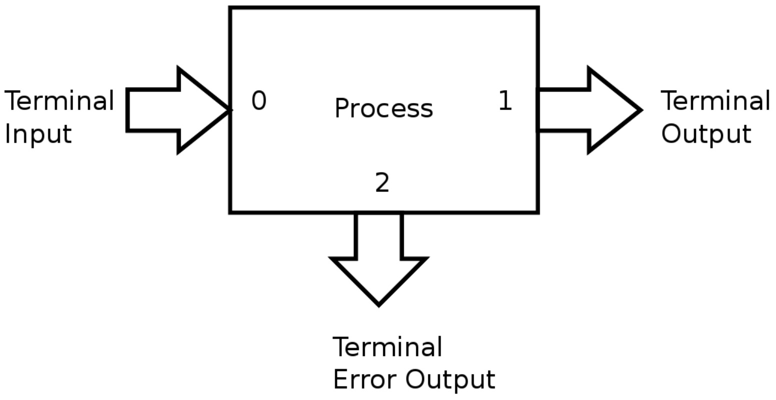
File I/O

Zilogic Systems

1. Input and Output

- Each process has a number of input and output sources associated with it.
- Examples:
  - Input Source: data from a file
  - Output Source: data sent to printer
  - Input Source: input from keyboard
  - Output Source: message printed to screen
- Each input/output source is given a unique number within the process.
- There are three input/output sources available by default to a process.
  - 0 - standard input, input source, usually associated with the terminal
  - 1 - standard output, output source, usually associated with the terminal
  - 2 - standard error, output source, also associated with the terminal
- The unique numbers give to each source is called a file descriptor.

Figure 1. Standard Input, Output and Error



- Redirecting input, output and error.

Figure 2. Redirecting Standard Input

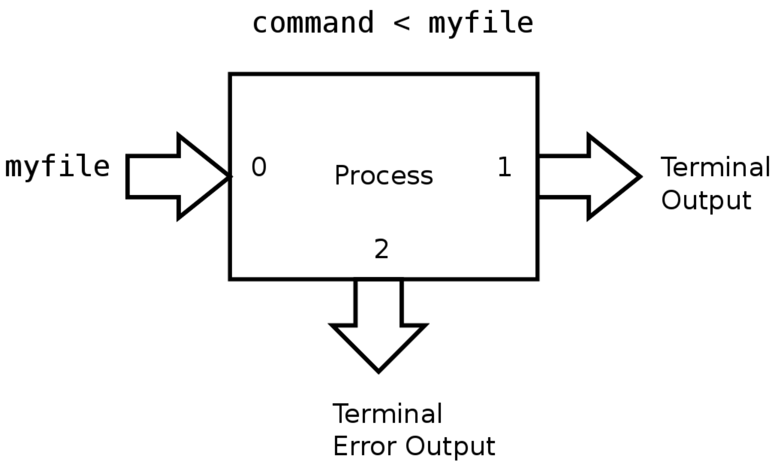
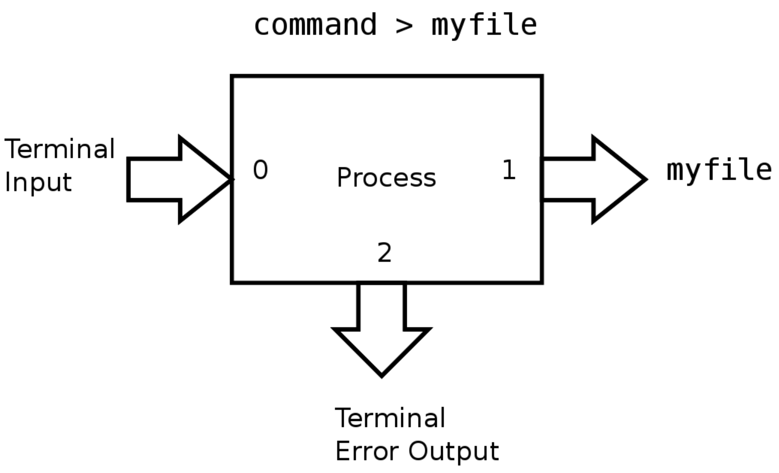


Figure 3. Redirecting Standard Output



2. Read and Write

- Data can be got from input sources using `read()` system call.
- Data can be sent to output sources using `write()` system call.

```
#include <unistd.h>

ret = read(fd, buf, len)

int fd
char *buf
size_t len
ssize_t ret
```

- `fd` specifies the file descriptor to read from
- `buf` is a buffer (char array) into which read data will be stored
- `len` is the size of the buffer
- returns the no. of bytes read
- if `ret` is `0`, indicates end-of-file
- if `ret` is `-1`, indicates error has occured

```
#include <unistd.h>

ret = write(fd, buf, len)

int fd
char *buf
size_t len
ssize_t ret
```

- `fd` specifies the file descriptor to write to
- `buf` is a buffer from which data will be written
- `len` is the no. of bytes in the buffer
- returns the no. of bytes written
- if `ret` is `-1`, indicates error has occured

Char array vs. String

- The buffer passed to `read()` and `write()` is an character array not a string.

3. Implementing `tr`

`tr` initial version.

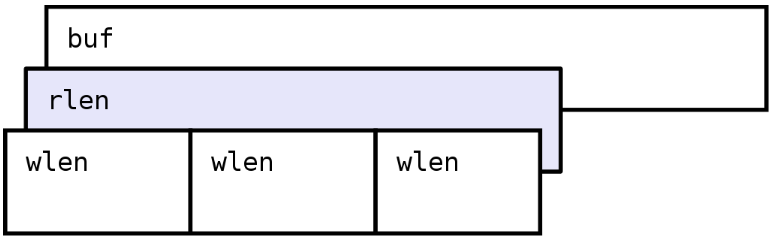
```
#include <unistd.h>

#define BUF_SIZE 256

int main(int argc, char *argv[])
{
        char buf[BUF_SIZE];
        ssize_t rlen;
        int i;
        char from;
        char to;

        from = 'e';
        to = 'a';

        while (1) {
                rlen = read(0, buf, sizeof(buf));
                if (rlen == 0)
                        return 0;

                for (i = 0; i < rlen; i++) {
                        if (buf[i] == from)
                                buf[i] = to;
                }

                write(1, buf, rlen);
        }

        return 0;
}
```

`tr` with error checking.

```
#include <unistd.h>
#include <error.h>
#include <errno.h>

#define BUF_SIZE 256

int main(int argc, char *argv[])
{
        char buf[BUF_SIZE];
        ssize_t rlen, wlen;
        int i;
        char from;
        char to;
        from = 'e';
        to = 'a';

        while (1) {
                rlen = read(0, buf, sizeof(buf));
                if (rlen == -1) /* ❶ */
                        error(1, errno, "error reading input"); /* ❷ */
                if (rlen == 0)
                        return 0;

                for (i = 0; i < rlen; i++) {
                        if (buf[i] == from)
                                buf[i] = to;
                }

                wlen = write(1, buf, rlen);
                if (wlen == -1) /* ❸ */
                        error(1, errno, "error writing to output");
        }

        return 0;
}
```

❶ `read()` returns -1 on error. Check for errors and terminate.
❷ `error()` prints an error message and terminates program.
❸ `write()` returns -1 on error. Check for errors and terminate.

Figure 4. Handling Partial Writes



`tr` with partial write handling.

```
#include <unistd.h>
#include <error.h>
#include <errno.h>

#define BUF_SIZE 256

int main(int argc, char *argv[])
{
        char buf[BUF_SIZE];
        char *bufp;
        ssize_t rlen, wlen;
        int i;
        char from;
        char to;
        from = 'e';
        to = 'a';

        while (1) {
                rlen = read(0, buf, sizeof(buf));
                if (rlen == -1)
                        error(1, errno, "error reading input");
                if (rlen == 0)
                        return 0;

                for (i = 0; i < rlen; i++) {
                        if (buf[i] == from)
                                buf[i] = to;
                }

                wlen = 0;
                bufp = buf;
                do { /* ❶ */
                        wlen = write(1, bufp, rlen);
                        if (wlen == -1)
                                error(1, errno, "error writing to output");

                        rlen -= wlen;
                        bufp += wlen;
                } while (rlen != 0); /* ❷ */
        }

        return 0;
}
```

❶❷ Loop till all data has been written.

`tr` with macro names for fds.

```
#include <unistd.h>
#include <error.h>
#include <errno.h>

#define BUF_SIZE 256

int main(int argc, char *argv[])
{
        char buf[BUF_SIZE];
        char *bufp;
        ssize_t rlen, wlen;
        int i;
        char from;
        char to;

        from = 'e';
        to = 'a';

        while (1) {
                rlen = read(STDIN_FILENO, buf, sizeof(buf));
                if (rlen == -1)
                        error(1, errno, "error reading input");
                if (rlen == 0)
                        return 0;

                for (i = 0; i < rlen; i++) {
                        if (buf[i] == from)
                                buf[i] = to;
                }

                wlen = 0;
                bufp = buf;
                do {
                        wlen = write(STDOUT_FILENO, bufp, rlen);
                        if (wlen == -1)
                                error(1, errno, "error writing to output");

                        rlen -= wlen;
                        bufp += wlen;
                } while (rlen != 0);
        }

        return 0;
}
```

`tr` with arguments.

```
#include <unistd.h>
#include <error.h>
#include <errno.h>

#define BUF_SIZE 256

int main(int argc, char *argv[])
{
        char buf[BUF_SIZE];
        char *bufp;
        ssize_t rlen, wlen;
        int i;
        char from;
        char to;

        if (argc != 3) {
                error(1, 0, "too many args or required args not specified");
        }

        from = argv[1][0];
        to = argv[2][0];

        while (1) {
                rlen = read(STDIN_FILENO, buf, sizeof(buf));
                if (rlen == -1)
```

```
                                error(1, errno, "error reading input");
                if (rlen == 0)
                        return 0;

                for (i = 0; i < rlen; i++) {
                        if (buf[i] == from)
                                buf[i] = to;
                }

                wlen = 0;
                bufp = buf;
                do {
                        wlen = write(STDOUT_FILENO, bufp, rlen);
                        if (wlen == -1)
                                error(1, errno, "error writing to output");

                        rlen -= wlen;
                        bufp += wlen;
                } while (rlen != 0);
        }

        return 0;
}
```

4. Open and Close

- Additional file descriptors can be obtained by opening files, sockets, device nodes, etc.

```
#include <fcntl.h>

fd = open(name, flags)

char *name
int flags
int fd
```

- `name` is the path name to the file.
- `flags` - `O_RDONLY`, `O_WRONLY`, `O_RDWR`
- returns a file descriptor on success
- returns `-1` on error

```
#include <fcntl.h>

ret = close(fd)

int fd
int ret
```

- `fd` is the file descriptor to be closed
- returns `0` on success and `-1` on error

5. Implementing `cat`

`cat` initial version.

```
#include <unistd.h>
#include <fcntl.h>

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        char *filename;
        int fd;
        char buf[1024];
        ssize_t ret = 1;
        int i;

        if (argc != 2) {
                printf("Usage: mycat <filename>\n");
                exit(1);
        }

        filename = argv[1];

        fd = open(filename, O_RDONLY);
        while (ret != 0) {
                /* Reading into buffer */
                ret = read(fd, buf, sizeof(buf));
                for (i = 0; i < ret; i++) {
                        putchar(buf[i]);
                }
        }

        close(fd);

        return 0;
}
```

`cat` with error checking.

```
#include <unistd.h>
#include <fcntl.h>

#include <stdio.h>
#include <stdlib.h>
#include <error.h>
#include <errno.h>

int main(int argc, char *argv[])
{
        char *filename;
        int fd;
        char buf[1024];
        ssize_t ret = 1;
        int i;

        if (argc != 2) {
                printf("Usage: mycat <filename>\n");
                exit(1);
        }

        filename = argv[1];

        fd = open(filename, O_RDONLY);
        if (fd == -1) { /* ❶ */
                error(1, errno, "error opening file %s", filename); /* ❷ */
        }
        while (ret != 0) {
                ret = read(fd, buf, sizeof(buf));
                if (ret == -1) { /* ❸ */
                        error(1, errno, "error reading file %s", filename);
                }
                for (i = 0; i < ret; i++) {
                        putchar(buf[i]);
                }
        }

        if (close(fd) == -1) {
                error(1, errno, "error closing file %s", filename);
        }

        return 0;
}
```

❶ Checking for error in `open()` syscall.
❷ Prints an error message and terminates the process.
❸ Checking for error in `read()` syscall.

6. Implementing `cp`

`cp` initial version.

```
#include <unistd.h>
#include <fcntl.h>

#include <stdio.h>
#include <stdlib.h>
#include <error.h>
#include <errno.h>

void write_buf(int fd, char *bufp, size_t nbytes, char *filename)
{
        int written;

        while (nbytes != 0) {
                written = write(fd, bufp, nbytes);
                if (written == -1)
```

```
                        error(1, errno, "error writing to file %s", filename);

                if (written <= nbytes) {
                        bufp += written;
                        nbytes -= written;
                }
        }
}

int main(int argc, char *argv[])
{
        char *source;
        char *dest;
        int source_fd, dest_fd;
        char buf[1024];
        ssize_t read_len;

        if (argc != 3) {
                printf("Usage: mycp <src-filename> <dest-filename>\n");
                exit(1);
        }

        source = argv[1];
        dest = argv[2];

        source_fd = open(source, O_RDONLY);
        if (source_fd == -1)
                error(1, errno, "error opening file %s", source);

        dest_fd = open(dest, O_WRONLY);
        if (dest_fd == -1)
                error(1, errno, "error opening file %s", dest);

        while (1) {
                read_len = read(source_fd, buf, sizeof(buf));
                if (read_len == -1)
                        error(1, errno, "error reading file %s", source);

                if (read_len == 0)
                        break;

                write_buf(dest_fd, buf, read_len, dest);
        }

        if (close(source_fd) == -1) {
                error(1, errno, "error closing file %s", source);
        }

        if (close(dest_fd) == -1) {
                error(1, errno, "error closing file %s", dest);
        }

        return 0;
```

```
}
```

`cp` with creation and truncation.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

#include <stdio.h>
#include <stdlib.h>
#include <error.h>
#include <errno.h>

void write_buf(int fd, char *bufp, size_t nbytes, char *filename)
{
        int written;

        while (nbytes != 0) {
                written = write(fd, bufp, nbytes);
                if (written == -1)
                        error(1, errno, "error writing to file %s", filename);

                if (written <= nbytes) {
                        bufp += written;
                        nbytes -= written;
                }
        }
}

int main(int argc, char *argv[])
{
        char *source;
        char *dest;
        int source_fd, dest_fd;
        char buf[1024];
        ssize_t read_len;
        struct stat stat;
        int ret;

        if (argc != 3) {
                printf("Usage: mycp <src-filename> <dest-filename>\n");
                exit(1);
        }

        source = argv[1];
        dest = argv[2];

        source_fd = open(source, O_RDONLY);
        if (source_fd == -1)
                error(1, errno, "error opening file %s", source);

        ret = fstat(source_fd, &stat);
        if (ret == -1)
                error(1, errno, "error getting mode bits of %s", source);

        dest_fd = open(dest, O_WRONLY | O_TRUNC | O_CREAT, /* ❶ */
                       stat.st_mode & 0777); /* ❷ */
        if (dest_fd == -1)
                error(1, errno, "error opening file %s", dest);

        while (1) {
                read_len = read(source_fd, buf, sizeof(buf));
                if (read_len == -1)
                        error(1, errno, "error reading file %s", source);

                if (read_len == 0)
                        break;

                write_buf(dest_fd, buf, read_len, dest);
        }

        if (close(source_fd) == -1) {
                error(1, errno, "error closing file %s", source);
        }

        if (close(dest_fd) == -1) {
                error(1, errno, "error closing file %s", dest);
        }

        return 0;
}
```

❶ Create file if not present. Truncate file if already present.

❷ Mode to use when file is created.