```
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 ();
```

```c
/* #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 */
}
```

```
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.

```c
#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);
```