BITS Pilani
Pilani Campus

Network Programming

K Hari Babu
Department of Computer Science & Information Systems

Outline

- C Library
- Error Handling
- File I/O

innovate achieve lead

BITS Pilani
Pilani Campus

Sys Calls & C Library

Sys Calls & C Library

- System call is a controlled entry point into the kernel.
- For every sys call there is a wrapper function in C library.
  - All library functions are not sys calls.
  - We use wrapper functions in programs
  - fork(), execve() …
- Wrapper function copies the arguments into specific registers. Also copies sys call number into a specific CPU register.
- Wrapper functions executes trap instruction int 0x80.
  - Causes CPU to switch from user mode to kernel mode.
- Kernel executes *system_call* routine at *arch/x86/kernel/entry.S*

Sys Calls & C Library

- On i386, the parameters of a system call are transported via registers.
  - The system call number goes into %eax
  - the first parameter in %ebx
  - the second in %ecx
  - the third in %edx
  - the fourth in %esi
  - the fifth in %edi
  - the sixth in %ebp.

Sys Calls & C Library

- *system_call()* routine:
  - Saves register values onto the stack
  - Checks the validity of the system call number.
  - Invokes the corresponding service routine.
  - Service routine returns a result status to the system_call routine.
  - Restores register values from the kernel stack and places the system call return value on the user process stack.
  - Returns to the wrapper function, simultaneously returning the process to enter user mode.
- Wrapper function checks the return value and if it is an error it sets the *errno* variable.
- System calls incur an appreciable overhead.
  - Calling a C library wrapper function is synonymous with invoking the corresponding system call routine.

C Library

- Many library functions do not make use of system calls.
- Often library functions provide a more caller-friendly interface than the underlying sys call.
  - fopen() uses open()
  - printf uses write()
  - malloc() uses brk()
- GNU C library (*glibc*)
  - libc.so.6
  - Where is it stored? List dynamic dependencies
    - `ldd a.out`
  - Finding the version
    - ./libc.so.6

BITS Pilani
Pilani Campus

Handling Errors

Errors from System Calls

- A service routine in kernel returns a negative number in case of error. The negative number corresponds to standard error codes.

/usr/include/asm-generic/errno-base.h

```
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define EPERM        1  /* Operation not permitted */
#define ENOENT       2  /* No such file or directory */
#define ESRCH        3  /* No such process */
#define EINTR        4  /* Interrupted system call */
#define EIO      5  /* I/O error */
#define ENXIO        6  /* No such device or address */
#define E2BIG        7  /* Argument list too long */
#define ENOEXEC      8  /* Exec format error */
#define EBADF        9  /* Bad file number */
#define ECHILD      10  /* No child processes */
#define EAGAIN      11  /* Try again */
#define ENOMEM      12  /* Out of memory */
#define EACCES      13  /* Permission denied */
#define EFAULT      14  /* Bad address */
#define ENOTBLK     15  /* Block device required */
```

Errors from System Calls

- Incase of error, wrapper function sores the positive value of the error code into errno variable and returns -1.

```
cnt = read(fd, buf, numbytes);
if (cnt == -1) {
    if (errno == EINTR)
        fprintf(stderr, "read was interrupted by a signal\n");
    else {
        /* Some other error occurred */
    }
}
```

- perror() and strerror() can be used to print the error.

```
fd = open(pathname, flags, mode);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

errno variable

- It is present one per each process.
- It is set in wrapper function after sys call returns error.
    - Every time it is over written.
    - So only after a sys call returns -1, we refer to errno.

Handling Errors from Library Functions

- Some library functions return error information exactly like system calls.
  - Return -1
  - errno is set
- Some return value other than -1 but set errno
  - fopen
- Others do not use errno at all
  - gethostbyname

Tracing System Calls

- *strace* command allows us to trace the system calls made by a program.

```
haribabuk@haribabuk-VirtualBox ~ $ strace ./a.out
execve("./a.out", ["./a.out"], [/* 48 vars */]) = 0
brk(0)                                  = 0x10e9000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd26b5e2000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=128950, ...}) = 0
mmap(NULL, 128950, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fd26b5c2000
close(3)                                = 0
```

- Each system call is displayed with input and output arguments
- Arguments are printed in symbolic form.
- *ltrace* is used for tracing libray calls.

```
haribabuk@haribabuk-VirtualBox ~ $ ltrace ./a.out
__libc_start_main(0x4005f4, 1, 0x7fffd10e0ba8, 0x400700, 0x400790 <unfinished ...>
socket(2, 3, 1)                                                          = -1
recvfrom(0xffffffff, 0x7fffd10e04d0, 1500, 0, 0)                         = -1
+++ exited (status 0) +++
```

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.

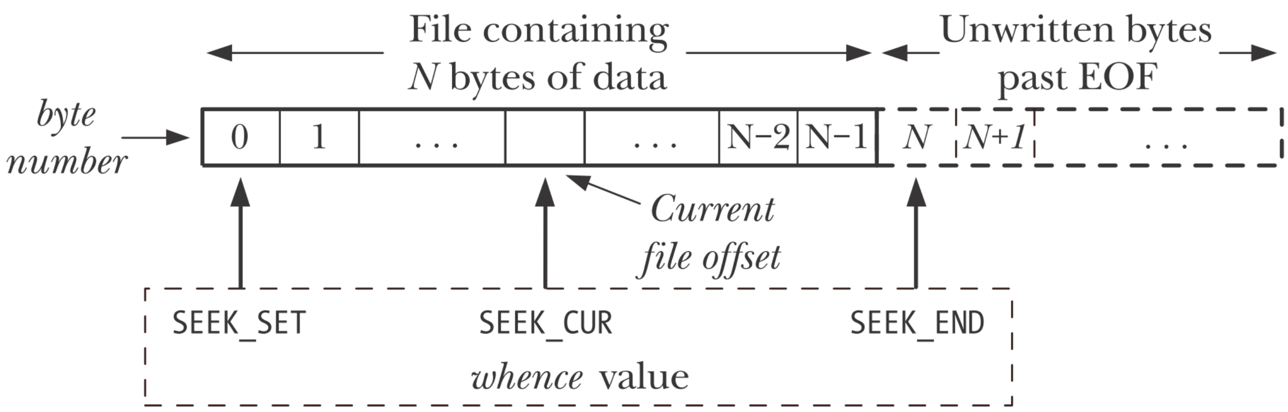
File I/O: lseek()

```c
#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()*

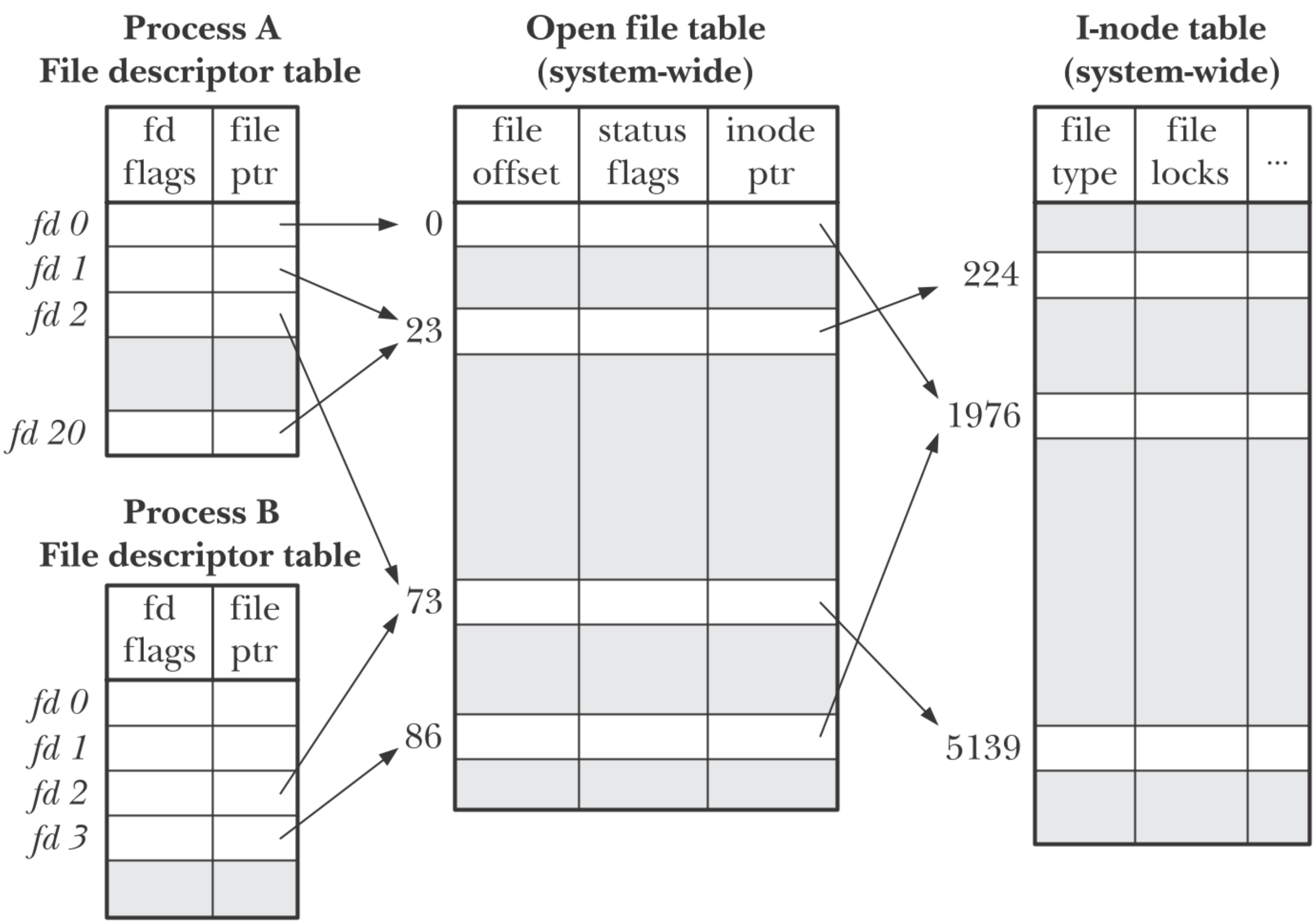
- e.g. for updating flags of inode, *ioctl()* is used.

```
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()

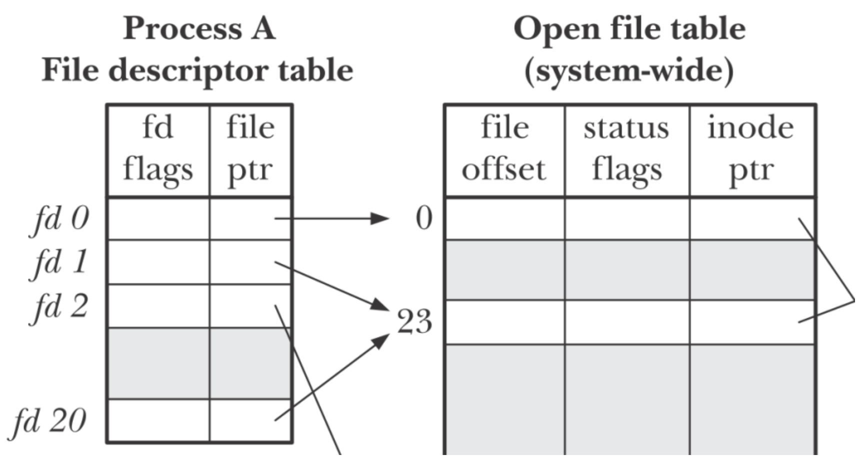
```
$ ./myscript > results.txt 2>&1
```

- 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.

```c
#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

```c
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.

File I/O Buffering

- Buffer cache:
  - Kernel reads the data from the disk and stores it in a buffer.
  - When a process request read(), the data is copied from kernel buffer to 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.
- Use fsync(int fd) to forcefully sync buffers with disk.

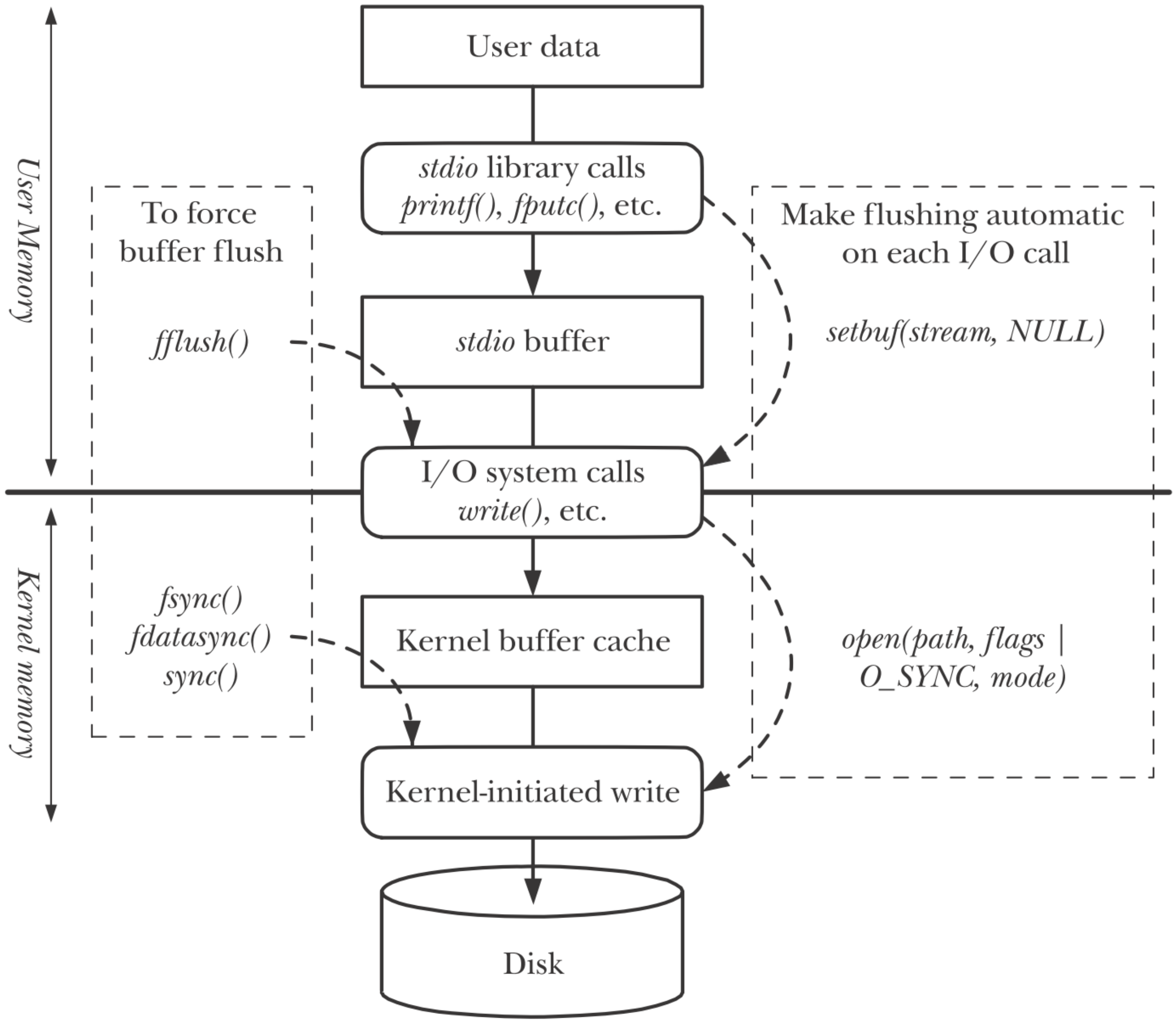
Buffering in *stdio* Library

- C library buffers the data to reduce the number system calls (read, write).

```
#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.
    - Use fflush() for flushing the buffer.

Buffering



User Memory
Kernel memory
User data
stdio library calls
printf(), fputc(), etc.
stdio buffer
I/O system calls
write(), etc.
Kernel buffer cache
Kernel-initiated write
Disk
To force
buffer flush
fflush()
fsync()
fdatasync()
sync()
Make flushing automatic
on each I/O call
setbuf(stream, NULL)
open(path, flags |
O_SYNC, mode)

Q&A

Next Time

- Please read through R1: chapters 6,7, 24-28 innovate achieve lead

BITS Pilani
Pilani Campus

Thank You