系統程式設計
Systems Programming

鄭卜壬教授
臺灣大學資訊工程系

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Contents

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

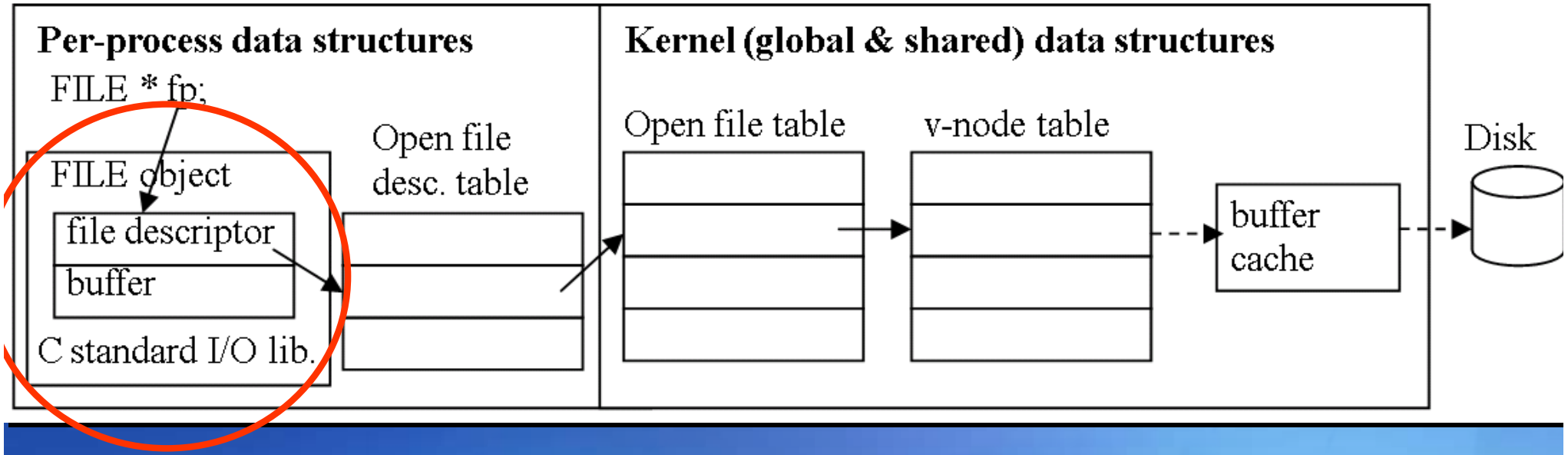
Standard I/O Library

- **Major revision by Dennis Ritchie in 1975**
- **An ANSI C standard**
  - Easy to use and portable
  - Details handled:
    - Buffer allocation, optimal-sized I/O chunks, better interface, etc.

Standard I/O Library

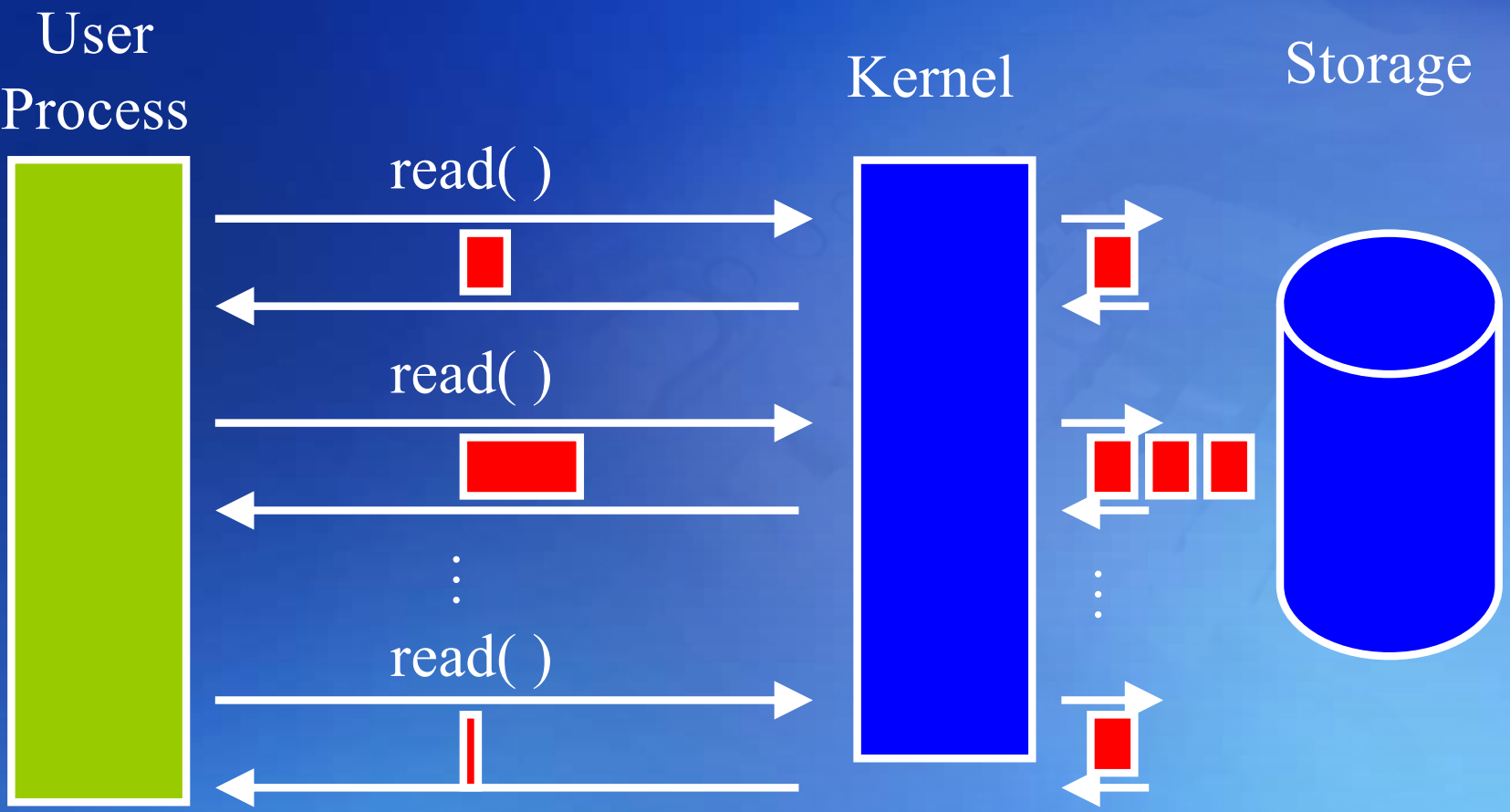
- **Difference from File I/O (unbuffered I/O)**
  - File Pointers vs File Descriptors
  - fopen vs open
    - When a file is opened/created, a *stream* is associated with the file.
    - FILE object
      - File descriptor, pointer to buffer, buffer size,
        # of remaining chars, an error flag,
        an end-of-file flag, and the like.
  - stdin, sdtout, stderr defined in <stdio.h>
    - STDIO_FILENO, STDOUT_FILENO, STDERR_FILENO

Buffering



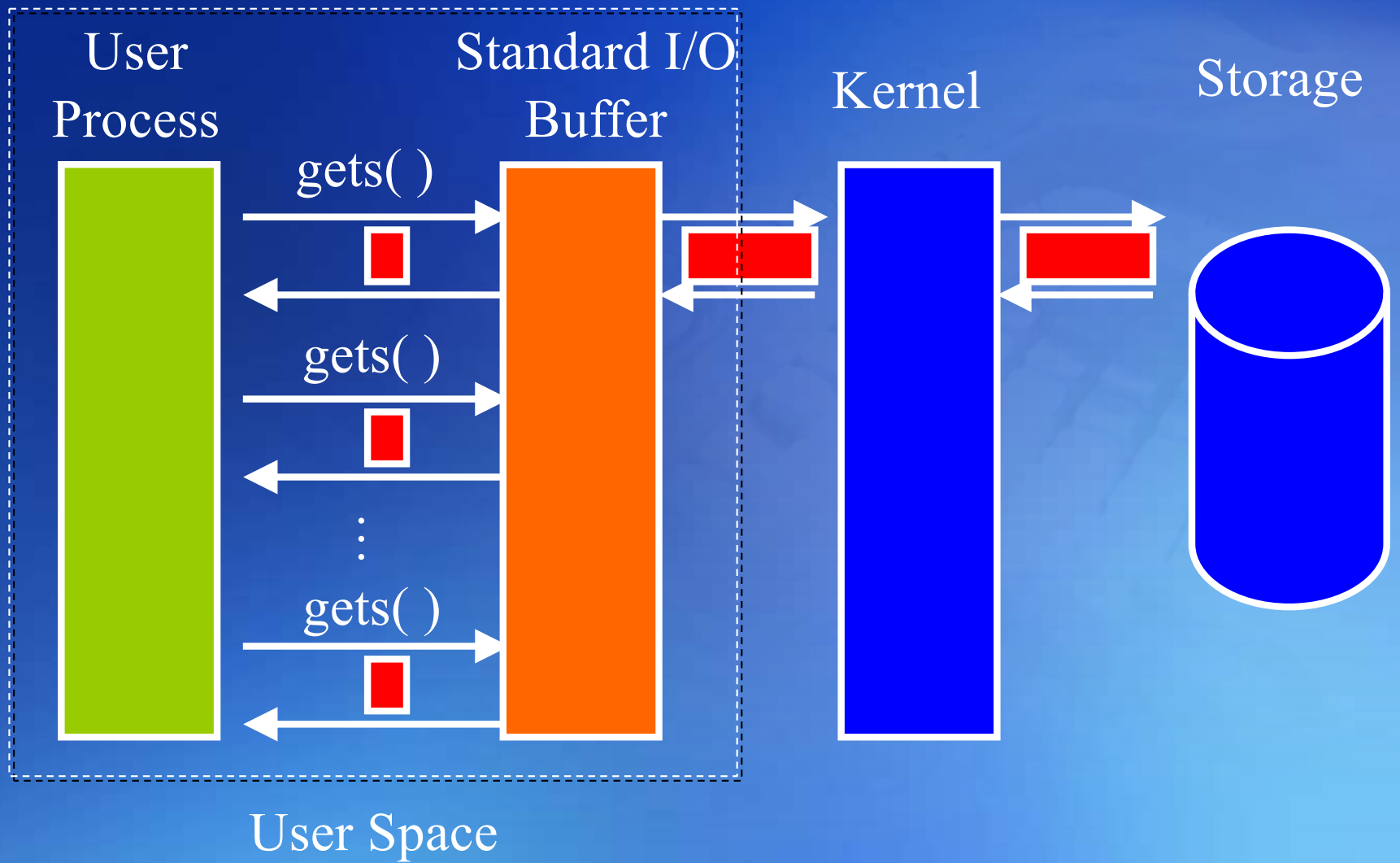
- A wrapper of unbuffered I/O
- Linux shell command: strace

Unbuffered I/O

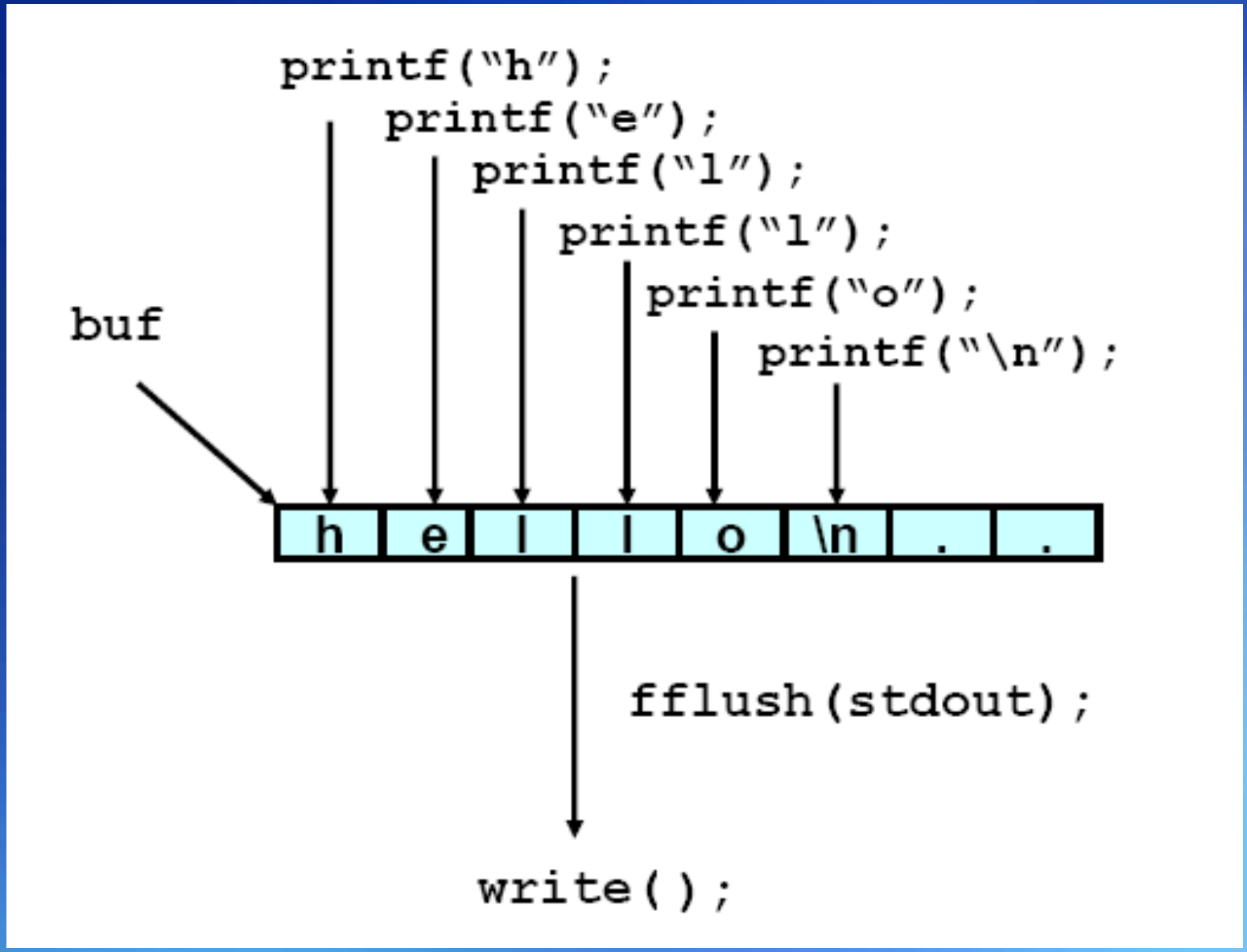


Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Buffered I/O



Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University


printf("h");
printf("e");
printf("l");
printf("l");
printf("o");
printf("\n");
buf
h
e
l
l
o
\n
.
.
fflush(stdout);
write();

Standard I/O Library - Open

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char *type);
```

- text file vs. binary file: b (in rb, wb, ab, r+b, …) stands for binary file – no effect for Unix kernel
- Append mode supports multiple access (atomic operation)

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			•			•

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Example of freopen()

FILE *freopen(const char *pathname,
const char *type, FILE *fp);

- close fp stream first and clear a stream's orientation
- typically used to open a specified file as one of the predefined streams: standard input/output/error.

The example logs all standard output to the /tmp/logfile file.

```
FILE *fp;
fp = freopen ("/tmp/logfile", "a+", stdout);
printf("Sent to stdout and redirected to /tmp/logfile");
fclose(stdout);
```

fdopen & fclose

FILE *fdopen(int fildes, const char *type);

- Associate (standard) I/O stream with an existing file descriptor – POSIX.1
  - Pipes, network channels
  - No truncating for the file for “w”

int fileno(FILE *fp);

- Get filedes for fcntl, dup, etc.

int fclose(FILE *fp);

- Flush buffered output
- Discard buffered input
- All I/O streams are closed after the process exits.
- The relocated buffers must be valid before the stream is closed.

Potential problem when using fclose

- **Never be referenced after their stack frames (see Ch.7) are released.**

```
#define DATAFILE    "datafile"

FILE *
open_data(void)
{
    FILE    *fp;
    char    databuf[BUFSIZ];   /* setvbuf makes this the stdio buffer */

    if ((fp = fopen(DATAFILE, "r")) == NULL)
        return(NULL);
    if (setvbuf(fp, databuf, _IOLBF, BUFSIZ) != 0)
        return(NULL);
    return(fp);     /* error */
}
```

Explain later

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2006
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Buffering

- **Goal**
  - Use the minimum number of read and write calls.
- **Allocation**
  - Automatically allocated when the first-time I/O (malloc) is performed on a stream.
  - Call setbuf() or setvbuf()
- **Types**
  - Fully Buffered
    - Actual I/O occurs when the buffer is filled up.
    - Disk files, pipes, and sockets are normally fully buffered.

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

- Line Buffered
  - Perform I/O when a newline char is encountered! – usually for terminals (e.g., stdin, stdout).
  - Caveats
    - The filling of a fixed buffer could trigger I/O.
    - The flushing of all line-buffered outputs if input is requested (from the kernel).

    Example:

```
char buf[100];
printf("$ ");            // "prompt" in shell
scanf("%s", buf);   // trigger the output of $
```

- Unbuffered
  - Expect to output ASAP, e.g. using write()
  - E.g., stderr

- **ANSI C Requirements**
  - Fully buffered for stdin and stdout unless interactive devices are referred to.
    - SVR4/4.3+BSD – line buffered
  - Standard error is never fully buffered.
    - SVR4/4.3+BSD – unbuffered

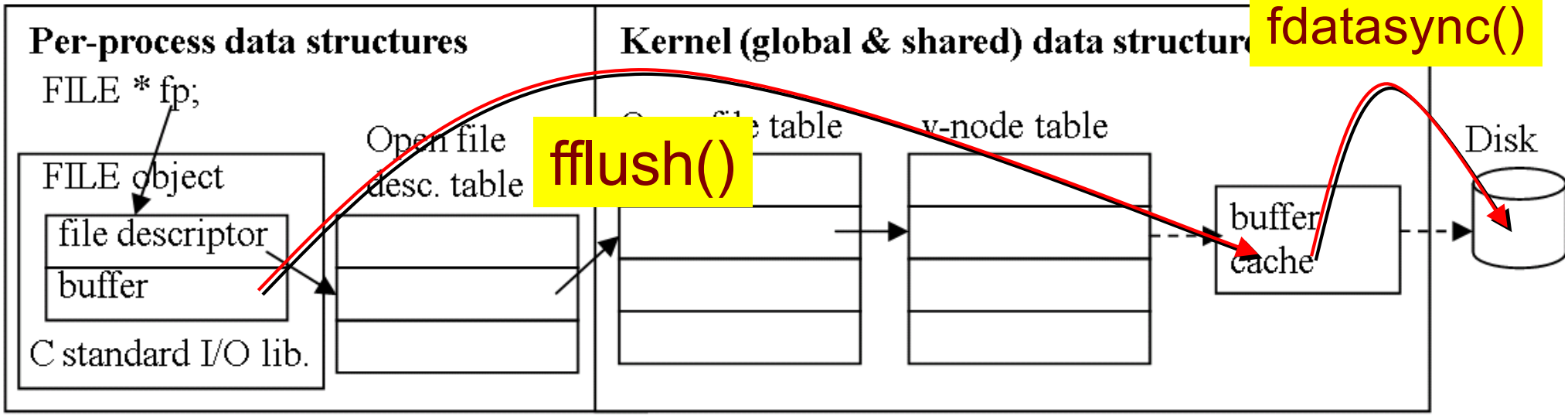
Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Way to Synchronize Data

int fflush(FILE *fp);

- Any unwritten data in the stream are passed to kernel
- All output streams are flushed if fp == NULL
- Call fsync() after each call to fflush() if necessary. Calling fsync() without calling fflush() might do nothing if all the data are still in the C library buffer.

sync()
fsync()
fdatasync()



Build up your own buffer

Must be called before any op is performed on streams!

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

- Full/line buffering if buf is not NULL (BUFSIZ)
  - Ex: line buffer for terminals
- Turn buffering off if buf is NULL
- #define BUFSIZ 1024 (<stdio.h>)

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

- Specify the type of buffering
- mode: _IOFBF, _IOLBF, _IONBF (<stdio.h>)
  - Optional size → st_blksize (stat())

- **Possible Memory Access Errors**
  - Note buffer's life cycle (close the stream before terminating a process/function if automatic variable is used)
- **Part of the buffer for internal bookkeeping**

	Mode	*Buf*	Buffer and Length	Type of Buffering
setbuf		Non-null	User *buf* of length BUFSIZE	Fully buffered or line buffered
		NULL	(no buffer)	Unbuffered
setvbuf	_IOFBF	Non-null	User *buf* of length *size*	Fully buffered
		NULL	System buffer of appropriate length	
	_IOLBF	Non-null	User *buf* of length *size*	Line buffered
		NULL	System buffer of appropriate length	
	_IONBF	(ignored)	(no buffer)	Unbuffered

Positioning-a-Stream

#include <stdio.h>

long ftell(FILE *fp);

- Current file offset in bytes

int fseek(FILE *fp, long offset, int whence);

void rewind(FILE *fp);

- Assumption: a file's position can be stored in a long
- Another version: fello, fseeko (off_t)

  fgetpos, fsetpos (fpos_t) (ANSI C Standard)
- whence: same as lseek
  - Binary files: No requirements for SEEK_END under ANSI C (good under Unix, possible padding for other systems).
  - Text files: SEEK_SET only – 0 or returned value by ftell

Standard I/O Library – Reading/Writing

- **Unformatted I/O**
  - Character-at-a-time I/O, e.g., getc
    - Buffering handled by standard I/O lib
  - Line-at-a-time I/O, e.g., fgets
    - Buffer limit might need to be specified.
  - Direct I/O, e.g., fread
    - Read/write a number of objects of a specified size.
    - An ANSI C term, e.g., = object-at-a-time I/O, binary I/O, record/structure-oriented I/O, etc.

Character-at-a-Time I/O

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

- getchar == getc(stdin)
- Differences between getc and fgetc
  - getc could be a macro
  - Argument's side effect (e.g., getc(f*++) will fail, exec time, passing of the function address.
- unsigned char converted to int in returning to return error number.
- Error value: -1 for EOF and error. (How do you know exactly what happens?)

- **The following code works correctly on some machines, but not on others. What could be the problem?**

```
int main()
{
    char    c;
    while ((c = getchar()) != EOF)
            putchar(c);
}
```

A system could choose “unsigned” or “signed” for “char” type.

• unsigned char → the character's value no longer equals -1, so the loop never terminates
• signed char → the character’s value equals -1 when reading char 255, so the loop terminates before EOF

Standard I/O Library – Reading/Writing

```
#include <stdio.h>
int ferror(FILE *fp);
int feof(FILE *fp);
void clearerr(FILE *fp);
```

- Clear both of error and EOF flags

```
int ungetc(int c, FILE *fp);
```

- No pushing back of EOF (i.e., -1)
- No need to be the same char read!
- Clear EOF flag
- The character is stored in IO buffer, instead of file.

Using ungetc to remove white space at the start of a line.

```
While (ftell(my_stream) != EOF){
do
  in_char = getc (my_stream);
while (isspace (in_char));

/* Back up to first non-whitespace character */
ungetc (in_char, my_stream);

getline (&my_string, &nchars, my_stream);
}
```

Character-at-a-Time I/O

```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

- putchar(c) == putc(c, stdout)
- Differences between putc and fputc
  - putc() can be a macro.

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Line-at-a-Time I/O

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

```
char *fgets(char *buf, int n, FILE *fp);
```

- Include '\n' and be terminated by *null*
- Could return a partial line with (n-1) bytes if the line is too long.

```
char *gets(char *buf);
```

- Read from stdin.
- No buffer size is specified → overflow (unsafe)
- *buf does not include '\n' and is terminated by *null.*

Example of Unsafe gets()

```
#include <stdio.h>

typedef struct MyStruct
{
  char buf[5];
  int  i;
} MyStruct_t;

int main(void)
{
  MyStruct_t my;

  my.i = 10;

  printf ("my.i is %d\n", my.i);
  printf ("Enter a 10 digit number:");  /
* Too big on purpose  */
```

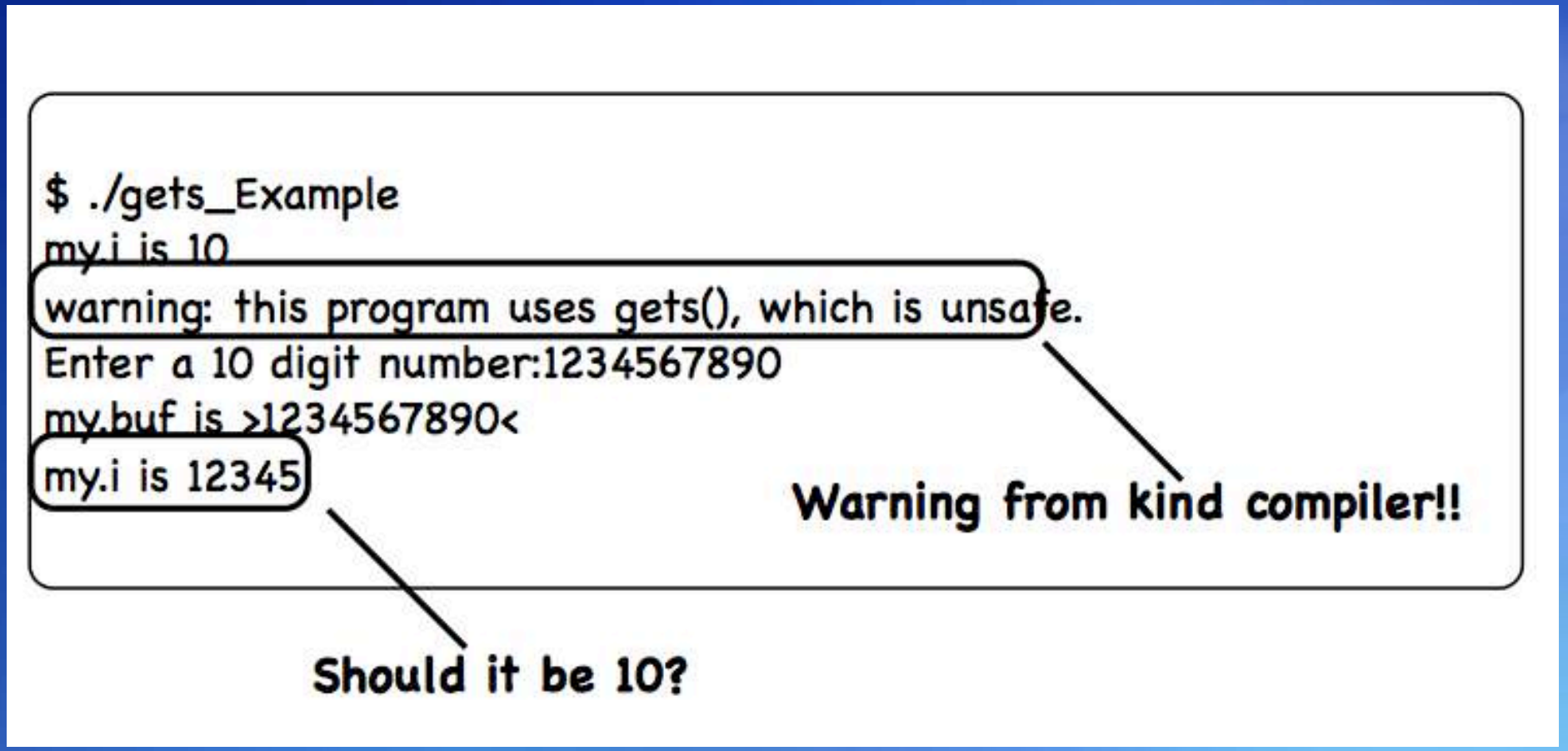
(cont.)

```
  gets(my.buf);

  printf ("my.buf is >%s<\n",
my.buf);
  printf ("my.i is %d\n", my.i);

  return(0);
}
```

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University


$ ./gets_Example
my.i is 10
warning: this program uses gets(), which is unsafe.
Enter a 10 digit number:1234567890
my.buf is >1234567890<
my.i is 12345
Warning from kind compiler!!
Should it be 10?

Line-at-a-Time I/O

```
#include <stdio.h>
char *fputs(const char *str, FILE *fp);
```

- Include '\n' and be terminated by *null*.
- Newline is not required for line-at-a-time output.

```
char *puts(const char *str);
```

- *str does not include '\n' and is terminated by *null*.
- puts() then writes '\n' to stdout.

Binary I/O

- **Objectives**
  - Read/write a structure at a time, which could contains null or '\n'.

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
```

  - Reads less than the specified number of objects → error or EOF → ferror, feof
  - Write error if less than the specified number of objects are written.

Binary I/O

- **Example 1**

```
float data[10];
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("fwrite error");
```

- **Example 2**

```
struct {
    short count;
    long total;
    char name[NAMESIZE];
} item;
if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("fwrite error");
```

Binary I/O

- **NOT PORTABLE for programs using fread and fwrite**
  1. The offset of a member in a structure can differ between compilers and systems (due to alignment).
  2. The binary formats for various data types, such as integers, could be different over different machines.

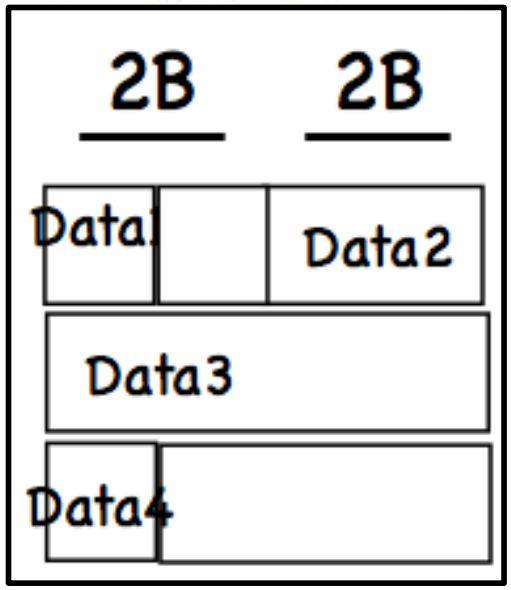
Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Memory Alignment

8B

```
struct MixedData
{
    char Data1;
    short Data2;
    int Data3;
    char Data4;
};
```

```
struct MixedData
{
    char Data1;          /* 1 byte */
    char Padding1[1];    /* 1 byte */
    short Data2;         /* 2 bytes */
    int Data3;           /* 4 bytes */
    char Data4;          /* 1 byte */
    char Padding2[3];    /* 3 bytes */
};
```



Memory Alignment

- Beneficial to allocate memory aligned to cache lines
- The type of each member of the structure usually has a default alignment, meaning that it will be aligned on a pre- determined boundary unless otherwise requested by the programmer.

A char (one byte) will be 1-byte aligned.

A short (two bytes) will be 2-byte aligned.

An int (four bytes) will be 4-byte aligned.

A float (four bytes) will be 4-byte aligned.

A double (eight bytes) will be 8-byte aligned on Windows and 4-byte aligned on Linux (8-byte with -malign-double compile time option).

A long double (twelve bytes) will be 4-byte aligned on Linux.

Any pointer (four bytes) will be 4-byte aligned on Linux. (e.g.: char*, int*)

Formatted I/O

Input Functions:

```
int scanf(const char *format, …);
int fscanf(FILE *fp, const char *format, …);
int sscanf(char *buf, const char *format, …);
```

Output Functions:

```
int printf(const char *format, …);
int fprintf(FILE *fp, const char *format, …);
int sprintf(char *buf, const char *format, …);
```

- Overflow is possible for sprintf() – '\0' appended at the end of the string. A better substitute: snprintf()

```
int vprintf(const char *format, va_list arg);
int vfprintf(FILE *fp, const char *format, va_list arg);
int vsprintf(char *buf, const char *format, va_list arg);
```

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Problem: No Range Checking

- **strcpy does not check input size**
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until “\0” is encountered, ignoring the size of area allocated to buf
- **Many C library functions are unsafe**
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - sprintf(char *buf, const char *format, …);

Interleaved R&W restrictions

- **Output [fflush | fseek | fsetpos | rewind] Input**
  Intput [fseek | fsetpos | rewind | EOF] Output

- Note that standard IO functions are implemented by UNIX IO system calls.
  - fread() calls read() to fill local buffer when needs.
  - fwrite() calls write() to flush local buffer when needs.
- When there is no need to trigger system calls, standard IO read/write data from buffer, directed by R and W pointers, respectively.
- The inconsistent on R/W pointers leads to errors

Errors may occur in some systems

```
// In this example, we try to check the buffering
// mechanism for Standard IO library.
// One character is read and another character
// is written to the same file.
// No file position functions such as fseek is called in between.

int main( void )
{
    FILE* fp = fopen( "./test.txt", "r+" );
    // test.txt: 12345


    char c;
    fread( &c, 1, 1, fp );
    // fseek(fp,0,SEEK_CUR);

    fwrite( &c, 1, 1, fp );

    fclose( fp );
    return 0;
}
```

The Results

In Linux:
Without fseek, 11345

In Linux:

With fseek, 11345

In OSX:
Without fseek, 12345

In OSX:

With fseek, 11345

In Windows:
Without fseek, 12345 \n 1

In Windows:

With fseek, 11345

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Workaround on the Restrictions

Flush stream after every write!!

Or,

Open two streams on the same descriptor, one for reading and one for writing:

```
FILE *fpin, *fpout;

fpin = fdopen(fd, "r");
fpout = fdopen(fd, "w");
```

However, you need to position the pointers separately and close two file streams at the end.

```
fclose(fpin);
fclose(fpout);
```

Standard I/O Efficiency

98.5 MB files with 3 million lines

System times are comparable

Function	User CPU (seconds)	System CPU (seconds)	Clock Time (seconds)	Bytes of program text
Best time using read/write	0.01	0.18	6.67	
fgets, fputs	2.59	0.19	7.15	139
getc, putc	10.84	0.27	12.07	120
fgetc, fputc	10.44	0.27	11.42	120
Single byte using read/write	124.89	161.65	288.64	

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Copy stdin to stdout using getc and putc

```
#include "apue.h"

int
main(void)
{
        int     c;

        while ((c = getc(stdin)) != EOF)
            if (putc(c, stdout) == EOF)
                err_sys("output error");

        if (ferror(stdin))
            err_sys("input error");

        exit(0);
}
```

Copy stdin to stdout using fgets and fputs

```
#include "apue.h"

int
main(void)
{
    char    buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Multibyte Files

```
#include <wchar.h>
int fwide(FILE *fp, int mode)
```

- Standard I/O file streams can be used with single byte and multiple byte character sets.
- By default, there is no orientation
- After the stream becomes byte-oriented or wide-oriented, the orientation of a stream will be fixed and can not be changed until the stream is closed.
- Mode:
  - Negative: set byte-oriented
  - Positive: set wide-oriented
  - 0: No change on orientation.
- Return values:
  - Positive: wide oriented
  - Negative: byte oriented
  - 0: no orientation.
- Related functions: fwprintf, fwscanf, fgetwc, fputwc, wmemcpy

What You Should Know

- **What is buffered I/O**
  - Goal and types (fully/lined buffered & unbuffered)
- **Kernel data structure for buffered I/O**
- **Flushing data**
  - fflush() vs. fsync()
- **I/O efficiency**
- **Unformatted I/O vs. Formatted I/O**