File and Console I/O

CS449 Spring 2016

What is a Unix(or Linux) File?

- File: “a resource for storing information [sic] based on some kind of durable storage” (Wikipedia)
- Wider sense: “In Unix, everything is a file.” (a.k.a “In Unix, everything is a stream of bytes.”)
  - Traditional files, directories, links
  - Inter-process communication (pipes, shared memory, sockets)
  - Devices (interactive terminals, hard drives, printers, graphic card)
    - Usually mounted under /dev/ directory
  - Process Links (for getting process information)
    - Usually mounted under /proc/ directory

Stream of Bytes Abstraction

- A file, in abstract, is a stream of bytes
- Can be manipulated using five system calls:
  - open: opens a file for reading/writing and returns a file descriptor
    - File descriptor: index into an OS array called open file table
  - read: reads current offset through file descriptor
  - write: writes current offset through file descriptor
  - lseek: changes current offset in file
  - close: closes file descriptor
- Some files do not support certain operations (e.g. a terminal device does not support lseek)

C Standard Library Wrappers

- C Standard Library wraps file system calls in library functions
  - For portability across multiple systems
  - To provide additional features (buffering, formatting)
- All C wrappers buffered by default
  - Buffering can be controlled using “setbuf” or “setlinebuf” calls (remember those?)
- Works on FILE * instead of file descriptor
  - FILE is a library data structure that abstracts a file
  - Contains file descriptor, current offset, buffering mode etc.

Wrappers for the Five System Calls

Function Prototype	Description
FILE ***fopen**(const char *path, const char *mode);	Opens the file whose name is the string pointed to by path and associates a stream with it. Returns NULL if error.
size_t **fread**(void *ptr, size_t size, size_t nmemb, FILE *stream);	Reads nmemb elements of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.
size_t **fwrite**(const void *ptr, size_t size, size_t nmemb, FILE *stream);	Writes nmemb elements of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr.
int **fseek**(FILE *stream, long offset, int whence);	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, which can be one of SEEK_SET, SEEK_CUR, SEEK_END.
int **fclose**(FILE *fp);	Flushes the stream pointed to by fp (using fflush) and closes the underlying file descriptor.

File Open Modes

Mode	Description
r	Open text file for reading. The stream is positioned at the beginning of the file.
r+	Open for reading and writing. The stream is positioned at the beginning of the file.
w	Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
w+	Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
a	Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
a+	Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

- Each mode string can in addition contain a “b” character for binary mode

Binary File Dump Example

```
#include <stdio.h>
int main()
{
  FILE *rfile;
  int nread;
  char buf[100];

  if((rfile=fopen(”main.c","rb"))==NULL) return 1;
  while(nread = fread(buf, sizeof(char), 100, rfile)) {
    fwrite(buf, sizeof(char), nread, stdout);
    if(nread != 100) break;
  }
  fclose(rfile);
  return 0;
}
```

```
>> ./a.out
#include <stdio.h>
int main()
{
  FILE *rfile;
  int nread;
  char buf[100];

  if((rfile=fopen(”main.c","rb"))==NULL) return 1;
  while(nread = fread(buf, sizeof(char), 100, rfile)) {
    fwrite(buf, sizeof(char), nread, stdout);
    if(nread != 100) break;
  }
  fclose(rfile);
  return 0;
}
```

Feof and Ferror

Function Prototype	Description
int **feof**(FILE *stream);	Tests the end-of-file indicator for the stream pointed to by stream, returning non-zero if it is set.
int **ferror**(FILE *stream);	Tests the error indicator for the stream pointed to by stream, returning non-zero if it is set.

- fread() does not distinguish between end-of-file and error, and callers must use feof and ferror to determine which occurred. E.g.:

```
if(fread(buf, sizeof(char), 100, rfile) != 100) {
  if(feof(rfile)) printf(“End of file.\n”);
  else if(ferror(rfile)) printf(“A read error has occurred.\n”);
  else { /* unreachable */ }
}
```

Additional (Text) File I/O Functions

Function Prototype	Description
int **fgetc**(FILE *stream);	Reads the next character from stream and returns it.
int **fputc**(int c, FILE *stream);	Writes the character c, cast to an unsigned char, to stream.
char ***fgets**(char *s, int size, FILE *stream);	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. A '\0' is stored after the last character in the buffer.
int **fputs**(const char *s, FILE *stream);	Writes the string s to stream, without its trailing '\0'.
int **fscanf**(FILE *stream, const char *format, ...);	Same as scanf, except reading from stream rather than stdin.
int **fprintf**(FILE *stream, const char *format, ...);	Same as printf, except writing to stream rather than stdout.

Text File Dump Example

```
#include <stdio.h>
int main()
{
  FILE *rfile;
  char buf[100];

  if((rfile=fopen("main.c","r"))==NULL)
    return 1;
  while(fgets(buf, 100, rfile)) {
    fputs(buf, stdout);
  }
  fclose(rfile);
  return 0;
}
```

```
>> ./a.out
#include <stdio.h>
int main()
{
  FILE *rfile;
  char buf[100];

  if((rfile=fopen("main.c","r"))==NULL)
    return 1;
  while(fgets(buf, 100, rfile)) {
    fputs(buf, stdout);
  }
  fclose(rfile);
  return 0;
}
```

Standard I/O

- There are three standard file descriptors for console I/O declared in <stdio.h>
  - extern FILE *stdin; (for standard input)
  - extern FILE *stdout; (for standard output)
  - extern FILE *stderr; (for standard error)
- stdout is used for normal output; stderr is used to output error messages
- All three are line buffered by default
- Underlying file descriptors for stdin, stdout, stderr are 0, 1, 2 respectively

Standard I/O Redirection

- Unix shells allow standard I/O to be redirected to/from another file

Operator	Description
command [N]< file	Redirect input of file descriptor N to file while running command. N defaults to 0 (stdin).
command [N] > file	Redirect output of file descriptor N to file while running command. N defaults to 1 (stdout). If file does not exist, it is created. If it exists, it is truncated.
command [N] >> file	Redirect output of file descriptor N to file while running command. N defaults to 1 (stdout). If file does not exist, it is created. If it exists, output is appended to end of file.

```
– cat < file.in > file.out 2> file.err
– cat < file.in &> file.out.err     (redirecting both stderr and stdout)
– cat < file.in 2>&1                (redirecting stderr to stdout)
– cat < file.in 1>&2                (redirecting stdout to stderr)
– cat < file.in | tee file.out      (keep console and redirect stdout to file.out)
– cat < file.in 2>&1 | tee file.out     (What does this do?)
```

File Redirection in Action

- ls -l /proc/self/fd (listing default file descriptors)

```
(84) thoth $ ls -l /proc/self/fd
total 0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 0 -> /dev/pts/0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 1 -> /dev/pts/0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 2 -> /dev/pts/0
lr-x------ 1 wahn UNKNOWN1 64 Sep 11 13:37 3 -> /proc/16970/fd
```

- ls -l /proc/self/fd > /tmp/ls.out (listing with redirect)

```
(85) thoth $ ls -l /proc/self/fd > /tmp/ls.out && cat /tmp/ls.out
total 0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:41 0 -> /dev/pts/0
l-wx------ 1 wahn UNKNOWN1 64 Sep 11 13:41 1 -> /tmp/ls.out
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:41 2 -> /dev/pts/0
lr-x------ 1 wahn UNKNOWN1 64 Sep 11 13:41 3 -> /proc/17025/fd
```

Tracing System Calls

- strace ./a.out
  (a.out is the text file dumper run on a Hello World main.c file)

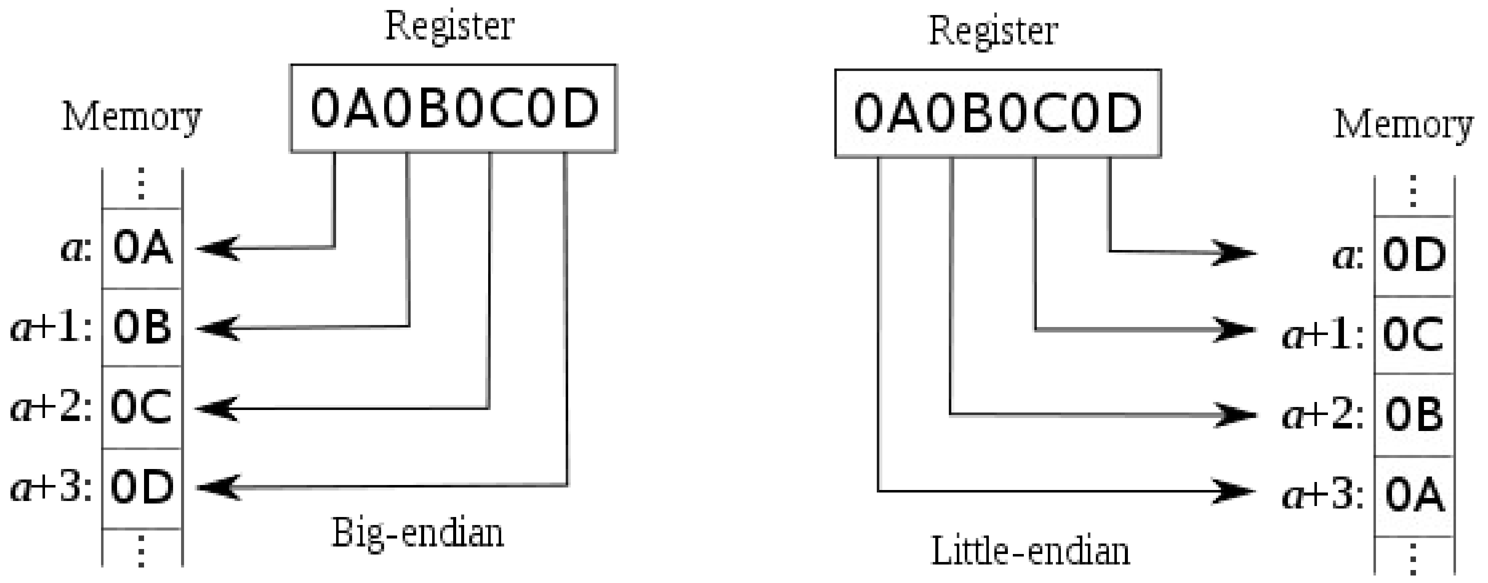
```
//TextFileDumper
#include <stdio.h>
int main()
{
  FILE *rfile;
  char buf[100];

  if((rfile=fopen("main.c","r"))==NULL)
    return 1;
  while(fgets(buf, 100, rfile)) {
    fputs(buf, stdout);
  }
  fclose(rfile);
  return 0;
}
```

```
(116) thoth $ strace ./a.out 2> /tmp/a.err && cat /tmp/a.err
open("main.c", O_RDONLY)               = 3
read(3, "#include <stdio.h>\nint main()\n{\n"..., 4096) = 75
write(1, "#include <stdio.h>\n", 19)    = 19
write(1, "int main()\n", 11)            = 11
write(1, "{\n", 2)                      = 2
write(1, "  printf(\"Hello world!\\n\");\n", 28) = 28
write(1, "  return 0;\n", 12)           = 12
write(1, "}\n", 2)                      = 2
write(1, "\n", 1)                       = 1
read(3, "", 4096)                       = 0
close(3)                                = 0
```

- Notice the difference in buffering for “read” and “write”

Big Endian vs. Little Endian



- When reading multi-byte primitive types (e.g. int), need to swap ordering when moving between big-endian and little-endian systems

Big Endian vs. Little Endian

Test program:

```
int main()
{
  int x = 1;
  char *y =
(char*)&x;

printf("%c\n",*y+48);
}
```

Part of ASCII table:

…	…
45	-
46	.
47	/
48	0
49	1
50	2
…	…

Try it on **thoth.cs.pitt.edu**!

```
              Assume 32-bit machine.

If it is little endian, the x in the memory
will be something like:

       higher memory
          ----->
    +----+----+----+----+
    |0x01|0x00|0x00|0x00|
    +----+----+----+----+
    A
    |
   &x

so (char*)(*x) == 1, and *y+48 == '1'


If it is big endian, it will be:
    +----+----+----+----+
    |0x00|0x00|0x00|0x01|
    +----+----+----+----+
    A
    |
   &x
so this one will be '0'.
```

File Formats

- Programmer's responsibility to design formats using delimiters and/or fixed sizes
- Text format (e.g. config files, log files)
  - + Human readable
  - -Numbers are represented as strings ➔ Bloated representation
  - -Usually formatted using delimiters (e.g. spaces, newlines) ➔ Impossible to do random access (must do sequential access)
- Binary format (e.g. image file, compressed file, database file)
  - + Compact representation
  - + Fixed length records ➔ Easy to do random access using lseek
  - - Needs translation for human comprehension