Using the C Stream I/O Functions

Ray Seyfarth

August 5, 2011

Why use the C stream I/O functions?

- The basic `open`, `lseek`, `read`, `write` and `close` system calls work
- The C stream I/O library buffers data in your process
- If you use `read` to read 1 billion bytes, there will be 1 billion system calls
- If you read 1 billion bytes using `getchar` there will be perhaps 1 system call per 8192 bytes
- Using `getchar` can be over 20 times as fast
- The operating system uses buffers too - you probably can't really 1 byte from a disk in one operation
- For small sized records, using the stream I/O functions will be faster
- You could implement your own specialized buffering system and do better than the C library, but you'll pay for the efficiency with time

Outline

Opening a file using fopen

```
FILE *fopen ( char *pathname, char *mode );
```

- pathname is the null-terminated name of the file to open
- mode is a string defining how you wish to use the file

r	read only mode
r+	read and write
w	write only, truncates or creates
w+	read and write, truncates or creates
a	write only, appends or creates
a+	read and write, appends or creates

- fopen returns an "opaque" FILE pointer (or NULL on error)
- A FILE is probably a struct with a file descriptor and a pointer to a buffer

Assembly code to open a file using fopen

```
        segment .data
name    db      "customers.dat",0
mode    db      "w+",0
fp      dq      0
        segment .text
        global  fopen
        lea     rdi, [name]
        lea     rsi, [mode]
        call    fopen
        mov     [fp], rax
```

Using `fscanf` and `fprintf`

```
int fscanf ( FILE *fp, char *format, ... );
int fprintf ( FILE *fp, char *format, ... );
```

- `scanf` is a function calling `fscanf` with `stdin` as the `FILE` pointer (more or less)
- The behavior of `fscanf` is like `scanf`, except that it reads from any file
- `printf` is a function calling `fprintf` with `stdout` as the `FILE` pointer
- The behavior of `fprintf` is like `printf`, except that it writes to any file

Using fgetc and fputc

```
int fgetc ( FILE *fp );
int fputc ( int c, FILE *fp );
int ungetc ( int c, FILE *fp );
```

- `fgetc` reads 1 character
- It returns `EOF` which is -1 on end of file or error
- `fputc` writes the character `c` to a file
- It returns `c` on success or EOF
- You can use `ungetc` to "push back" a character

Copying data using fgetc and fputc

```
more    mov     rdi, [ifp]   ; input file pointer
        call    fgetc
        test    eax, -1
        je      done
        mov     edi, eax
        mov     rsi, [ofp]   ; output file pointer
        call    fputc
        jmp     more
done:
```

Using fgets and fputs

```
char *fgets ( char *s, int size, FILE *fp );
int fputs ( char *s, FILE *fp );
```

- The parameter s is the array to read or write
- size is the number of characters in s
- fgets will read until it has read a new-line character, or it has filled s, or it hits end-of-file
- The new-line character will be placed in s
- No matter what fgets places a null byte (0) at the end of s
- fgets returns s on success or NULL on end-of-file or error
- fputs writes s to the file
- It returns EOF (-1) on error

Selectively copying lines of text

- The code below copies all lines of text which do not start with ';'

```
more     lea     rdi, [s]
         mov     esi, 200
         mov     rdx, [ifp]
         call    fgets
         test    rax, 0
         je      done
         mov     al, [s]
         test    al, ';'
         je      more
         lea     rdi, [s]
         mov     rsi, [ofp]
         call    fputs
         jmp     more
done:
```

Using fread and fwrite

```
int fread ( void *p, int size, int nelts, FILE *fp );
int fwrite ( void *p, int size, int nelts, FILE *fp );
```

- The parameter p is the address of a variable or array
- size is the size of each element to read or write
- nelts is the number of elements to read or write
- Both return the number or elements read or written
- The return value could be less than nelts or 0
- The code below writes 100 Customer objects

```
mov     rdi, [customers]  ; allocated array
mov     esi, Customer_size
mov     edx, 100
mov     rcx, [fp]
call    fwrite
```

Using fseek and ftell

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

- `fseek` sets the stream's position like `lseek`
- `ftell` returns the current position
- If `whence` is 0, `offset` is the byte position
- If `whence` is 1, `offset` is relative to the current position
- If `whence` is 2, `offset` is relative to the end of file

Function to write a customer record

```
write_customer:
.fp     equ     0
.c      equ     8
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     [rsp+.fp], rdi      ; file pointer
        mov     [rsp+.c], rsi       ; save Customer pointer
        mul     rdx, Customer_size  ; record number * size
        mov     rsi, rdx            ; 2nd parameter to ftell
        mov     rdx, 0              ; whence meaning position
        call    ftell
        mov     rdi, [rsp+.c]       ; pointer to start writing from
        mov     rsi, Customer_size  ; size of each element
        mov     rdx, 1              ; write 1 element
        mov     rcx, [rsp+.fp]      ; file pointer
        call    fwrite
        leave
        ret
```

Closing a file

```
int fclose(FILE *fp);
```

- The `FILE` object has a buffer and may contain data which has not been written
- Failure to close with `fclose` could result in lost data
- The system will close the underlying file, but will not call `fclose` automatically when your process ends