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 */
}

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

```
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);
}
```

Second Example: monitor.c
uses 3 new advanced system calls:

```
  stat():     obtains status information about a file
  fstat():    similar to stat
  getdents(): obtains directory entries
```

monitor.c program allows the user to monitor a collection of files and obtain information whenever any of them are modified.

```
  % monitor [-t delay] [-l count] {fileName}+
```

scans all specified files every delay seconds and displays information about any of the files that were modified since the last scan.

if fileName is a directory, all of the files inside the directory are scanned.

file modification is indicated as follows:

```
  ADDED   : indicates that the file was created since the last scan
            every file is given this label in the first scan
  CHANGED : indicates that the file was modified since the last scan
  DELETED : indicates that the file was deleted since the last scan
```

By default monitor will scan forever unless overridden by [-l count] option (in which case it scans only count times)
The default delay between scans is 10 seconds which can be overridden by the [-t delay] option.

Algorithm + Data Structure:

```
  struct statStruct {
   char   fileName[MAX_FILENAME];   /* File name */
   int    lastCycle, thisCycle;     /* To detect changes */
   struct stat status;              /* Information from stat () */
  };


  /* Globals */
  char* fileNames [MAX_FILES]; /* One per file on command line */
  int fileCount; /* Count of files on command line */
  struct statStruct stats [MAX_FILES]; /* One per matching file */
  int loopCount = DEFAULT_LOOP_COUNT; /* Number of times to loop */
  int delayTime = DEFAULT_DELAY_TIME; /* Seconds between loops */

  - monitor program continually scans the specified files/directory
  - It uses stat() to get information (type and last modification time)
  - It calls getdents() to scan directories
  - If the file is not in the scan table, it is ADDED
    if the file is already in scan table and has been modified, then
      MODIFIED message
    At the end of a scan, if a file is not present in current scan
     but was present in prevous scan, it is marked DELETED
```

```
System call:

  int stat(const char* name, struct stat* buf)

  stat() fill the buffer buf with information about file name
  The stat structure is defined in /usr/include/sys/stat.h
      and includes the following fields:

  st_dev   device number
  st_ino   the inode number
  st_mode  the permission flags
  st_nlink the hard-link count
  st_uid   the user ID
  st_gid   the group ID
  st_size  the file size
  st_atime the last access time
  st_mtime the last modification time
  st_ctime the last status-change time

  macros:
  S_ISDIR(mode) returns true if file is a directory
  S_ISCHR(mode) returns true if file is a character special device
  S_ISBLK(mode) returns true if file is a block special device
  S_ISREG(mode) returns true if file is a regular file
  S_ISFIFO(mode) returns true if file is a pipe
```

lstat() : same as stat() except it returns information about the symbolic link itself rather than the file it refers to

fstat() : same as stat() except it takes file descriptor as first parameter

All return 0 if successful and -1 otherwise

```
ex. result = stat(fileName, &statBuf);

    if (S_ISDIR(statBuf.st_mode))
          processDirectory(fileName);
```

processDirectory function applies monitorFile() recursively to each of the entries in the directory

if the file is a regular file, character special file or block special file, it calls updateStat() which either adds or updates the file's status entry; if status has changes, updateEntry() is called to display file's new status

the decoding of time is done using localtime() and asctime()

Reading Directory Information: getdents()

```
int getdents(int fd, struct dirent* buf, int structSize)
```

reads the directory file with descriptor fd from its current position and fills structure pointed to by buf with the next entry. The dirent struct is defined in /usr/include/sys/dirent.h and contains the following fields:

d_ino	: the inode number
d_off	: the offset of the next directory entry
d_reclen	: length of the dirent struct
d_name	: the fileName

```
returns length of the directory entry if successful
        0 if last directory entry has already been read
        -1 if error
```

processDirectory function skips . and .. and uses lseek to advance to the next directory entry

monitor.c

```
#include <stdio.h>           /* For printf, fprintf */
#include <string.h>          /* For strcmp */
#include <ctype.h>           /* For isdigit */
#include <fcntl.h>           /* For O_RDONLY */
#include <dirent.h>      /* For getdents */
#include <sys/stat.h>        /* For IS macros */
#include <sys/types.h>       /* For modet */
#include <time.h>            /* For localtime, asctime */
```

```
void parseCommandLine (int argc, char* argv[]);
int findEntry (char* fileName);
void fatalError ();
void usageError ();
void processOptions (char* str);
int getNumber (char* str, int* i);
void monitorLoop ();
void monitorFiles ();
void monitorFile (char* fileName);
void processDirectory (char* dirName);
void updateStat (char* fileName, struct stat* statBuf);
int findEntry (char* fileName);
int addEntry (char* fileName, struct stat* statBuf);
int nextFree ();
void updateEntry (int index, struct stat* statBuf);
void printEntry (int index);
void printStat (struct stat* statBuf);
void fatalError ();
```

```
/* #define Statements */
#define MAX_FILES           100
#define MAX_FILENAME        50
#define NOT_FOUND           -1
#define FOREVER             -1
#define DEFAULT_DELAY_TIME   10
#define DEFAULT_LOOP_COUNT   FOREVER

/* Booleans */
enum { FALSE, TRUE };

/* Status structure, one per file. */
struct statStruct {
   char fileName [MAX_FILENAME]; /* File name */
   int lastCycle, thisCycle; /* To detect changes */
   struct stat status; /* Information from stat () */
};

/* Globals */
char* fileNames [MAX_FILES]; /* One per file on command line */
int fileCount; /* Count of files on command line */
struct statStruct stats [MAX_FILES]; /* One per matching file */
int loopCount = DEFAULT_LOOP_COUNT; /* Number of times to loop */
int delayTime = DEFAULT_DELAY_TIME; /* Seconds between loops */
int main (int argc, char* argv[]) {
 parseCommandLine (argc, argv); /* Parse command line */
 monitorLoop (); /* Execute main monitor loop */
 return (/* EXIT_SUCCESS */ 0);
}

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

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

 int i;

 for (i = 1; ( (i < argc) && (i < MAX_FILES) ); i++) {
   if (argv[i][0] ==  '-')
     processOptions (argv[i]);
   else
     fileNames[fileCount++] = argv[i];
 }
 if (fileCount == 0) usageError ();
}

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

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

 for (j = 1; str[j] != ’\0’; j++) {
   switch(str[j]) /* Switch on option letter */ {
     case ’t’:
       delayTime = getNumber (str, &j);
       break;
     case ’l’:
       loopCount = getNumber (str, &j);
       break;
   }
 }
}

int getNumber (char* str, int* i) {
/* Convert a numeric ASCII option to a number */
  int number = 0;
  int digits = 0; /* Count the digits in the number */

  while (isdigit (str[(*i) + 1])) /* Convert chars to ints */ {
    number = number * 10 + str[++(*i)] - ’0’;
    ++digits;
  }
  if (digits == 0) usageError (); /* There must be a number */
  return (number);
}
```

```
void usageError () {
  fprintf (stderr, "Usage: monitor -t<seconds> -l<loops> {filename}+\n");
  exit (/* EXIT_FAILURE */ 1);
}

void monitorLoop () {
/* The main monitor loop */
  do {
    monitorFiles (); /* Scan all files */
    fflush (stdout); /* Flush standard output */
    fflush (stderr); /* Flush standard error */
    sleep (delayTime); /* Wait until next loop */
  } while (loopCount == FOREVER || --loopCount > 0);
}

void monitorFiles () {
/* Process all files */
  int i;

  for (i = 0; i < fileCount; i++)
    monitorFile (fileNames[i]);

  for (i = 0; i< MAX_FILES; i++) /* Update stat array */ {
    if (stats[i].lastCycle && !stats[i].thisCycle)
      printf ("DELETED %s\n", stats[i].fileName);
    stats[i].lastCycle = stats[i].thisCycle;
    stats[i].thisCycle = FALSE;
  }
}
```

```
void monitorFile (char* fileName) {
/* Process a single file/directory*/
  struct stat statBuf;
  mode_t mode;
  int result;

  result = stat (fileName, &statBuf); /* Obtain file status */

  if (result == -1) /* Status was not available */ {
    fprintf (stderr, "Cannot stat %s\n", fileName);
    return;
  }

  mode = statBuf.st_mode; /* Mode of file */

  if(S_ISDIR (mode)) /* Directory */
    processDirectory (fileName);
  else if (S_ISREG (mode) || S_ISCHR (mode) || S_ISBLK (mode))
    updateStat (fileName, &statBuf); /* Regular file */
}
```

```c
void processDirectory (char* dirName) {
/* Process all files in the named directory */
  DIR *dp;
  struct dirent *dep;
  char fileName [MAX_FILENAME];

  dp = opendir (dirName); /* Open for reading */
  if (dp == NULL) fatalError ();

  while (dep = readdir(dp)) /* Read all directory entries */ {
    if (strcmp (dep->d_name, ".") != 0&&
        strcmp (dep->d_name, "..") != 0) /* Skip . and .. */ {
      sprintf (fileName, "%s/%s", dirName, dep->d_name);
      monitorFile (fileName);  /* Call recursively */
    }
  }
  closedir (dp); /* Close directory */
}
```

```
void updateStat (char* fileName, struct stat* statBuf) {
/* Add a status entry if necessary */
  int entryIndex;

  entryIndex = findEntry (fileName); /* Find existing entry */

  if (entryIndex == NOT_FOUND)
    entryIndex = addEntry (fileName, statBuf); /* Add new entry */
  else
    updateEntry (entryIndex, statBuf); /* Update existing entry */

  if (entryIndex != NOT_FOUND)
    stats[entryIndex].thisCycle = TRUE; /* Update status array */
}

int findEntry (char* fileName) {
/* Locate the index of a named filein the status array */
  int i;

  for (i = 0; i < MAX_FILES; i++)
    if (stats[i].lastCycle &&
        strcmp (stats[i].fileName, fileName) == 0) return (i);

  return (NOT_FOUND);
}
```

```
int addEntry (char* fileName, struct stat* statBuf) {
/* Add a new entry into the status array */
  int index;

  index = nextFree (); /* Find the next free entry */
  if (index == NOT_FOUND) return (NOT_FOUND); /* None left */
  strcpy (stats[index].fileName, fileName); /* Add filename */
  stats[index].status = *statBuf; /* Add status information */
  printf ("ADDED "); /* Notify standard output */
  printEntry (index); /* Display status information */
  return (index);
}

int nextFree () {
/* Return the nextfree index in the status array */
  int i;

  for (i = 0; i < MAX_FILES; i++)
    if (!stats[i].lastCycle && !stats[i].thisCycle) return (i);

  return (NOT_FOUND);
}
```

```
void updateEntry (int index, struct stat* statBuf) {
/*Display information if the file has been modified */
  if (stats[index].status.st_mtime != statBuf->st_mtime) {
    stats[index].status = *statBuf; /* Store stat information */
    printf ("CHANGED "); /* Notify standard output */
    printEntry (index);
  }
}

void printEntry (int index) {
/* Display an entry of the status array */
  printf ("%s ", stats[index].fileName);
  printStat (&stats[index].status);
}

void printStat (struct stat* statBuf) {
/* Display a status buffer */
  printf ("size %lu bytes, mod. time = %s",  statBuf->st_size,
           asctime (localtime (&statBuf->st_mtime)));
}

void fatalError () {
  perror ("monitor: ");
  exit (/* EXIT_FAILURE */ 1);
}
```

Miscellaneous File Management System Calls

```
int chown (const char* fileName, uid_t ownerId, gid_t groupId)
int lchown (const char* fileName, uid_t ownerId, gid_t groupId)
int chown (int fd, uid_t ownerId, gid_t groupId)
```

These cause the owner and group IDs of fileName to be set to ownerId and groupId respectively (a value of -1 indicates that the ID should not change)

```
int main() {
  int flag;
  flag = chown("test.txt", -1, 62);
  if (flag == -1) perror("error changing group ID for test.txt");
}
```

62 is the groupID for group name (see /etc/group for group IDs)

```
int chmod (const char* fileName, int mode)
int fchmod (int fd, int mode)

these change the mode of fileName to mode (specified as octal;
        ex. 0600)
set UID and set GID flags have values 04000 and 02000 respectively

int main() {
  int flag;
  flag = chmod("test.txt",0600);
  if (flag == -1) perror("problem setting mode");
}
```

Duplicating a file descriptor: dup() and dup2()

```
int dup (int oldFd);
int dup2 (int oldFd, int newFd);
```

dup() finds the smallest file descriptor entry and points it to the same file oldFd points to.

dup2() closes newFd if it is currently active and points it to the same file which oldFd points to.

In both cases, the original and copied file descriptors share the same file pointer and access mode.

They both return the index of the new file descriptor if successful and a value of -1 otherwise.

```
#include <stdio.h>
#include <fcntl.h>
int main() {
  int fd1, fd2, fd3;
  fd1 = open("test.txt", O_RDWR | O_TRUNC | O_CREAT);
  printf("fd1 = %d\n", fd1);
  write(fd1,"what’s",6);
  fd2 = dup(fd1);
  printf("fd2 = %d\n", fd2);
  write(fd2," up",3);
  close(0);
  fd3 = dup(fd1);
  printf("fd3 = %d\n", fd3);
  write(fd3," doc",4);
  dup2(3,2);
  write(2,"?\n",2);
}
```

File Descriptor Operations:

```
int fcntl (int fd, int cmd, int arg)
```

performs the operation encoded in cmd on the file associated with descriptor fd; arg is an optional argument for cmd.

```
cmd = F_SETFL  sets the current file-status flags to arg
cmd = F_GETFL  returns a number corresponding to the current
               file-status flags and mode
etc. see P 417 of text
```

```
ex. fcntl(fd, F_SETFL, O_WRONLY | O_APPEND);
```

PROCESS MANAGEMENT:

Every process in UNIX has

- some code
- some data
- a stack
- a unique process ID (PID)

When UNIX starts (boots), there is only one process, called init, with process ID = 1.

The only way to create a new process is to duplicate an existing process; So init is the ancestor of all subsequent processes.

Initially, init duplicates (forks) several times and each child process replaces its code (execs) with the code of the executable getty which is responsible for user logins.

It is very common for a parent process to suspend itself until one of its child processes terminates. For example:

Consider the way a shell process executes a utility in the foreground:

- The shell first duplicates itself using fork()
- The child shell process replaces its code with that of the utility using exec(). (Differentiates)
- The parent process waits for the child process to terminate using wait()
- When the child process terminates using exit(), the parent process is informed (using a signal) and the parent process presents the user with the shell prompt to accept the next command.

System calls:

fork(), getpid(), getppid(), exit(), wait(), exec()

pit_t fork(void)

causes a process to duplicate. The child process is almost exactly a duplicate of the parent process; it inherits a copy of its parent's code, data, stack, open file descriptors, and signal tables. The only difference is in the process IDs (pid) and parent process IDs (ppid)

If fork() succeeds, it returns a value of 0 to the child process and the PID of the child process to the parent.

If fork() fails, it returns a -1 to the parent process and the child process is not created.

```
 pid_t getpid(void)
 pid_t getppid(void)
```

return the process’ ID and parent process ID respectively.
They always succeed. (The PPID value for process with ID=1 is 1)

```
#include <stdio.h>
main () {
 int pid;
 printf ("I'm the original process with PID %d and PPID %d.\n",
         getpid (), getppid ());
 pid = fork (); /* Duplicate. Child and parent continue from here */
 if (pid != 0) /* pid is non-zero, so I must be the parent */ {
   printf ("I'm the parent process with PID %d and PPID %d.\n",
            getpid (), getppid ());
   printf ("My child's PID is %d\n", pid);
 }
 else /* pid is zero, so I must be the child */ {
   printf ("I'm the child process with PID %d and PPID %d.\n",
            getpid (), getppid ());
 }
 printf ("PID %d terminates.\n", getpid () ); /* Both processes execute this */
}
```

```
$ myfork
I'm the original process with PID 639 and PPID 416.
I'm the parent process with PID 639 and PPID 416.
My child's PID is 640
PID 639 terminates.
I'm the child process with PID 640 and PPID 1.
PID 640 terminates.
$
```

Orphan Processes: If a parent process terminates before its child terminates, the child process is automatically adopted by the init process;

```
#include <stdio.h>
main () {
 int pid;
 printf ("I'm the original process with PID %d and PPID %d.\n",
         getpid (), getppid ());
 pid = fork (); /* Duplicate. Child and parent continue from here */
 if (pid != 0) /* Branch based on return value from fork () */ {
   /* pid is non-zero, so I must be the parent */
   printf ("I'm the parent process with PID %d and PPID %d.\n", getpid (), getppid ());
   printf ("My child's PID is %d\n", pid);
 }
 else {
   /* pid is zero, so I must be the child */
   sleep (5); /* Make sure that the parent terminates first */
   printf ("I'm the child process with PID %d and PPID %d.\n", getpid (), getppid ());
 }
 printf ("PID %d terminates.\n", getpid () ); /* Both processes execute this */
}
```

```
$ orphan
I'm the original process with PID 680 and PPID 416.
I'm the parent process with PID 680 and PPID 416.
My child's PID is 681
PID 680 terminates.
$ I'm the child process with PID 681 and PPID 1.
PID 681 terminates.
```

Terminating a Process: exit()

```
  void exit(int status)
```

closes all of a process’ file descriptors, deallocates its code, data, and stack; then terminates the process.

When a child process terminates, it sends its parent a SIGCHLD signal and waits for its termination code (status) to be accepted. (only lower 8 bits of status are used; so value 0-255).

A process which is waiting for its parent process to accept its return code is called a zombie process.

A parent accepts a child’s termination code by executing wait()

The kernel makes sure that all of a terminating process’ children are adopted by init by setting their PPID’s to 1;

init always accepts its children’s termination code.

exit() never returns

```
% cat myexit.c
#include <stdio.h>
main() {
  printf("I am going to exit with status code 42\n");
  exit(42);
}

% myexit
I am going to exit with status code 42
% echo $status
42
```

Zombie Process:

A process that cannot leave the system (even if it has exit-ed) until its parent process accepts its termination code.
(If parent process is dead; init adopts process and accepts code).
If the parent process is alive but is unwilling to accept the child's termination code (because it never executes wait()); the child process will remain a zombie process.

Zombie processes do not take up system resources but do use up one entry in the system's fixed-size process table. Too many zombie processes is a problem.

```
#include <stdio.h>
main () {
 int pid;
  pid = fork (); /* Duplicate */
 if (pid != 0) /* Branch based on return value from fork () */ {
     while (1)  /* Never terminate, and never execute a wait () */
       sleep (1000);
 }
 else {
   exit (42); /* Exit with a silly number */
 }
}

$ zombie &
[1] 684
$ ps
  684  p4 S    0:00 zombie
  685  p4 Z    0:00 (zombie <zombie>)
  686  p4 R    0:00 ps
$ kill 684
[1]+  Terminated              zombie
$ ps
  688  p4 R    0:00 ps
$
```

Waiting for a Child: wait()

```
pid_t wait(int *status)
```

causes a process to suspend until one of its child processes terminates. A successful call to wait() returns the PID of the child process that terminated abd places a status code into status that is encoded as follows:

if the rightmost byte of status is zero, the leftmost byte contains the low 8 bits of the value returned by the child's exit() or return() call

if the rightmost byte of status is non-zero, the rightmost 7 bits are equal to the Signal Number that caused the child to terminate and the last bit is set to 1 if the child core dumped.

If a process executes a wait() and has no children, wait() returns immediately with a value of -1

If a process executes a wait() and one or more of its children are already zombies, wait() returns immediately with the status of one of the zombies.

```
#include <stdio.h>
main () {
 int pid, status, childPid;
 printf ("I'm the parent process and my PID is %d\n", getpid ());
 pid = fork (); /* Duplicate */
 if (pid != 0) /* Branch based on return value from fork () */ {
     printf ("I'm the parent process with PID %d and PPID %d\n",
              getpid (), getppid ());
     childPid = wait (&status); /* Wait for a child to terminate. */
     printf ("A child with PID %d terminated with exit code %d\n",
             childPid, status >> 8);
 }
 else {
     printf ("I'm the child process with PID %d and PPID %d\n",
              getpid (), getppid ());
     exit (42); /* Exit with a silly number */
 }
  printf ("PID %d terminates\n", getpid () );
}
```

```
$ mywait
I'm the parent process and my PID is 695
I'm the parent process with PID 695 and PPID 418
I'm the child process with PID 696 and PPID 695
A child with PID 696 terminated with exit code 42
PID 695 terminates
```

Differentiating a Process: exec()

A process may replace its current code, data, and stack with those of another executable by using one of the exec() family of system calls.

When a process executes exec() its PID and PPID stay the same; only the code that the process is executing changes.

```
  int execl(const char* path, const char* arg0, ...,
                              const char* argn, NULL)
  int execv(const char* path, const char* argv[])
```

The following two use $PATH variable to find the executable:

```
  int execlp(const char* path, const char* arg0, ...,
                               const char* argn, NULL)
  int execvp(const char* path, const char* argv[])
```

path: executable (relative or absolute path for execl/execv
name of executable for execlp/execvp)

argi or argv[i]:
ith command line argument for executable (arg0: name of executable)

If executable is not found, -1 is returned otherwise the calling process replaces its code, data, and stack with those of the executable and starts executing the new code.

A successful exec() never returns.

```
#include <stdio.h>
main () {
 printf ("I’m process %d and I’m about to exec an ls -l\n", getpid ());
 execl ("/bin/ls", "ls", "-l", NULL); /* Execute ls */
 printf ("This line should never be executed\n");
}

$ myexec
I’m process 710 and I’m about to exec an ls -l
total 38
-rw-rw-r--   1 raj      raj           187 Jul 22 20:24 alarm.c
-rw-rw-r--   1 raj      raj           226 Jul 22 20:22 background.c
-rw-rw-r--   1 raj      raj           284 Jul 22 20:22 mychdir.c
-rw-rw-r--   1 raj      raj          2058 Jul 22 20:23 vount.c
-rwxrwxr-x   1 raj      raj          4174 Jul 24 12:08 zombie
-rw-rw-r--   1 raj      raj           298 Jul 22 20:20 zombie.c
```

Changing Directories: chdir()

Every process has a current working directry that is used when processing a relative path name.

A child process inherits the current working directory from its parent; e. when a utility is run from a shell, the utility process inherits the shell's current working directory.

```
int chdir(const char* pathname)
```

sets a process' current working directory to pathname. The process must have execute permission from the directory for chdir() to succeed.

chdir() returns -1 if it fails; 0 if it succeeds

```
#include <stdio.h>
main () {
 system ("pwd"); /* Display current working directory */
 chdir ("/"); /* Change working directory to root directory */
 system ("pwd"); /* Display new working directory */
 chdir ("/home/naveen"); /* Change again */
 system ("pwd"); /* Display again */
}

$ mychdir
/home/raj/3320/ch12
/
/home/naveen
$
```

Changing Priorities: nice()

Child process inherits the priority of the parent process and can change the priority using nice()

```
  int nice(int delta)
```

adds delta to the priority (legal values of priority -20 to +19)
Only super user can change to negative priority.

Smaller the priority value, faster the process will run.

Getting and Setting User and Group Ids:

```
  uid_t getuid()              uid_t setuid(uid_t id)
  uid_t geteuid()             uid_t seteuid(uid_t id)
  gid_t getgid()              uid_t setgid(gid_t id)
  gid_t getegid()             uid_t setegid(gid_t id)
```

The get functions always succeed; The set methods will succeed only when executed by super user or if id equals real or effective id of the process.

Program which uses fork() and exec() to execute a program in the background.

The original process creates a child process to exec the specified executable and terminates.

The orphaned child process is adopted by init.

```
#include <stdio.h>
main (int argc, char* argv[]) {
 if (fork () == 0) /* Child */ {
     execvp (argv[1], &argv[1]); /* Execute other program */
     fprintf (stderr, "Could not execute %s\n", argv[1]);
 }
}
```

```
$ background gcc mywait.c
```