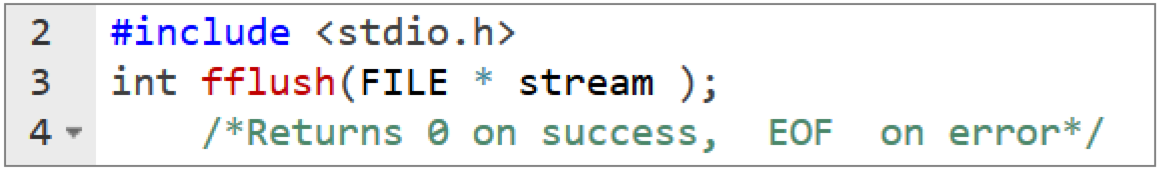
- C library buffers the data to reduce the number system calls (read, write). *fopen()* call opens a buffered stream for a file.



```c
#include <stdio.h>
int setvbuf(FILE * stream , char * buf , int  mode , size_t  size );
    /*Returns 0 on success, or nonzero on error*/
```

  - This is a library function that controls the type of buffering.
  - This function must be called before any I/O operation.
  - If *buf* is null, stdio automatically allocates the buffer for use with the *stream*.
  - mode
    - _IONBF: no buffering. E.g. stderr
    - _IOLBF: line buffering. Default for terminal devices. Output is buffered until newline char. Data is read a line at a time.
    - _IOFBF: fully buffered I/O. data is read or written in units of buffer size. Default for disk files.

Flushing a stdio Buffer

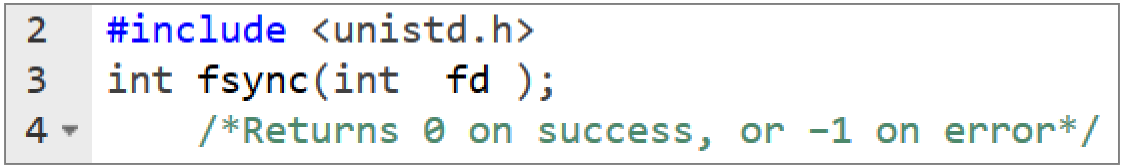


```c
#include <stdio.h>
int fflush(FILE * stream );
    /*Returns 0 on success,  EOF  on error*/
```

- Regardless of the current buffering mode, at any time, we can force the data to written.
- fflush() function flushes the output buffer for that particular FILE stream.
    - It can be used on input stream also.

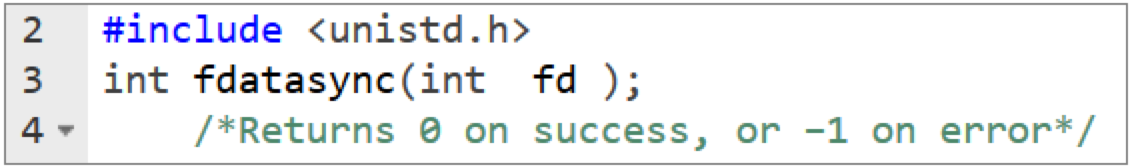
Controlling Kernel Buffering of File I/O

- Two type of synchronization
  - Synchronized I/O file integrity
    - Both data and meta data about the file are synchronized with disk.
  - Synchronized I/O data integrity
    - Only data is synchronized with the disk.



```c
#include <unistd.h>
int fsync(int  fd );
    /*Returns 0 on success, or -1 on error*/
```

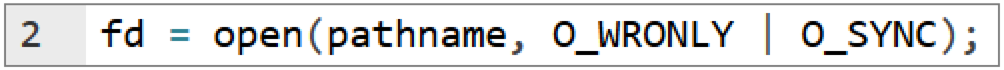
- Flushes both data and metadata such as file size, time stamps etc associated with *fd*.



```c
#include <unistd.h>
int fdatasync(int  fd );
    /*Returns 0 on success, or -1 on error*/
```

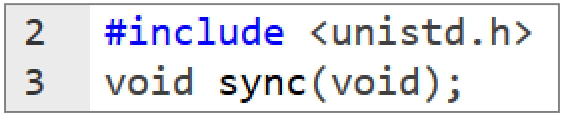
- Flushes only data buffers of file descriptor *fd.*

Controlling Kernel Buffering of File I/O



```c
fd = open(pathname, O_WRONLY | O_SYNC);
```

- If we use O_SYNC flag while opening a file, after every write, both data and metadata will be flushed to disk.

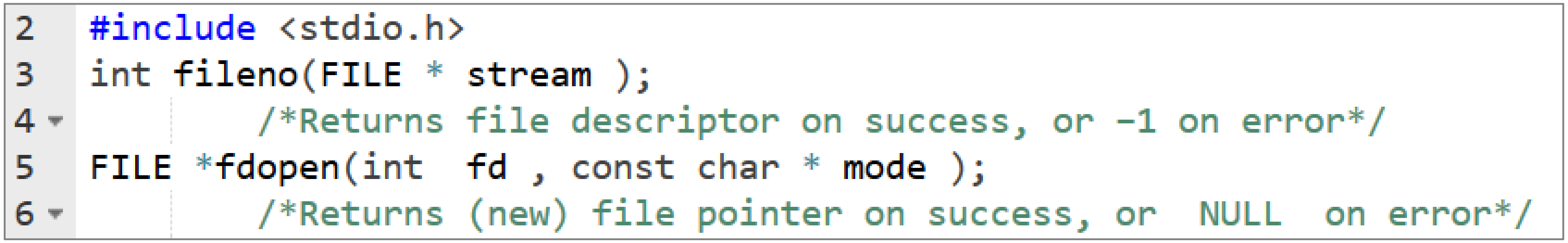


```c
#include <unistd.h>
void sync(void);
```

- It causes all kernel buffers containing modified data including metadata to be flushed to disk.
- Call returns only after syncing.

Mixing Library Functions and I/O Sys Calls

- Sys calls: open(). read(), write(), close()
  - Work with file descriptors
- Library functions: fopen(), fprintf(), fscanf(), fclose(), …
  - Work with FILE streams



```
#include <stdio.h>
int fileno(FILE * stream );
        /*Returns file descriptor on success, or -1 on error*/
FILE *fdopen(int  fd , const char * mode );
        /*Returns (new) file pointer on success, or  NULL  on error*/
```

  - Given a stream, *fileno*() returns the corresponding *fd*
  - Given a file descriptor, *fdopen()* creates corresponding FILE stream.
- *fdopen()* is useful while dealing with pipes and sockets.

Q&A

innovate achieve lead

BITS Pilani

Pilani Campus

Thank You