Database Management Systems Lab

Vijaya Saradhi

IIT Guwahati

Sat, 25th Jan 2020

Overview

Block Diagram

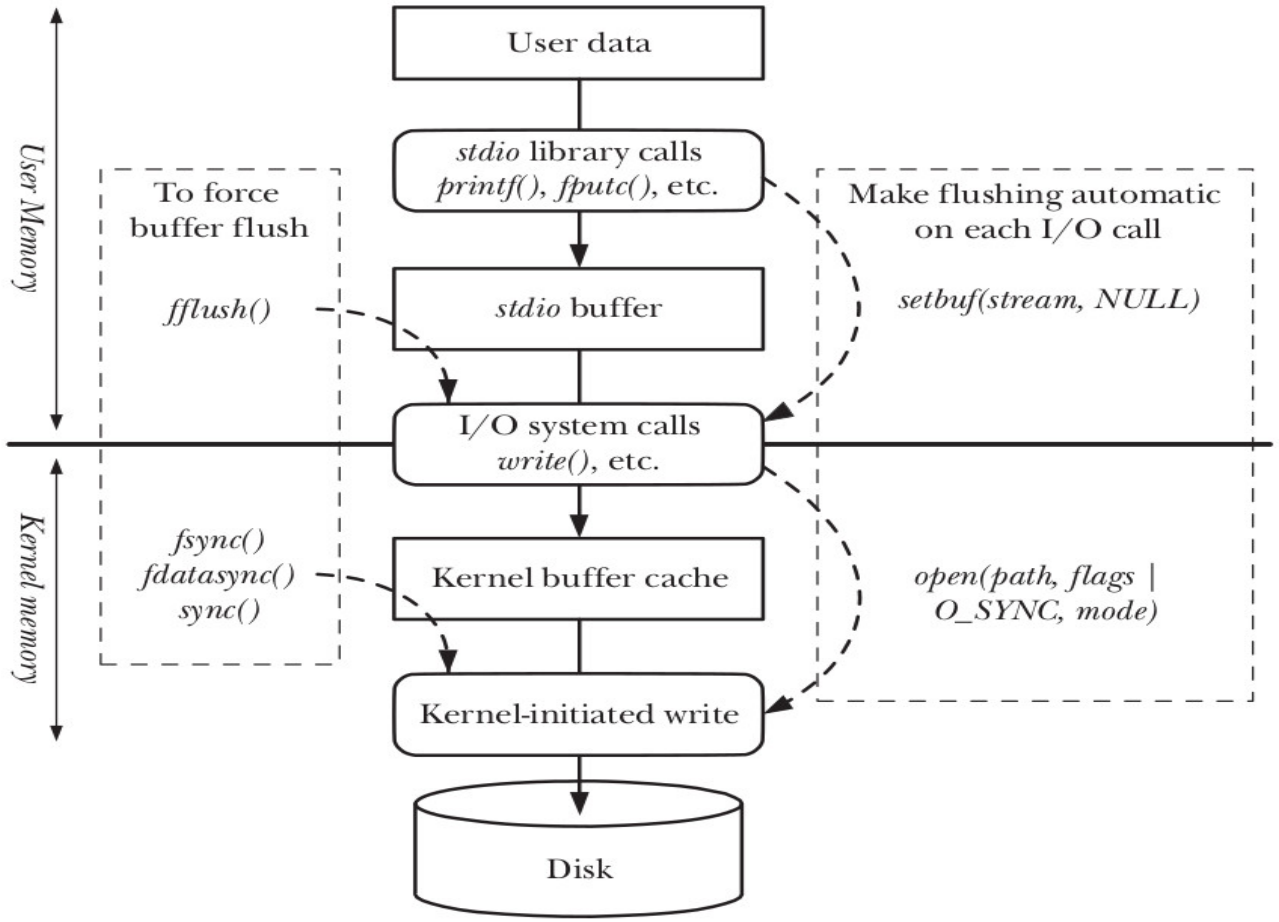

Figure 13-1: Summary of I/O buffering

Description

System Calls Introduction

- A fundamental interface between an application and the (Linux) kernel
- System call is an entry point into the kernel
- System calls are not invoked directly
- They are invoked through wrapper functions in library functions
- Most system calls have corresponding library functions
- Example: stdio: fscanf()
- Making a system call looks same as invoking a normal library function
- Every system call is associated with a unique number

Wrapper Functions

Responsibilities

Work In many cases C library functions does no more than the following

Copying arguments and the unique system call number to the registers where the kernel expects them

Mode Switching Trapping to kernel mode, at which point the kernel does the real work of the system call

Errors setting errno if the system call returns an error number when the kernel returns the CPU to user mode.

Wrapper Functions

Work

Work In a few cases, wrapper functions work beyond copying such as:

Preprocessing the arguments before entering into the kernel mode

Postprocessing the returned values by the system call before getting into the user mode

Errors processing of errors

- When system call fail, they return negative values.
- The library wrapper functions copies the absolute values into errno variable
- Returns -1 as the value in case of failure

Manual Pages

Systems' manuals

```
man <section-number> <title>
```

Section numbers About

- Manual pages are organized into several sections
- Each section has a designated number
- Each section contains specific topic

section-number	topic
1	Executable programs or shell commands
2	System calls (functions provided by the kernel)
3	Library calls (functions within program libraries)
4	Special files (usually found in `/dev`)
5	File formats and conventions eg `/etc/passwd`
6	Games
7	Miscellaneous (macro packages and conventions)
8	System administration commands (root only)
9	Non standards (Kernel routines)

Manual Pages

Systems' manuals - Examples

section-number	topic
1	`man ls`
2	`man 2 open`
3	`man 3 printf`
4	Special files (usually found in `/dev`)
5	`man 5 /etc/wgetrc`
6	Games
7	`man 7 groff_mdoc`
8	`man 8 mount`
9	Non standards (Kernel routines)

Manual Pages

Structure

Name Name of the manual page

Synopsis A brief summary of the command or function's interface.

Configuration Configuration details for a device

Description An explanation of what the program, function, or format does

- Options
- Exit status
- Return value
- Errors
- Environment
- Files
- Versions
- Attributes
- Conforming to
- Notes
- Authors
- Bugs
- Example

See Also Related manual pages

Overview

File Descriptor Overview

- System calls performing I/O refers to open files using file descriptor
- It is a non-negative integer
- Refers to files, pipes, FIFOs, sockets, terminals, devices and regular files
- Each process has its own set of file descriptors. Your fd is different from mine.

Standard

File Descriptor

0 Standard input `stdin`

1 Standard output `stdout`

2 Standard error `stderr`

Standard Output

Example

```
#include<stdio.h>
int main()
{
    fprintf(stdout, "Hello world\n");
    return 0;
}
```

Note: fprintf arguments expect FILE pointer

Standard Input

Example

```
#include<stdio.h>
int main()
{
    int i = 0, j = 0;
    fscanf(stdin, "%d%d", &i, &j);
    fprintf(stdout,"%d %d\n", i, j);
    return 0;
}
```

Note: fscanf arguments expect FILE pointer

File I/O

System calls for file I/O

- open
- read
- write
- close

File opening

Input arguments

- `pathname`: string
- `flags`: binary number constructed using bitwise operations
- `mode`: binary number constructed using bitwise operations (optional)

Return parameter

a file descriptor (integer)

First input argument

Pathname

- A string referring to an existing or non-existing file name
- Path can be relative or absolute

Second input argument - 01

flags

- Specifies how the file should be opened
- There is comprehensive list for this flag construction

Second input argument - 02

flags

Flag	Purpose
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing

Second input argument - 03

flags usage - Reading alone

```
open("grades01.csv", O_RDONLY);
```

flags usage - Writing alone

```
open("grades-sorted-Q.csv", O_WRONLY);
```

flags usage - Reading & Writing

```
open("allow-read-write.txt", O_RWR);
```

Second input argument - 04

flags

Flag	Purpose
O_CREAT	Create file if it doesn't exist already
O_TMPFILE	Create a temporary file (deleted upon closing file)
O_DIRECT	File I/O bypass buffer cache
O_TRUNC	Truncate existing file to zero length

Second input argument - 05

More on flags

- pathname assumes existence of the file
- If the file is not present in the path, errno is set appropriately (and -1) is returned
- Can tell open what to do in case file is not found
- Using bit-wise operators to compose desired result
- Pipe: | denotes bit-wise OR operation

```
open("grades01.csv", O_WRONLY | O_CREAT | O_TRUNC);
```

Second input argument - 06

flags

Flag	Purpose
O_APPEND	Writes always appended to the end of file
O_ASYNC	Generates a signal when I/O is possible
O_NONBLOCK	Open in non-blocking mode

Third input argument - 01

mode/permissions

- We know file has permissions
- Owner: Read, write, execute
- Group: Read, write, execute
- Other: Read, write, execute
- Specify the file permission

Third input argument - 02

mode/permissions bits

- Owner: S_IRUSR, S_IWUSR, S_IXUSR
- Group: S_IRGRP, S_IWGRP, S_IXGRP
- Other: S_IRUSR, S_IWOTH, S_IXOTH

Third input argument - 03

Example

```
file_permissions = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
    S_IROTH | S_IWOTH;
//  rw-rw-rw-

file_permissions = S_IRUSR | S_IRGRP | S_IROTH;
//r--r--r--
```

values

USR		GRP		OTH	
Variable	Value	Variable	Value	Variable	Value
S_IRUSR	00400	S_IRGRP	00400	S_IROTH	00400
S_IWUSR	00200	S_IWGRP	00200	S_IWOTH	00200
S_IXUSR	00100	S_IXGRP	00100	S_IXOTH	00100

Return value

What open returns?

- Returns a file descriptor (an integer)
- First three values are reserved: 0, 1 & 2 for stdin, stdout & stderr
- Upon failure, `errno` will be set appropriately
- -1 will be returned
- 35 possible error conditions are listed in the documentation

open - example

Example

```
int outputFd, openFlags;
mode_t filePerms;
openFlags = O_CREAT | O_WRONLY | O_TRUNC;
filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH |
    S_IWOTH;

outputFd = open(argv[2], openFlags, filePerms);
```

read input/output arguments

Input arguments

- `file descriptor`: int
- `buffer`: pointer to an array (pointer is of any data type)
- `size`: int number of bytes to be read into the buffer

Return parameter

Number of bytes read into the buffer

read - example

Example

```
#define BUF_SIZE 1024

int inputFd, openFlags;
mode_t filePerms;
ssize_t numRead;
char buf[BUF_SIZE]

openFlags = O_RDONLY;

inputFd = open(argv[1], openFlags);

/* Read 1024 bytes into character array buf */
numRead = read(outputFd, buf, BUF_SIZE);
```

read - pitfalls - 01

Example

```
#define BUF_SIZE 20
char buf[BUF_SIZE];

/* Read BUF_SIZE bytes into character array buf */
numRead = read(STDIN_FILENO, buf, BUF_SIZE);
printf("Input data was: %s\n", buf);
```

read - pitfalls - 02

NULL character?

- If you have read all BUF_SIZE bytes there is no issue
- If less than BUF_SIZE was read then additional characters will also gets included
- Reason: this system call reads bytes and it interpretation is not the business of read system call
- read system call reads both text and binary integers
- There is no way to tell what is the terminating character!
- User must explicitly place the NULL character

write input/output arguments

Input arguments

- `file descriptor`: int
- `buffer`: pointer to an array (pointer is of any data type)
- `size`: int number of bytes to be written

Return parameter

Number of bytes written

write - example

Example

```
#define BUF_SIZE 1024

int outputFd, openFlags;
mode_t filePerms;
ssize_t numRead;
char buf[BUF_SIZE]

openFlags = O_CREAT | O_WRONLY | O_TRUNC;
filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH |
    S_IWOTH;

outputFd = open(argv[2], openFlags, filePerms);

/* Read 1024 bytes into character array buf */
numRead = write(outputFd, buf, BUF_SIZE);
```

About write

Discussion

- On success write returns number of bytes actually written
- This may be less than size
- May be disk was full and hence partial write happened
- A successful return from `write` does not guaranttee that data has been transferred to disk
- Kernel performs the buffering of disk I/O

close

input arguments

- file descriptor to be closed

Return value

- 0 on success
- -1 on error; errno set appropriately

close - example

Example

```
close(outputFd);
```

Complete example - 01

Example

```
/* for open */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* for read, write and close */
#include <unistd.h>

/* User buffer size */
#define BUF_SIZE 1024
int main(int argc, char *argv[])
{
   int inputFd, outputFd, openFlags;
   mode_t filePerms;
   ssize_t numRead;
   char buf[BUF_SIZE];
```

Complete example - 02

Example

```
if (argc != 3 || strcmp(argv[1], "--help") == 0)
{
   printf("%s old-file new-file\n", argv[0]);
       exit(EXIT_FAILURE);
}
/* Open input and output files */
inputFd = open(argv[1], O_RDONLY);
if (inputFd == -1)
   printf("opening file %s", argv[1]);

/* rw-rw-rw- */
openFlags = O_CREAT | O_WRONLY | O_TRUNC;
filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
            S_IROTH | S_IWOTH;

outputFd = open(argv[2], openFlags, filePerms);
if (outputFd == -1)
   printf("opening file %s", argv[2]);
```

Complete example - 03

Example

```
/* Transfer data until we encounter end of input or an
   error */
while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0)
   {
   if (write(outputFd, buf, numRead) != numRead)
   {
      printf("couldn't write whole buffer");
            exit(EXIT_FAILURE);
   }
   }
if (numRead == -1)
   printf("read error");
if (close(inputFd) == -1)
   printf("closing the input file error");
if (close(outputFd) == -1)
   printf("closing the output file error");
exit(EXIT_SUCCESS);
}
```

Kernel Buffering of File I/O - 01

Buffering

- The `read()` and `write()` do not directly initiate disk access
- They copy data between user-space buffer to kernel space buffer

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

- `write()` transfers three bytes data from user buffer space to buffer in kernel space
- Once buffer to buffer copying is complete, `write` will return
- Kernel writes its buffer to disk
- Therefore system call not synchronized

Kernel Buffering of File I/O - 02

Block Diagram

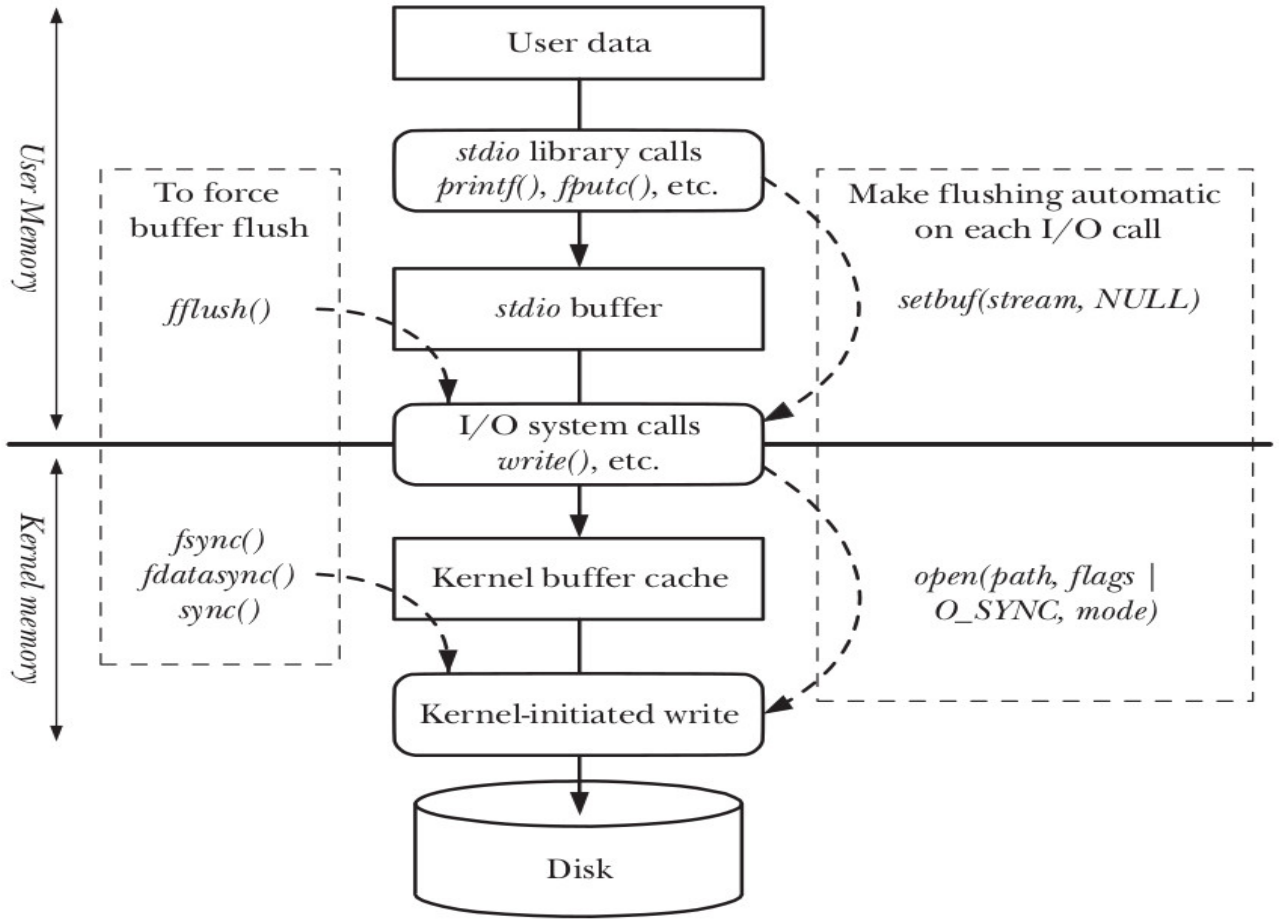

Figure 13-1: Summary of I/O buffering

Kernel Buffering of File I/O - 03

Discussion

- Between the time kernel writes to a file, when a read requests arrives, kernel supplies data from the buffer
- For input, kernel reads data from the disk and stores in kernel buffer
- Calls to `read()` fetch data from the kernel buffer
- Until the buffer is empty
- At this point, kernel initiates next segment of file into the buffer

Effect of buffer size on I/O system call performance - 01

Discussion

- Kernel performs same number of disk accesses regardless of
  - We perform 1000 writes of a single byte OR
  - We perform single write of 1000 bytes
- First one incurrs 1000 write system calls
- Where as the second one incurrs one write system call
- However, buffer size will impact the performance of the system
- When BUF_SIZE is varied between 1 and 65536 in the powers of 2, the time is as shown

Effect of buffer size on I/O system call performance - 02

Read 100 Million bytes

With various BUF_SIZE

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	107.43	107.32	8.20	99.12
2	54.16	53.89	4.13	49.76
4	31.72	30.96	2.30	28.66
8	15.59	14.34	1.08	13.26
16	7.50	7.14	0.51	6.63
32	3.76	3.68	0.26	3.41
64	2.19	2.04	0.13	1.91
128	2.16	1.59	0.11	1.48
256	2.06	1.75	0.10	1.65
512	2.06	1.03	0.05	0.98
1024	2.05	0.65	0.02	0.63
4096	2.05	0.38	0.01	0.38
16384	2.05	0.34	0.00	0.33
65536	2.06	0.32	0.00	0.32

Effect of buffer size on I/O system call performance - 03

Write 100 Million bytes

With various BUF_SIZE

Table 13-2: Time required to write a file of 100 million bytes

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	72.13	72.11	5.00	67.11
2	36.19	36.17	2.47	33.70
4	20.01	19.99	1.26	18.73
8	9.35	9.32	0.62	8.70
16	4.70	4.68	0.31	4.37
32	2.39	2.39	0.16	2.23
64	1.24	1.24	0.07	1.16
128	0.67	0.67	0.04	0.63
256	0.38	0.38	0.02	0.36
512	0.24	0.24	0.01	0.23
1024	0.17	0.17	0.01	0.16
4096	0.11	0.11	0.00	0.11
16384	0.10	0.10	0.00	0.10
65536	0.09	0.09	0.00	0.09

Effect of buffer size on I/O system call performance - 04

Read Vs Write of 100 Million bytes

- Read took 2.06 seconds for BUF_SIZE = 65536
- Write took 0.09 seconds for the same BUF_SIZE
- There is no actual disk operation that took place with write system call

stdio Buffering

Library buffers

- Buffering of data into large blocks
- To reduce number of system calls by standard libraries
- stdio relieves the users the task of buffering data for writing output or reading input
- These (library) buffers can be modified

stdio Buffering

setvbuf()

controls buffering employed by stdio library

Input argument 1 FILE pointer

Input argument 2 Character pointer buf

Input argument 3 mode (integer)

- _IONBF: No buffering
- _IOLBF: Employ line-buffering (for terminals)
- _IOFBF: Employ fully buffered I/O

Input argument 4 Size of the buffer (size_t)

Return value nonzero value on error

stdio Buffering - Example

setvbuf()

```
#define BUF_SIZE 1024
static char buf[BUF_SIZE];

if( setvbuf(stdout, buf, _IOFBF, BUF_SIZE) != 0 )
{
    printf("Error setting output buffer size\n");
    exit(EXIT_FAILURE);
}
```

stdio Buffering

setbuf()

a function layered on top of setvbuf()

Input argument 1 FILE pointer

Input argument 2 Character pointer buf

Return value No return value

Equivalent to the following call using setvbuf()

BUFSIZ a macro defined in <stdio.h>

```
setvbuf(fp, buf, (buf != NULL) ? _IOFBF : _IONBF, BUFSIZ);
```

Introduction

Context

- Force flushing of kernel buffers for output files
- This is used in database journaling for ensuring output really written to the disk
- Two important definitions
  - Synchronized I/O data integrity
  - Synchronized I/O file integrity
  - Difference is that file has metadata which is to be written along with data
  - examples: last accessed, last modified, permissions, etc.

Synchronized I/O Completion

Definition

I/O operation that has either successfully transferred or diagnosed as unsuccessful

Synchronized I/O data integrity

Discussion

For Read Requested file data has been transferred from disk

Pending write operations affecting read are transferred to the disk before performing read

For Write Data has been transferred to the disk along with file metadata

Not all modified metata of file need to be transferred

Example: Modified file size vs modified file timestamps. The later is not needed immediately; but the former is needed

System calls for controlling kernel buffering - 01

fsync

`fsync()` : input: file descriptor; output: integer

- Causes the buffered data and all metadata associated with `fd` to be flushed to disk
- Forces the file to the synchronized I/O file integrity completion state
- Needs `unistd.h` header
- `fsync()` returns only after the transfer to the disk device has completed

System calls for controlling kernel buffering - 02

fdatasync

`fdatasync()` : input: file descriptor; output: integer

- This potentially reduces number of disk operations from two to one
- Example: file data has changed but file size has not
- Calling `fdatasync()` forces data to be updated
- However, `fsync()` would force metadata to be transferred to disk

System calls for controlling kernel buffering - 03

sync

`sync()` : input: void; output: void

- Causes all kernel buffers containing updated file information to be flushed to disk
- Example: data blocks, pointer blocks, metadata, ...
- Returns only after all data has been transferred to the disk device

Synchronous writes

Details

- Making all writes synchronous
- `open(pathname, O_WRONLY | O_SYNC)`
- After the above call, every `write()` to file automatically flushes the file data and metadata to disk
- However it has performance impact

Synchronous writes

Experiment

Table 13-3: Impact of the O_SYNC flag on the speed of writing 1 million bytes

BUF_SIZE	Time required (seconds)			
	Without O_SYNC		With O_SYNC	
	Elapsed	Total CPU	Elapsed	Total CPU
1	0.73	0.73	1030	98.8
16	0.05	0.05	65.0	0.40
256	0.02	0.02	4.07	0.03
4096	0.01	0.01	0.34	0.03

Direct I/O - 01

Bypass buffers

- Direct I/O does not mean faster I/O
- Using this for general applications considerably degrade performance
- Intended for applications with special I/O needs
- Direct I/O is not portable. The discussion is very specific to a particular version (and above) of Linux kernel
- On other systems, this has it's equivalence
- Database systems is one such application; it performs own caching, I/O optimization

Direct I/O - 02

Requirements

- The data buffer being transferred must be aligned on a memory boundary that is multiple of block size
- The offset must be a multiple of block size
- length of data to be transferred must be multiple of block size

Direct I/O - 03

Steps

- Step 1: `open` with flag O_DIRECT
- Create memory using `posix_memalign()`. You have learned about
  - `malloc`
  - `free`
- The siblings of the above are
  - `calloc`
  - `realloc`

Direct I/O - 04

Steps

- Step 2: `posix_memalign` takes three arguments
- Example: `posix_memalign(&p, 32, 512)`
  - Create 512 bytes of memory
  - Whose address start address is guaranteed to be multiple of 512
  - That address is stored in `p`
- Step 3: Perform `read()` or `write`