CS 201

The Unix System Interface

Gerson Robboy
Portland State University

System Calls

A system call is a request to the operating system for a service.

- typically made via a trap into the kernel

The UNIX system call interfaces are defined in section 2 of the man pages

- "man 2 intro" shows how they are actually called

These are the "real" UNIX services, everything else is

- An abstraction
- Built on top of them
- See unistd.h in Linux kernel tree for details

System calls for process control

We've already seen these:

fork

execl, execv, …

signal, sigaction, …

wait, waitpid

alarm

sleep

exit

File Manipulation

open – get a file descriptor for named file

close – free a file descriptor

read – read data from a file descriptor

write – write data to a file descriptor

stat – get file meta data

dup2 – duplicate a descriptor

lseek – change the current offset

creat – create/rewrite a named file

unlink – remove a directory entry

chmod: change permissions associated with file

fcntl: file control

mmap: map file contents

We've seen these in action already

We'll look at these

creat System Call

creat is used to create new files

- int creat(const char *path, mode_t mode);
- Equivalent to
  - open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)

If file already exists, truncates to zero

```
#include <fcntl.h>
    ...
    int fd;
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
    char *filename = "/tmp/file";
    ...
    fd = creat(filename, mode);
    ...
```

unlink System Call

Removes a directory entry

- int unlink(const char *path);

The unlink() function removes a link to a file.

- If path names a symbolic link, unlink() removes the symbolic link named by path

Exercise

Write a simple *"remove"* utility that removes a single file. The program must take one command argument, the name of the file to remove. The program must check to make sure there is exactly one argument, then unlink the file and check to see whether it was successfully unlinked.

chmod System Call

Change access permission mode of a file

- int chmod(const char *path, mode_t mode);
- int fchmod(int fildes, mode_t mode);

The effective user ID of the process must match the owner of the file (or be 0)

Mode is constructed by the bitwise OR operation of the access permission bits

- mode_t mode = S_ISUID | S_IRWXU | S_IRWXG;

chmod Example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>

extern int errno;
```

```
int
main(int argc, char *argv[])
{
        int rc;
        mode_t newMode = S_IRUSR | S_IRGRP;

        if (argc != 2) {
                fprintf(stderr, "Usage: %s <file>\n", argv[0]);
                return -1;
        }

        rc = chmod(argv[1], newMode);
        if (rc < 0) {
                fprintf(stderr, "Error: %s\n", strerror(errno));
                return -1;
        }

        return 0;
}
```

fcntl System Call

```
#include <fcntl.h>
int fcntl(int fd, int cmd, /* arg */ ...);  /* Number of args depends on
   cmd */
```

A whole collection of things you can do with files

- Make I/O synchronous or asynchronous
- File locking
- Receive signal when file is available for read/write
- Receive signal when file is modified by another process
- Keep file open across exec
- Anything else a particular device driver needs to do

F_DUPFD duplicates a file descriptor

- Find the lowest numbered available file descriptor greater than or equal to arg and make it be a copy of fd
- Compare to dup2 system call
  - dup2 returns the named descriptor and will close the named descriptor if it was in use before the call

fcntl example

You want to write tests for a new processor

- You want to do this using a Linux kernel
  - Ability to run many processes, do I/O, handle interrupts, etc
  - Really exercise the system, not just individual CPU tests in isolation
- There are things you need to do in supervisor mode
- You don't want to rewrite the Linux kernel or define a new system call interface

Write a device driver for a pseudo-device

- dynamically linked to the kernel at run time
- Open this "device" as a file, and you can do fcntl on it
- Individual supervisor mode operations can be fcntl commands

Write your tests as user-space programs

- Can be multi-process, multi-threaded, do I/O, do whatever you want with memory
- System calls to do small, specific operations

mmap System Call

- Establishes a mapping between a process's address space and a file or shared memory object
  - void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
  - pa = mmap(addr, len, prot, flags, fildes, off);
- Allows files to be treated as memory buffers
- Unmapping saves changes automatically
  - int munmap(void *start, size_t length);

mmap() maps a file into your memory space.

Instead of a read/write paradigm, you use a memory access paradigm.

What does this buy you?

Does it improve performance by avoiding the overhead of system calls?

Directory Manipulation

Directories, while technically files, are handled in a special manner

opendir

readdir

closedir

rewinddir

opendir System Call

Open a directory stream corresponding to the named directory

- DIR *opendir(const char *name)

Equivalent to the open system call for files

readdir

Read a directory entry

- int readdir(unsigned int fd, struct dirent *dirp, unsigned int count)

Reads the next directory entry to *dirp*. Count is ignored.

```
struct dirent
        {
           long d_ino;                      /* inode number */
           off_t d_off;                     /* offset to this dirent */
           unsigned short d_reclen;         /* length of this d_name */
           char d_name [NAME_MAX+1];        /* file name (null-terminated) */
        }
```

closedir and rewinddir

closedir closes the directory stream

- Equivalent to the close system call for files

rewinddir resets the position of the directory stream to the beginning

- Seeks to the first entry in the directory

seekdir sets the location of the directory stream

- Equivalent to the seek system call for files

What kind of program would use these?

Finding file permissions

Suppose you want to know if a certain path is an executable file

- /usr/home/oscar/a/b/c

How can you do it without traversing the path and reading all the individual directories?

Memory Allocation

Malloc, calloc, realloc, free, alloca

These are really library calls, not system calls.

The actual system call is *sbrk*

- Sets the limit on the heap you are allowed to use
- Essentially, gives us a hunk of virtual address space to use
- But as users, we don't normally call sbrk

Malloc, free and friends manage the memory

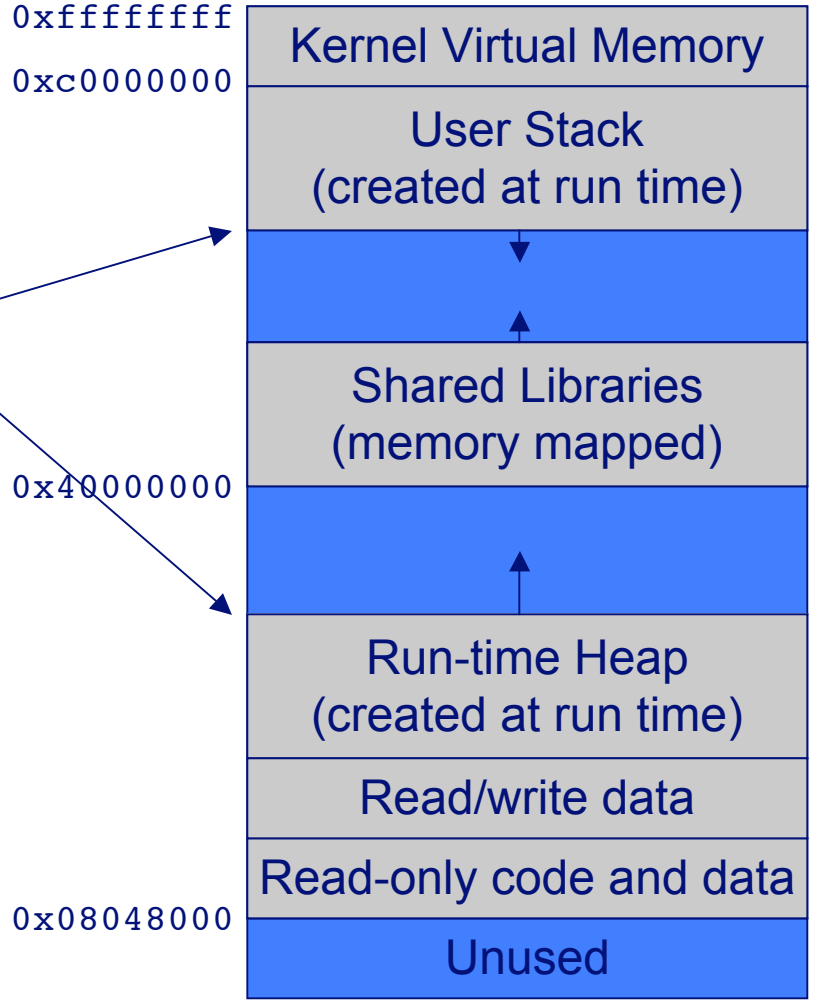
- No system calls
- Just touch the virtual address, and the VM system allocates the actual memory we need.

Memory Allocation

The main players

- **malloc**
- **calloc**
- **realloc**
- **free**

- **alloca**



A Note on Alignment

Alignment is important when accessing data

Memory allocators really allocate chunks of bytes, so misalignment is easy to do

Making sure that you allocate memory in proper "chunks" and casting the return to the "chunk size" is critical

- Always cast return value of malloc to data type you want to point to

alloca

alloca allocates memory in the stack frame of the caller

- The specific function (main, foobar, etc.)
- Alloca'ing beyond the current stack limit results in undefined behavior

There is no corresponding free call

- The memory is automatically freed when the function returns
- Warning: don't pass pointers to alloca'd memory back up the call stack!
  - Can safely pass down, though

malloc

- Returns a pointer to a block of at least size bytes – not zero filled! Allocated from the heap
  - void *malloc(size_t size);
  - int *ptr = (int *) malloc(value * sizeof(int));
- Be careful to allocate enough memory
  - Overrun on the space is undefined
  - Common error:
    - char *cptr = (char *) malloc (strlen(buf) * sizeof(char))
      - » strlen doesn't account for the NULL terminator
  - Fix:
    - char *cptr = (char *) malloc ((strlen(buf)+1) * sizeof(char))

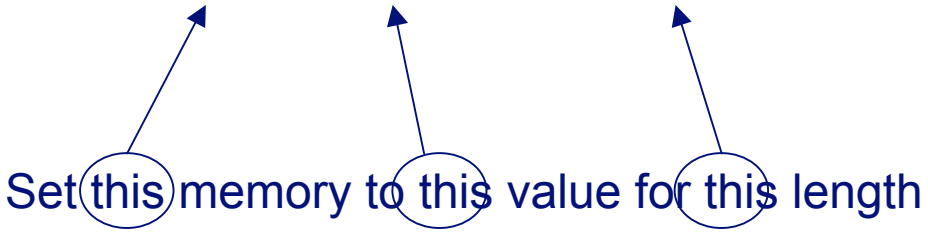
Zeroing Memory

Sometimes before you use memory returned by malloc, you want to zero it

- **Or maybe set it to a specific value**

memset sets a chunk of memory to a specific value

- **void *memset(void *s, int c, size_t n);**



calloc

void *calloc(size_t nelem, size_t elsize);

Will always zero memory that is returned

Essentially equivalent to malloc + memset

It takes time to zero the memory, so frequent calls to calloc can be more costly that just malloc

- A design consideration for your program

realloc

Allows modification of the specified block

- void *realloc(void *ptr, size_t size)

Special semantics

- Changes the size of the block pointed to by ptr to size bytes and returns a pointer to the (possibly moved) block.
- Contents unchanged up to the smaller of the new or old sizes
  - Implied copy when block is moved
- If ptr is NULL, behaves like malloc
- If pts in non-NULL and size is 0, behaves like free

free

Returns memory to the process for (possible) later reallocation

- Memory is not returned to the system until the process actually terminates

Memory is automatically free'd on process termination

- However, it is always a good idea to explicitly free any memory that is allocated from the heap
  - Helps to avoid memory leaks; aids program checkers

During program execution you should always free alloc'd memory when you don't need it anymore

- In this class it is considered an error not to do so
- But you don't have to free everything before exiting.
- Only during execution.

malloc vs. calloc

Sometimes performance considerations dictate which one to use

- Writing to memory is really bad for performance if you don't have to.

What if you are allocating a buffer and you are going to copy a string into it?

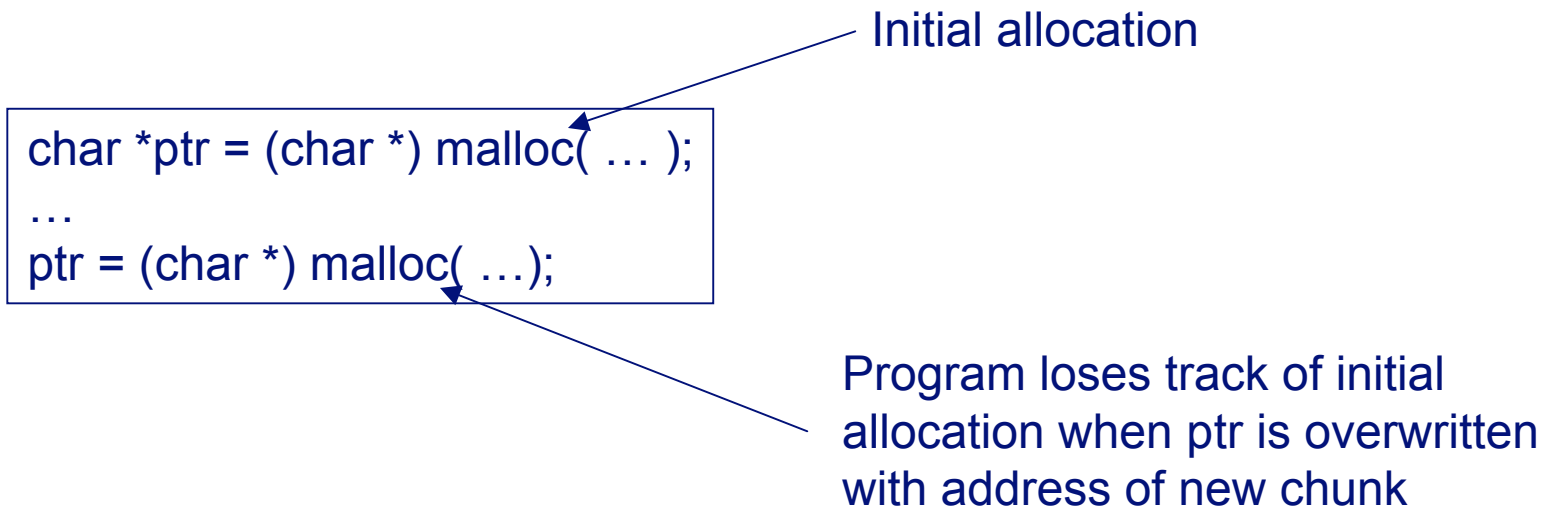
- What if the string is not as big as the buffer?

What if you are allocating a data structure containing pointers?

Memory Leak

A leak in a program's dynamic store allocation logic that causes it to fail to reclaim memory in the heap after it has finished using it, eventually causing the program to fail due to lack of memory

Simple example:



Summary of kernel-level I/O

The Unix kernel gives us files as an abstraction: a named stream of bytes

System calls for access: open, close, read, write, …

Higher level abstractions are at the application level …

… or libraries

Standard I/O Functions

The C standard library (`libc.a`) contains a collection of higher-level standard I/O functions

- **Documented in Appendix B of K&R.**

Examples of standard I/O functions:

- **Opening and closing files (`fopen` and `fclose`)**
- **Reading and writing bytes (`fread` and `fwrite`)**
- **Reading and writing text lines (`fgets` and `fputs`)**
- **Formatted reading and writing (`fscanf` and `fprintf`)**

Simple Buffered I/O

What does buffering buy us?

```
Simple Unbuffered I/O:
int getchar(void)
{
        char c;
        return (read(0, &c, 1) ==1) ?
                (unsigned char) c : EOF;
}
```

Note where EOF comes from.

```
Simple Buffered I/O:
int getchar(void)
{
        static char buf[BUFSIZE];
        static char *bufp = buf;
        static int n = 0;
        if (n == 0) {  // buffer is empty
                n = read(0, buf, sizeof(buf));
                bufp = buf;
        }
        return (- - n >= 0) ?
                (unsigned char) *bufp++ : EOF;
}
```

Standard I/O Streams

Standard I/O models open files as *streams*

- Abstraction for a file descriptor and a buffer in memory.

C programs begin life with three open streams (defined in `stdio.h`)

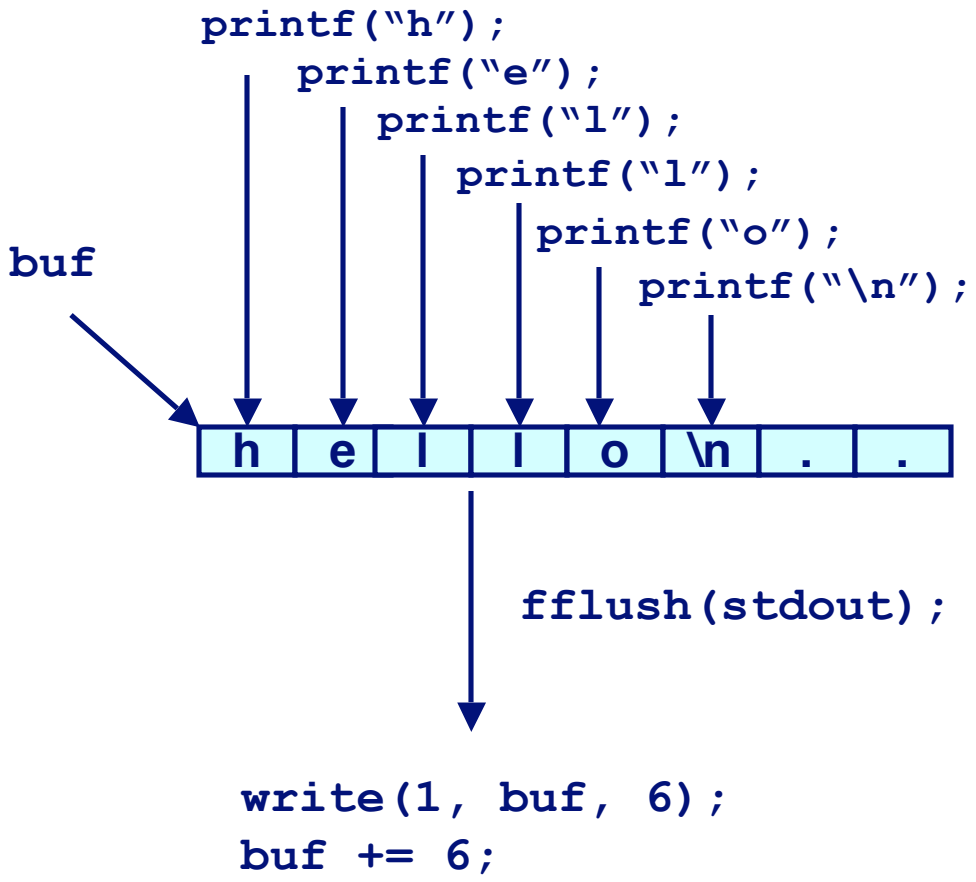
- `stdin` (standard input)
- `stdout` (standard output)
- `stderr` (standard error)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

Buffering in Standard I/O

Standard I/O functions use buffered I/O



Standard I/O Buffering in Action

You can see this buffering in action for yourself, using the Unix `strace` program:

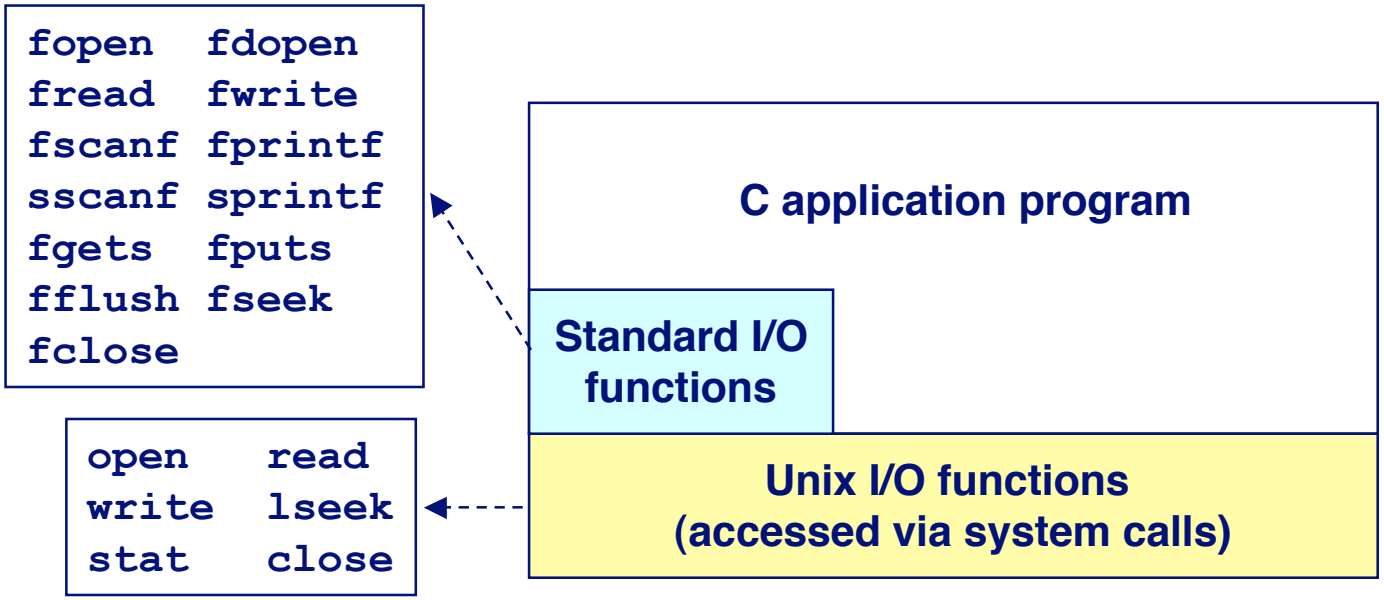
```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6...)                = 6
...
_exit(0)                                 = ?
```

Unix I/O vs. Standard I/O

Standard I/O is implemented using low-level Unix I/O.



Which ones should you use in your programs?

Exercise

A program needs to seek to random places in a file and write one small record in each place.

Should it use the stdio library or Unix system calls?

What about a compiler, which reads one character at a time sequentially through a file containing source code?

Pros and Cons of Unix I/O

Pros

- Unix I/O is the most general and lowest overhead form of I/O.
  - All other I/O packages are implemented using Unix I/O functions.
- Unix I/O provides functions for accessing file metadata.

Cons

- Dealing with short counts is tricky and error prone.
- Efficient reading of text lines requires some form of buffering, also tricky and error prone.
- Both of these issues are addressed by the standard I/O package.

Pros and Cons of Standard I/O

Pros:

- Buffering increases efficiency by decreasing the number of `read` and `write` system calls.
  - Usually
  - Unless you're transferring blocks of data >= the buffer size
  - Or unless you're seeking randomly, not using the buffer
- Higher level of abstraction makes it easier to use
  - Usually
  - Example: Short counts are handled automatically.
    - » There's a concept of EOF
  - More likely to avoid bugs
- Portability
  - De-facto part of the C language, independent of O. S.

Pros and Cons of Standard I/O

Cons:

- "Usually" more efficient and easier to use
  - For big blocks of data, an extra layer of library calls reduces efficiency
  - For many seeks, it's even worse: Fill a buffer and don't use it
- Provides no function for accessing file metadata
- Not appropriate for input and output on network sockets
- There are poorly documented restrictions on streams that interact badly with restrictions on sockets

Pros and Cons of Standard I/O (cont)

Restrictions on streams:

- Restriction 1: input function cannot follow output function without intervening call to `fflush`, `fseek`, `fsetpos`, or `rewind`.
  - Latter three functions all use `lseek` to change file position.
- Restriction 2: output function cannot follow an input function with intervening call to `fseek`, `fsetpos`, or `rewind`.

Restriction on sockets:

- You are not allowed to change the file position of a socket.

Choosing I/O Functions

Use the highest-level I/O functions you can.

- Many C programmers are able to do all of their work using the standard I/O functions.

When to use standard I/O?

- Usually.
- Especially when working with disk or terminal files.
- Especially when you're doing something that standard I/O does anyway.
  - Don't re-invent library functions yourself!

When to use raw Unix I/O

- When you need to fetch file metadata.
- In rare cases when you need absolute highest performance.

Question

Suppose a program generates huge arrays of data in memory, multi-megabytes, and periodically the program writes a whole array to a file. Each array is written in its entirety, and just once.

Is it more efficient to use *write* system calls or the stdio library to write the array?

Question

The operating system kernel has a pool of buffers for disk blocks. If you read a file one character at a time, the kernel does not go to the disk for each character, because it has the whole disk block in a buffer.

When a program uses *fread*, the library function has its own buffer in user space. So essentially, data is being copied from the buffer pool to *fread*'s buffer, and then to the application's own buffer. So the data is copied at least twice.

In that case, is it more efficient to use *fread* to read a character at a time, than to use *read*?

For Further Information

W. Richard Stevens, Advanced Programming in the Unix Environment, Addison Wesley, 1993.

Brian W. Kernighan and Rob Pike, The Unix Programming Environment, Prentice-Hall, 1984