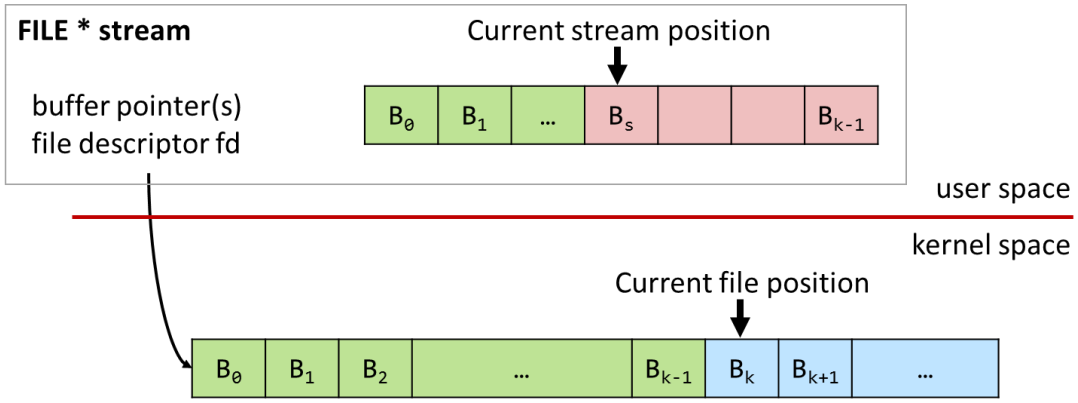
Input/Output

Direct and Buffered I/O



Module Outline

- Unix I/O
- Standard I/O
- Interaction of Standard I/O with Unix I/O
- Module Summary

```
int fd = open(“unix.io”,
              O_CREAT | O_WRONLY,
              S_IRWXU | S_IRGRP);
```

Unix I/O

Unix I/O: Overview

- **Unix I/O refers to the interface of *nix kernels to perform input/output**
  - lowest level of I/O an application programmer can perform
  - Unix I/O is implemented with system calls

- **Design concept: one standardized, general interface to access files of any type**
  - regular files, directories
  - block devices: disks, partitions, …
  - character devices: memory, keyboards, mice, …
  - pipes, sockets, …

  all input and output is handled in a consistent and uniform way!

One single file interface to interact with any kind of device

(well, almost)

Unix I/O: File Abstraction

- An open file is identified by a **file descriptor (handle)**
  - uniquely identifies underlying file
  - handle required for methods to update the state of a file
    - returned when opening/creating a file, used in all subsequent file operations

- Each open file has an internal pointer pointing to the **current position in the file**
  - read/write operations start from that position and advance it
  - subtleties arise when sharing file descriptors

Unix I/O: File Descriptors

- **File descriptors are positive integers uniquely identifying open files of a process**

- Each process begins life with three open file descriptors (defined in unistd.h)
  - 0: STDIN_FILENO standard input
  - 1: STDOUT_FILENO standard output
  - 2: STDERR_FILENO standard error

  - by default, these file descriptors are connected to the terminal and used for input, output, and error output

- Any other file descriptor must be obtained using the Unix I/O API

Unix I/O API

- System calls offered by the kernel
- Convenient wrappers provided by the C standard library (headers fcntl.h, unistd.h)

Operation	API	Variants
Open and create	`int open(const char *pathname, int flags)` `int creat()`	`openat`
Read and write	`ssize_t read(int fd, void *buf, size_t count)` `ssize_t write(int fd, void *buf, size_t count)`	`pread, readv` `pwrite, writev`
Positioning	`off_t lseek(int fd, off_t offset, int whence)`	`llseek, lseek64`
Close	`int close(int fd)`	
Remove from filesystem	`int remove(const char *pathname)`	`unlink, unlinkat, rmdir`

Hint: finding the base type of unknown type *t*

```
$ echo "#include <stdlib.h>" | gcc -E - | grep -w _*t
```

Opening and Creating Files

- Opening a file informs the kernel that you are getting ready to access that file

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

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

int creat(const char *pathname, mode_t mode);
```

  - pathname: name (and path) to file
  - flags: status and creation flags such as O_RDONLY, O_APPEND, …
  - mode: file mode for newly created files such as S_IRWXU, S_IRGRP, …
  - refer to manpage for details

- Returns a file descriptor
  - -1 indicates that an error has occurred

Opening and Creating Files

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
  perror("Cannot open file");
  exit(EXIT_FAILURE);
}
```

```
int fd = creat(“log", S_IRUSR|S_IWUSR);

if (fd < 0) {
  perror("Cannot create file");
  exit(EXIT_FAILURE);
}
```

Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
#include <unistd.h>

int close(int fd);
```

  - fd: (open) file descriptor

- Returns 0 on success, -1 on error

- Closing an already closed file is an error
  - may cause unexpected failures in multithreaded programs

Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;
int retval;

…

if ((retval = close(fd)) < 0) {
  perror("Cannot close file");
  exit(EXIT_FAILURE);
}
```

Reading and Writing Files

- Reading/writing a file copies *count* bytes from the current file position to memory or vice-versa, and then updates the file position

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

  - fd: file descriptor
  - buf: buffer that holds data
  - count: number of bytes to read/write

- Returns number of bytes read/written
  - Return value of < 0 indicates that an error occurred (see manpage)
  - Short counts, i.e., retval < sizeof(buf), are possible and not necessarily errors (more details later)

Reading Files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];
int fd;
ssize_t nbytes;

…

if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
  perror("Cannot read from file");
  exit(EXIT_FAILURE);
}
```

Writing Files

- Writing a file copies bytes from memory to the current file position, and then updates current file position

```
char buf[512];
int fd;
ssize_t nbytes;

…

if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
  perror("Cannot write to file");
  exit(EXIT_FAILURE);
}
```

“Hello, world!” with Unix I/O

- Write contents of buffer containing “Hello, world!” to STDOUT_FILENO
  - write() requires us to specify the number of bytes to write
    - use ‘strlen()’ from the string API (string.h) is correct here because the string ‘str’ includes a terminating \0 character at its end which is included in sizeof(str) but ignored by ‘strlen(str)’

```c
#include <stdlib.h>
#include <unisth.h>
#include <string.h>

char str[] = "Hello, world\n";

int main(void)
{
  write(STDOUT_FILENO, str, strlen(str));
  return EXIT_SUCCESS;
}
```

unixio/helloworld.c

  - in principle, we should always check the return value of write()

“Hello, world!” with Unix I/O

- Write contents of buffer containing “Hello, world!” into a file
  - While STDOUT_FILENO is already open & available to be written to, we have to open (create) the output file explicitly here

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <stdio.h>

char str[] = "Hello, world!\n";

int main(void) {
  int fd = open("./output.txt", O_WRONLY | O_CREAT | O_APPEND,
                                S_IRUSR|S_IWUSR | S_IRGRP | S_IROTH);

  if (fd == -1) {
    perror("Cannot open/create file");
    return EXIT_FAILURE;
  }

  write(fd, str, strlen(str));
  close(fd);

  return EXIT_SUCCESS;
}
```

unixio/helloworld2file.c

Seeking in Files

- Change the file position

```
#include <sys/types.h>
#include <unistd.h>

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

  - fd: file descriptor
  - offset: offset in relation to whence
  - whence: base
    - SEEK_SET: filepos = offset
    - SEEK_CUR: filepos = filepos + offset
    - SEEK_END: filepos = end + offset (can seek beyond end of file)

- Returns new absolute position or -1 to indicate that an error has occurred

Seeking in Files

- Change the file position

```
int fd;

…

if (lseek(fd, 100, SEEK_SET) < 0) {
  perror("Cannot seek in file");
  exit(EXIT_FAILURE);
}
```

Simple Unix I/O example

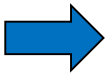
- Copying standard in to standard out, one byte at a time

```
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  char c;

  while (read(STDIN_FILENO, &c, 1) > 0) {
    write(STDOUT_FILENO, &c, 1);
  }

  return EXIT_SUCCESS;
}
```

Short Counts

- A **short count** is a situation when Unix I/O reads/writes fewer bytes than requested, but does not report an error (-1).

```
ssize_t nbytes = read(fd, buf, 512);
printf("Read %ld bytes.\n", nbytes);
```



```
Read 302 bytes.
```

- Short counts can occur in many situations:
  - Encountering EOF (end-of-file) on reads
  - Filesystem full on write
  - Reading text lines from a terminal
  - Reading and writing network sockets or Unix pipes
  - Interrupts and signals sent to the process

- Dealing with short counts requires careful inspection of the return value and errno
  - see manpage for read

Full Unix I/O Example: Copying Data

- Append *n* bytes from offset *ofs* in file *input* to file *output*
  - Take arguments from command line
  - Create output file if necessary
  - Check for errors, in particular, short counts

```
$ ./unixio data.dat test.dat 400000 100000
Opening input file 'data.dat'...
Opening output file 'test.dat'...
Seeking input file to position 400000...
Copying 100000 bytes...
  Short count of 27776 when trying to read 34464 bytes from input
  copied a total of 93312 bytes.
Closing files...
```

```
FILE *f = fopen("standard.io",
                "r+");
```

Standard I/O

Shortcomings of Unix I/O

- Applications often read/write character data
  - read/write one character at a time
  - read/write one line of text at a time

- Unix I/O
  - calls are mapped 1:1 to system calls
    - rather expensive: > 10,000 clock cycles
  - not well suited for string handling

- Copying 10 MiB byte-by-byte: Unix I/O vs Standard I/O

```
$ time ./copy_unixio …

real    0m2.679s
user    0m0.363s
sys     0m2.316s
```

```
$ time ./copy_stdio …

real    0m0.261s
user    0m0.261s
sys     0m0.000s
```

Standard I/O

- **Goals**
  1. reduce overhead of I/O
  2. provide formatted I/O

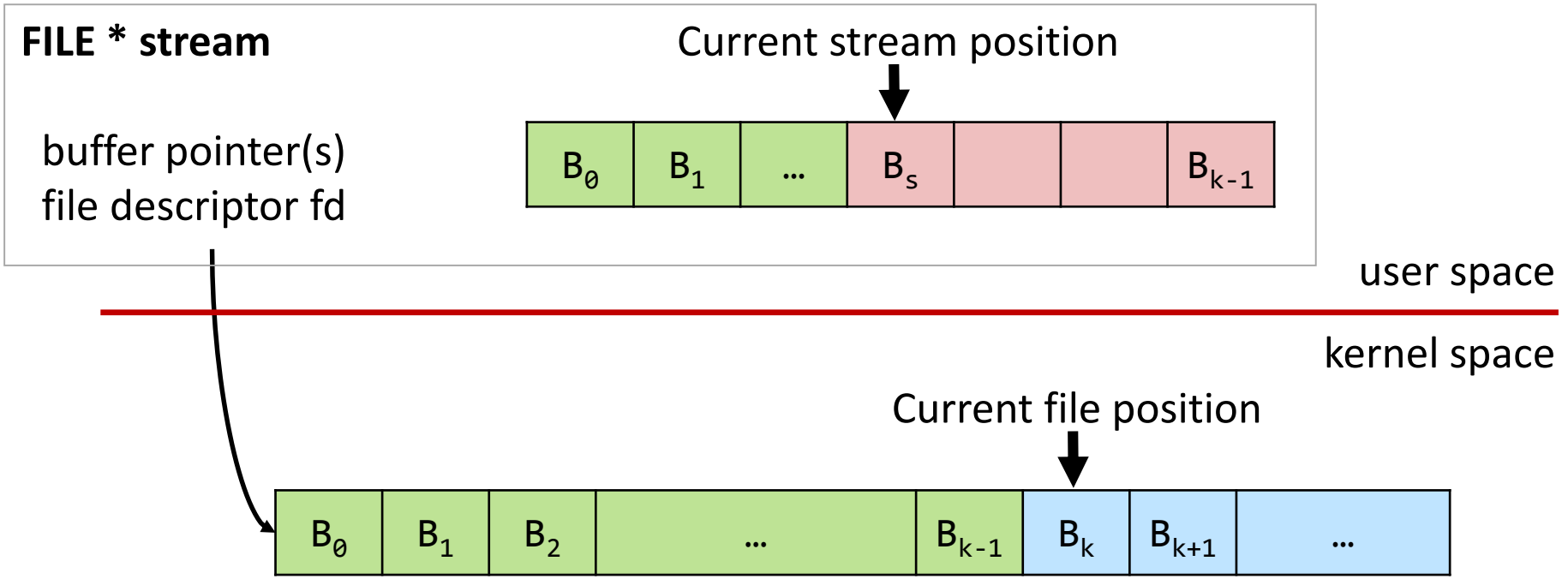
Standard I/O

- **Achieving the first goal:** use a buffer to reduce number of read() and write() system calls
  - Buffering is (mostly) transparent to the user
  - Three types of buffering: fully buffered, line buffered, and unbuffered

- **Achieving the second goal: provide a formatted I/O library (fprintf/fscanf family)**
  - Operates on top of Standard I/O streams
  - Independent of buffering mode

Unix I/O vs Standard I/O

- Standard I/O: buffered read
  - Use Unix read to grab block of bytes
  - User input functions take one byte at a time from buffer
    - Refill buffer when empty



Standard I/O API

- Collection of higher-level standard I/O functions implemented in C standard library (header stdio.h)

Operation	API	Variants
Open and create	`FILE *fopen(const char *path, const char *mode)`	`fdopen`, `fdreopen`
Read and write	`size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)` `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)`	
Positioning	`int fseek(FILE *stream, long offset, int whence)` `long ftell(FILE *stream)`	`rewind`, `fsetpos` `fgetpos`
Close	`int fclose(FILE *stream)`	`fcloseall`

Others	`int fflush(FILE *stream)`
	`int feof(FILE *stream)`
	`int ferror(FILE *stream)`
	`int fileno(FILE *stream)`	

Standard I/O API

- Standard I/O also includes well-known functions for formatted input/output

Operation	API	Variants
Read character or line	`char *fgets(char *s, int size, FILE *stream)`	~~`gets`~~, `fgetc`, `getc`, `getchar`, `ungetch`
Read formatted input	`int fscanf(FILE *stream, const char *format, …)`	`scanf`, `sscanf`, `vscanf`, …
Write character or line	`char fputs(const char *s, FILE *stream)`	`fputc`, `putc`, `putchar`, `puts`
Write formatted output	`int fprintf(FILE *stream, const char *format, …)`	`printf`, `dprintf`, `sprint`, `snprintf`, `vprintf`, …

Standard I/O Streams

- The default Unix I/O STD*_FILENO file descriptors are mapped to corresponding I/O streams (defined in stdio.h)
  - stdin stream for STDIN_FILENO standard input
  - stdout stream for STDOUT_FILENO standard output
  - stderr stream for STDERR_FILENO standard error

```
#include <stdio.h>
#include <stdlib.h>

extern FILE *stdin;  // standard input  (file desc: STDIN_FILENO)
extern FILE *stdout; // standard output (file desc: STDOUT_FILENO)
extern FILE *stderr; // standard error  (file desc: STDERR_FILENO)

int main(void)
{
  fprintf(stdout, "Hello, world\n");
  return EXIT_SUCCESS;
}
```

“Hello, world!” with Standard I/O

- Write contents of buffer containing “Hello, world!” to stdout
  - [f]printf() supports formatted output and calculates the length of the string for you

```c
#include <stdio.h>
#include <stdlib.h>

char str[] = "Hello, world!\n";

int main(void)
{
  fprintf(stdout, "%s", str); // specify output stream
  printf("%s", str);          // implicitly print to stdout

  return EXIT_SUCCESS;
}
```

stdio/helloworld.c

“Hello, world!” with Standard I/O

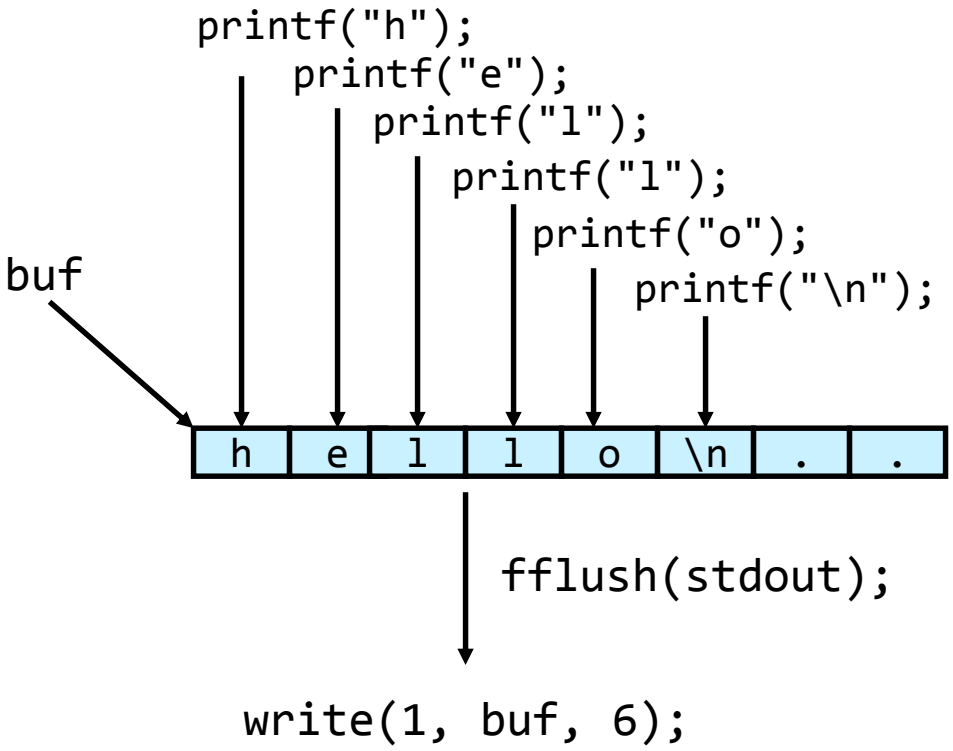
- Write contents of buffer containing “Hello, world!” into a file
  - Also with Standard I/O, we have to explicitly open/close files

```c
#include <stdio.h>
#include <stdlib.h>

char str[] = "Hello, world!\n";

int main(void) {
  FILE *out = fopen("./output.txt", "a+");
  if (out == NULL) {
    perror("Cannot open/create file");
    return EXIT_FAILURE;
  }

  fprintf(out, str);

  fclose(out);

  return EXIT_SUCCESS;
}
```

stdio/helloworld2file.c

Understanding Buffering in Standard I/O

- To flush, or not to flush, that is the question



- Buffer flushed to output fd on "\n" or `fflush()` call
  - on "\n" only when output is a terminal (see following slides)!

Standard I/O Buffering Types

- Buffering is (mostly) transparent to the user, but depends on the underlying file type

- Standard I/O supports three types of buffering

Type	Mode	Default for
Fully buffered	_IOFBF	file descriptor points to a file
Line buffered	_IOLBF	file descriptor connected to a terminal
Unbuffered	_IONBF	standard error or by user request

  - Manually set with setvbuf()

Standard I/O: Fully buffered

- For files

- Unix I/O occurs when
  - fread(): buffer is empty and needs to be filled
  - fwrite(): buffer is full and needs to be emptied

- Flushing (writing the buffer to disk) occurs
  - Automatically when the buffer is full
  - Manually when calling fflush()

Standard I/O: Line Buffered

- Typically used for terminal devices
- Unix I/O occurs when a newline character is encountered in input or output
- Caveats
  - When buffer is full, it is flushed even if there is no newline
  - Lots of head scratching when mixing input and output operations
    - Automatic flushing before any input operation requested by unbuffered or line-buffered streams

Standard I/O: Unbuffered

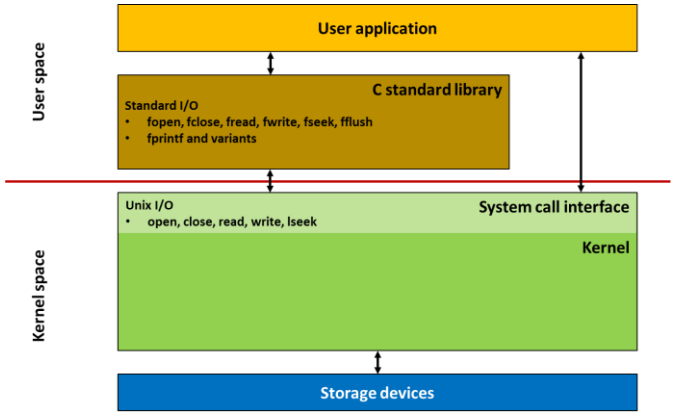
- Uses no buffering
  - Standard I/O FILE interface for Unix I/O
- By default, the standard error stream is unbuffered
  - All output is written immediately to the terminal

Standard I/O Buffering in Action

- Observe buffering in action using the always fascinating Unix strace program:

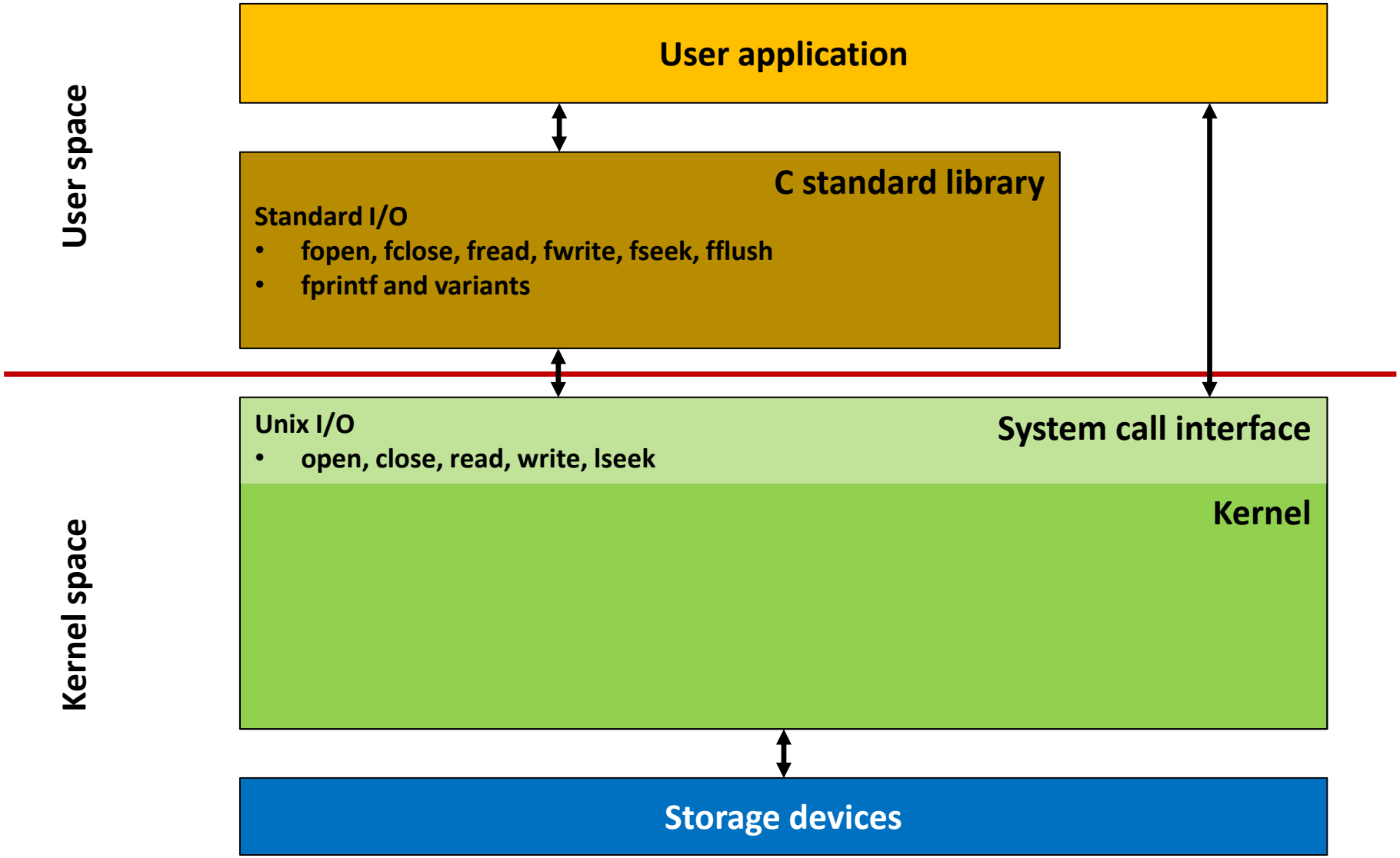
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  setvbuf(stdout, NULL, _IOLBF, 0);

  printf("h");
  printf("e");
  printf("l");
  fflush(stdout);
  printf("l");
  printf("o");
  printf("\n");
  return EXIT_SUCCESS;
}
```

```
$ strace -o log ./bufferedio x
…
$ cat log
execve("./bufferedio", ["bufferedio", "x"], …)
…
write(1, "hello\n", 6)                  = 6
…
write(1, "hello,", 6)                   = 6
…
write(1, "world\n", 6)                  = 6
exit_group(0)                          = ?
```

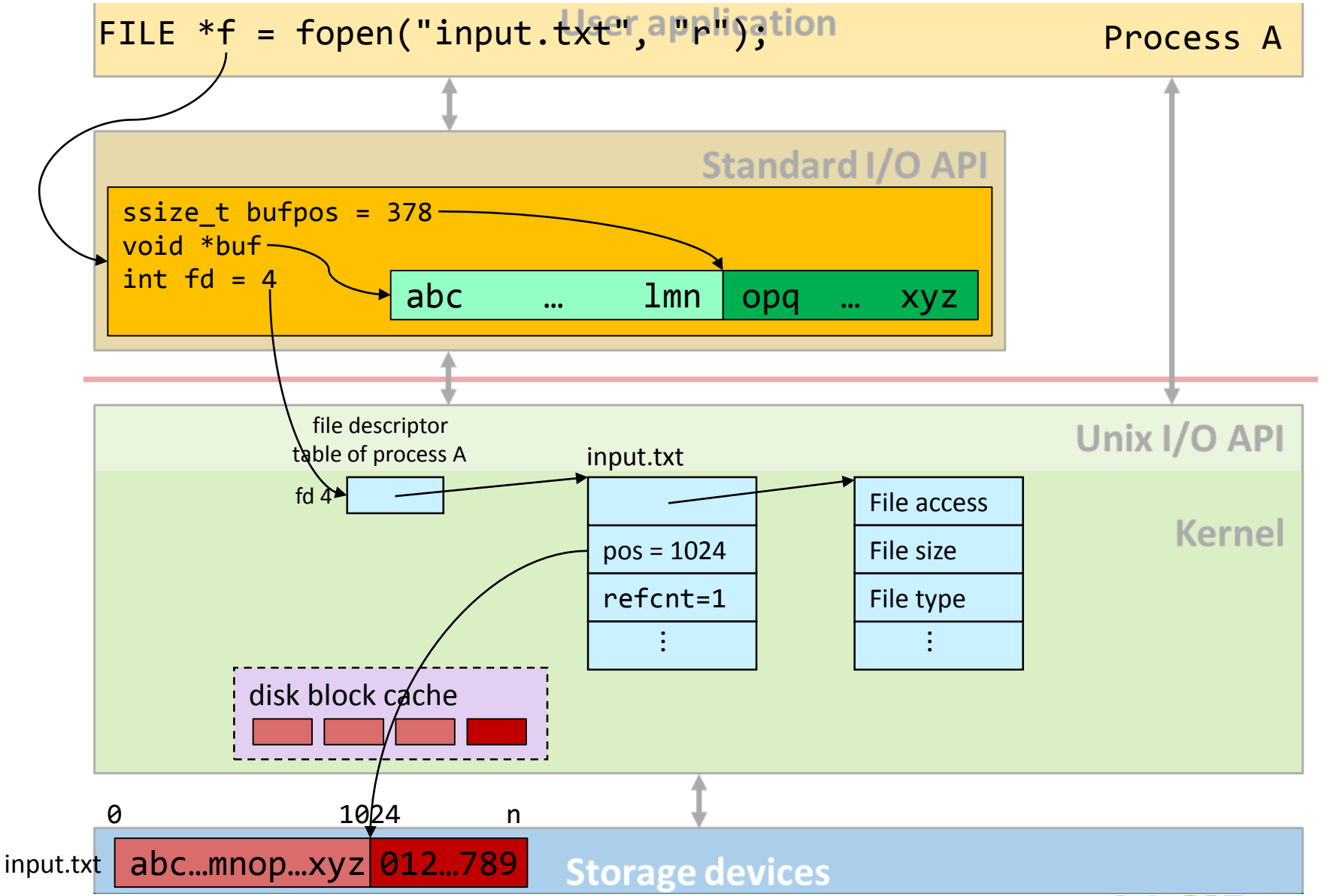


Interaction of Standard I/O with Unix I/O

Standard I/O and Unix I/O



Standard I/O and Unix I/O



Standard I/O and Unix I/O

- **Pseudo code**

```
FILE* fopen(char *path, char* mode) {
  int fd = open(path, …);
  if (fd == -1) return NULL;

  FILE *stream = malloc(sizeof(FILE));
  stream->fd     = fd;
  stream->buffer = malloc(bufsize);
  stream->bufpos = 0;

  refill_buffer(stream);

  return stream;
}
```

Standard I/O and Unix I/O

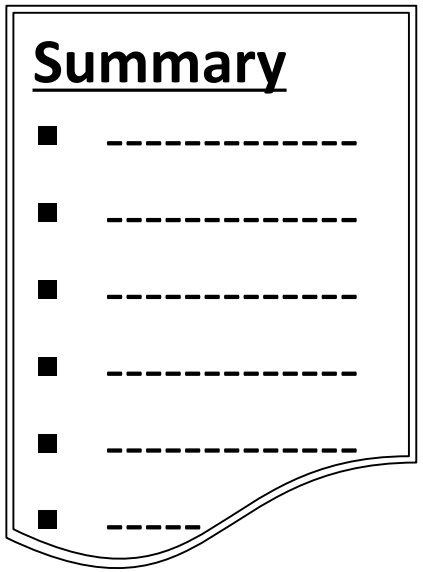
- **Pseudo code**

```
void refill_buffer(FILE *stream) {
  read(stream->fd, stream->buffer, bufsize);
  stream->bufpos = 0;
}


int fclose(FILE *stream) {
  close(stream->fd);
  free(stream->buffer);
  return 0;
}


int fileno(FILE *stream) {
  return stream->fd;
}
```

Standard I/O and Unix I/O

- **Pseudo code**

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) {
  size_t total_bytes = size*nmemb;
  size_t read_bytes   = 0;

  while (read_bytes < total_bytes) {
    int copy_bytes = min(total_bytes-read_bytes,
                         bufsize-stream->bufpos);

    memcpy(ptr, stream->buf+stream->bufpos, copy_bytes);
    read_bytes += copy_bytes;
    stream->bufpos += copy_bytes;

    if (stream->bufpos == bufsize) refill_buffer(stream);
  }

  return read_bytes;
}
```



Module Summary

Summary: Unix I/O

- Pros
  - Unix I/O is the most general and lowest overhead form of I/O.
    - All other I/O packages are implemented using Unix I/O functions.
  - Unix I/O provides functions for accessing file metadata.
  - Unix I/O functions are async-signal-safe and can be used safely in signal handlers.

- Cons
  - Dealing with short counts is tricky and error prone.
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone.
  - Both of these issues are addressed by the standard I/O packages.

Summary: Standard I/O

- Pros
  - Buffering increases efficiency by decreasing the number of read and write system calls
  - Short counts are handled automatically
- Cons
  - Provides no function for accessing file metadata
  - Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers.
  - Standard I/O is not appropriate for input and output on network sockets
    - There are poorly documented restrictions on streams that interact badly with restrictions on sockets (CS:APP3e, Sec 10.9)

Choosing I/O Functions

- General rule: use the highest-level I/O functions you can
  - Many C programmers are able to do all of their work using the standard I/O functions
- When to use standard I/O
  - When working with disk or terminal files
- When to use raw Unix I/O
  - Inside signal handlers, because Unix I/O is async-signal-safe.
  - In rare cases when you need absolute highest performance.