Input/Output

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>
CPU

Memory

I/O

- disk
- terminal
- shared memory
- printer
- network

…
- vast number of different mechanisms
- but overlapping requirements:
  - read/write operations
  - metadata (e.g., name, position)
  - robustness, thread-safety
programming concerns:
- how are I/O endpoints represented?
- how to perform I/O?
- how to perform I/O efficiently?
focus on *Unix system-level I/O*
§Unix I/O & Filesystem Architecture Brief
2 general classes of I/O devices:

- \textit{block}: accessed in fixed-size chunks; support for seeking & random access

- \textit{character}: char-by-char streaming access; no seeking / random access
block device

char device

... | ...
2 general classes of I/O devices:
- block: e.g., disk, memory
- character: e.g., network, mouse
the filesystem acts as a namespace for data residing on different devices
regular files consist of ASCII or binary data, stored on a block device
special files may represent directories, in-memory structures, sockets, or raw devices
i.e., “files” are a general OS abstraction for arbitrary data objects!
each file has a unique **inode** data structure in the filesystem, tracking:

- ownership & permissions
- size, type, and location
- number of **links**
a given i-node can be referenced using one or more fully qualified path(s), e.g.,

- /home/lee/.emacs
- /proc/sys/kernel/version
- /dev/tty
“/home/lee/.emacs”

OS’s filesystem module

inode table
"/home/lee/.emacs"

OS’s filesystem module

* filename → inode association = link

inode table

locate & load inode structure
every currently open file has a *single* in-memory inode, aka. “vnode”

**vnode**
- ownership
- permissions
- size & location
each open file is also tracked by the kernel using an *open file description* structure

**open file desc**
- position
- access mode

**vnode**
- ownership
- permissions
- size & location
can have *multiple open file descriptions* referencing a *single vnode* (e.g., to track separate read/write positions)

**open file desc**
- position
- access mode

**vnode**
- ownership
- permissions
- size & location
for each process, the kernel maintains a table of pointers to its open file structures
all these structures reside in *kernel memory* (off-limits to user processes)!
to let a process reference an open file, the kernel returns *an index into* the table

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFD</td>
<td>vnode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OFD</td>
<td>vnode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OFD</td>
<td>vnode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OFD</td>
<td>vnode</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*protected memory*
call this a **file descriptor** (FD)
by convention, processes ...

- read from FD 0 for *standard input*
- write to FD 1 for *standard output*
- write to FD 2 for *standard error*
after opening a file, *all file operations are performed using file descriptors!"
kernel space

per process

0
1
2
0
1
2
0
1
2

system-wide

OFD

OFD

OFD

vnode

vnode

vnode

terminal

network

disk
FDs obscure kernel I/O & FS implementation details from the user, and enable an elegant, abstract I/O API
§ System-level I/O API
int open ( const char *path, int oflag, ... );
int fstat ( int fd, struct stat *buf );
int dup ( int fd );
int dup2 ( int fd1, int fd2 );
int close ( int fd );
off_t lseek ( int fd, off_t offset, int whence );
ssize_t read ( int fd, void *buf, size_t nbytes );
ssize_t write ( int fd, const void *buf, size_t nbytes );
int open(const char *path, int oflag, ...);

- loads *vnode* for file at *path* (if not already loaded)

- creates & inits a new OFD

- returns a *FD* referring to the new OFD
int open(const char *path, int oflag, ...);

- oflag is an or-ing of O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_TRUNC, etc.

- if O_CREAT, must specify access permissions of new file ("rwx" flags)
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
```
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
```

```
$ ls -l foo.txt
-rw-r--r-- 1 lee staff 5 Feb 15 18:23 foo.txt
```

rwx flags from (octal) 0644
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);

struct stat stat;

/* query file metadata */
fstat(fd1, &stat);

printf("Inode # : %lu\n", stat.st_ino);
printf("Size    : %lu\n", stat.st_size);
printf("Links   : %lu\n", stat.st_nlink);

Inode # : 19603149
Size    : 0
Links   : 1
† a process inherits its parent’s open files across a `fork`, and `retains them post-exec`!
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
fork();
```

**Diagram:**

```
parent
3
4

child
3
4

OFD    vnode    empty file

i.e., parent and child share position and file access mode
```

**Text:**

```
per-process   cross-system
```

---

**IIT College of Science**

**ILLINOIS INSTITUTE OF TECHNOLOGY**
sharing an OFD can be very handy — e.g., for coordinating output to terminal
can also explicitly “share” position from separate FDs using `dup` system calls
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = dup(fd1);
```

i.e., reading/writing FD 4 is equivalent to doing so with FD 3
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
dup2(fd1, 2); /* second arg is "destination" fd */
```

i.e., reading/writing FD 2 (stderr) is equivalent to doing so with FD 3.

(original FD is automatically closed)
int close(int fd);

- delete OFD pointer in file table for fd
- if the OFD has no referring FDs (in any process), deallocate it
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
```
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
close(fd1);
```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
close(fd1);
close(fd2);
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = dup(fd1);

```
0
1
2
3
4
```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = dup(fd1);
close(fd1);
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = dup(fd1);
close(fd1);
close(fd2);
application: *input/output redirection*

- leverage FD usage conventions

- \(0 = \text{stdin}, 1 = \text{stdout}, 2 = \text{stderr}\)

- recall: FD says nothing about the actual file/device it refers to!
Demo:

Shell I/O redirection
int main(int argc, char *argv[]) {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    dup2(fd, 1);
    printf("Arg: %s\n", argv[1]);
}

$ ./a.out hello!
$ ls -l foo.txt
-rw-r--r-- 1 lee staff 12 Feb 19 20:36 foo.txt
$ cat foo.txt
Arg: hello!
int main(int argc, char *argv[]) {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    dup2(fd, 1);
    printf("Arg: %s\n", argv[1]); /* printf prints to stdout */
}

(by default: terminal)
int main(int argc, char *argv[]) {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    dup2(fd, 1);
    printf("Arg: %s\n", argv[1]); /* printf prints to stdout */
}

(output)
```c
int main() {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    if (fork() == 0) {
        dup2(fd, 1);
        execlp("echo", "echo", "hello!", NULL);
    }
    close(fd);
}
```

```
$ ./a.out
$ cat foo.txt
hello!
```
```c
int main() {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    if (fork() == 0) {
        dup2(fd, 1);
        execlp("echo", "echo", "hello!", NULL);
    }
    close(fd);
}
```

illustrates a powerful technique that requires separating **fork & exec**

- original program sets up *new process environment* before **exec-ing**
off_t lseek(int fd,
    off_t offset, int whence);
    
    - changes file position relative to “whence” 
    - if whence = SEEK_SET, pos = offset 
    - if whence = SEEK_CUR, pos += offset 
    - returns current position
```c
int main() {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    if (fork() == 0) {
        dup2(fd, 1);
        execlp("echo", "echo", "hello world!", NULL);
    }
    wait(NULL);
    lseek(fd, -7, SEEK_CUR);
    write(fd, "planet!\n", 8);
    printf("Curr pos: %ld\n", lseek(fd, 0, SEEK_CUR));
}
```

```
$ ./a.out
Curr pos: 14
$ cat foo.txt
hello planet!
```
† not all files are `lseek`-able!

e.g., character special files
ssize_t read(int fd, void *buf, size_t nbytes);

- reads up to nbytes bytes from open file at fd into buf
  - by default, block for at least 1 byte
  - returns # bytes read (or -1 for error)
ssize_t **write**(int *fd*, const void **buf**, size_t **nbytes**);

- writes up to nbytes bytes into open file at *fd* from *buf*
  - by default, *block* for at least 1 byte
- returns # bytes written (or -1 for error)
“up to nbytes bytes”

i.e., *short counts* can occur

— process asks OS to write $k$ bytes, but only $l < k$ bytes are actually written
why?
reads:

- EOF, unreadable FD, “slow” file, interrupt, etc.

writes:

- out of space, unwritable FD, “slow” file, interrupt, etc.
read/write are the lowest level I/O calls
— kernel objective is to support maximum performance & minimum latency
e.g., if reading from slow network, return to process asap and allow it to decide to read again or do something else
(but usually, short counts are a royal pain)
ssize_t robust_read(int fd, void *buf, size_t n) {
    size_t nleft = n;
    ssize_t nread;
    char *p = buf;

    while (nleft > 0) {
        if ((nread = read(fd, p, nleft)) < 0)
            return -1; /* error in read */
        else if (nread == 0)
            break; /* read returns 0 on EOF */
        nleft -= nread;
        p += nread;
    }

    return (n - nleft);
}
(yuck)

good news: short counts only occur on EOF for reads on regular files
but there’s another concern…
char buf[10];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);
read(fd, buf, 2);  buf[2] = 0;
x = atoi(buf);

read(fd, buf, 2);  buf[2] = 0;
y = atoi(buf);

read(fd, buf, 2);  buf[2] = 0;
z = atoi(buf);

printf("%d %d %d", x, y, z);
char buf[10];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);

read(fd, buf, 2); buf[2] = 0;
x = atoi(buf);

read(fd, buf, 2); buf[2] = 0;
y = atoi(buf);

read(fd, buf, 2); buf[2] = 0;
z = atoi(buf);

printf("%d %d %d", x, y, z);

one syscall per integer read = inefficient!!!
fd = open("data.txt", O_RDONLY);

read(fd, buf, 2);  buf[2] = 0;  x = atoi(buf);

read(fd, buf, 2);  buf[2] = 0;  y = atoi(buf);

read(fd, buf, 2);  buf[2] = 0;  z = atoi(buf);

printf("%d %d %d", x, y, z);

$ strace ./a.out
execve("./a.out", ["./a.out"], [/* 67 vars */]) = 0
...
open("data.txt", O_RDONLY) = 3
read(3, "10", 2) = 2
read(3, "20", 2) = 2
read(3, "30", 2) = 2
write(1, "10 20 30", 8) = 8
...
solution: buffering
step 1: read more bytes than we need into a separate *backing buffer*

user

kernel
step 1: read more bytes than we need into a separate *backing buffer*
char buf[10], bbuf[80];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);
read(fd, bbuf, sizeof(bbuf));

102030
step 2: avoid syscalls and process future “reads” from that buffer

user

-----------------------------------------------

kernel
step 2: avoid syscalls and process future “reads” from that buffer

user

-----------------------------------------------

kernel
step 2: avoid syscalls and process future “reads” from that buffer

user

kernel
char buf[10], bbuf[80];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);
read(fd, bbuf, sizeof(bbuf));

buf[2] = 0;
memcpy(buf, bbuf, 2);
x = atoi(buf);
char buf[10], bbuf[80];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);
read(fd, bbuf, sizeof(bbuf));

buf[2] = 0;
memcpy(buf, bbuf, 2);
x = atoi(buf);

memcpy(buf, bbuf+2, 2);
y = atoi(buf);
```c
char buf[10], bbuf[80];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);
read(fd, bbuf, sizeof(bbuf));

buf[2] = 0;
memcpy(buf, bbuf, 2);
x = atoi(buf);

memcpy(buf, bbuf+2, 2);
y = atoi(buf);

memcpy(buf, bbuf+4, 2);
z = atoi(buf);
```
fd = open("data.txt", O_RDONLY);
read(fd, bbuf, sizeof(bbuf));

buf[2] = 0;
memcpy(buf, bbuf, 2);
x = atoi(buf);

memcpy(buf, bbuf+2, 2);
y = atoi(buf);

memcpy(buf, bbuf+4, 2);
z = atoi(buf);

$ strace ./a.out
execve("./a.out", ["./a.out"], [/* 67 vars */]) = 0
...
open("data.txt", O_RDONLY) = 3
read(3, "102030\n", 80) = 7
write(1, "10 20 30", 8) = 8
...
to generalize, bundle together:

(1) FD
(2) backing buffer
(3) num unused bytes
(4) pointer to next byte
typedef struct {
    int    fd;    /* (1) wrapped FD */
    char   buf[100]; /* (2) backing buffer */
    int    count; /* (3) num unused bytes */
    char  *nextp; /* (4) pointer to next byte */
} bufio_t;

void bufio_init(bufio_t *bp, int fd) {
    bp->fd    = fd;
    bp->count = 0;
    bp->nextp = bp->buf;
}
ssize_t bufio_read(bufio_t *bp, char *buf, size_t n) {
    int ncpy;

    /* fill backing buffer if empty */
    if (bp->count <= 0) {
        bp->count = read(bp->fd, bp->buf, sizeof(bp->buf));
        if (bp->count <= 0)
            return bp->count; /* EOF or read error */
        else
            bp->nextp = bp->buf; /* re-init buf position */
    }

    /* copy from backing buffer to user buffer */
    ncpy = (bp->count < n)? bp->count : n;
    memcpy(buf, bp->nextp, ncpy);
    bp->nextp += ncpy;
    bp->count -= ncpy;

    return ncpy;
}

```c
char buf[10];
int fd, x, y, z;
bufio_t bbuf;

fd = open("data.txt", O_RDONLY);
bufio_init(&bbuf, fd);

buf[2] = 0;
bufio_read(&bbuf, buf, 2);
x = atoi(buf);

bufio_read(&bbuf, buf, 2);
y = atoi(buf);

bufio_read(&bbuf, buf, 2);
z = atoi(buf);
```
open is now a distraction... we never use the FD directly (except to initialize buffer)
next step: hide syscalls from user — wrap `open` together with buffer initialization
bufio_t *buf_open(const char *path) {
    bufio_t *buf = malloc(sizeof(bufio_t));
    int fd = open(path, O_RDWR);
    bufio_init(buf, fd);
    return buf;
}

int main() {
    bufio_t *bbuf = buf_open("data.txt");
    char buf[10];
    int x, y, z;

    bufio_read(bbuf, buf, 2);
    ...
}
Stop!

<stdio.h> does all this for us!
fclose  fdopen  feof  ferror  ffflush  fgetc  fgets  fopen  fprintf  fputs
fread  freopen  fscanf  fseek  fsetpos  fwrite
ferror  printf  putc  putchar  puts
remove  rewind  scanf  sprintf
sscanf  strlen  strerror  tmpfile  ungetc
vfprintf  vfprintf  vscanf ...
... all use buffered I/O
stdio functions operate on *stream* objects
i.e., buffered wrappers on FDs
FILE* fopen ( const char *filename, const char *mode );
FILE* fdopen ( int fd, const char *mode );
int fclose ( FILE *stream );
int fseek ( FILE *stream, long offset, int whence );
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 );
int fprintf ( FILE *stream, const char *format, ... );
int fscanf ( FILE *stream, const char *format, ... );
char* fgets ( char *str, int size, FILE *stream );
```c
int x, y, z;
FILE *infile = fopen("data.txt", "r");

fscanf(infile, "%2d", &x);
fscanf(infile, "%2d", &y);
fscanf(infile, "%2d", &z);

printf("%d %d %d", x, y, z);

fclose(infile); /* or memory leak! */
```

```
$ strace ./a.out
execve("./a.out", ["./a.out"], [/* 67 vars */]) = 0
...
open("data.txt", O_RDONLY) = 3
read(3, "102030\n", 4096) = 7
write(1, "10 20 30", 8) = 8
close(3) = 0
...
```
printf("h");
printf("e");
printf("l");
printf("l");
printf("o");

$ strace ./a.out
...
write(1, "hello", 5) = 5
...

(writes are buffered too!)
stream buffer can *absorb* multiple writes before being flushed to underlying file
flush happens on:
- buffer being filled
- (normal) process termination
- newline, in a line-buffered stream
- explicitly, with fflush
```c
int main() {
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    fork();
}
```

```
$ ./a.out
hellohello
```

```
@#$%^&*!!!
```
```c
int n, fd = open("fox.txt", O_RDONLY);
char buf[10];

n = read(fd, buf, sizeof(buf));
write(1, buf, n);
if (fork() == 0) {
    n = read(fd, buf, sizeof(buf));
    write(1, buf, n);
    exit(0);
}
wait(NULL);
n = read(fd, buf, sizeof(buf));
write(1, buf, n);
```

```bash
$ ./a.out
The quick brown fox jumps over
```
```c
int n;
FILE *stream = fopen("fox.txt", "r");
char buf[10];

n = fread(buf, 1, sizeof(buf), stream);
write(1, buf, n);
if (fork() == 0) {
    n = fread(buf, 1, sizeof(buf), stream);
    write(1, buf, n);
    exit(0);
}
wait(NULL);
n = fread(buf, 1, sizeof(buf), stream);
write(1, buf, n);
```

```
$ ./a.out
The quick brown fox brown fox

@#$%^&*!!!
```
things gets even more confusing when we perform both input & output
```c
int fd = open("fox.txt", O_RDWR);
char buf[10];

/* output followed by input */
write(fd, "a playful ", 10);
read(fd, buf, sizeof(buf));

write(1, buf, sizeof(buf));
```

```
$ ./a.out
brown fox
```

*fox.txt*

```
a playful brown fox jumps over the lazy dog
```
```c
int fd = open("fox.txt", O_RDWR);
FILE *stream = fdopen(fd, "r+");
char buf[10];

/* output followed by input */
fwrite("a playful ", 1, 10, stream);
read(fd, buf, sizeof(buf));

write(1, buf, sizeof(buf));
```

```
$ ./a.out
the quick
@#$%^&*!!!
```
```c
int fd = open("fox.txt", O_RDWR);
char buf[10];

/* input followed by output */
read(fd, buf, sizeof(buf));
write(fd, "green cat ", 10);
write(1, buf, sizeof(buf));
```

```
$ ./a.out
the quick green cat jumps over the lazy dog
```

`fox.txt`

```
the quick fox jumps over the lazy dog
```
FILE *stream = fopen("fox.txt", "r+");  
char buf[10];

/* input followed by output */
fread(buf, 1, sizeof(buf), stream);
fwrite("green cat ", 1, 10, stream);
write(1, buf, sizeof(buf));

$ ./a.out
the quick

@#$%^&&*!!!
When a file is opened with update mode ..., both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the `fflush` function or to a file positioning function ..., and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end- of-file.

*ISO C99 standard, 7.19.5.3 (par 6)*
input shall not be directly followed by output without an intervening call to a file positioning function

but not all files support “file positioning functions”! (e.g., no seeks on character devices)
lessons:

- buffered stdio functions help minimize system overhead & simplify I/O

- use whenever possible!
lessons:

- but need to beware of glitches
- don’t mix buffered & unbuffered I/O
- and not appropriate for some devices (e.g., network)
  - use low-level, robust I/O for these