Streams

R. Inkulu
http://www.iitg.ac.in/rinkulu/

Memory hierarchy

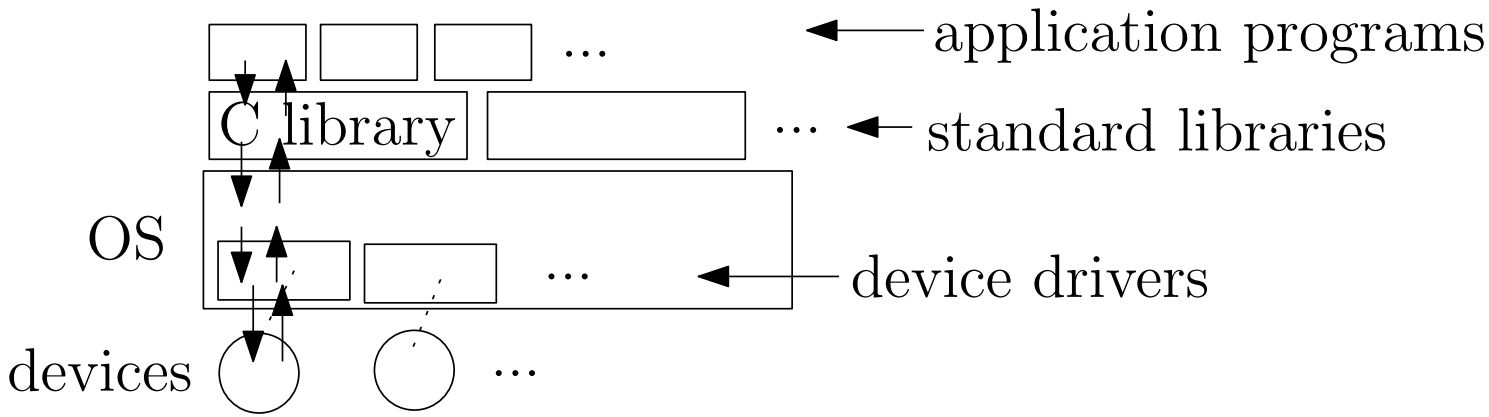
- internal: processor registers and cache
- main: RAM
- secondary: harddisk
- tertiary: off-line storage

going down the hierarchy, typically, storage space increases but the access speed (together with the cos per unit memory) decrease

Storage of a data file

- Memory allocated for a file to store data on a device (ex. hard disk) need not be contiguous, but this fact is hidden from user and the operating system manages the details (via *file tables*).

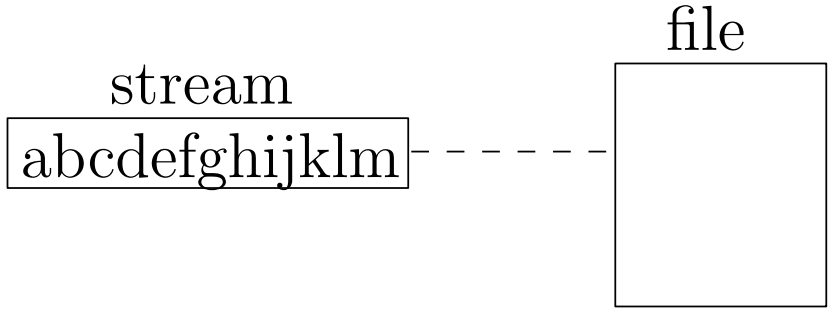
Files and devices



- In operating systems' that are variants of unix [1], every device (disk, floppy, cd, printer, scanner, communication port, graphics hardware, etc.,) is mounted as a file.
- Writing/reading file corresponding to a device is the way to access the device: device details are hidden from the user.

[1] for convenience, let this lecture be specific to linux

Description of a stream



- motivation: buffering in main memory helps in accessing secondary/tertiary devices intermittently, significantly reducing the execution time of a program

- a *stream* is a source or destination of data that is associated with a file; OS allocates memory for a stream from the process address space

- C library together with OS takes care of reading (resp. writing) buffered/unbuffered stream of bytes from (resp. to) any file

Kinds of streams

text:

- data is converted to characters back and forth
- adv: easy to inspect (ex. using gedit)
- disadv: precision of binary data may get affected
- ex. (vaguely) any stream decipherable using gedit
- primarily, fgetc, fputc, fgets, fputs, fprintf, fscanf, sprintf, sscanf does the job

binary:

- data stored in the same form as in RAM
- adv: compactness, speed, integrity
- disadv: delimiters needed to be handled
- ex. a jpg or pdf file
- primarily, fwrite, fread functions does the job

fopen, fclose

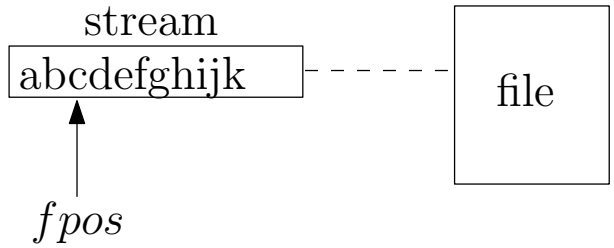
```
FILE *fp = fopen("/home/rinkulu/exp/abc.txt", "w");

if (fp != NULL) {
   ...
   fclose(fp);
}
```

- a stream (buffer) is connected to a file by opening it; the connection is broken by closing the stream
- within a process, only FOPEN_MAX number of files can be open at any instance (apart from FILE object, many other corresponding objects are instantiated in the operating system)
- important modes in opening are:
  "r", "w", "a",
  "r+", "w+", "a+", [2]
  "rb", "wb", "ab",
  "r+b", "w+b", a+b"

[2] "r+": open a file for update (both reading and writing); the file must exist though; "w+": creates an empty file for both reading and writing; "a+": open a file reading and appending; file is created if it does not exist

FILE structure




```
typedef struct {
   char *fpos;      //current position in stream
   void *base;
   unsigned short handle;
   short flags;     //read, write, etc., flags
   short unget;
   unsigned long alloc;
   unsigned short buffincrement;
} FILE;
```

- FILE structure [3] contains a pointer to a position in the associated buffer (stream), model flags, etc.,;
- fpos is made to point to the appropriate location in the stream (buffer) by C library; *fpos is the location to write/read next

[3] members with no remarks are not relevant for this course

fputc, fflush

```
FILE *fp = fopen("abc.txt", "w");

if (fp != NULL) {
   fputc('a', fp));
   fputc('b', fp);
   fputs("cde", fp);

   fflush(fp);
   fclose(fp);
}
```

- int fputc(int c, FILE* stream)
- int fputs(const char *s, FILE *stream): writes string pointed by s and a newline char to stream
- int fflush(FILE *stream): flushes output stream

  new line character implicitly flushes text streams (hence, texts streams are known as *line buffered* streams)

fprintf

```
FILE *fp = fopen("abc.txt", "w");

if (fp != NULL) {
   fprintf(fp, "%d, %c, %s", 25, 'I', "hello");

   fflush(fp);
   fclose(fp);
}
```

- int fprintf(FILE *stream, const char *format, ...)

fgets

```
char buf[30];
FILE *fp = fopen("readme.txt", "r");
if (fp != NULL) {
   fgets(buf, 20, fp);
   printf("%s", buf);

   fclose(fp);
}
```

- char *fgets(char *s, int n, FILE *stream): reads into buffer pointed by s from stream; stops when either $n-1$ characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

fscanf

```
char buf[20]; int num; char c;
FILE *fp = fopen("readme.txt", "r");
if (fp != NULL) {
   fscanf(fp, "%d%c%s\n", &num, &c, buf);
   printf("%d\n %c\n %s\n", num, c, buf);

   fclose(fp);
}
```

- int fscanf(FILE *stream, const char *format, ...): like the familiar scanf function, it does the pattern matching with respect to *format* specified
- avoid intertwining fscanf and fgets: due to issues with newlines (for example, fscanf %d won't consume a trailing newline)

fgetc, ungetc, feof

```
char c, buf[256];
FILE *fp = fopen("readme.txt", "r");
if (fp != NULL) {
   while (!feof(fp)) {
      c = fgetc(fp);
      if (c == 'a') ungetc('b', fp);

      fgets(buf, 255, fp);
      printf("%s", buf);
   } }
```

- int fgetc(FILE *stream)
- int ungetc(int c, FILE *stream): pushes c onto stream so that this is available for the next read operation
- int feof(FILE *steam): returns non-zero if the end of file indicator is set (C library sets error code to EOF after an input routine has tried to read, and failed as it reached the end; EOF itself won't be explicitly stored in the file)

ferror, clearerr

```
FILE *fp = fopen("input.txt", "r");

fputc('a', fp);
if (ferror(fp)) {
   printf("writing to file opened in readonly mode\n");
}
clearerr(fp);
if (ferror(fp)) {
   printf("more errors\n");
}
```

- int ferror(FILE *stream): tests the error indicator for the given stream
- void clearerr(FILE *stream): clears the end-of-file and error indicators for the given stream

Standard text streams

- when the program starts three special text streams get open by default: standard input (stdin), standard output (stdout), (unbuffered) standard error (stderr)

- stdin, stdout, and stderr are pointers to FILE objects

Few functions to access standard text streams

```
int putchar(int c)
int puts(const char *s)
  //puts string pointed by s appended
  //with a new line to stdout
int printf(const char *format, ...)

int getchar(void)
char *gets(char *s)
  //reads next input line (or, till the end of
  //stream) from stdin into buffer pointed by s
int scanf(const char *format, ...)

void perror(const char *s)
  //prints string pointed by s appended with
  //an implemnetation-defined error message
  //(based on errono) to stderr
```

Few functions for string streams

```
int sprintf(char *s, const char *format, ...)
    //prints to s

int sscanf(const char *s, const char *format, ...)
   //reads from s
```

Binary input and output

```
size_t fwrite(void *buffer, size_t len,
              size_t count, FILE *stream);
   //writes count number of chunks each of size len
   //into stream from buffer


size_t fread(void *buffer, size_t len,
             size_t count, FILE *stream);
   //reads count number of chunks each of size len
   //from stream into buffer
```

- copies byte-by-byte from (resp. to) the memory to (resp. from) stream
- *deserialization* order of objects must be same as the *serialization* order of objects (i.e., types corresponding to reading order of objects must precisely match with the types corresping to writing order of types)

fwrite

```
struct Point {
    double x;
    double y;
} pt[2];

pt[0].x=14.50000; pt[0].y=89.3939;
pt[1].x=9.89; pt[1].y=78.38;

FILE *fp = fopen("abc.bin", "wb");

if (fp != NULL) {
    fwrite(pt, sizeof(struct Point), 2, fp);

    fflush(fp);
    fclose(fp);
}
```

fread

```
struct Point {
   double x;
   double y;
} pt[2];

FILE *fp = fopen("abc.bin", "rb");

if (fp != NULL) {
   fread(pt, sizeof(struct Point), 2, fp);

   printf("%lf %lf %lf %lf",
          pt[0].x, pt[0].y, pt[1].x, pt[1].y);
      //prints 14.500000 89.393900 9.890000 78.380000

   fclose(fp);
}
```

fseek, ftell, rewind

```
fseek(fp, 0, SEEK_END);
len = ftell(fp);
printf("%d \n", len);

rewind(fp);
len = ftell(fp);
printf("%d \n", len);
```

- long int ftell(FILE *stream)

  returns the current position of the stream from the beginning
- int fseek(FILE *stream, long offset, int origin)

  sets stream->fpos to $offset$ from $origin$

  origin can be SEEK_SET (beginning of file), SEEK_CUR, or SEEK_END
- void rewind(FILE *stream)

  makes steam->fpos to point to the beginning of the stream

Associating user-specific buffer to a stream

```
char buf[1024];
memset(buf, '\0', sizeof(buf));
setvbuf(fp, buf, _IOFBF, 1024);
```

- int setvbuf(FILE * stream, char * buffer, int mode, size_t size)

  buffer: user allocated buffer to use as stream

  size: buffer size in bytes

  mode: _IOFBUF (full buffering), _IOLBUF (line buffering), _IONBF (no buffer used)

  when buffer is NULL, system automatically allocates

Miscellaneous functions

```
int remove(const char *filename)
int rename(const char *oldname, const char *newname)

FILE *tmpfile(void)
   //opens a temporary file with "wb+" mode;
   //removes when closed or when the program
   //terminates normally
```

Custom printf implementation

```
#include <stdarg.h>
void customprintf(char *fmt, ...) {
   va_list ap;         //va_list structure
   char *p; int ival; char *str;
   va_start(ap, fmt); //initialize ap with the arg before ellipsis
   for (p = fmt; *p; p++) {
       if (*p != '%') {
          putchar(*p); continue; }
       switch(*++p) {
          case 'd':
             ival = va_arg(ap, int);   //get the next unnamed arg
             ...
             break;
          case 's':
             str = va_arg(ap, char*);  //get the next unnamed arg
             ...
             break;
          default:
             putchar(*p);
             break; } }
   va_end(ap);  //to clean up }
```

- va_start, va_arg and va_end are macros whereas the va_list is a type
- constraints in using *variable argument lists*: there must be at least one named arg; the ellipsis operator must be the last arg

functions with va_list instead of ellipsis operator

```
int vprintf(const char *format, va_list arg)

int vfprintf(FILE *stream,
             const char *format, va_list arg)

int vsprintf(char *s, const char *format, va_list arg)
```

homework: get familiarize with these functions