1 System-level I/O

I/O is the process of copying data between the main memory and external devices, e.g. disk drivers, terminal and network. All runtime systems offer high-level I/O tools, e.g. `stdio.h` of C and `iostream` of C++. Unix system offer system-level I/O functions to implement these high-level I/O functions. Learning Unix I/O can help us understand other system concepts, and sometimes we have no other option but to use Unix I/O.

1.1 Unix I/O interface

A Linux file is a sequence of bytes. All I/O devices are modeled as files, and all input and output is performed by reading and writing the appropriate files. This elegant mapping allows the Linux kernel to export a simple low-level I/O application interface.

- Opening files. An application asks the kernel to open a file in order to access the related IO device. The kernel returns a small non-negative integer called the **descriptor** that identifies the file in subsequent operations. The kernel keeps all information of the opened file, while the application keeps nothing but the descriptor. Each process created by the Linux shell has 3 open files as it begins: standard input (0), standard output (1) and standard error (2). `unistd.h` defines the constants `STDIN_FILENO, STDOUT_FINENO` and `STDERR_FILENO`.
- Change the current file position. Linux keeps a file position k for each opened file, initialized as 0. It describes the byte offset from the beginning of the file. An application can explicitly set the current file position k with the `seek` operation.
- Read from / Write to a file. Read: copy $n > 0$ bytes from the file to the memory, starting from the current file position k, and increasing k to k+n. Given a file of size m, a read when $k \geq m$ will trigger the EOF condition, which can be detected by the application. There exists no explicit "EOF" symbol at the end of a file. Write: copy $n > 0$ files from the memory to a file, starting from the file position k and update k.
- Close a file. When an application has finished accessing a file, it informs the kernel to close the file. The data structures created when opening the file are freed, and the file descriptor is restored back to the pool of available descriptors. When a process terminates for some reason, the kernel closes all open files and frees their memory resources.

1.2 Files

Each Linux file has a type to indicate its role in the system.

- A **regular** file contains any data: text file or binary file. There is no difference between the two for the kernel.
- A **directory** contains a series of links, each maps a file name to a file. Each directory contains at least two items: . to itself and .. to its parent directory in the directory hierarchy.
- A **socket** is used to communicate with another process across a network.
- Other file types include named pipe, symbolic link, character and block device, etc.

All files are organized in a single directory hierarchy. Each process has a current working directory to indicate its current position in the hierarchy. It can be changed by the `cd` command and specified with a pathname, which can be absolute or relative.

1.2.1 Open & close a file

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(char *filename, int flags, mode_t mode);
```

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

`flags`: indicate how the process wants to access the file:

O_RDONLY Read only

O_WRONLT Write only

O_RDWR Read/Write

O_CREAT Create a truncated (empty) version if the file does not exist.

O_TRUNC Truncate the file if it exists.

O_APPEND Set the file position to the end of the file before each write.

`mode`: Access permission bits. P = R(read), W(write), X(execute).

S_IPUSR User (Owner)

S_IPGRP The group to which the owner belongs

S_IPOTH Others (Anyone)

Each process has a `umask` as part of its context. The access permission bits are set as `mode & ~umask`, i.e. it is used to forbid some permissions, such as execute permission of anyone (S_IXOTH).

1.2.2 Read & write a file

```
#include <unistd.h>
ssize_t read(int fd, void* buf, size_t n);
ssizee_t write(int fd, const void* buf, size_t n);
```

In x86-64, `ssize_t` is a typedef of `long` while `size_t` is a typedef of `unsigned long`, so that the former can return -1 to indicate error.

Short count `read` or `write` transfers fewer bytes than the application specifies, for the following reasons:

- Encountering EOF on reads.
- Reading lines from a terminal: `read` transfers a text line each time.
- Reading / Writing network sockets.

1.3 Robust reading / writing with RIO package

Unbuffered IO functions Transfer data directly between memory and a file, with no application-level buffering. Useful for binary r/w to and from networks.

Buffered input functions Read text lines and binary data from a file whose contents are cached in an application-level buffer, similar to `printf`.

1.3.1 Unbuffered IO

```
/* Unbuffered read. Returns short count when encountering EOF.*/
ssize_t rio_readn(int fd, void *userbuf, size_t n) {
  size_t nleft = n;
  ssize_t nread;
  char *bufp = userbuf;

  while (nleft > 0) {
    if ((nread = read(fd, bufp, nleft)) < 0) { //error
      if (errno == EINTR) //Interrupted by sig handler. Should call read again
        nread = 0;
      else
        return -1;
    } else if (nread == 0) //EOF
      break;
    nleft -= nread;
    bufp += nread;
  }

  return n - nleft;
}

/* Unbuffered write. Never returns short count. */
ssize_t rio_writen(int fd, void *userbuf, size_t n) {
  size_t nleft = n;
  ssize_t nwritten;
  char *bufp = userbuf;

  while(nleft > 0) {
    if((nwritten = write(fd, bufp, nleft)) < 0) {
      if(error == EINTR)
        nwritten = 0;
      else
        return -1;
    }
    nleft -= nwritten;
    bufp += nwritten;
  }
  return n;
}
```

1.3.2 Buffered input

```
#define RIO_BUFSIZE 8192
typedef struct {
  int rio_fd;                  /* Descriptor */
  int rio_cnt;                 /* Num of unread bytes in internal buffer.  */
  char *rio_bufptr;            /* Next unread byte in internal buffer */
  char rio_buf[RIO_BUFSIZE];   /* Internal buffer */
} rio_t;

/* Called once per open descriptor */
void rio_readinitb(rio_t *rp, int fd) {
  rp->rio_fd = fd;
  rp->rio_cnt = 0;
  rp->rio_bufptr = rp->rio_buf;
}

/* Internal buffered read (buffered version of Linux read)*/
static ssize_t rio_read(rio_t *rp, char *userbuf, size_t n) {
  int cnt;
  while (rp->rio_cnt <= 0) {
    rp->rio_cnt = read(rp->rio_fd, rp->rio_buf, sizeof(rp->rio_buf));
    if (rp->rio_cnt < 0) {
      if (errno != EINTR)
        return -1;
    } else if (rp->rio_cnt == 0) {
      return 0;
    } else {
      rp->rio_bufptr = rp->rio_buf;
    }
    cnt = n < rp->rio_cnt ? n : rp->rio_cnt;
    memcpy(userbuf, rp->rio_bufptr, cnt);
    rp->rio_bufptr += cnt;
    rp->rio_cnt -= cnt;
    return cnt;
  }
}

/* Buffered readn */
ssize_t rio_readnb(rio_t *rp, void *userbuf, size_t n) {
  size_t nleft = n;
  ssize_t nread;
  char *bufp = userbuf;

  while (nleft > 0) {
    if ((nread = rio_read(rp, bufp, nleft)) < 0)//error
      return -1; //no need to check EINTR because it is guaranteed by rio_read
    else if (nread == 0) //EOF
      break;
    nleft -= nread;
    bufp += nread;
  }

  return n - nleft;
}

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen) {
  int n, rc;
  char c, *bufp = usrbuf;
  for(n = 1; n < maxlen; ++n) {
    rc = rio_read(rp, &c, 1);
    if(1 == rc) {
      *bufp++ = c;
      if('\n' == c) {
        ++n;
        break;
      }
    } else if (0 == rc) {
      if (1 == n)
        return 0;
      else
        break;
    } else
      return -1;
  }
  *bufp = '\0';
  return n - 1;
}
```

1.4 Read metadata of files

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

`struct stat` contains `mode_t st_mode` that encodes the visit permission bits of the file, and `off_t st_size` which is the bytes of the file.

1.5 Read directory content

```
#include <sys/types.h>
#include <dirent.h>
struct direct {
  ino_t d_ino;        /* inode number */
  char  d_name[256]; /* file name */
};
DIR *opendir(const char *name);
/* DIR: directory stream. Stream is the abstraction of an ordered sequence of
    entries.*/
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

1.6 Sharing files

Descriptor table Each process has its own descriptor table indexed by the process's open file descriptors. Each fd points to an entry in the file table.

File table Shared by all processes. Each entry represents an open file, containing the current file position, reference count and a pointer to the v-node table. Closing an fd decrements the reference count of the file table entry it points to. The kernel won't delete the entry until its reference count is reduced to 0.

v-node table Shared by all processes. Each entry contains most information in the `stat` struct.

Each fd has its own current position, so different descriptors related to the same file point to separate entries in the file table, while the same fd in a parent process and its children processes point to the same entry.

1.7 IO redirection

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
/*Copies fd table entry of oldfd to that of newfd.*/
```