Systems Programming

```
- UNIX System calls: C functions that provide access to the
     file system, processes, and error handling.

- System Calls grouped into 3 main categories:
    * File Management (Fig. 12.1)
    * Process Management (Fig. 12.2)
    * Error Handling (Fig. 12.3)
```

```
Error Handling:
  - Most system calls are capable of failing in some way.
  - ex. open a file may fail because file does not exist!
  - System call returns a value of -1 when it fails.
  - This value does not tell much about the cause of the failure.

global variable errno (holds the numeric code of the last system-call
                       error)
function perror() describes the system-call error

  void perror(char* str)   /* standard C function in stdio.h */

displays str followed by : followed by a description of the last
  system call error. (Error 0 is displayed if no error)
```

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main () {
 int fd;
  /* Open a non-existent file to cause an error */
 fd = open ("nonexist.txt", O_RDONLY);
 if (fd == -1) /* fd == -1 =, an error occurred */ {
  printf ("errno = %d\n", errno);
  perror ("main");
 }
 fd = open ("/", O_WRONLY); /* Force a different error */
 if (fd == -1) {
  printf ("errno = %d\n", errno);
  perror ("main");
 }
 /* Execute a successful system call */
 fd = open ("nonexist.txt", O_RDONLY | O_CREAT, 0644);
 printf ("errno = %d\n", errno); /* Display after successful call */
                                 /* will display previous error num (21) */
 perror ("main");
 errno = 0; /* Manually reset error variable */
 perror ("main");
}
```

FILE MANAGEMENT:
  System Calls: open, fcntl, read, write, lseek, unlink, close

Typical sequence:

```
int fd;  /* file descriptor 0: std. in, 1: std. out, 2: std error*/
fd = open(fileName,...);
if (fd == -1)  /* deal with error */
fcntl(fd,...)  /* set IO flags if necessary */
read(fd,...)  /* read from file */
write(fd,...)  /* write to file */
lseek(fd,...)  /* seek within file */
close(fd);  /* close file */
```

```
- Can open file several times (with different descriptors)
- Each descriptor has its own private set of properties

  * file pointer (stores offset within file;
                  changes on read/write/lseek)
  * flag indicating if the file descriptor should be closed
    or not when process execs
  * flag indicating if output to file should be appended to end
    of file or not
  * others

System calls open() and fcntl() both allow you to manipulate these
 flags.
```

```
Opening a file: open()
  int open(char *fileName, int mode [, int permissions])

  - allows you to open an existing file or create a new file for r/w
  - fileName: absolute or relative path name
  - mode: bitwise or-ing of a r/w flag together with zero or more
          miscellaneous flags
          r/w flags: O_RDONLY, O_WRONLY, O_RDWR
          misc. flags:
             O_APPEND : position file pointer at the end of file
                        before each write
             O_CREAT  : if the file does not exist, create it and set
                        the owner ID = process' eff. uid (uses umask
                        value to set permissions)
             O_EXCL   : if O_CREAT is set and file exists then open()
                        fails
             O_NONBLOCK: for named pipes
             O_TRUNC: if file exists it is truncated to length 0
  - permissions: supplied only when file is created (ex. 0600 octal)

  ex. tmpfd = open(tmpName, O_CREAT | O_RDWR, 0600);
      fd = open (fileName, O_RDONLY);
```

Reading from regular file: read()

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

- typedef int ssize_t;
- typedef unsigned int size_t;
- copies upto count bytes from the file referenced by fd
      into buffer buf
- the bytes are read from current position (file pointer) which
  is then updated accordingly
- it returns number of bytes copied (returns 0 if it attempts to
    copy after end of file)
- returns -1 if unsuccessful

ex. charsRead = read(fd,buffer,BUFFER_SIZE);
    if charsRead == 0 break;
    if charsRead == -1 fatalError();
```

Writing to regular file: write()

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

- copies upto count bytes from a buffer buf into the file referenced by fd
- the bytes are written into current position (file pointer) which is then updated accordingly
- if O_APPEND was set, file pointer is set to end of file before each write
- it returns number of bytes copied (should check this return value)
- returns -1 if unsuccessful

```
ex. if (standardInput) {
      charsWritten = write(tmpfd,buffer,BUFFER_SIZE);
      if (charsWritten != charsRead) fatalError();
    }
```

Moving file pointer: lseek()

```
off_t lseek (int fd, off_t offset, int mode)
```

- typedef long off_t
- changes file pointer
- mode determines how offset is to be used.
- mode = SEEK_SET : offset relative to start of file
  mode = SEEK_CUR : offset relative to current position of file
  mode = SEEK_END : offset relative to end of file

- lseek fails if moved before start of file
- returns new file position if successful
- returns -1 if unsuccessful

```
ex. lseek(fd, lineStart[i], SEEK_SET);
    charsRead = read(fd,buffer, lineStart[i+1] - lineStart[i]);
```

- to find out current position use:

```
currentOffset = lseek(fd, 0, SEEK_CUR);
```

- if you move past the end of file and write there,
  Unix automatically extends the size of the file and
  treats intermediate area as NULLS (0)

  Unix does not allocate disk area for intermediate space!!
  see next example.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
/*****************************************************/
main () {
 int i, fd;
  /* Create a sparse file */
 fd = open ("sparse.txt", O_CREAT | O_RDWR, 0600);
 write (fd, "sparse", 6);
 lseek (fd, 60006, SEEK_SET);
 write (fd, "file", 4);
 close (fd);
  /* Create a normal file */
 fd = open ("normal.txt", O_CREAT | O_RDWR, 0600);
 write (fd, "normal", 6);
 for (i = 1; i <= 60000; i++)
   write (fd, "/0", 1);
 write (fd, "file", 4);
 close (fd);
}

% sparse
% ls -ls *.txt
   0 -rw-r--r--   1 raj      other          0 Jul 14 11:29 nonexist.txt
 118 -rw-------   1 raj      other      60010 Jul 14 13:44 normal.txt
  22 -rw-------   1 raj      other      60010 Jul 14 13:44 sparse.txt
```

Closing a file: close()

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

frees fd; releases resources; When a process terminates, all fds are automatically closed; still it is a goof idea to close yourself.

returns 0 if success; -1 if failure

Deleting a file: unlink()

```
int unlink(const char *fileName)
```

removes the hard link from the fileName to its file;
if fileName is the last hard link, file resources are deallocated.

ex.

```
  if (standardInput) unlink(tmpName);
```

```
% reverse -c fileName
reverses all lines in fileName
the -c option reverses each line;
if fileName is missing, standard input is used

Algorithm:
  Step 1: Parse Command Line (parseCommandLine, processOptions)
          System Calls used: none

  Step 2: If reading from standard input, create temp. file to
          store input; otherwise open file for input (pass1)
          System Calls used: open()

  Step 3: Read from file in chunks storing starting offsets of
          each line in an array. If reading from std. input, write
          each chunk into temp. file. (pass1, trackLines)
          System Calls used: read(), write()

  Step 4: Read input file again, but backward, copying each line
          to std. output; reverse the line if -c option
          (pass2, processLine, reverseLine)
          System Calls used: lseek()

  Step 5: Close file; remove it if temp. file; (pass2)
          System Calls used: close(), unlink()
```

```
#include <fcntl.h> /* For file mode definitions */
#include <stdio.h>
#include <stdlib.h>

/* Enumerator */
enum { FALSE, TRUE }; /* Standard false and true values */
enum { STDIN, STDOUT, STDERR }; /* Standard I/O channel indices */

/* #define Statements */
#define BUFFER_SIZE    4096   /* Copy buffer size */
#define NAME_SIZE      12
#define MAX_LINES      100000 /* Max lines in file */

/* Function Prototypes */
void parseCommandLine (int argc, char* argv[]);
void processOptions (char* str);
void usageError ();
void pass1 ();
void trackLines (char* buffer, int charsRead);
void pass2 ();
void processLine (int i);
void reverseLine (char* buffer, int size);
void fatalError ();
```

```
/* Globals */
char *fileName = NULL; /* Points to file name */
char tmpName [NAME_SIZE];
int charOption = FALSE; /* Set to true if -c option is used */
int standardInput = FALSE; /* Set to true if reading stdin */
int lineCount = 0; /* Total number of lines in input */
int lineStart [MAX_LINES]; /* Store offsets of each line */
int fileOffset = 0; /* Current position in input */
int fd; /* File descriptor of input */
/****************************************************************/

int main (int argc, char* argv[]) {
  parseCommandLine (argc,argv); /* Parse command line */
  pass1 (); /* Perform first pass through input */
  pass2 (); /* Perform second pass through input */
  return (/* EXITSUCCESS */ 0); /* Done */
}

/****************************************************************/
```

```
void parseCommandLine (int argc, char* argv[]) {
/* Parse command line arguments */
  int i;

  for (i= 1; i < argc; i++) {
    if(argv[i][0] == '-')
      processOptions (argv[i]);
    else if (fileName == NULL)
      fileName= argv[i];
    else
      usageError (); /* An error occurred */
  }
  standardInput = (fileName == NULL);
}
```

```
/****************************************************************/
void processOptions (char* str) {
/* Parse options */
  int j;

  for (j= 1; str[j] != NULL; j++) {
    switch(str[j]) /* Switch on command line flag */ {
      case 'c':
        charOption = TRUE;
        break;
      default:
        usageError ();
        break;
    }
  }
}

/****************************************************************/

void usageError () {
  fprintf (stderr, "Usage: reverse -c [filename]\n");
  exit (/* EXITFAILURE */ 1);
}
```

```
/****************************************************************/

void pass1 () {
/* Perform first scan through file */
  int tmpfd, charsRead, charsWritten;
  char buffer [BUFFER_SIZE];

  if (standardInput) /* Read from standard input */ {
    fd = STDIN;
    sprintf (tmpName, ".rev.%d",getpid ()); /* Random name */
    /* Create temporary file to store copy of input */
    tmpfd = open (tmpName, O_CREAT | O_RDWR, 0600);
    if (tmpfd == -1) fatalError ();
  }
  else /* Open named file for reading */ {
    fd = open (fileName, O_RDONLY);
    if (fd == -1) fatalError ();
  }

  lineStart[0] = 0; /* Offset of first line */
```

```
  while (TRUE) /* Read all input */ {
    /* Fill buffer */
    charsRead = read (fd, buffer, BUFFER_SIZE);
    if (charsRead == 0) break; /* EOF */
    if (charsRead == -1) fatalError (); /* Error */
    trackLines (buffer, charsRead); /* Process line */
    /* Copy line to temporary file if reading from stdin */
    if (standardInput) {
      charsWritten = write (tmpfd, buffer, charsRead);
      if(charsWritten != charsRead) fatalError ();
    }
  }

  /* Store offset of trailing line, if present */
  lineStart[lineCount + 1] = fileOffset;

  /* If reading from standard input, prepare fd for pass2 */
  if (standardInput) fd = tmpfd;
}
```

```
/****************************************************************/

void trackLines (char* buffer, int charsRead) {
/* Store offsets of each line start in buffer */
  int i;

  for (i = 0; i < charsRead; i++) {
    ++fileOffset; /* Update current file position */
    if (buffer[i] == '\n') lineStart[++lineCount] = fileOffset;
  }
}

/****************************************************************/

void pass2 () {
/* Scan input file again, displaying lines in reverse order */
  int i;

  for (i = lineCount - 1; i >= 0; i--)
    processLine (i);

  close (fd); /* Close input file */
  if (standardInput) unlink (tmpName); /* Remove temp file */
}
```

```
/****************************************************************/

void processLine (int i) {
/* Read a line and display it */
  int charsRead;
  char buffer [BUFFER_SIZE];

  lseek (fd, lineStart[i], SEEK_SET); /* Find the line and read it */
  charsRead = read (fd, buffer, lineStart[i+1] - lineStart[i]);
  /* Reverse line if -c option was selected */
  if (charOption) reverseLine (buffer, charsRead);
  write (1, buffer, charsRead); /* Write it to standard output */
}

/****************************************************************/
```

```c
void reverseLine (char* buffer, int size) {
/* Reverse all the characters in the buffer */
  int start = 0, end = size - 1;
  char tmp;

  if (buffer[end] == '\n') --end; /* Leave trailing newline */

  /* Swap characters in a pairwise fashion */
  while (start < end) {
    tmp = buffer[start];
    buffer[start] = buffer[end];
    buffer[end] = tmp;
    ++start; /* Increment start index */
    --end; /* Decrement end index */
  }
}

/****************************************************************/

void fatalError () {
  perror ("reverse: "); /* Describe error */
  exit (1);
}
```