I/O SYSTEM CALLS

Alok Jhaldiyal
Assistant Professor
Dept. of Cyber Platform, SoCSE,UPES

System Calls for I/O

- They look like regular procedure calls but are different
  - A system call makes a request to the operating system by trapping into kernel mode
  - A procedure call just jumps to a procedure defined elsewhere in your program
- Some library procedure calls may themselves make a system call
  - e.g., fopen() calls open()

File Statistics

```
#include <sys/stat.h>
int stat(const char* name, struct stat* buf);
```

- Get information about a file
- Returns:
  - 0 on success
  - -1 on error, sets errno
- Parameters:
  - name: Path to file you want to use
    - Absolute paths begin with “/”, relative paths do not
  - buf: Statistics structure
    - off_t st_size: Size in bytes
    - time_t st_mtime: Date of last modification. Seconds since January 1, 1970

File: Open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char* path, int flags [, int mode ]);
```

- Open (and/or create) a file for reading, writing or both
- Returns:
  - Return value ≥ 0 : Success - New file descriptor on success
  - Return value = -1: Error, check value of errno
- Parameters:
  - path: Path to file you want to use
    - Absolute paths begin with "/", relative paths do not
  - flags: How you would like to use the file
  - O_RDONLY: read only, O_WRONLY: write only, O_RDWR: read and write,
  - O_CREAT: create file if it doesn't exist, O_EXCL: prevent creation if it already exists

Open: Example

```
#include <fcntl.h>
#include <errno.h>
extern int errno;

main() {
    int fd;
    fd = open("foo.txt", O_RDONLY | O_CREAT);
    printf("%d\n", fd);
    if (fd=-1) {
        printf ("Error Number %d\n", errno);
        perror("Program");
    }
}
```

Argument: string
Output: the string, a colon, and a description of the error condition stored in `errno`

File: Close

```
#include <fcntl.h>
int close(int fd);
```

- Close a file
  - Tells the operating system you are done with a file descriptor
- Return:
  - 0 on success
  - -1 on error, sets errno
- Parameters:
  - fd: file descriptor

Close: Example

```
#include <fcntl.h>
main(){
    int fd1;

    if(( fd1 = open("foo.txt", O_RDONLY)) < 0){
        perror("c1");
        exit(1);
    }
    if (close(fd1) < 0) {
        perror("c1");
        exit(1);
    }
    printf("closed the fd.\n");
```

File: Read

```
#include <fcntl.h>
size_t read (int fd, void* buf, size_t cnt);
```

- Read data from one buffer to file descriptor
  - Read `size` bytes from the file specified by `fd` into the memory location pointed to by `buf`
- Return: How many bytes were actually read
  - Number of bytes read on success
  - 0 on reaching end of file
  - -1 on error, sets `errno`
  - -1 on signal interrupt, sets `errno` to `EINTR`
- Parameters:
  - `fd`: file descriptor
  - `buf`: buffer to read data from
  - `cnt`: length of buffer

File: Read

```
size_t read (int fd, void* buf, size_t cnt);
```

- Things to be careful about
  - `buf` needs to point to a valid memory location with length not smaller than the specified size
    - Otherwise, what could happen?
  - `fd` should be a valid file descriptor returned from `open()` to perform read operation
    - Otherwise, what could happen?
  - `cnt` is the requested number of bytes read, while the return value is the actual number of bytes read
    - How could this happen?

Read: Example

```
#include <fcntl.h>
main() {
    char *c;
    int fd, sz;

    c = (char *) malloc(100
              * sizeof(char));
    fd = open("foo.txt",
              O_RDONLY);
    if (fd < 0) {
        perror("r1");
        exit(1);
    }
    sz = read(fd, c, 10);
    printf("called
        read(%d, c, 10).
        returned that %d
        bytes  were
        read.\n", fd, sz);
    c[sz] = '\0';

    printf("Those bytes
        are as follows:
        %s\n", c);
    close(fd);
}
```

File: Write

```
#include <fcntl.h>
size_t write (int fd, void* buf, size_t cnt);
```

- Write data from file descriptor into buffer
  - Writes the bytes stored in `buf` to the file specified by `fd`
- Return: How many bytes were actually written
  - Number of bytes written on success
  - 0 on reaching end of file
  - -1 on error, sets `errno`
  - -1 on signal interrupt, sets `errno` to `EINTR`
- Parameters:
  - `fd`: file descriptor
  - `buf`: buffer to write data to
  - `cnt`: length of buffer

File: Write

```
size_t write (int fd, void* buf, size_t cnt);
```

- Things to be careful about
  - The file needs to be opened for write operations
  - `buf` needs to be at least as long as specified by `cnt`
    - If not, what will happen?
  - `cnt` is the requested number of bytes to write, while the return value is the actual number of bytes written
    - How could this happen?

Write: Example

```
#include <fcntl.h>
main()
{
    int fd, sz;

    fd = open("out3",
        O_RDWR | O_CREAT |
        O_APPEND, 0644);
    if (fd < 0) {
        perror("r1");
        exit(1);
    }
```

```
sz = write(fd, "cs241\n",
    strlen("cs241\n"));

printf("called write(%d,
    \"cs360\\n\", %d).
    it returned %d\n",
    fd, strlen("cs360\n"),
    sz);

close(fd);
}
```

Unix Error Model

- Error Model
  - `errno` variable
    - Unix provides a globally accessible integer variable that contains an error code number
  - Return value
    - 0 on success
    - -1 on failure for functions returning integer values
    - NULL on failure for functions returning pointers
  - Examples (see `errno.h`)

```
#define EPERM     1      /* Operation not permitted */
#define ENOENT    2      /* No such file or directory */
#define ESRCH     3      /* No such process */
#define EINTR     4      /* Interrupted system call */
#define EIO       5      /* I/O error */
#define ENXIO     6      /* No such device or address */
```