Input & Output

1. I/O Management I

1.1 CSCI 330

1.2 Unit Overview

Unit Overview

- System calls
- File I/O
  - open, creat
  - read, write
  - close
- more:
  - unlink, stat, chmod, dup

1.3 UNIX System Call

UNIX System Call

- a system call is how a program requests a service from the operating system, i.e. UNIX kernel
- system call executes code in the kernel and makes direct use of facilities provided by the kernel

versus:

- library function is linked to executable, becomes part of the executable

1.4 System Call Categories

System Call Categories

- file management
  - create/delete file, open/close, read/write, get/set attributes
- process control
  - create/terminate process, wait/signal event, allocate/free memory
- communication
  - create/delete connection, send/receive messages, remote devices
- information management
  - get/set system data and attributes, e.g. time
- device management
  - attach/request/release/detach device, read/write/position

1.5 System Call Invocation

System Call Invocation

- declare system call via appropriate C header file
  - read man page carefully (section 2 of manual)
- prepare parameters using basic C data types
  - no C++ classes, i.e. string
- prepare suitable return value variable

<u>call like any other function</u>

1.6 File Management

File Management

open	open a file
read	read data from a file
write	write data to a file
close	close a file
creat	make a new file

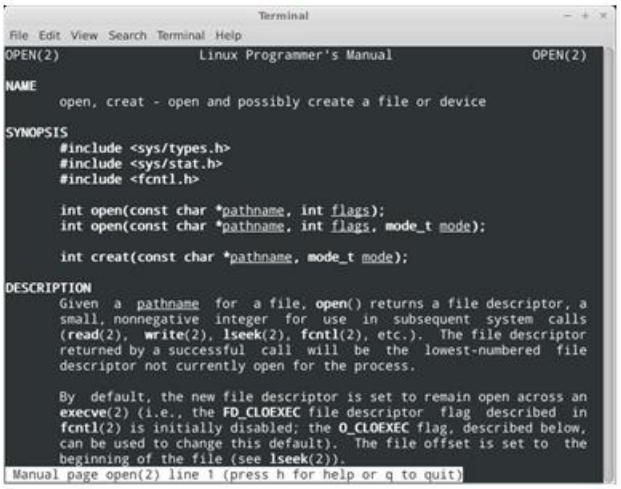
- next:

unlink	remove file
stat	get file information
chmod	change permissions
dup	duplicate file descriptor

- all calls share file descriptor, i.e. number, to identify file

1.7 System Call: open

System Call: open



1.8 System Call: open

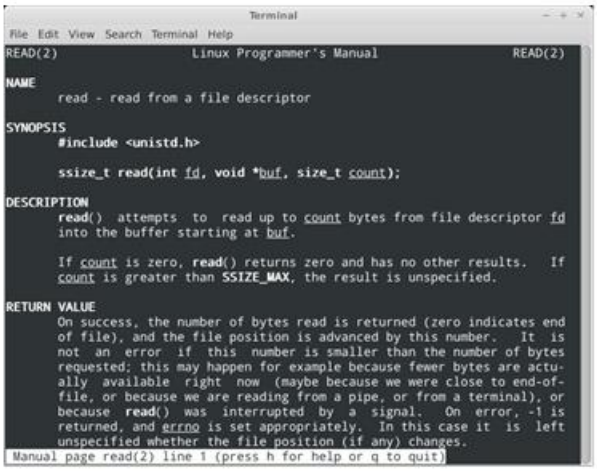
System Call: open

```
int open(const char* pathname, int flags)
```

- opens file specified as **pathname** for access
- **flags** determine access type
  - **O_RDONLY** read only
  - **O_WRONLY** write only
  - **O_RDWR** read and write
- returns file descriptor, to be used in read/write/close
- returns -1 on error

1.9 System Call: read

System Call: read



1.10 System Call: read

System Call: read

```
ssize_t read(int fd, void *buf, size_t count)
```

- attempts to read **`count`** bytes from file descriptor **`fd`** into the buffer starting at **`buf`**
  - **`ssize_t`** and **`size_t`** are system specific integer types
- returns the number of bytes read
  - maybe smaller than count, zero indicates end of file
  - file position is advanced by this number
- returns -1 on error

1.11 System Call: close

System Call: close

```
int close(int fd)
```

- closes file specified by **`fd`** file descriptor
  - makes file descriptor available
- returns zero on success

1.12 System Call: read example

System Call: read example

read.cxx - /home/student/Desktop/Unit 12 - IO Mgmt I - Geany

File Edit Search View Document Project Build Tools Help

read.cxx

```
18  int main() {
19      int fd, count;
20      char buffer[1024], filename[] = "example.txt";
21      // open file
22      fd = open(filename, O_RDONLY);
23      if (fd == -1) {
24          perror(filename);
25          exit(EXIT_FAILURE);
26      }
27      // read from file
28      count = read(fd, buffer, 1024);
29      if (count == -1) {
30          perror(filename);
31          exit(EXIT_FAILURE);
32      }
33      cout << "read " << count << " bytes from file\n";
34      // close  file
35      close(fd);
36      return 0;
37  }
```

line: 1 / 39 col: 0 sel: 0 INS TAB mode: Unix (LF) encoding: UTF-8 filetype: C++ ...

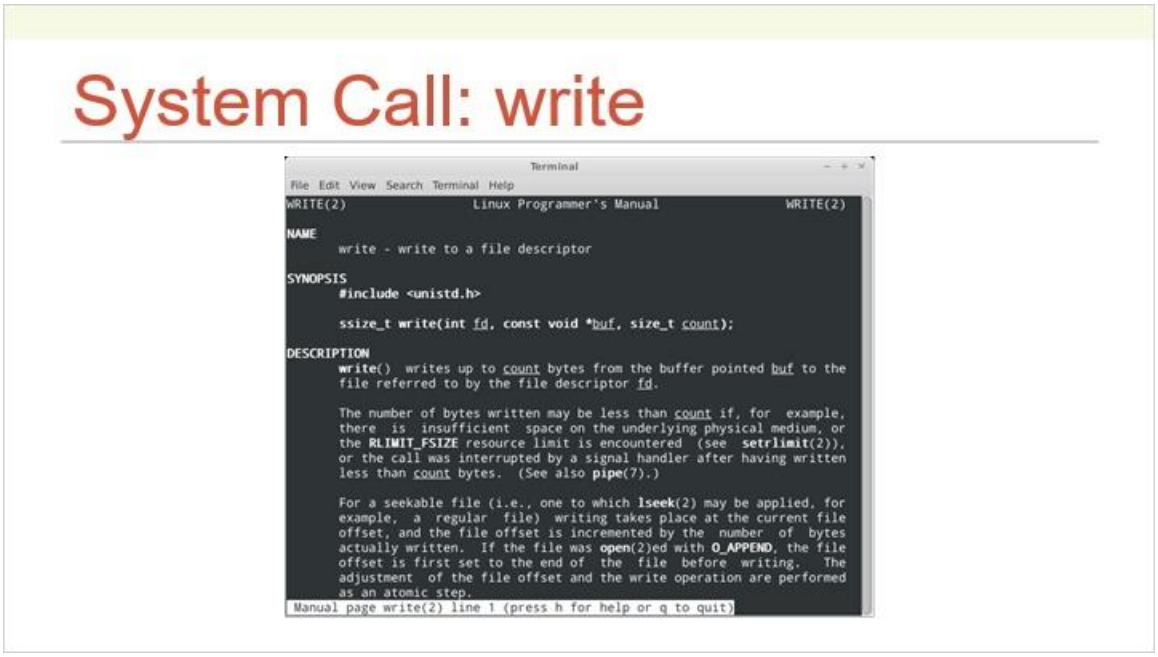
1.13 System Call: readAll example

System Call: readAll example

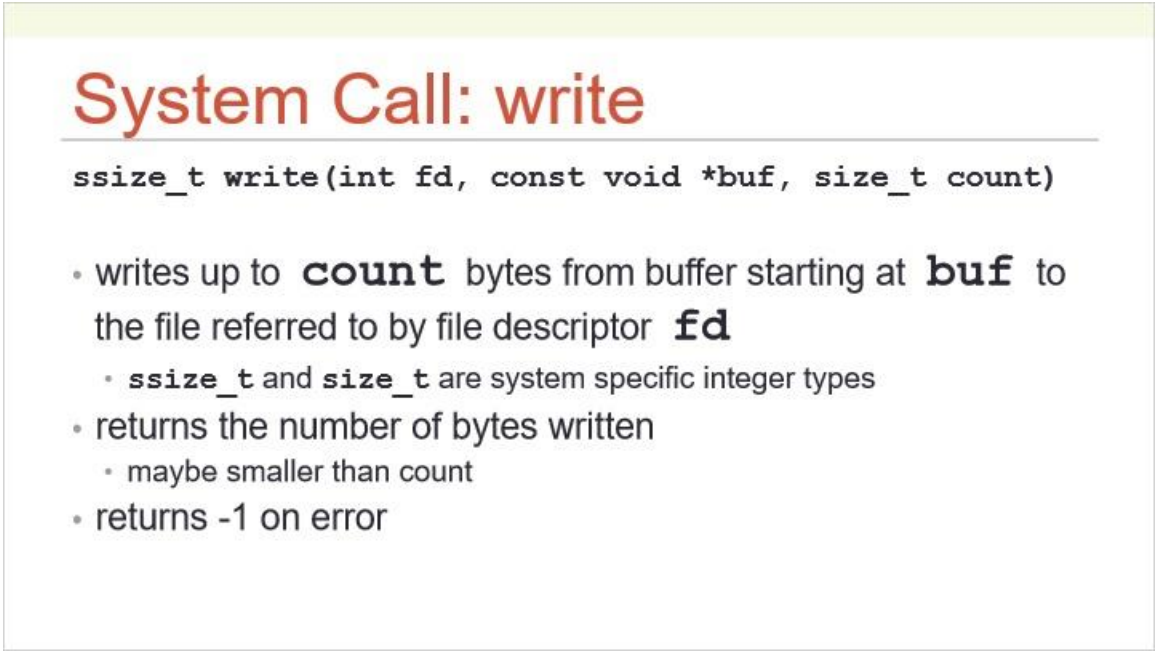
readAll.cxx

```
18  int main(int argc, char* argv[]) {
19      int fd, count, sum=0;
20      // check command line arguments
21      if (argc < 2) {
22          cerr << "Usage: readAll filename\n";
23          exit(EXIT_FAILURE);
24      }
25      char buffer[1024];
26      // open file
27      fd = open(argv[1], O_RDONLY);
28      if (fd == -1) {
29          perror(argv[1]);
30          exit(EXIT_FAILURE);
31      }
32      // read from file
33      while ((count = read(fd, buffer, 1024)) != 0) {
34          if (count == -1) {
35              perror(argv[1]);
36              exit(EXIT_FAILURE);
37          }
38          sum += count;
39          cout << "read " << count << " bytes from file\n";
40      }
41      cout << "read all " << sum << " bytes from " << argv[1] << endl;
42      // close  file
43      close(fd);
44      return 0;
45  }
```

1.14 System Call: write



1.15 System Call: write



1.16 System Call: write example

System Call: write example

write.cxx

```
17  int main() {
18      int fd, count;
19      char buffer[] = "one line to write";
20      // open existing file, will overwrite current content
21      fd = open("example.txt", O_WRONLY);
22      if (fd == -1) {
23          perror("example.txt");
24          exit(fd);
25      }
26      // write to file
27      count = write(fd, buffer, sizeof(buffer));
28      if (fd == -1) {
29          perror("example.txt");
30          exit(fd);
31      }
32      cout << "wrote " << count << " bytes to file\n";
33      // close file
34      close(fd);
35      return 0;
36  }
```

1.17 revisit System Call: open

revisit System Call: open

Terminal

File Edit View Search Terminal Help

```
OPEN(2)                    Linux Programmer's Manual                    OPEN(2)

NAME
       open, creat - open and possibly create a file or device

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);

DESCRIPTION
       Given  a  pathname  for  a file, open() returns a file descriptor, a
       small, nonnegative  integer  for  use  in  subsequent  system  calls
       (read(2),  write(2), lseek(2), fcntl(2), etc.).  The file descriptor
       returned by a successful  call  will  be  the  lowest-numbered  file
       descriptor not currently open for the process.

       By  default, the new file descriptor is set to remain open across an
       execve(2) (i.e., the FD_CLOEXEC file descriptor  flag  described  in
       fcntl(2) is initially disabled; the O_CLOEXEC flag, described below,
       can be used to change this default).  The file offset is set to  the
       beginning of the file (see lseek(2)).
 Manual page open(2) line 1 (press h for help or q to quit)
```

1.18 System Call: open

System Call: open

```
int open(const char* pathname, int flags)
```

- opens file specified as **`pathname`** for access
- **`flags`** determine access type
  - **`O_RDONLY`** read only
  - **`O_WRONLY`** write only
  - **`O_RDWR`** read and write
- returns file descriptor, to be used in read/write/close
- returns -1 on error

1.19 System Call: open

System Call: open

- additional **`flags`**, used with **`O_WRONLY`**:
  - **`O_APPEND`** to append to an existing file
  - **`O_TRUNC`** existing file will overwritten (default)

Ex.:

```
O_WRONLY | O_TRUNC
O_WRONLY | O_APPEND
```

- additional flag: O_CREAT
  - creates file, if file does not exist

1.20 revisit System Call: open

revisit System Call: open

Terminal

File Edit View Search Terminal Help

```
OPEN(2)                    Linux Programmer's Manual                    OPEN(2)

NAME
       open, creat - open and possibly create a file or device

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);

DESCRIPTION
       Given  a  pathname  for  a file, open() returns a file descriptor, a
       small, nonnegative  integer  for  use  in  subsequent  system  calls
       (read(2),  write(2), lseek(2), fcntl(2), etc.).  The file descriptor
       returned by a successful  call  will  be  the  lowest-numbered  file
       descriptor not currently open for the process.

       By  default, the new file descriptor is set to remain open across an
       execve(2) (i.e., the FD_CLOEXEC file descriptor  flag  described  in
       fcntl(2) is initially disabled; the O_CLOEXEC flag, described below,
       can be used to change this default).  The file offset is set to  the
       beginning of the file (see lseek(2)).
 Manual page open(2) line 1 (press h for help or q to quit)
```

1.21 System Call: open with O_CREAT

System Call: open with O_CREAT

`int open(const char* pathname, int flags, mode_t mode)`

- must specify file mode, i.e. permissions, of type **`mode_t`**

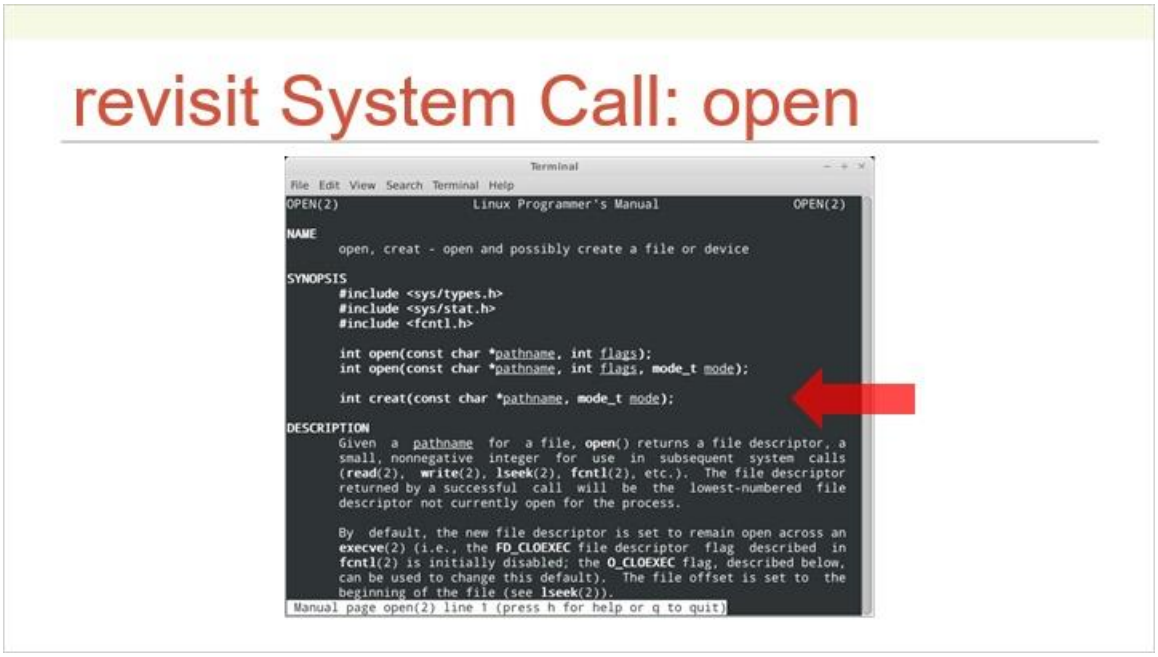
```
S_IRWXU  00700  user has read, write and execute permission
S_IRUSR  00400  user has read permission
S_IWUSR  00200  user has write permission
S_IXUSR  00100  user has execute permission
S_IRWXG  00070  group has read, write and execute permission
S_IRWXO  00007  others have read, write and execute perm.
...
```

  - C++ requires leading '0' for octal numbers
  - actual file mode is further modified by umask

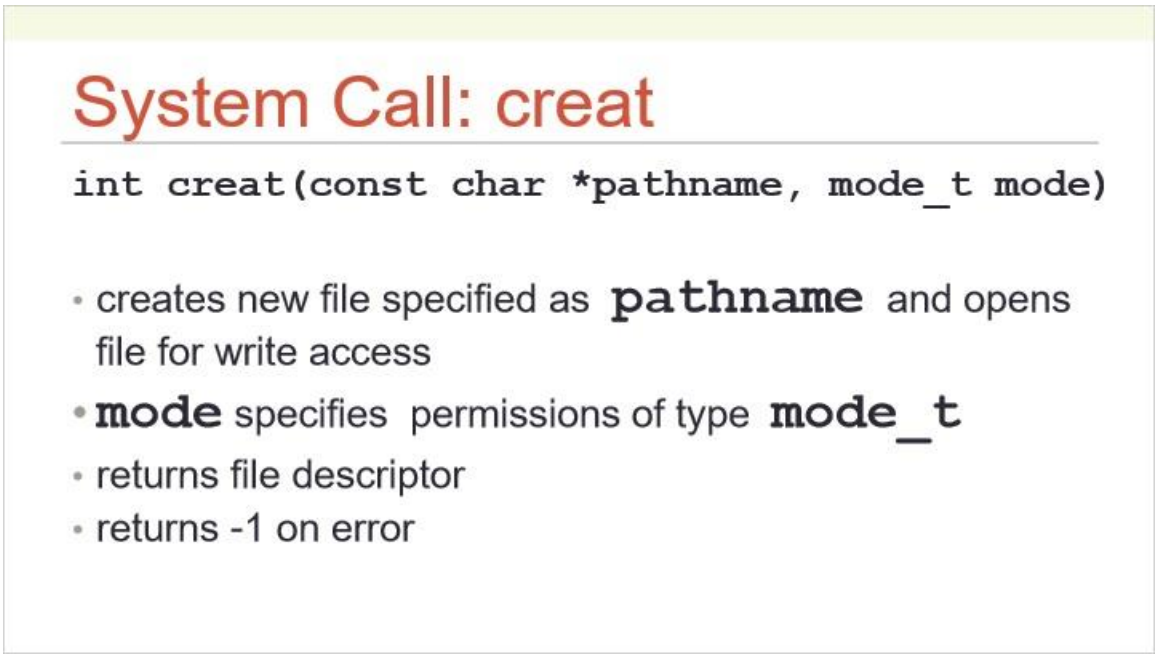
Example:

```
open("ex.txt", O_WRONLY | O_APPEND | O_CREAT, 00666)
```

1.22 revisit System Call: open



1.23 System Call: creat



1.24 Example: copy file

Example: copy file

copy.cxx

```
/*
 * copy.cxx
 *
 *  Example Program for CSCI 330
 *  implement copy by reading file, and writing file
 *  via read/write system calls
 *
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <cstdlib>
#include <cstdio>
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {

    int ifd, ofd, count, sum=0;
    char buffer[1024];

    // check command line arguments
    if (argc < 3) {
        cerr << "Usage: copy fromFile toFile\n";
        exit(EXIT_FAILURE);
    }
    // open file to read
    ifd = open(argv[1], O_RDONLY);
    if (ifd == -1) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
```

1.25 Summary

Summary

- System calls: open, creat, read, write, close

- next:

unlink	remove file
stat	get file information
chmod	change permissions
dup	duplicate file descriptor