Real-Time Programming

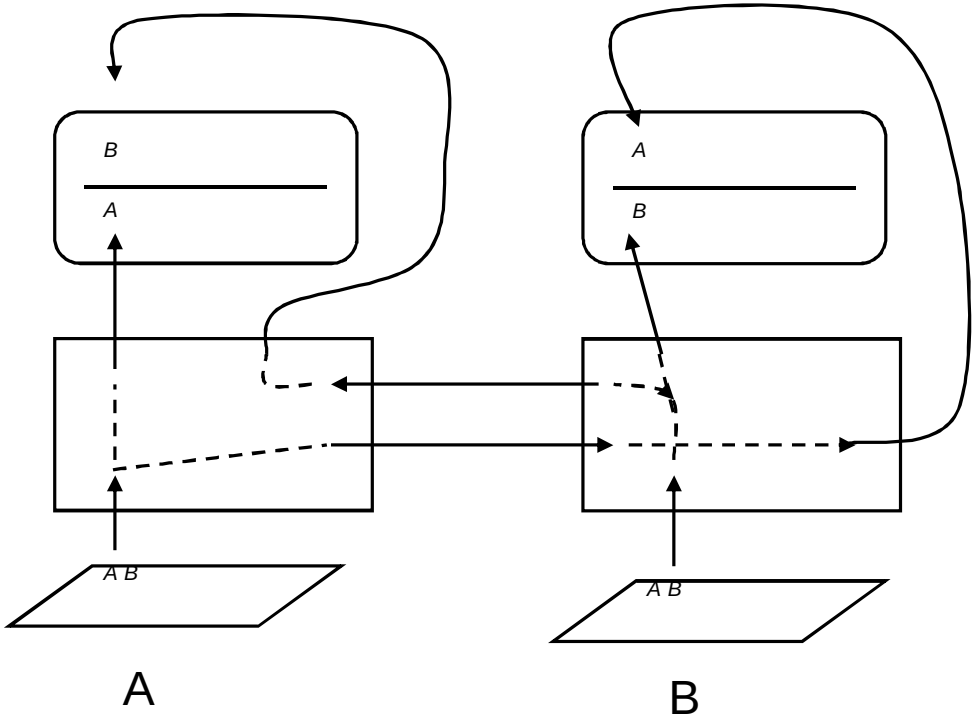
TI00AA55

Lecture 2 - 21.01.2015

Jarkko.Vuori@metropolia.fi

Simple “real-time” application

- We have now learned the concept of real-time system
- As an example application we use a simple chat application between two people
  - The chat application displays the text entered by user A at computer A and the text entered by the fellow user B at computer B
  - The computers are connected via any kind of connection (like Bluetooth, Ethernet or serial)
- The chat system is real-time system if it is capable to display ”fast enough” (for example in 0.1 second) the input entered from the keyboard of computer A as well as the input read from the communication channel (from the keyboard of the computer B).
- The figure on the right illustrates our system



First trial to implement real-time software

- Now we move to study how the software in the real-time system could be implemented
- We could first try to implement the application using the following program
- This program does not work in real time, because the input instructions block the application if data is not ready to be read
- In practice this means for example that
  - If the user A does not enter any text, he cannot see the text possibly entered by the user B either
  - Or if user B does not enter any text the user A cannot see even his own text (if the terminal driver does not echo it) and the computer A cannot send text to the computer B

```
int main (void) {
  FILE *serial;
  char chr;
  serial = fopen("//dev//ttyS0", "rw");
  while(1) {
    chr = getchar();//blocks if data is not rdy
    fwrite(&chr, 1, 1, serial);
    fread(&chr, 1, 1, serial); // blocks
    putchar(chr);
  }
}
```

Polling method

- We could fix the problem and make a working program in the following way (pseudo language is used here)
- This demonstrates the so called polling method

```
int main (void) {
  FILE* serial;
  char chr;
  serial = fopen("//dev//ttyS0", "rw");
    while(1) {
      succeed=try_to_read_kb(&chr);
      if (succeed) {
        putchar(chr);
        fwrite(&chr, 1, 1, serial);
      }

      succeed=try_to_read_serial(&chr);
      if (succeed) {
        putchar(chr);
      }
  }
}
```

Dividing tasks to different processes 1

- The program on the previous page works in real time. The problem is that it wastes CPU time by observing (”polling”) all the time whether data is ready to be read. This is done by executing “try” instructions all the time. This testing or trying would be done maybe millions of times per second
- The polling method actually represents a simple operating system where several tasks are done concurrently. In this case these tasks are reading and processing keyboard input and reading and processing input from the communication channel. This program is capable to respond ”immediately” to different events from the outside world

```
int main (void) {
  FILE* serial;
  char chr;
  serial = fopen("//dev//ttyS0", "rw");
    while(1) {
      succeed=try_to_read_kb(&chr);
      if (succeed) {
        putchar(chr);
        fwrite(&chr, 1, 1, serial);
      }

      succeed=try_to_read_serial(&chr);
      if (succeed) {
        putchar(chr);
      }
  }
}
```

Dividing tasks to different processes 2

- The problem of wasting processor time can be avoided by dividing the different tasks in different processes (A and B) as it has been done on the right
- This method does not waste CPU time, because
  - When the process has initiated the i/o, it is moved to a blocked state and the operating system does the rescheduling allowing another process (one in a ready state) to run (to use the CPU)
  - After the i/o becomes ready, the operating system moves the process back to a ready state
- Next we study more this method and learn a bit more how the computer and the operating system are working when our char application is running

Process A

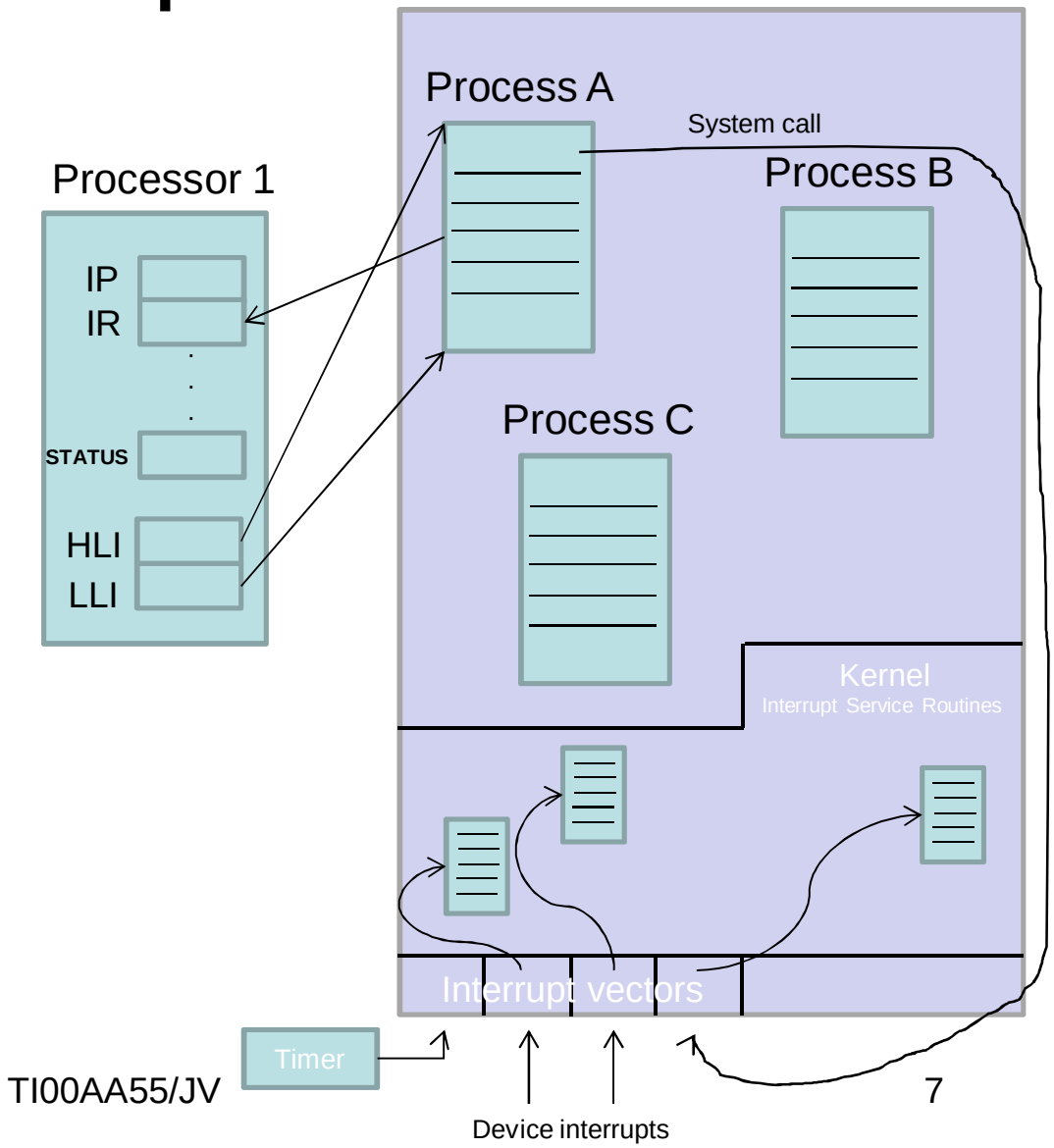
```
int main (void) {
  FILE *serial;
  char chr;
  serial = fopen("//dev//ttyS0", "w");
  while(1) {
    chr = getchar();//Can block without problem
    fwrite(&chr, 1, 1, serial);
  }
}
```

Process B

```
int main (void) {
  FILE *serial;
  char chr;
  serial = fopen("//dev//ttyS0", "r");
  while(1) {
    fread(&chr, 1, 1, serial); // Can block
    putchar(chr);
  }
}
```

Concurrent processes

- The picture on the right can be used to illustrate how concurrent processes are run and how the operating system and the processor are co-operating with them
  - The kernel is a computer program that manages input/output requests from software and translates them into data processing instructions for the central processing unit and other electronic components of a computer.
  - The kernel is a fundamental part of a modern computer's operating system

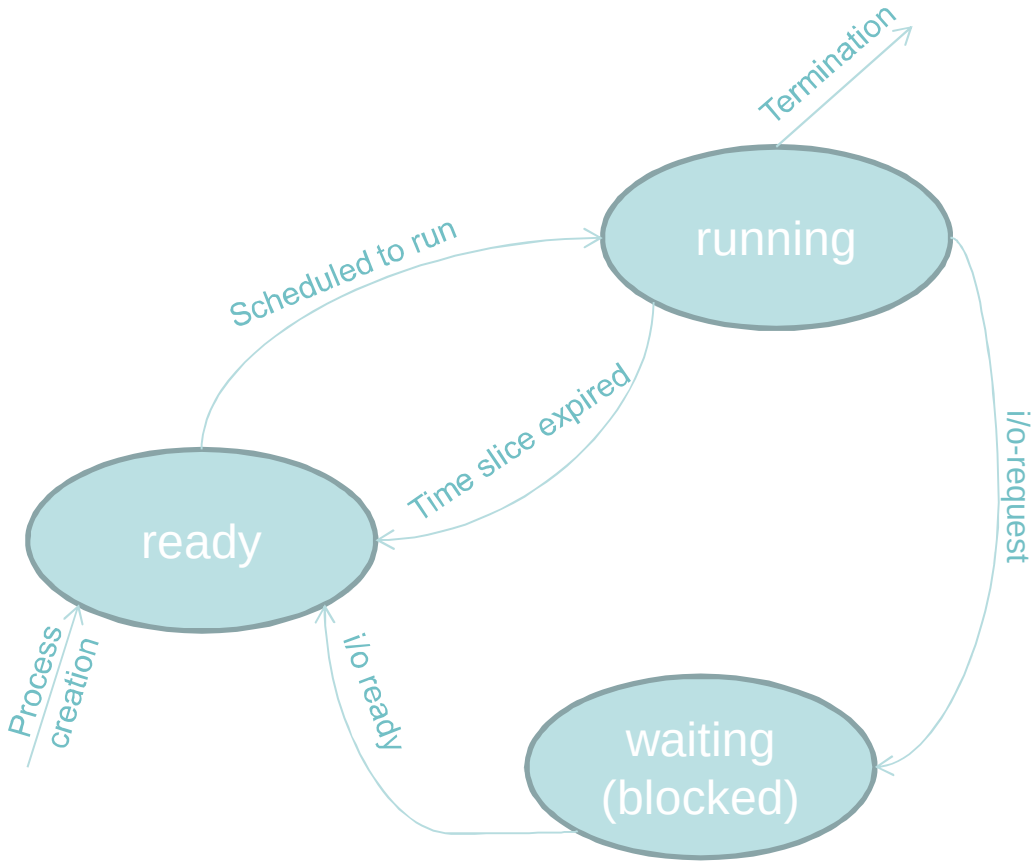


Main things to note in the figure

- Registers of the processor
  - IP (Instruction pointer; Points to the instruction in the memory that will be executed next)
  - IR (Instruction register; Register that contains the instruction under execution)
  - STATUS (Status register that contains among other things the bit(s) that indicate the privilege state of the processor.
  - HLI (High Limit register)
  - LLI (Low Limit register)
    - If the process under execution tries to access memory area outside these limit registers, an error happens
- Interrupt vector and interrupt service routines
  - Interrupt vector that contains addresses of interrupt service routines
  - Interrupt service routines are program code that belongs to the operating system
  - The execution of the operating system code starts always when interrupt (either hardware or software) interrupt happens. Operating system code is run in privileged mode
- Interaction between a process and operating system
  - When the process needs a service from the operating system it generates a software interrupt (makes a system call)

Process states

- Remember also the state diagram of the process you learned last time
  - The program is always running as a process under the control of the operating system
- The state diagram looks as on the right



”Maintaining the order”

- Because several processes are running concurrently in the computer, strict order should be maintained to make sure for example:
  1. That all processes can access peripheral devices (disks and others)
  2. That all processes can use processor to proceed
  3. That processes cannot mess up each other’s data
- The basis for this protection is that processor can run in two different mode:
  1. **privileged** mode (kernel mode, monitor mode, super user mode)
  2. **user** mode (unprivileged mode)
- The processor mode is determined by a bit in the processor status word
- The basic difference between modes is that some machine language instructions can be executed only in the privileged mode

Kernel mode and user mode

- Instructions that can be executed only in the kernel mode are:
  1. read and write from/to device registers
  2. setting the memory limit registers
  3. setting the counter register of the timer
- In the user mode the program can access only memory addresses between memory limit registers (otherwise a segmentation fault or access violation error is generated and the program is stopped)
- The consequence is that a process cannot refer to the memory area of the operating system, the memory area of other processes or the interrupt vector
- Only the operating system code is run in the kernel mode. Other programs are run in the user mode
- Processor can move from the user mode to the kernel mode only via an interrupt (device interrupt or software interrupt). This also means that the only way to move to run operating system code is by generating an interrupt. When an interrupt service routine is completed (IRET), the processor is restored to the user mode
  - On Intel architectures, 64-bit OS uses a special SYSCALL instruction instead of INT software interrupt instruction
- If a program needs to access a peripheral device, it is not possible to access directly device register, because it cannot know if some other process is already using the device. The program needs to ask device access service from the operating system. Asking service from the operating system is called **system call**

Kernel mode and user mode

- Processor hardware prohibits i/o instruction execution in user mode

```
[jarkkov@edunix lecture02]$ a.out
Address of the main() function is 0x40051e
Address of the p is 0x7fffc6dc6858
Now we try to write directly to the I/O
device
Segmentation fault
[jarkkov@edunix lecture02]$
```

- Memory management prohibits writing to memory area not owned by the current user

```
[jarkkov@edunix lecture02]$ a.out
Address of the main() function is 0x400504
Address of the p is 0x7fff787c53f8
Now we try to write to the OS memory area
Segmentation fault
[jarkkov@edunix lecture02]$
```

```
#include <stdio.h>
#include <sys/io.h>
#define IOTEST 1

int main() {
  char *p = (char *)0x80000000;

  printf("Address of the main() function is %p\n", main);
  printf("Address of the p is %p\n", &p);

#if !defined(IOTEST)
  printf("Now we try to write to the OS memory area\n");
  *p = 0;
  printf("It should not work\n");
#else
  printf("Now we try to write directly to the I/O device\n");
  outb(0x00A0, 0x00); // output byte 00H to the i/o port A0H
  printf("It should not work\n");
#endif
}
```

Timer interrupt and context switch

- System calls look like ordinary C-function calls, but actually these C-functions are only a nice cover for a software interrupt
- In practice the processor starts to execute operating system code, when system call is done or when hardware interrupt occurs (see page 7)
- Timer interrupt has a special role in the maintenance of the order
  - Timer interrupt occurs by default in Linux 100 times per second
  - This guarantees that the operating system gets the control 100 times in a second and a scheduling can be done so that no process can use processor too long time
- Everybody should recall the basic principles of interrupt handling

- Context switch:
- When operating system does scheduling and gives turn to an another process to run, a context switch is needed. All information needed to continue the process later are stored and the information for the next process is restored. This information is called a process context and it contains for example:
  - value of instruction pointer IP (PC)
  - other registers
  - memory limits
  - list of open files
  - book keeping data (for example used CPU time)
- Remark. These information is usually kept in a so called PCB (Process Control Block)

System calls, interrupts, scheduling and process states

- Let’s study, using a simple practical example, how system calls, interrupts, scheduling and process states belong together (see the figures on pages 7 and 9)

- The following things are considered:
  a) What happens, when the system call read is done (device is available / device is not available)
  b) What happens, when i/o is completed
  c) What happens (or can happen), when timer interrupt occurs

```
int main(void) {
  char chr;
  while(1) {
    read(0, &chr, 1);
      ---
      ---
      ---
  }
}
```

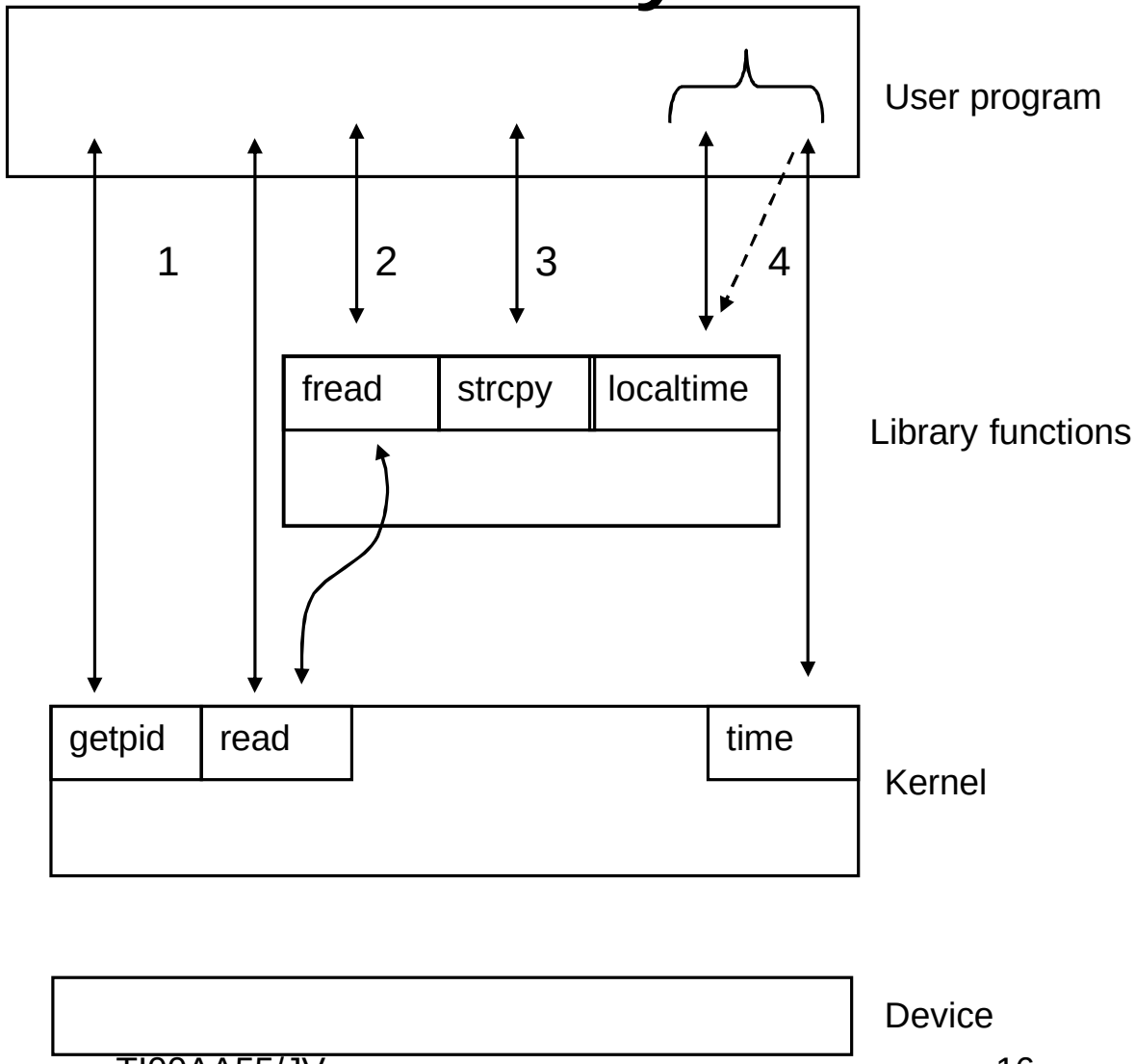
timer-interrupt can occur

System calls and library calls

- System call is a service request that a process uses to ask service directly from the kernel
- As a result of the system call, the processor executes the code of operating system on behalf of the running process
- Although the system call is made like calling a C-function, there is a big difference between system call and ordinary C-function call
- The reason for this is that actually software interrupt is generated to enter a system call code and that’s why the processor is in privileged mode when executing the code of operating system
- System calls offer minimal and reduced set of services. The tasks are small and elementary
- In addition to the system calls, programmer can use services of functions in the run-time function library, that can be used to ”further refine” the data
- The drawing on the next transparent illustrates in what ways a programmers can use system calls and library calls

System calls and library calls

1. Application → system call
2. Application → library call → system call (implicit)
3. Application → library call
4. Application → system call → library call (explicit)



Examples of system calls and library functions

- System calls are really basic operations and they are done using C-function call procedure
- Library functions provide more versatile collection of functions to ”refine data further”
  - Often library functions use a system call to complete it’s task, as we saw on the previous transparent (fread calls system function read)

System call	**Library function**
Memory allocation sbrk	Memory allocation malloc
Read and write from and to read, write	Read and write from and to fread, fscanf, fgets, fwrite, fprintf, fputs
lseek	fseek
Reading system time time Proto: time_t **time** (time_t *ptime);	Time handling Proto: char ***ctime**(const time_t *time); Proto: struct tm* **localtime**(const time_t *time);
Creating and waiting a child process: fork, wait	Running a program from inside a program: system
Executing a program: execxy	(Library function system uses system calls fork, wait, dup2, execxy)

Broken-down time structure tm

- ”Basic time” (calendar time) is represented in Unix as seconds from January 1. 1970, 00:00:00
- Data type time_t is used for that purpose and you can ask the kernel “what is the current time” using system call time (see the previous transparent)
- There are several library functions to further refine the time to more suitable forms (see the previous transparent)

```
struct tm {
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;  // 1, ..., 31
  int tm_mon;   // 0, ...,11
  int tm_year;  //year calculated from year 1900
  int tm_wday;  //0 = Sunday, 1 = Monday, ...
  int tm_yday;  //order no of day of year, 0 is 1st
  int tm_isdst; // >0, daylight saving time in effect
} ;
```

You need to include the header file **time.h** to use times and time handling functions

Error handling of system calls

- If a system call does not succeed, it usually returns -1 indicating that error occurred
- In this case the system sets an error code in the global integer variable errno, indicating the reason for the error
- If the system call succeeds, the **variable errno is not cleared**
- Descriptive names are defined for each error code in the header file errno.h (#define constants)
- The include file errno.h contains these definitions and other constants and function prototypes needed for the error handling
- Remark. **Test the errno variable only if system call has returned -1**

- Some useful error handling functions:

```
<stdio.h>
void perror(const char *message);
```

Displays the error message (in plain text) after the text passed as a parameter. The error message of course corresponds the current value of global variable errno.

```
<string.h>
char *strerror(int error_num);
```

Converts an integer error code to an error message in plain text (string)

Some system call errors

EACCES	Permission denied
EAGAIN	Resource temporarily unavailable
EBADF	Bad file descriptor
ECANCELED	Operation cancelled
ECHILD	No child processes
EDEADLK	Resource deadlock avoided
EDOM	Domain error
EEXIST	File exists
EFAULT	Bad address
EFBIG	File too large
EINPROGRESS	Operation in progress
EINTR	Interrupted function call
EINVAL	Invalid argument
EIO	Input/output error
EISDIR	Is a directory
EMFILE	Too many open files
ENAMETOOLONG	Filename too long
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOTDIR	Not a directory
ENOTSUP	Not supported
EPERM	Operation not permitted
ERANGE	Result too large
ESPIPE	Invalid seek
ETIMEDOUT	Operation timed out

Primitive System Data types

- The primitive data types are used in system calls and may be dependent on the implementation of the operating system (they all are “relatives” of integers)
- By using these type names instead of “real” types it is guaranteed, that the programs are portable from one Unix-operating system to an another
- They are defined in <sys/types.h>

clock_t	time in clock ticks
fpos_t	file position
gid_t	numerical group id
ino_t	i-node number
mode_t	file creation flags
off_t	the size and offset of the file
pid_t	process id
ptrdiff_t	pointer difference
size_t	the size of object (unsigned)
ssize_t	return type for functions, that return number of bytes or error (read, write) (signed size)
time_t	calendar time in internal form (time in seconds from 00:00:00 1.1.1970)
uid_t	numerical user id
wchar_t	wide char

System limits and properties

- Different systems can have different system limits and features. To make a program portable we need to take these differences into account in the program
- The program can determine these limits and supported features
- Limits and features can be **determined during compile time and/or during run time**
- Compile time determination is based on defined constants in the header files
- Run time determination is based on the system functions sysconf, pathconf and fpathconf

- Limits that can be determined during compile time:
  - POSIX (limits.h | unistd.h | nowhere)
  - These are real limits in the system:

MAX_INPUT	The size of input buffer
NAME_MAX	The maximum length of file name
OPEN_MAX	The maximum number of open files per process
PATH_MAX	The maximum length of path name
SSIZE_MAX	The maximum value in the data type ssize_t

POSIX compatibility limits

- In addition to the real limits of the operating system, there are limits that the operating system in minimum needs to fulfil to be POSIX-compliant operating system
  - See some examples of these kind of limits on the right
- These are not usually used in programs
- They are anyway useful to know, if it is important to make a software portable to all POSIX compliant systems
- Remark. Don’t forget the basic limits defined in C-standards (see the next page)

- POSIX (bits/posix1_lim.h)
  - POSIX specifies that the maximum values must be at least these in POSIX compliant system

_POSIX_MAX_INPUT	The size of input buffer
_POSIX_NAME_MAX	The maximum length of file name
_POSIX_OPEN_MAX	The maximum number of open files per process
_POSIX_PATH_MAX	The maximum length of path name
_POSIX_SSIZE_MAX	The maximum value in the data type ssize_t

ANSI C limit in files float.h and limits.h

float.h	(limits for floating point numbers)
FLT_MAX	Max value of float
FLT_MIN	Min value of float
FLT_DIG	Precision of float in digits
FLT_EPSILON	Smallest x that makes 1.0 + x ≠ 1.0
DBL_MAX	Max value of double
DBL_MIN	Min value of double
DBL_DIG	Precision of double in digits
DBL_EPSILON	Smallest x that makes 1.0 + x ≠ 1.0

limits.h	(limit for integers)
CHAR_BIT	Bits in a byte
CHAR_MAX	Max value of char
CHAR_MIN	Min value of char
SCHAR_MAX	Max value of signed char
SCHAR_MIN	Min value of signed char
UCHAR_MAX	Max value of unsigned char
SHRT_MAX	Max value of a short int
SHRT_MIN	Min value of a short int
USHRT_MAX	Max value of an unsigned short int
INT_MAX	Max value of an integer
INT_MIN	Min value of an integer
UINT_MAX	Max value of an unsigned int
LONG_MAX	Max value of a long int
LONG_MIN	Min value of a long int
ULONG_MAX	Max value of an unsigned long int

Obtaining run-time limits

- Function prototypes (file unistd.h):

```
long sysconf(int feature_name);
long pathconf(const char *pathname,
              int feature_name);
long fpathconf(int file_desc,
               int feature_name);
```

- Example:

```
int posixversion;
posixversion = sysconf(_SC_VERSION);
printf("This system has posix
version%ld", posixversion);
// the format is yyyymm (yyyy is year,
mm is month)
```

- The system does not necessarily support all limits
- If the function above returns error (-1) and the error code (errno) is not set (is 0), it means that system does not support the asked feature or has no limit available at run-time
  - Remark. You need assign 0 to errno variable before calling sysconf, pathconf or fpathconf function

- Examples of run-time limits (feature_name)

limit or feature	feature_name
Clock ticks/s	_SC_CLK_TCK
OPEN_MAX	_SC_OPEN_MAX
_POSIX_VERSION	_SC_VERSION
NAME_MAX	_PC_NAME_MAX
PATH_MAX	_PC_PATH_MAX
_POSIX_NO_TRUNCATE	_PC_NO_TRUNC

File handling

- Remark. Remember that conceptually devices are files at operating system level
- The file services of the operating system are unbuffered
- File descriptors are used to “identify” the file
- File descriptor is an abstraction of a file at the operating system level
- Internally file descriptor is an integer number
- Keyboard and display are files that are opened by default. Their file descriptors are 0 and 1
  - File descriptor 2 is reserved for the error output console (often the display)
- POSIX standard defines names for these descriptors
  - STDIN_ FILENO (keyboard by default)
  - STDOUT_FILENO (display by default)
  - There is file descriptor STDERR_FILENO too (2 by default)
- The most important system calls of file handling are:
  - open
  - write
  - read
  - lseek
  - close

Prototypes of file handling system calls

```
int open(const char *path, int open_flags, …);
```

The third parameter may be mode_t right_flags

```
ssize_t write(int file_desc, const void *buff, size_t n);
ssize_t read(int file_desc, void *buff, size_t n);
off_t lseek(int file_desc, off_t offset, int whence);
int close(int file_desc);
```

- Parameter whence in the function lseek can be one of the following:
  1. SEEK_SET
  2. SEEK_CUR
  3. SEEK_END
- Remark. See an example on the pages 30 and 31, that illustrates the use of the functions above and at the same time compares them to corresponding library functions
- Remark. To use these functions you need headers fcntl.h (function open) and unistd.h (read, write, lseek and close functions)

File related constants 1

- Parameter open_flags is constructed by combining (ORing) one of the next three values (bit field of 2 bits)

O_RDONLY	Read only (bit field value 0)
O_WRONLY	Write only (bit field value 1)
O_RDWR	Read and write (bit field value 2)

- Only one of the possible first values for the bit field can be selected
- Other flag values consist of one bit only and it is possible to combine them quite freely with OR
- Remark. These flags are defined in the header fcntl.h

- and some of the following bits:

O_APPEND	Append
O_CREAT	The file is created, if it does not exist (parameter 3., access permission bits, is needed in function open)
O_EXCL	Open returns error, if bit O_CREAT is set and the file already exists
O_TRUNC	The file is truncated , if it exists (can be used with O_WRONLY or O_RDWR)
O_NONBLOCK	Non blocking mode

File related constants 2

- Bits needed to construct parameter access permission bits,(or right_flags):

S_IRUSR	user read
S_IWUSR	user write
S_IXUSR	user execute
S_IRGRP	group read
S_IWGRP	group write
S_IXGRP	group execute
S_IROTH	other read
S_IWOTH	other write
S_IXOTH	other execute

- All of these bits can freely be combined using OR operator
- Remark. These flags are defined in the header sys/stat.h
- Remark. Later on we also learn flags S_ISUID (Set User ID) and S_ISGID (Set Group ID)

File handling example (system calls)

```
#include <unistd.h>
#include <fcntl.h>
int main (void) {
    int fd, n;
    char buff[10];

    fd = open("file.txt", O_WRONLY | O_CREAT, 0700);
    if (fd < 0) {
        perror("File creation");
        exit(1);
    }
    write(fd, "AAAAAAAAAA", 10);
    close(fd);

    fd = open("file.txt", O_RDONLY);
    if (fd < 0) {
        perror("File open");
        exit(1);
    }
    n = read(fd, buff, 10);
    if (n > 0)
        write(STDOUT_FILENO, buff, n);
        close(fd);
        return 0;
}
```

File handling with system calls / library functions

Simple file handling with system calls:

```
#include <unistd.h>
#include <fcntl.h>
int main (void) {
    int fd;
    char buff[10];

    fd = open("file.txt", O_RDONLY);
    read(fd, buff, 10);
    close(fd);
    return 0;
}
```

Same thing using library functions:

```
#include <stdio.h>
int main (void) {
    FILE *fp;
    char buff[10];

    fp = fopen("file.txt", "r");
    fread(buff, 10, 1, fp);
    fclose(fp);
    return 0;
}
```

Why it is important that the user program cannot modify the interrupt vector?

- This is additional information about the protection things
- Limit register also prevent a process from accessing the interrupt vector
- This small example demonstrates, why it is important that a process cannot modify the interrupt vector

```
void f(void);
int main (void) {
    // assign the address of function f in the element X
    // of the interrupt vector
     …
    // generate the interrupt number X (INT X);
}

void f(void) {
    // This code would run now in the privileged mode
    // and could do what ever
}
```