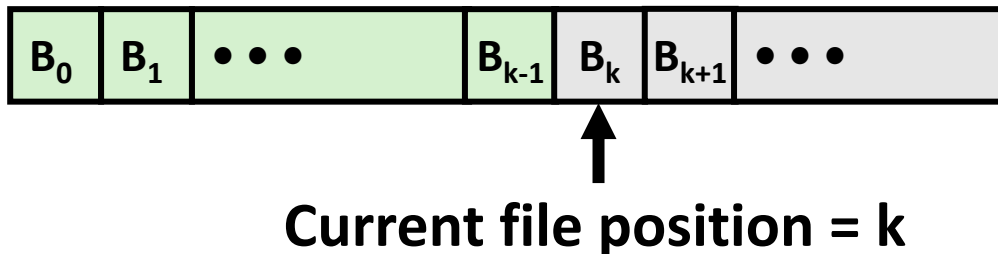


System-Level I/O:

Unix I/O: System calls

Unix I/O Syscalls

- Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O*:
 - Opening and closing files
 - `open()` and `close()`
 - Reading and writing a file
 - `read()` and `write()`
 - Changing the *current file position* (seek)
 - indicates next offset into file to read or write
 - `lseek()`



Opening Files

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;    /* file descriptor */  
  
if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {  
    perror("open");  
    exit(1);  
}
```

- Returns a small identifying integer *file descriptor*
 - `fd == -1` indicates that an error occurred

Opening Files

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;    /* file descriptor */  
  
if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {  
    perror("open");  
    exit(1);  
}
```

- Returns a small identifying integer *file descriptor*
 - `fd == -1` indicates that an error occurred
- Each process created by a Linux shell begins life with three open files associated with a terminal:
 - 0: standard input (stdin)
 - 1: standard output (stdout)
 - 2: standard error (stderr)

Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;      /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
    perror("close");
    exit(1);
}
```

Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;      /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
    perror("close");
    exit(1);
}
```

- Closing an already closed file is a recipe for disaster in threaded programs (more on this later)
- Moral: Always check return codes, even for seemingly benign functions such as `close()`

Reading Files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];  
int fd;          /* file descriptor */  
int nbytes;      /* number of bytes read */  
  
/* Open file fd ... */  
/* Then read up to 512 bytes from file fd */  
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {  
    perror("read");  
    exit(1);  
}
```

Reading Files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];
int fd;          /* file descriptor */
int nbytes;      /* number of bytes read */

/* Open file fd ... */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```

- Returns number of bytes read from file `fd` into `buf`
 - Return type `ssize_t` is signed integer
 - `nbytes < 0` indicates that an error occurred
 - **Short counts** (`nbytes < sizeof(buf)`) are possible and are not errors!

Writing Files

- Writing a file copies bytes from memory to the current file position, and then updates current file position

```
char buf[512];
int fd;          /* file descriptor */
int nbytes;      /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```

Writing Files

- Writing a file copies bytes from memory to the current file position, and then updates current file position

```
char buf[512];
int fd;          /* file descriptor */
int nbytes;      /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```

- Returns number of bytes written from `buf` to file `fd`
 - `nbytes < 0` indicates that an error occurred
 - As with reads, short counts are possible and are not errors!

Simple Unix I/O example

- Copying stdin to stdout, one byte at a time

```
#include "csapp.h"

int main(void)
{
    char c;

    while (Read(STDIN_FILENO, &c, 1) != 0)
        Write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

Accessing Directories

■ Overly complicated syscalls

- `getdents64(2)`: “These are not the interfaces you are interested in.”

Accessing Directories

■ Overly complicated syscalls

- `getdents64(2)`: “These are not the interfaces you are interested in.”

■ Only use `libc`’s `opendir`, `readdir`

- `dirent` structure contains information about a directory entry
- `DIR` structure contains info about `dir` while stepping through its entries

```
#include <sys/types.h>
#include <dirent.h>
void print_files_in (const char *dir_name) {
    DIR *directory;
    struct dirent *de;
    if (!(directory = opendir(dir_name)))
        error("Failed to open directory");
    while (0 != (de = readdir(directory))) {
        printf("Found file: %s\n", de->d_name);
    }
    closedir(directory);
}
```

Accessing Directories

```
void print_files_in (const char *dir_name);  
  
int main (int argc, char **argv) {  
    if (argc) print_files_in (argv[1]);  
}
```

```
$ gcc main.c print_files.c -o filesin  
$ mkdir empty-dir  
$ ./filesin empty-dir  
Found file: .  
Found file: ..  
$ ./filesin /proc/self/fd  
Found file: .  
Found file: ..  
Found file: 0  
Found file: 1  
Found file: 2  
Found file: 3
```

File Metadata

- **Metadata** is data about data, in this case file data
- Per-file metadata maintained by kernel
 - accessed by users with the `stat` and `fstat` functions

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t      st_dev;      /* Device */
    ino_t      st_ino;      /* inode */
    mode_t     st_mode;     /* Protection and file type */
    nlink_t    st_nlink;    /* Number of hard links */
    uid_t      st_uid;      /* User ID of owner */
    gid_t      st_gid;      /* Group ID of owner */
    dev_t      st_rdev;     /* Device type (if inode device) */
    off_t      st_size;     /* Total size, in bytes */
    unsigned long st_blksize; /* Blocksize for filesystem I/O */
    unsigned long st_blocks; /* Number of blocks allocated */
    time_t     st_atime;    /* Time of last access */
    time_t     st_mtime;    /* Time of last modification */
    time_t     st_ctime;    /* Time of last change */
};
```

Example of Accessing File Metadata

```
int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode))      /* Determine file type */
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* Check read access */
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

statcheck.c

Example of Accessing File Metadata

```
int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode)) /* Determine file type */
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* Check read access */
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

```
linux> ./statcheck statcheck.c
type: regular, read: yes
linux> chmod 000 statcheck.c
linux> ./statcheck statcheck.c
type: regular, read: no
linux> ./statcheck ..
type: directory, read: yes
```

statcheck.c