Input/Output – part 1

Last modification: 06.11.2017

Streams and descriptor-based I/O

POSIX

- **File descriptor** - a per-process unique, non-negative integer used to identify an open file for the purpose of file access. The value of a file descriptor is from zero to {**OPEN_MAX**}. A process can have no more {**OPEN_MAX**} file descriptors open simultaneously.
- A **stream** is a file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the **fdopen( ), fopen( )**, or **popen**( ) functions, and are associated with a file descriptor. A process can have no more {**FOPEN_MAX**} streams open.

Comments

- File descriptors, used for low-level I/O, are represented as objects of type **int**, while streams are represented as **FILE*** type „file pointers”.
- File descriptors have to be used when one wants to perform control operations that are specific to a particular device or to do I/O operation in special modes, e.g. non-blocking/polled input.
- Streams are built on top of the file descriptor facilities, so it is possible to get related descriptor of a stream (**fileno**()). It is also possible to build a stream atop a descriptor (**fdopen()**).
- I/O streams are **portable** across systems running ISO C, while file descriptor based I/O is supported by POSIX and UNIX-like systems.

Stream buffering

Streams can be:

- **Unbuffered** (_IONBF) - characters written to or read from an *unbuffered* stream are transmitted individually to or from the file **as soon as possible**.
- **Line buffered** (_IOLBF) **–** characters are transmitted when a newline character is encountered.
- **Fully buffered** (_IOBF) - bytes are intended to be transmitted as a block when a buffer is filled.

Good size of buffer (`BUFSIZ`) is given in <stdio.h>

Newly opened streams are **fully buffered**, except the streams connected to an interactive device such as a terminal are initially **line buffered**.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
        // sets buffering mode (_IOFBF, _IOLBF, _IONBF) and
        // possibly the buffer of given size (if buf not NULL)

void setbuf (FILE *stream, char *buf); // sets stream mode
        //to _IOBF unless buf==NULL (then mode is _IOBF)
int fflush (FILE *stream); //  buffered output of the stream
        // delivered to file, if stream==NULL → all buffered
        // output is flushed
```

Opening/closing streams

- stdin, stdout, stderr – streams available to a process upon start-up.
- Opening / closing other streams (to files, pipes):

```
FILE *fopen(const char *filename, const char *opentype);
        // opens a stream for I/O to the file filename, and
        // returns a pointer to the stream. opentype:
        // "r", "w", "r+" - read-only, write-only, read-write
        // "a", "a+" - appends write-only, appends read-write
        // after flushing the stream buffer
        // "w+"- truncates (to 0 length) then updates
```

Note: Opening with append mode ('a')shall cause all subsequent writes to the file to be forced to the then current end-of file, regardless intervening calls to **fseek()**

Note: When opened, a stream is fully buffered if and only f it can be determined to not refer to an interactive device.

Note: When in append mode ('+') mixing reading/writing is allowed if separated by **fflush()** or positioning (**fseek(),fsetpos() ,rewind()**)

```
FILE *fdopen (int fildes, const char *mode); // associates
                // a stream with a file descriptor
int fclose(FILE *stream); // closes the stream
int fcloseall(void); // closes all streams
```

Stream locking; EOF and errors

- The POSIX standard requires that by default the stream operations are **atomic**. i.e., issuing two stream operations for the same stream in two threads at the same time will cause the operations to be executed as if they were issued sequentially.
- Explicit stream locking is also available to control stream access:

```
void flockfile (FILE *stream);
int ftrylockfile (FILE *stream);
void funlockfile (FILE *stream);
```

- Testing End-Of-File indicator on the stream:

```
int feof(FILE *stream);
```

- Testing and clearing error indicator on the stream:

```
void ferror(FILE *stream);// testing error indicator
void clearerr(FILE *stream); // clearing the EOF
        // and error indicators for the stream
```

Stream position

For streams that refer to randomly-accessed devices/files it is possible to get and set stream position (for other an error condition occurs).

Getting position in a stream:

```
long int ftell(FILE *stream);// returns -1L on error
off_t ftello(FILE *stream);// used for very long files
int fgetpos(FILE *stream, fpos_t *position); // returns error
// code; shall store to *position the current values of the
// parse state (if any) and position indicator for the stream
```

Setting position in a stream:

```
int fseek(FILE *stream, long int offset, int whence); // sets
      // absolute or relative position
int fseeko(FILE *stream, off_t offset, int whence); // long f.
void rewind(FILE *stream);//reset the file position indicator
int fsetpos(FILE *stream, const fpos_t *position);// sets
      // absolute position in a stream
```

Note: position indicator contains unspecified information usable by fsetpos() for repositioning the stream to its position at the time of the call to fgetpos() .

Note: to write programs compatible with non-POSIX systems that implement binary and text files differently use fgetpos/fsetpos and binary flag ‘b’ in the fopen mode

Stream I/O operations

- Basic (byte-oriented) input functions (also ISO C):

```
int fgetc(FILE *stream); int getc(FILE *stream);
int getchar(void);
char * fgets(char *buf, int buflen, FILE *stream);
tnt fscanf(FILE *stream, const char *template,…)
```

- Basic (byte-oriented) output functions (also ISO C):

```
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int fputs(const char *s, FILE *stream); // s points at a C-string
int puts(const char *s);
int fprintf(FILE *stream, const char *template, …);
int printf(const char *template, …)
```

- Binary read/write functions (note stream format is **machine/OS dependent**!):

```
size_t fread(void *ptr,size_t size,size_t nitems,FILE *stream);
size_t fwrite(void *ptr,size_t size,size_t nitems,FILE *stream);
```

Directories

POSIX:

- **Working (current) directory** - a directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash
- **Root directory** - directory, associated with a process, that is used in pathname resolution for pathnames that begin with a slash.

```
int chroot(const char *path);  // defines new path
                               // to the root directory
int chdir(const char *path);   // changes the path
                               // to the working directory
char *getcwd(char *buf, size_t size);  // returns
        // null-terminated path to the working directory
        // see getcwd(3) for details
```

Directory stream

- **Directory entry (or link)** - an object that associates a filename with a file. Several directory entries can associate names with the same file.

POSIX:
directory entry is represented by `struct dirent` (see `<dirent.h>`), which includes:

`ino_t d_ino`	- file serial number (e.g. node number)
`char d_name[]`	- name of entry.

Note:

- Length of the d_name is available as `strlen(d_name)` **not** `sizeof(d_name)`
- Attributes of a directory entry can be retrieved with `stat()` function call (or `lstat()` for symbolic link entry)
- **Directory stream** - a sequence of all the directory entries in a particular directory

POSIX:

Header file `<dirent.h>` defines **DIR** - a (possibly incomplete) type representing a **directory stream**.

Directory stream operations

```
DIR * opendir(const char *dirname);
```
// opens and returns directory stream

```
int    closedir (DIR *dirstream );
```
// closing the directory stream

```
struct dirent * readdir (DIR *dirstream);
```
// reading directory entry

Note: the pointer returned points to `readdir` maintained data buffer which may be overwritten by another `readdir()` call on the same stream.

```
int readdir_r(DIR *restrict dirp,
        struct dirent *restrict entry,
        struct dirent **restrict result);
```
// reading directory entry to a user specified buffer (`entry`)
// with `d_name` field of at least `{NAME_MAX}+1` bytes

```
void   rewinddir (DIR *dirstream);
```
// resetting the directory stream position

```
int    telldir (DIR *dirstream);
```
// reporting position in the stream

```
void   seekdir (DIR *dirstream, long int pos);
```
// setting stream pos.

Example

Example of simple current working directory listing code:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <errno.h>
#define ERR(source) (perror(source),\
        fprintf(stderr,"%s:%d\n",__FILE__,__LINE__), exit(EXIT_FAILURE))
int main(int argc, char** argv) {
   DIR *dirp;
   struct dirent *dp;
   struct stat filestat;
   if (NULL == (dirp = opendir(".")) ERR("opendir");
   do {
        errno = 0;
        if ((dp = readdir(dirp)) != NULL) {
           if (lstat(dp->d_name, &filestat)) ERR("lstat");
           printf("%16s : %10zd B\n",dp->d_name, filestat.st_size);
        }
   } while (dp != NULL);
   if (errno != 0) ERR("readdir");
   if(closedir(dirp)) ERR("closedir");
   return EXIT_SUCCESS;
}
```

Handling file links

`int ret=link(const char *oldpath, const char *newpath)`

makes a new link to the existing file named by `oldpath`, under the new name `newpath`

`int ret=symlink(const char *oldpath, const char *newpath)`

makes a symbolic link to `oldpath` (does not have to exist) named `newpath`

```
int ret=readlink(const char *filename, char *buf,
                 size_t size)
```

gets the value of the symbolic link `filename` and stores it in a buffer `buf` of given `size`.

`int ret=rename(const char *old, const char *new)`

renames the file `old` to `new`.

`int ret=unlink(const char *path)` – deletes the file name `path`.

The above functions return `ret==0` on success or `ret== -1` otherwise (error code in `errno`).