SYSTEM PROGRAMMING

Module 1

EGCI 252
SYSTEM PROGRAMMING

Computer Engineering Department
Faculty of Engineering
Mahidol University

System Calls and Device Drivers

Files and devices can be controlled and accessed using a small number of functions known as system calls.

System calls are the interface to the core of the operating system called **kernel**. **In the kernel**, **there are** a number of low-level interfaces known as **device drivers for controlling system hardware**.

Low-Level System Calls

open – Open a file or device.

read – Read from an open file or device.

write – Write to a file or device.

close – Close the file or device.

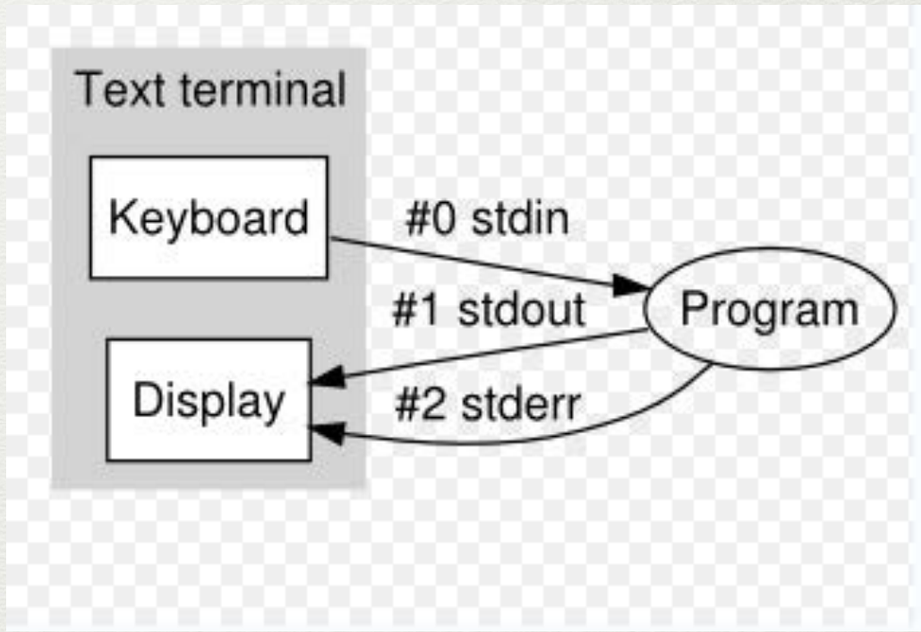
ioctl* – Specific control the device.

*The ioctl function is used to provide some necessary hardware-specific control. Its use varies from device to device.

Low-Level File Access

When a program starts, there are three file descriptors* automatically opened:

0	standard input
1	standard output
2	standard error



*A file descriptor is an abstract key or a reference for accessing a file.

Open

The “open” system call is used to create a new file descriptor.

The “creat” system call is another option supported by LINUX;
however its use is almost obsolete.

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

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

Open (cont.)

The oflags are specified as a bitwise OR of a mandatory file access mode and other optional modes.

Mandatory access modes:

O_RDONLY Open for read-only

O_WRONLY Open for write-only

O_RDWR Open for reading and writing

Open (cont.)

Optional access modes:

O_APPEND Place written data at the end of the file.

O_TRUNC Set the length of the file to zero, discarding existing contents.

O_CREAT Create the file, if necessary; with permissions given in mode.

O_EXCL Used with O_CREAT, ensures that the caller creates the file. The open is atomic, i.e. it's performed with just one function call. This protects against two programs creating the file at the same time. If the file already exists, open will fail.

Open (cont.)

open returns the new file descriptor as a non-negative integer if successful, or -1 if fails.

Initial permissions for a new file can be specified in the third parameter of the open function

(e.g., open(“file”, O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);)

The permission flags are defined in the sys/stat.h header file.

S_IRUSR	Read permission for owner	S_IRGRP	Read permission for group
S_IWUSR	Write permission for owner	S_IWGRP	Write permission for group
S_IXUSR	Execute permission for owner	S_IXGRP	Execute permission for group
S_IROTH	Read permission for others		
S_IWOTH	Write permission for others		
S_IXOTH	Execute permission for others		

Write

The write system call writes data from the memory to a file.

```
#include <unistd.h>
size_t write(int fd, const void *buf, size_t count);
```

- the number of bytes to be written specified by count
- buf is the pointer to the location of data
- fd is the file descriptor

On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately. If count is zero and the file descriptor refers to a regular file, 0 will be returned without causing any other effect. For a special file, the results are not portable.

Write (cont.)

Example:

```
#include <unistd.h>
int main(void)
{
    if ((write(1, “Some data is written\n”,  21)) != 21)
        write(2, “An error has occurred on the stdout\n”, 36);
    return 0;
}
```

Read

The read system call reads data from file to the memory.

```
#include <unistd.h>

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

- the number of bytes to be read specified by count
- buf is the pointer to the location of data
- fd is the file descriptor

Read (cont.)

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. On error, -1 is returned, and errno is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

Read (cont.)

Example:

```
#include <unistd.h>
int main(void)
{
    int nread; char buffer[120];
    nread = read(0, buffer, 120);
    if (nread == -1)  write(2, “An error has occurred\n”, 21);
     else write(1, buffer, nread);
    return 0;
}
```

Close

The close system call is used to terminate the association between a file descriptor and its file. It return 0 if successful and -1 on error

```
#include <unistd.h>

int close(int fd);
```

Example: Low-Level File Access

```
$ gcc -o char_copy char_copy.c
$ time ./char_copy
```

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

int main(void)
{
    char c; int in, out;
    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT,  S_IRUSR|S_IWUSR);
    while (read(in, &c, 1) == 1)  write(out, &c, 1);
    return 0;
}
```

Example: Low-Level File Access (Cont.)

```
$ gcc -o block_copy block_copy.c
$ time ./block_copy
```

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

int main(void)
{
    char block[512]; int in, out, nread;
    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT,  S_IRUSR|S_IWUSR);
    while ((nread = read(in, block, sizeof(block))) > 0)  write(out, block, nread);
    return 0;
}
```

Lseek

lseek returns the length in bytes from the beginning of the file

lseek : The lseek can set the read/write pointer of a file descriptor. It allow a user to set the location in the file for the next reading or writing.

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

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

offset specifies the length in bytes from the reference position

whence specifies the reference position which can be one of the following:

SEEK_SET: the beginning of the file
SEEK_CUR: the current position
SEEK_END: the end of the file

File with Hole Using Lseek

Write a simple program that creates a file with 16-null character long (or 16-character hole) in the middle of the file.

For example:

abcdefghij\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\01234567890

I/O Efficiency

Buffer Size	Time (seconds)			No.of loops
	User	System	Real	
1	23.8	397.9	423.4	1468802
2	12.3	202.0	215.2	734401
4	6.1	100.6	107.2	367201
8	3.0	50.7	54.0	183601
16	1.5	25.3	27.0	91801
32	0.7	12.8	13.7	45901
64	0.3	6.6	7.0	22950
128	0.2	3.3	3.6	11475
256	0.1	1.8	1.9	5738
512	0.0	1.0	1.1	2869
1024	0.0	0.6	0.6	1435
2048	0.0	0.4	0.4	718
4096	0.0	0.4	0.4	359
8192	0.0	0.3	0.3	180
16384	0.0	0.3	0.3	90
32768	0.0	0.3	0.3	45

Function Calls VS System Calls

A library function is an ordinary function that resides in a library external to your program. Most of the library functions we've presented so far are in the standard C library, libc. For example, scanf and printf are functions provided in the C library. A call to a library function is just like any other function call.The arguments are placed in processor registers or onto the stack, and execution is transferred to the start of the function's code, which typically resides in a loaded shared library.

Function Calls VS System Calls (cont.)

A system call is implemented in the Linux kernel.When a program makes a system call, the arguments are packaged up and handed to the kernel, which takes over execution of the program until the call completes. A system call isn't an ordinary function call, and a special procedure is required to transfer control to the kernel. However, the GNU C library (the implementation of the standard C library provided with GNU/Linux systems) wraps Linux system calls with functions so that you can call them easily. Low-level I/O functions such as open and read are examples of system calls on Linux.

Standard I/O Library

The standard I/O library provides a versatile interface to low-level I/O system calls.

It gives more sophisticated functions for formatting output, scanning input.

It uses buffer to minimize number of read and write system calls. Each I/O stream is automatically buffered by the standard I/O library.

Standard I/O Library (cont.)

When a program starts, there are three streams* automatically opened with file pointers** pointed to each stream:

stdin	->	standard input
stdout	->	standard output
stderr	->	standard error

*A stream is a representation of an associated file.

**A file pointer is an abstract key or a reference for accessing a file, which is equivalent to the low-level file descriptor.

Buffering

There are three types of buffering

- Fully buffered – Actual I/O take place when the standard I/O buffer is filled.
- Line buffered – Actual I/O is performed when a new line character is encountered on input or output.
- Unbuffered – I/O is performed as quickly as possible without buffering.

ANSI C requires the following buffering characteristics:

- Standard input and output are fully buffered, if and only if they do not refer to an interactive device.
- Standard error is never fully buffered.

Changing buffering type

Buffering type can be changed by calling the following functions:

```
#include <stdio.h>

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

Returns: 0 if OK, nonzero on error. These functions must be called after the stream has been opened.

Changing buffering type (cont.)

Function	**Mode**	**Buf**	**Buffer & Length**	**Type of buffering**
setbuf		Non-Null	User buf of length BUFSIZ (defined in stdio.h)	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

fopen

fopen opens a specified file

#include <stdio.h>

FILE *fopen(const char *pathname, const char *type);

Type	Description
r or rb	Open for reading
w or wb	Truncate to 0 length or create for writing
a or ab	Append; open for writing at the end of file, or create for writing
r+ or r+b or rb+	Open for reading and writing
w+ or w+b or wb+	Truncate to 0 length or create for reading and writing
a+ or a+b or ab+	Open or create for reading and writing at the end of file

fopen (cont.)

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 the end			•			•

fread

The fread library function is used to read data from a stream.
Data is read into a data buffer given by ptr.

#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

Returns: number of items successfully read into data buffer if OK,
If an error occurs, or the end-of-file is reached, the return value is a short item count (or zero).

fwrite

The fwrite library function is used to write data to a stream.
Data from the data buffer is written to the output stream.

#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t nmemb,
FILE *stream);

Returns: number of items successfully written to the output stream
if OK, If an error occurs, or the end-of-file is reached,
the return value is a short item count (or zero).

fclose

An open stream is closed by calling fclose. Any buffered output data is flushed before the file is closed. Any input data that may be buffered is discarded. If the standard I/O library had automatically allocated a buffer for the stream, the buffer is released.

```
#include <stdio.h>
int fclose(FILE *fp);
```

Returns: 0 if OK, EOF on error

fseek

The fseek function sets the file position indicator for the stream pointed to by stream. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence. If whence is set to SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the fseek function clears the end-of-file indicator for the stream and undoes any effects of the ungetc(3) function on the same stream.

fseek (cont.)

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
```

The rewind function returns no value. Upon successful completion, fseek, return 0. Otherwise, -1 is returned and the global variable errno is set to indicate the error.

fflush

The function fflush forces a write of all user-space buffered data for the given output or update stream via the stream's underlying write function. The open status of the stream is unaffected. If the stream argument is NULL, fflush flushes all open output streams.

```
#include <stdio.h>

int fflush(FILE *stream);
```

Upon successful completion 0 is returned. Otherwise, EOF is returned and the global variable errno is set to indicate the error.

fgetc, getc, getchar, ungetc

fgetc() reads the next character from stream and returns it as an unsigned char cast to an int, or EOF on end of file or error.

getc() is equivalent to fgetc() except that it may be implemented as a macro which evaluates stream more than once.

getchar() is equivalent to getc(stdin).

ungetc() pushes c back to stream, cast to unsigned char, where it is available for subsequent read operations. Pushed - back characters will be returned in reverse order; only one pushback is guaranteed.

fgetc, getc, getchar, ungetc (cont.)

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
```

fgetc(), getc() and getchar() return the character read as an unsigned char cast to an int or EOF on end of file or error.ungetc() returns c on success, or EOF on error.

fputc, putc, putchar

fputc() writes the character c, cast to an unsigned char, to stream.
putc() is equivalent to fputc() except that it may be implemented as a macro which evaluates stream more than once.
putchar(c); is equivalent to putc(c,stdout).

fputc, putc, putchar (cont.)

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

fputc(), putc() and putchar() return the character written as an unsigned char cast to an int or EOF on error.

fgets, gets

```
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
```

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with ‘\0’. No check for buffer overrun is performed.

fgets, gets (cont.)

fgets() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A '\0' is stored after the last character in the buffer.

gets() and fgets() return s on success, and NULL on error or when end of file occurs while no characters have been read.

fputs, puts

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

fputs() writes the string s to stream, without its trailing '\0'.

puts() writes the string s and a trailing newline to stdout.

puts() and fputs() return a non-negative number on success, or EOF on error.

File Stream Copy Program

Write a program similar to earlier versions (char_copy) using functions referenced in stdio.h (getc, putc)

Write a program similar to earlier versions (block_copy) using functions referenced in stdio.h (fgets, fputs)

Timing results using standard I/O

Function	User CPU (s)	System CPU (s)	Clock time (s)	Bytes of program text
Best time System call	0.0	0.3	0.3	
fgets, fputs	2.2	0.3	2.6	184
getc, putc	4.3	0.3	4.8	384
fgetc, fputc	4.6	0.3	5.0	152
Worst time System call	23.8	397.9	423.4	

End of Module