ΕΡΓΑΣΤΗΡΙΟ 5

Αρχεία και κατάλογοι

Να πληκτρολογηθεί και να εκτελεστεί ο κώδικας των επόμενων παραδειγμάτων που επιδεικνύουν τη χρήση των συναρτήσεων διαχείρισης αρχείων και καταλόγων.

Παράδειγμα 1. Χρήση της stat για την εμφάνιση των πληροφοριών του i-node και για αρχείο που διαβιβάζεται ως όρισμα στη γραμμή εντολών (αρχείο filestat.c).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  struct stat sb;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **filestat name**, όπου name το όνομα ενός αρχείου ή καταλόγου. Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με 2 (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 2, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
  if (argc != 2) {
     fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
     exit(EXIT_FAILURE); }
```

Η λήψη των ζητούμενων πληροφοριών γίνεται με την κλήση της συνάρτησης stat η οποία εάν κληθεί με επιτυχία επιστρέφει στη δομή **struct stat sb** τις τιμές των ιδιοτήτων του αρχείου. Εάν η επιστρεφόμενη τιμή είναι -1, έχει ανακύψει κάποιο σφάλμα και η συνάρτηση τερματίζει τη λειτουργία της.

```
  if (stat(argv[1], &sb) == -1) {
     perror("stat");
     exit(EXIT_FAILURE); }
```

Αρχικά εξετάζουμε την τιμή του πεδίου **st_mode** η οποία ορίζει τον τύπο του αρχείου με τον ακόλουθο τρόπο: εάν είναι ίση με **S_IFBKL** το αρχείο είναι αρχείο παράλληλης συσκευής (block device file), εάν είναι ίση με **S_IFCHR** το αρχείο είναι αρχείο σειριακής συσκευής (character device file), εάν είναι **S_IFDIR** το αρχείο κατάλογος (directory), εάν είναι **S_IFIFO** το αρχείο είναι αρχείο επώνυμου αγωγού (FIFO/pipe), εάν είναι **S_IFLNK** το αρχείο είναι συμβολικός σύνδεσμος (symbolic link), εάν είναι **S_IFREG** το αρχείο είναι σύνηθες αρχείο (regular file), ενώ εάν είναι **S_IFSOCK** το αρχείο είναι αρχείο δικτυακού υποδοχέα (socket). Ο έλεγχος της τιμής του πεδίου st_mode και η εκτύπωση του κατάλληλου μηνύματος, γίνεται με την case / switch.

```
printf("File type:                ");
  switch (sb.st_mode & S_IFMT) {
   case S_IFBLK:  printf("block device\n");          break;
   case S_IFCHR:  printf("character device\n");      break;
   case S_IFDIR:  printf("directory\n");             break;
   case S_IFIFO:  printf("FIFO/pipe\n");             break;
   case S_IFLNK:  printf("symlink\n");               break;
   case S_IFREG:  printf("regular file\n");          break;
   case S_IFSOCK: printf("socket\n");                break;
   default:       printf("unknown?\n");              break; }
```

Στη συνέχεια εκτυπώνουμε τις τιμές των υπόλοιπων πεδίων της δομής που περιλαμβάνουν: την τιμή του i-node (το πεδίο **st_ino**), την τιμή της κατάστασης (το πεδίο **st_mode**) στην οποία τα δώδεκα τελευταία bits περιέχουν τη μάσκα δικαιωμάτων

ενώ τα υπόλοιπα τον τύπο του αρχείου (σύνηθες αρχείο ή κατάλογος) και τον τροποποιητή (τα τρία ειδικά bits), το πλήθος των συνδέσμων προς το αρχείο (το πεδίο **st_link**), το UID και το GID του αρχείου (δηλαδή τον κωδικό του κατόχου και της ομάδας του κατόχου του αρχείου) (τα πεδία **st_uid** και **st_gid**), το (προτιμώμενο) μέγεθος του block δεδομένων που μεταφέρεται από ή προς το δίσκο σε ένα βήμα κατά την πραγματοποίηση μιας διαδικασίας ανάγνωσης ή εγγραφής (το πεδίο **st_blksize**), το μέγεθος του αρχείου (**st_size**), το πλήθος των blocks στο δίσκο που δεσμεύονται από το αρχείο (το πεδίο **st_blocks**) , τη χρονική στιγμή της τελευταίας μεταβολής της κατάστασης του αρχείου (το πεδίο **st_ctime**), τη χρονική στιγμή της τελευταίας προσπέλασης του αρχείου (το πεδίο **st_atime**) και τη χρονική στιγμή της τελευταίας τροποποίησης του αρχείου (το πεδίο **st_mtime**).

```
printf("I-node number:          %ld\n", (long) sb.st_ino);
  printf("Mode:                 %lo (octal)\n", (unsigned long) sb.st_mode);
  printf("Link count:           %ld\n", (long) sb.st_nlink);
  printf("Ownership:            UID=%ld   GID=%ld\n", (long) sb.st_uid, (long) sb.st_gid);
  printf("Preferred I/O block size: %ld bytes\n", (long) sb.st_blksize);
  printf("File size:            %lld bytes\n", (long long) sb.st_size);
  printf("Blocks allocated:       %lld\n", (long long) sb.st_blocks);
  printf("Last status change:     %s", ctime(&sb.st_ctime));
  printf("Last file access:       %s", ctime(&sb.st_atime));
  printf("Last file modification:   %s", ctime(&sb.st_mtime));
  exit(EXIT_SUCCESS); }
```

Ένα παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια, για ένα σύνηθες αρχείο (πρώτη εικόνα) και για έναν κατάλογο (δεύτερη εικόνα). Παρατηρήστε τη διαφορά στην πρώτη γραμμή (File type).

```
amarg@amarg-vbox:~/shared/Lab5$ ./filestat ~/sample.c
File type:                regular file
I-node number:            279500
Mode:                     100664 (octal)
Link count:               1
Ownership:                UID=1000   GID=1000
Preferred I/O block size: 4096 bytes
File size:                1238 bytes
Blocks allocated:         8
Last status change:       Thu Oct  8 12:09:40 2020
Last file access:         Tue Nov  3 12:21:45 2020
Last file modification:   Thu Oct  8 12:09:40 2020
amarg@amarg-vbox:~/shared/Lab5$
```

```
amarg@amarg-vbox:~/shared/Lab5$ ./filestat /bin
File type:                directory
I-node number:            917523
Mode:                     40755 (octal)
Link count:               2
Ownership:                UID=0   GID=0
Preferred I/O block size: 4096 bytes
File size:                40960 bytes
Blocks allocated:         80
Last status change:       Wed Nov  4 11:05:01 2020
Last file access:         Wed Nov  4 11:05:09 2020
Last file modification:   Wed Nov  4 11:05:01 2020
```

Παράδειγμα 2. Έλεγχος της ύπαρξης και των δικαιωμάτων πρόσβασης του αρχείου με τη χρήση της access (αρχείο testAccess.c).

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

int main (int argc, char* argv[]) {
 char* path = argv[1];
 int rval;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **testAccess name**, όπου name το όνομα ενός αρχείου ή καταλόγου. Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με 2 (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 2, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του **(αυτός ο έλεγχος ΔΕΝ ΓΙΝΕΤΑΙ στο αρχείο του eClass, προσθέστε το μόνοι σας)**.

```
  if (argc != 2) {
     fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
     exit(EXIT_FAILURE); }
```

Καταρχήν θα πρέπει να ελέγξουμε εάν το αρχείο υπάρχει, αλλιώς δεν έχει νόημα να συνεχίσουμε. Αυτό γίνεται καλώντας τη συνάρτηση access με το όρισμα **F_OK**. Εάν η access επιστρέψει την τιμή 0 αυτό σημαίνει πως το αρχείο υπάρχει και μπορούμε να συνεχίσουμε, ενώ στην αντίθετη περίπτωση εξετάζουμε την τιμή της μεταβλητής errno. Εάν αυτή τιμή είναι ίση με **ENOENT** αυτό σημαίνει πως το αρχείο δεν υπάρχει, ενώ εάν είναι ίση με **EACCESS**, αυτό σημαίνει πως το αρχείο υπάρχει αλλά δεν είναι προσβάσιμο –σε κάθε περίπτωση η λειτουργία της εφαρμογής τερματίζεται.

```
 rval = access (path, F_OK);
 if (rval == 0)
    printf ("%s exists\n", path);
 else {
    if (errno == ENOENT)
        printf ("%s does not exist\n", path);
    else if (errno == EACCES)
         printf ("%s is not accessible\n", path);
  return 0;  }
```

Σε αυτό το σημείο εξετάζουμε εάν έχουμε δικαίωμα ανάγνωσης κάτι που γίνεται καλώντας τη συνάρτηση access με το όρισμα **R_OK**. Εάν η access επιστρέψει την τιμή 0 αυτό σημαίνει πως έχουμε δικαίωμα ανάγνωσης του αρχείου, κάτι που δε ισχύει στην αντίθετη περίπτωση. Η εφαρμογή εκτυπώνει το κατάλληλο μήνυμα.

```
 rval = access (path, R_OK);
 if (rval == 0) printf ("%s is readable\n", path);
 else printf ("%s is not readable (access denied)\n", path);
```

Σε αυτό το σημείο εξετάζουμε εάν έχουμε δικαίωμα εγγραφής κάτι που γίνεται καλώντας τη συνάρτηση access με το όρισμα **W_OK**. Εάν η access επιστρέψει την τιμή 0 αυτό σημαίνει πως έχουμε δικαίωμα εγγραφής του αρχείου, κάτι που δε ισχύει στην αντίθετη περίπτωση, όπου εξετάζουμε τι ακριβώς συμβαίνει (είτε το συγκεκριμένο αρχείο δεν είναι εγγράψιμο (**EACCESS**) είτε όλο το σύστημα αρχείων είναι μόνο για ανάγνωση (**EROFS**)). Η εφαρμογή εκτυπώνει το κατάλληλο μήνυμα.

```
 rval = access (path, W_OK);
 if (rval == 0) printf ("%s is writable\n", path);
 else if (errno == EACCES) printf ("%s is not writable (access denied)\n", path);
 else if (errno == EROFS) printf ("%s is not writable (read-only filesystem)\n", path);
 return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Για την επίδειξη της εφαρμογής δημιουργούμε τέσσερα αρχεία στα οποία ορίζουμε όλους τους δυνατούς συνδυασμούς δικαιωμάτων ανάγνωσης και εγγραφής για τον κάτοχο (δηλαδή - -, - w, r - , r w) και στη συνέχεια καλούμε την εφαρμογή acc1 για να ανακτήσουμε τα παραπάνω δικαιώματα πρόσβασης, με τα αποτελέσματα φυσικά να βρίσκονται σε πλήρη συμφωνία με αυτά που προκύπτουν από τη μελέτη της μάσκας δικαιωμάτων.

```
amarg@amarg-vbox:~$ chmod 144 file1.doc
amarg@amarg-vbox:~$ chmod 344 file2.doc
amarg@amarg-vbox:~$ chmod 544 file3.doc
amarg@amarg-vbox:~$ chmod 744 file4.doc
amarg@amarg-vbox:~$ ls -l *.doc
- -- xr--r-- 1 amarg amarg    1521 Σεπ  20 19:58 file1.doc
- -w xr--r-- 1 amarg amarg 7959821 Σεπ  20 19:58 file2.doc
- r- xr--r-- 1 amarg amarg    2797 Σεπ  20 19:59 file3.doc
- rw xr--r-- 1 amarg amarg     550 Σεπ  20 19:59 file4.doc
amarg@amarg-vbox:~$ ./acc1 file1.doc
file1.doc exists
file1.doc is not readable (access denied)
file1.doc is not writable (access denied)
amarg@amarg-vbox:~$ ./acc1 file2.doc
file2.doc exists
file2.doc is not readable (access denied)
file2.doc is writable
amarg@amarg-vbox:~$ ./acc1 file3.doc
file3.doc exists
file3.doc is readable
file3.doc is not writable (access denied)
amarg@amarg-vbox:~$ ./acc1 file4.doc
file4.doc exists
file4.doc is readable
file4.doc is writable
amarg@amarg-vbox:~$
```

Παράδειγμα 3. Αλλαγή δικαιωμάτων πρόσβασης του αρχείου με την εντολή chmod (αρχείο testChmod.c).

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

int main (int argc, char * argv[]) {
  int fd;
  struct stat info;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **testChmod name**, όπου name το όνομα ενός αρχείου ή καταλόγου. Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με 2 (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 2, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
  if (argc!=2) {
     printf ("Usage testChmod pathname\n");
     return (-1); }
```

Προκειμένου να γίνει κατανοητός ο τρόπος λειτουργίας της εφαρμογής, το πρόγραμμα ανακτά την παλαιά μάσκα δικαιωμάτων έτσι ώστε αυτή να εκτυπωθεί μαζί με τη νέα μάσκα δικαιωμάτων δίδοντας με τον τρόπο αυτό την ευκαιρία στο χρήστη να εντοπίσει τις αλλαγές που πραγματοποιήθηκαν. Αρχικά το πρόγραμμα καλεί τη συνάρτηση **access** που χρησιμοποιήσαμε και προηγουμένως για να ελέγξει εάν το αρχείο ή ο κατάλογος υπάρχει. Εάν η συνάρτηση επιστρέψει την τιμή -1, το εξεταζόμενο αντικείμενο δεν υπάρχει και η εφαρμογή εκτυπώνει το κατάλληλο μήνυμα σφάλματος και τερματίζει.

```
fd = access(argv[1], F_OK);
If (fd == -1){

    printf("Error Number : %d\n", errno);
    perror("Error Description");
    return (-2); }
```

Στην αντίθετη περίπτωση, κατά την οποία το αντικείμενο που όρισε ο χρήστης είναι ένα υπάρχον αντικείμενο, το πρόγραμμα καλεί τη συνάρτηση **stat** που είδαμε στην πρώτη άσκηση, η οποία επιστρέφει στη δομή info τα περιεχόμενα του i-node για το εν λόγω αντικείμενο (αρχείο ή κατάλογος). Η μάσκα δικαιωμάτων περιλαμβάνεται στο πεδίο **st_mode**, η τιμή του οποίου εκτυπώνεται στο **οκταδικό** σύστημα με **8** ψηφία όπως υπαγορεύεται από τον προσδιοριστή **%08o** που χρησιμοποιούμε στην printf (το %o (ο από τη λέξη octal → οκταδικό) εκτυπώνει την τιμή στο οκταδικό σύστημα ενώ το 08 από μπροστά προκαλεί την εκτύπωση με 8 ψηφία).

```
else {
   stat(argv[1], &info);
   printf("Original file permissions were: %08o\n", info.st_mode);
```

Τέλος, καλούμε τη συνάρτηση **chmod** για την αλλαγή της μάσκας δικαιωμάτων. Η τιμή S_IRWXU εκχωρεί στον κάτοχο του αρχείου (USER, S_IRWX**U**) δικαιώματα ανάγνωσης, εγγραφής και εκτέλεσης (S_I**RWX**U), ενώ η τιμή S_IRWXG εκχωρεί στην ομάδα του κατόχου του αρχείου (GROUP, S_IRWX**G**) δικαιώματα ανάγνωσης, εγγραφής και εκτέλεσης (S_I**RWX**G). Ο συνδυασμός αυτών των δύο τελεστών με τη δυαδική λογική διάζευξη (**|**), εκχωρεί στο αντικείμενο δικαιώματα ανάγνωσης, εγγραφής και εκτέλεσης, τόσο για τον κάτοχο όσο και για την ομάδα του κατόχου. Εάν η συνάρτηση chmod επιστρέψει τιμή διαφορετική του μηδενός αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα το οποίο εκτυπώνεται από τη συνάρτηση perror, ενώ στην αντίθετη περίπτωση καλείται ξανά η stat με τον τρόπο με τον οποίο αυτό έγινε προηγουμένως, προκειμένου να ανακτήσει και να εκτυπώσει τη νέα τιμή της μάσκας δικαιωμάτων του αρχείου.

```
   if (chmod(argv[1], S_IRWXU|S_IRWXG) != 0) perror("chmod() error");
   else {
     stat(argv[1], &info);
     printf("After chmod(), file permissions are: %08o\n", info.st_mode); }}
 return (0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ ls -l | grep results
-rw-rw-r--  1 amarg amarg     262 Σεπ  22 10:08 results
amarg@amarg-vbox:~$ ./testChmod results
Original file permissions were: 00100664
After chmod(), file permissions are: 00100770
amarg@amarg-vbox:~$ ls -l | grep results
-rwxrwx---  1 amarg amarg     262 Σεπ  22 10:08 results
```

Η αρχική μάσκα δικαιωμάτων είναι η **00100664** που διαχωρίζεται ως **0010 0 664**. Το **0010** σημαίνει πως το αντικείμενο είναι κανονικό αρχείο, η τιμή **0** του τροποποιητή σημαίνει πως **δεν έχουν τεθεί** τα ειδικά bits (setuid, setgid και sticky bit) ενώ η μάσκα **664** που στο δυαδικό είναι η 110 110 100 ορίζει τα δικαιώματα **r w – r w – r – –** . Μετά την εκτέλεση της εντολής η νέα μάσκα είναι η **00100770**. Τα bits **0010** και **0** είναι τα ίδια με πριν ενώ τα νέα δικαιώματα πρόσβασης είναι τα **770** δηλαδή **111 111 000** ή **r w x r w x – – –**. Παρατηρήστε πως πλέον **ο κάτοχος και η ομάδα έχουν δικαίωμα ανάγνωσης, εγγραφής και εκτέλεσης** όπως άλλωστε αναμέναμε και πως η αλλαγή που έγινε άλλαξε **ολόκληρη** τη μάσκα. Πράγματι, ενώ στην αρχή οι άλλοι χρήστε είχαν δικαίωμα ανάγνωσης, τώρα δεν έχουν **τίποτε**, αφού η συνάρτηση chmod **δεν όρισε** δικαιώματα για τους άλλους χρήστες και ως εκ τούτου αυτοί έχασαν και αυτά που είχαν προηγουμένως.

Παράδειγμα 4. Αλλαγή κατόχου και ομάδας κατόχου αρχείου με την εντολή chown (αρχείο testChown.c).

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include<errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main (int argc, char * argv[]) {
  int retVal;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **testChmod name**, όπου name το όνομα ενός αρχείου ή καταλόγου. Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με 2 (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 2, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
  if (argc!=2) {
      printf ("Usage testChown pathname\n");
      return (-1); }
```

Εάν η εφαρμογή κλήθηκε με το σωστό τρόπο, καλεί τη συνάρτηση **chown** για την αλλαγή του κατόχου και της ομάδας του κατόχου για το αρχείο ή τον κατάλογο που όρισε ο χρήστης (και το οποίο βρίσκεται καταχωρημένο στη συμβολοσειρά argv[1]). Σε αντίθεση με τη γραμμή εντολών του λειτουργικού όπου υπάρχουν **δύο** εντολές, η **chown** για την αλλαγή του κατόχου και η **chgrp** για την αλλαγή της ομάδας του κατόχου, η γλώσσα C διαθέτει **μία και μοναδική συνάρτηση** chown που μεταβάλλει και τις δύο αυτές παραμέτρους (δείτε τη διαφάνεια 15 της παρουσίασης 06.Files and Directories). Στο παράδειγμα της άσκησης ως κάτοχο και ομάδα κατόχου του αρχείου ορίζουμε τον χρήστη **SYS** και την ομάδα **SYS** οι κωδικοί των οποίων έχουν αμφότεροι την τιμή 3, όπως προκύπτει από τα περιεχόμενα των αρχείων **/etc/passwd** (που περιέχει τα στοιχεία των χρηστών) και **/etc/group** (που περιέχει τα στοιχεία των ομάδων χρηστών).

```
amarg@amarg-vbox:~/shared/Lab4$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

```
amarg@amarg-vbox:~/shared/Lab4$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog,amarg
```

```
  retVal = chown (argv[1], (uid_t)3, (gid_t)3);
```

Εάν η συνάρτηση επιστρέψει την τιμή -1, σημαίνει πως έλαβε χώρα κάποιο σφάλμα. Στην περίπτωση αυτή η συνάρτηση καλεί την **perror** για να εκτυπώσει μία λεκτική περιγραφή του σφάλματος που προέκυψε και τερματίζει τη λειτουργία της ενώ στην αντίθετη περίπτωση εκτυπώνει ένα ενημερωτικό μήνυμα που ενημερώνει το χρήστη πως η εφαρμογή λειτούργησε σωστά και πραγματοποίησε τη μεταβολή του κατόχου και της ομάδας του κατόχου του αρχείου.

```
  if(retVal == -1){
     printf("Error Number : %d\n", errno);
     perror("Error Description");
     return (-2); }
```

```
  printf ("Assignment to %s of user SYS and group SYS
                                    has been performed succesfully\n", argv[1]);
  return (0); }
```

Παράδειγμα χρήσης της εφαρμογή ακολουθεί στη συνέχεια. Παρατηρήστε πως εάν η εφαρμογή κληθεί με το συνήθη τρόπο **δεν λειτουργεί** αλλά εκτυπώνει το μήνυμα **Operation not permitted** (μη επιτρεπτή λειτουργία) – ο λόγος για αυτό είναι πως οι εντολές αυτές με τον τρόπο που καλούνται μπορούν να εκτελεστούν μόνο από το διαχειριστή του συστήματος.

```
amarg@amarg-vbox:~/shared/Lab5$ ./testChown ~/sample.c
Error Number : 1
Error Description: Operation not permitted
amarg@amarg-vbox:~/shared/Lab5$
```

Η εκτέλεση της εφαρμογής ως διαχειριστής πραγματοποιείται με την εντολή **sudo** η οποία απαιτεί την καταχώρηση του κωδικού του τρέχοντα χρήστη. Στην περίπτωση αυτή, η εφαρμογή εκτελείται χωρίς πρόβλημα.

```
amarg@amarg-vbox:~/shared/Lab5$ sudo ./testChown ~/sample.c
[sudo] password for amarg:
Assignment to /home/amarg/sample.c of user SYS and group SYS has been performed succesfully
amarg@amarg-vbox:~/shared/Lab5$
```

Η εκτέλεση της εντολής ls –l πιστοποιεί πως η εφαρμογή εκτελέστηκε με επιτυχία, αφού πλέον ο κάτοχος και η ομάδα του κατόχου έχουν την τιμή SYS.

```
-rwxrwxr-x  1 amarg amarg    20032 Οκτ  30 11:05 sample
-rw-rw-r--  1 sys   sys       1238 Οκτ   8 12:09 sample.c
-rw-rw-r--  1 amarg amarg     8888 Οκτ  30 11:05 sample.o
```

Παράδειγμα 5. Υπολογισμός μεγέθους αρχείου με την lseek (αρχείο fsize1.c).

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

int main(int argc, char * argv[]) {
   int fd;
   off_t filelength;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **fsize1 name**, όπου name το όνομα ενός αρχείου ή καταλόγου. Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με 2 (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 2, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
   if (argc !=2) {

      printf ("Usage: fsize1 pathname\n");
      return (-1); }
```

Προκειμένου η εντολή να λειτουργήσει σωστά, θα πρέπει το αρχείο που όρισε ο χρήστης **να είναι σε θέση να ανοίξει σε κατάσταση μόνο ανάγνωσης** κάτι που γίνεται καλώντας την εντολή **open** με όρισμα **O_RDONLY**. Εάν η εντολή open επιστρέψει

τιμή μικρότερη του μηδενός, αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα (π.χ. το αρχείο δεν υπάρχει ή δεν είναι επιτρεπτή η πρόσβαση σε αυτό). Η εφαρμογή λοιπόν δεν μπορεί να συνεχίσει και για το λόγο αυτό καλεί την **perror** για να εκτυπώσει μία λεκτική περιγραφή του σφάλματος και τερματίζει τη λειτουργία της.

```
fd = open(argv[1], O_RDONLY );
if (fd < 0) {
    printf("Error Number : %d\n", errno);
    perror("Error Description");
    return (-2); }
```

Η λογική που κρύβεται πίσω από τη χρήση της **lseek** για τον υπολογισμό του μεγέθους του αρχείου, είναι πως **αυτό το μέγεθος είναι ίσο με την απόσταση σε bytes ανάμεσα στην αρχή και στο τέλος του αρχείου**. Η συνάρτηση lseek μεταφέρει τον δείκτη τρέχουσας θέσης σε οποιαδήποτε θέση στο εσωτερικό του αρχείου (μην ξεχνάτε πως μιλάμε για **αρχεία τυχαίας προσπέλασης** που επιτρέπουν την μετακίνησή μας σε ένα και μόνο βήμα σε οποιαδήποτε θέση του αρχείου) και επιστρέφει **την απόσταση σε bytes ανάμεσα στην αρχή του αρχείου και στην τρέχουσα θέση**. Εάν λοιπόν μεταφερθούμε στο τέλος του αρχείου (δηλαδή αν ορίσουμε ως τρέχουσα θέση το τέλος του αρχείου), τότε η τιμή που επιστρέφεται από την lseek είναι η απόσταση ανάμεσα στην αρχή και στο τέλος του αρχείου, δηλαδή το **μέγεθος** του αρχείου σε bytes.

Η κλήση της συνάρτησης lseek με τη μορφή lseek (fd, A, SEEK_END) ορίζει ως τρέχουσα θέση για το αρχείο που περιγράφεται από τον περιγραφέα αρχείου fd, εκείνη που απέχει A bytes από το τέλος του αρχείου (SEEK_END). Εάν λοιπόν είναι A=0 δηλαδή εάν καλέσουμε την εντολή με τη μορφή **lseek (fd, 0, SEEK_END)**, τότε ως τρέχουσα θέση ορίζουμε το τέλος του αρχείου. Στην περίπτωση αυτή η τιμή filelength που επιστρέφεται από τη συνάρτηση lseek είναι σύμφωνα με τα όσα αναφέραμε παραπάνω, το μέγεθος του αρχείου.

```
filelength = lseek (fd, 0, SEEK_END);
```

Εάν η επιστρεφόμενη τιμή της lseek είναι αρνητική, έχει λάβει χώρα κάποιο σφάλμα και εκτυπώνεται το κατάλληλο μήνυμα σφάλματος, ενώ στην αντίθετη περίπτωση εκτυπώνεται η επιστρεφόμενη τιμή της lseek που είναι το μέγεθος του αρχείου.

```
if (filelength < 0) {
    printf("Error Number : %d\n", errno);
    perror("Error Description"); }
else printf ("Size of file %s is %d bytes\n", argv[1], (int)filelength);
close (fd);
return (0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab5$ ./fsize1 ~/sample.c
Size of file /home/amarg/sample.c is 1238 bytes
amarg@amarg-vbox:~/shared/Lab5$ ./fsize1 ~/sample.o
Size of file /home/amarg/sample.o is 8888 bytes
amarg@amarg-vbox:~/shared/Lab5$ ./fsize1 ~/sample
Size of file /home/amarg/sample is 20032 bytes
```

Παράδειγμα 6. Υπολογισμός μεγέθους αρχείου με την read (αρχείο fsize2.c).

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

#define BUFSIZE 100
```

```
int main (int argc, char *argv[]) {
   int fd, nread, total=0;
   char buf[BUFSIZE];
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **fsize2 name**, όπου name το όνομα ενός αρχείου ή καταλόγου. Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με 2 (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 2, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
   if (argc !=2) {
   printf ("Usage: flength pathname\n");
   return (-1); }
```

Η λογική που κρύβεται πίσω από τη χρήση της **read** για τον υπολογισμό του μεγέθους του αρχείου, είναι **πως το μέγεθος του αρχείου είναι ίσο με το πλήθος των bytes που είναι αποθηκευμένα σε αυτό**. Εάν λοιπόν διαβάσουμε όλα τα bytes που υπάρχουν αποθηκευμένα στο αρχείο και μετρήσουμε το πλήθος τους, ο αριθμός που θα βρούμε είναι ίσος με το μέγεθος του αρχείου.

Προκειμένου η εντολή να λειτουργήσει σωστά, θα πρέπει το αρχείο που όρισε ο χρήστης **να είναι σε θέση να ανοίξει σε κατάσταση μόνο ανάγνωσης** κάτι που γίνεται καλώντας την εντολή **open** με όρισμα **O_RDONLY**. Εάν η εντολή open επιστρέψει τιμή μικρότερη του μηδενός, αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα (π.χ. το αρχείο δεν υπάρχει ή δεν είναι επιτρεπτή η πρόσβαση σε αυτό). Η εφαρμογή λοιπόν δεν μπορεί να συνεχίσει και για το λόγο αυτό καλεί την **perror** για να εκτυπώσει μία λεκτική περιγραφή του σφάλματος και τερματίζει τη λειτουργία της.

```
   fd = open(argv[1], O_RDONLY );
   if (fd < 0) {
      printf("Error Number : %d\n", errno);
      perror("Error Description");
      return (-2);
```

Αν και μπορούμε να διαβάσουμε **ένα byte κάθε φορά**, ωστόσο για να επιταχύνουμε τη διαδικασία επιλέγουμε να διαβάζουμε **BUFSIZE-1 bytes** κάθε φορά, τα οποία αποθηκεύονται στη μεταβλητή **buf**. Η συνάρτηση **read** επιστρέφει το πλήθος των bytes που διάβασε και καλείται μέσα από ένα βρόχο που εκτελείται επ' άπειρον. Σε κάθε κύκλο αυτής της επαναληπτικής διαδικασίας ανάγνωσης, το πλήθος των bytes που διαβάστηκαν προστίθενται στην τιμή μιας μεταβλητής **total** η οποία αρχικά έχει τεθεί στη μηδενική τιμή και η οποία στο τέλος θα περιέχει το πλήθος των bytes που διαβάστηκαν και που όπως αναφέραμε είναι το μέγεθος του αρχείου. Ο επαναληπτικός βρόχος εκτελείται **για όσο χρονικό διάστημα υπάρχουν bytes για ανάγνωση** δηλαδή για όσο χρόνο η συνάρτηση read επιστρέφει μη μηδενική τιμή. Όταν η εν λόγω συνάρτηση επιστρέψει μηδενική τιμή, αυτό σημαίνει πως δεν βγήκε άλλα bytes για να διαβάσει και κατά συνέπεια **έχει φτάσει στο τέλος του αρχείου**. Στην περίπτωση αυτή εκτελείται η εντολή break που προκαλεί τον τερματισμό του βρόχου, ενώ η τρέχουσα τιμή της μεταβλητής total είναι το μέγεθος του αρχείου η τιμή του οποίου στη συνέχεια εκτυπώνεται στην οθόνη.

```
   while (1) {
      nread = read(fd,buf,BUFSIZE-1);
      if (nread == 0){
         break; }
       /* buf[nread] = '\0';*/
      total += nread; }
   printf("%d bytes total\n",total);
   close(fd);
   return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Παρατηρήστε πως τα μεγέθη των αρχείων που υπολογίζονται είναι τα ίδια με αυτά που υπολογίζονται από την προηγούμενη εφαρμογή, κάτι που ασφαλώς είναι αναμενόμενο.

```
amarg@amarg-vbox:~/shared/Lab5$ ./fsize2 ~/sample.c
1238 bytes total
amarg@amarg-vbox:~/shared/Lab5$ ./fsize2 ~/sample.o
8888 bytes total
amarg@amarg-vbox:~/shared/Lab5$ ./fsize2 ~/sample
20032 bytes total
amarg@amarg-vbox:~/shared/Lab5$
```

Τα ίδια αυτά μεγέθη εκτυπώνονται και από την εντολή ls –l.

```
amarg@amarg-vbox:~$ ls -l sample*
-rwxrwxr-x 1 amarg amarg 20032 Οκτ  30 11:05 sample
-rw-rw-r-- 1 sys   sys    1238 Οκτ   8 12:09 sample.c
-rw-rw-r-- 1 amarg amarg  8888 Οκτ  30 11:05 sample.o
```

Παράδειγμα 7. Αντιγραφή αρχείου με τις read και write (αρχείο myCopy.c).

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#define BUFSIZE 512

int main(int argc, char * argv[]) {
   char buffer[BUFSIZE];
   int src, dest, errno;
   int nread, nwritten, n;
   int totalr=0, totalw=0;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **myCopy source target**, όπου **source** το όνομα του αρχείου πηγή και **target** το όνομα του αρχείου προορισμού (πράγματι, για να γίνει η αντιγραφή απαιτείται να καθορίσουμε το όνομα του αρχείου που θα αντιγράψουμε και το όνομα του αρχείου που θα προκύψει από την αντιγραφή). Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με **3** (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 3, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
   if (argc!=3) {
      printf ("Usage: myCopy source destination\n");
      return (-1); }
```

Το αρχείο – πηγή που βρίσκεται στη συμβολοσειρά **argv[1]** και το περιεχόμενο του οποίου θα αντιγραφεί στο αρχείο προορισμού, ανοίγει υπό συνθήκες μόνο ανάγνωσης **(O_RDONLY)** – εάν η συνάρτηση open που χρησιμοποιείται για αυτή τη διαδικασία επιστρέψει **αρνητική** τιμή, αυτό σημαίνει πως πραγματοποιήθηκε κάποιο σφάλμα (συνηθισμένα σφάλματα είναι **να μην υπάρχει** το αρχείο ή τα δικαιώματα πρόσβασης σε αυτό να είναι τέτοια ώστε **να μην επιτρέπεται** το άνοιγμά του για ανάγνωση). Στην περίπτωση αυτή η εφαρμογή δεν μπορεί να συνεχίσει – καλεί λοιπόν τη συνάρτηση **perror** για να εκτυπώσει μία λεκτική περιγραφή του σφάλματος που προέκυψε και τερματίζει τη λειτουργία της.

```
   src = open(argv[1], O_RDONLY);
   if (src < 0) { perror("Source file could not be opened!"); return (-1); }
```

Από την άλλη πλευρά, το αρχείο – προορισμός που βρίσκεται στη συμβολοσειρά **argv[2]** μπορεί να υπάρχει αλλά μπορεί και όχι (οπότε δημιουργείται λόγω του ορίσματος **O_CREATE**). Το όρισμα **O_TRUNC** προκαλεί το **μηδενισμό** του μεγέθους του αρχείου προορισμού εάν αυτό υπάρχει και έχει ανοίξει με επιτυχία υπό καθεστώς O_WRONLY (δηλαδή μόνο για **εγγραφή**) ενώ τα δύο τελευταία ορίσματα **S_IRUSR** και **S_IWUSR** ενεργοποιούν τα bits ανάγνωσης και εγγραφής της μάσκας δικαιωμάτων για τον κάτοχο του αρχείου.

```
dest = open(argv[2], O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);
```

Εάν η συνάρτηση open που χρησιμοποιείται για αυτή τη διαδικασία επιστρέψει **αρνητική** τιμή, αυτό σημαίνει πως πραγματοποιήθηκε κάποιο σφάλμα Στην περίπτωση αυτή η εφαρμογή δεν μπορεί να συνεχίσει – καλεί λοιπόν τη συνάρτηση **perror** για να εκτυπώσει μία λεκτική περιγραφή του σφάλματος που προέκυψε και τερματίζει τη λειτουργία της

```
if (dest < 0) {
   perror("Target file could not be opened!");
   return (-2); }
```

Με ποιο τρόπο πραγματοποιείται η αντιγραφή του αρχείου? πολύ απλά **διαβάζοντας bytes από το αρχείο –πηγή και εγγράφοντάς τα στο αρχείο προορισμού**, για όσο χρονικό διάστημα **υπάρχουν** bytes για ανάγνωση. Αυτή η διαδικασία πραγματοποιείται μέσα από ένα βρόχο επανάληψης ο οποίος εκτελείται για όσο το πλήθος των bytes που διάβασε η read από το αρχείο πηγή και επιστρέφεται στη μεταβλητή nread είναι **μεγαλύτερος του μηδενός**. Η διαδικασία εγγραφής των bytes που διαβάστηκαν στο αρχείο προορισμού γίνεται με ένα ένθετο βρόχο **do { … } while ( )** οποίος εκτελείται **για όσο χρονικό διάστημα το πλήθος των bytes που έχουν εγγραφεί στο αρχείο προορισμού είναι μικρότερο από το πλήθος των bytes που διαβάστηκαν κατά την τρέχουσα επανάληψη του εξωτερικού βρόχου**. Εάν όλα τα bytes που διαβάστηκαν έχουν εγγραφεί στο αρχείο προορισμού, πραγματοποιείται η επόμενη επανάληψη του εξωτερικού βρόχου μέχρι τελικά το σύνολο των bytes του αρχείου πηγή να εγγραφεί στο αρχείο προορισμού, συνθήκη που σηματοδοτεί και την ολοκλήρωση της διαδικασίας αντιγραφής του αρχείου.

```
while ((nread = read(src, buffer, BUFSIZE))>0) {
            nwritten=0;
        do {
           n = write (dest, &buffer[nwritten], nread-nwritten);
           nwritten += n;
          } while (nwritten<nread); }

close (src);  close (dest);
return (0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Το αρχείο - πηγή hello1 δημιουργείται από την ανακατεύθυνση εξόδου της εντολής **echo** και περιέχει το μήνυμα **Hello world**. Στη συνέχεια καλείται η **myCopy** που αντιγράφει το αρχείο hello1 στο αρχείο **hello2** το οποίο είναι πανομοιότυπο με το αρχείο hello1 όπως διαπιστώνεται στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab5$ echo "Hello world" > hello1
amarg@amarg-vbox:~/shared/Lab5$ cat hello1
Hello world
amarg@amarg-vbox:~/shared/Lab5$ ./myCopy hello1 hello2
amarg@amarg-vbox:~/shared/Lab5$ ls -l hello*
-rwxrwxrwx 1 root root 12 Νοε   5 18:04 hello1
-rwxrwxrwx 1 root root 12 Νοε   5 18:04 hello2
amarg@amarg-vbox:~/shared/Lab5$ cat hello2
Hello world
amarg@amarg-vbox:~/shared/Lab5$
```

Παράδειγμα 8. Αντιγραφή αρχείου με τις συναρτήσεις της C (αρχείο myCopyc.c).

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>

#define BUFSIZE 512
```

Σε αυτό το παράδειγμα υλοποιούμε την εφαρμογή αντιγραφής αρχείου χρησιμοποιώντας **τις συναρτήσεις της γλώσσας C** προκειμένου να γίνει αντιληπτή η διαφορά στη σύνταξη. Αντί για τις συναρτήσεις **open**, **close**, **read** και **write**, χρησιμοποιούνται οι **fopen**, **fclose**, **fread** και **frite**. Σε αντίθεση με τις κλήσεις συστήματος χαμηλού επιπέδου που προσδιορίζουν το αρχείο χρησιμοποιώντας **έναν απλό περιγραφέα αρχείου** που είναι μία ακέραια τιμή, οι συναρτήσεις υψηλού επιπέδου της γλώσσας C χρησιμοποιούν μία ολόκληρη δομή τύπου **FILE**, με τον περιγραφέα αρχείου να αποτελεί ένα από τα πεδία αυτής της δομής. Επειδή η ανάγνωση και η εγγραφή γίνονται διαβάζοντας και γράφοντας **ένα byte κάθε φορά**, το αρχείο –πηγή ανοίγει σε κατάσταση **rb (read binary)** ενώ το αρχείο προορισμού ανοίγει σε κατάσταση **wb (write binary)**. Η διαδικασία της αντιγραφής πραγματοποιείται όπως και πριν μέσω μιας επαναληπτικής διαδικασίας και σε κάθε κύκλο επανάληψης μεταφέρεται ένα απλό byte από το αρχείο πηγή στο αρχείο προορισμού. Η δομή του κώδικα και ο τρόπος λειτουργίας του είναι ακριβώς ίδια με πριν και δεν απαιτείται να γίνει κάποιος πρόσθετος σχολιασμός.

```
int main(int argc, char * argv[]) {
   char cTemp;
   FILE * src, * dest;

   if (argc!=3) {
      printf ("Usage: myCopy source destination\n");
      return (-1); }

   src = fopen(argv[1], "rb");
   if (!src) {
      printf ("Source File could not be opened! Aborting...");
      return (-2); }

   dest = fopen(argv[2], "wb");
   if (!src) {
      printf ("Destination file could not be opened! Aborting...");
      return (-3); }

   while (fread(&cTemp, 1, 1, src) == 1) {
         fwrite(&cTemp, 1, 1, dest); }

   fclose(src);
   fclose(dest);
   return (0); }
```

Παράδειγμα 9. Δημιουργία και διαγραφή καταλόγου (αρχείο dirFun.c).

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

```
#include <errno.h>
#include <stdlib.h>
#include <dirent.h>

int main(int argc, char * argv[]) {
   long size; int retVal; DIR * dir;
   char * buf, * ptr, arg [20]="ls -l | grep ";
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **dirFun directory-name** όπου **directory-name** το όνομα του καταλόγου που θέλουμε να χρησιμοποιήσουμε. Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με **2** (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 2, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
   if (argc!=2) {
      printf ("Usage: dirFun dirname\n");
      return (-1); }
```

Ο στόχος της εφαρμογής είναι η **δημιουργία** του καταλόγου το όνομα του οποίου ορίζεται από το χρήστη. Για το λόγο αυτό καλούμε τη συνάρτηση **opendir** η οποία ανοίγει έναν κατάλογο και ελέγχουμε την επιστρεφόμενη τιμή της. Εάν αυτή η τιμή είναι διαφορετική του μηδενός, η εφαρμογή εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία της, γιατί προφανώς **δεν μπορούμε** να κατασκευάσουμε έναν κατάλογο που να έχει το ίδιο όνομα με έναν υπάρχοντα κατάλογο.

```
   dir = opendir(argv[1]);
   if (dir) {
      printf ("Directory exists. Aborting...\n");
   closedir(dir);
   return (-2);  }
```

Στο σημείο αυτό η εφαρμογή ανακτά και εκτυπώνει το όνομα του τρέχοντος καταλόγου καλώντας τη συνάρτηση **getcwd** η οποία επιστρέφει την απόλυτη διαδρομή αυτού του καταλόγου. Επειδή **δεν γνωρίζουμε** το μήκος σε χαρακτήρες αυτής της απόλυτης διαδρομής καλούμε τη συνάρτηση **pathconf** με ορίσματα τον τρέχοντα κατάλογο (**.**) και τη σταθερά **_PC_MAX_PATH** η οποία επιστρέφει την τιμή αυτού του μήκους. Στη συνέχεια καλούμε τη συνάρτηση **malloc** για να δεσμεύσουμε την περιοχή μνήμης **buf** με αυτό το μέγεθος η οποία στη συνέχεια χρησιμοποιείται ως όρισμα στην **getcwd** που επιστρέφει την απόλυτη διαδρομή του τρέχοντος καταλόγου.

```
   size = pathconf(".", _PC_PATH_MAX);
   if ((buf = (char *)malloc((size_t)size)) != NULL)
      ptr = getcwd(buf, (size_t)size);
   printf ("Current Working Directory is %s\n", buf);
   printf ("Creating directory %s inside directory %s\n", argv[1], buf);
   free(buf);
```

Μετά την εκτύπωση των κατάλληλων ενημερωτικών μηνυμάτων, η εφαρμογή καλεί τη συνάρτηση **mkdir** για να δημιουργήσει τον κατάλογο που όρισε ο χρήστης και το όνομα του οποίου είναι αποθηκευμένο στη συμβολοσειρά argv[1]. Ο χρήστης πρέπει επίσης να ορίσει και τα **δικαιώματα πρόσβασης** για το νέο κατάλογο που εδώ είναι τα **755 (r w x r - x r – x).** Εάν η συνάρτηση επιστρέψει την τιμή -1 αυτό σημαίνει πως έχει λάβει χώρα κάποιο σφάλμα, οπότε η εφαρμογή καλεί τη συνάρτηση **perror** για να εκτυπώσει μία λεκτική περιγραφή αυτού του σφάλματος και τερματίζει τη λειτουργία της.

```
   retVal = mkdir (argv[1],0755);
   if (retVal==-1) {
      printf("Error Number : %d\n", errno);
      perror("Error Description");
      return (-2); }
```

Προκειμένου να επαληθεύσουμε πως ο κατάλογος δημιουργήθηκε με επιτυχία, καλείται η εντολή **ls –l | grep argv[1]** (με τη συμβολοσειρά **argv[1]** να περιέχει το όνομα του καταλόγου που δημιουργήθηκε) για να εκτυπωθεί η γραμμή για τον τρέχοντα

κατάλογο. Η συμβολοσειρά ls –l | grep argv[1] προκύπτει από τη **συνένωση** της συμβολοσειράς ls –l | grep που βρίσκεται αποθηκευμένη στη μεταβλητή arg και της συμβολοσειράς argv[1] (διαδικασία η οποία πραγματοποιείται από τη συνάρτηση **strcat** του αρχείου string.h) και διαβιβάζεται ως όρισμα στη συνάρτηση **system** η οποία και εκτελεί την εντολή. Στο τέλος της διαδικασίας καλείται η συνάρτηση rmdir για τη διαγραφή του καταλόγου που δημιουργήσαμε παραπάνω και η εφαρμογή τερματίζει.

```
strcat (arg, argv[1]);
system (arg);
printf ("Removing directory %s\n", argv[1]);
rmdir (argv[1]);
return (0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab5$ ./dirFun myDir
Current Working Directory is /home/amarg/shared/Lab5
Creating directory myDir inside directory /home/amarg/shared/Lab5
drwxrwxrwx 1 root root     0 Νοε   6  2020 myDir
Removing directory myDir
amarg@amarg-vbox:~/shared/Lab5$
```

Παράδειγμα 10. Εκτύπωση περιεχομένων καταλόγου (αρχείο myLS.c).

```
#include <stdio.h>
#include <sys/types.h>
#include <errno.h>
#include <dirent.h>

int main(int argc, char * argv[]) {
   DIR * dip;
   struct dirent * dit;
   int files = 0;
```

Η κατάσταση είναι ακριβώς ίδια με πριν. Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **myLs directory-name** όπου **directory-name** το όνομα του καταλόγου τα περιεχόμενα του οποίου θέλουμε να εμφανίσουμε. Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με **2** (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 2, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του

```
if (argc!=2) {
   printf ("Usage: myLs dirname\n");
   return (-1); }
```

Ο στόχος της εφαρμογής είναι η **εμφάνιση των περιεχομένων** του καταλόγου το όνομα του οποίου ορίζεται από το χρήστη και ο οποίος φυσικά θα πρέπει να υπάρχει. Για το λόγο αυτό καλούμε τη συνάρτηση **opendir** η οποία ανοίγει έναν κατάλογο και ελέγχουμε την επιστρεφόμενη τιμή της. Εάν αυτή η τιμή είναι η τιμή **NULL** αυτό σημαίνει πως η κλήση της opendir δεν είχε επιτυχία. Για το λόγο αυτό, η εφαρμογή εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία της.

```
if ((dip = opendir (argv[1]))==NULL) {
   printf("Error Number : %d\n", errno);
   perror("Error Description");
   return (-2); }
```

Η εμφάνιση των περιεχομένων του καταλόγου πραγματοποιείται από την **επαναληπτική** κλήση της συνάρτησης **readdir** η οποία επιστρέφει έναν **δείκτη** προς την επόμενη καταχώρηση του καταλόγου ή τιμή **NULL** εάν έχει φτάσει στο τέλος του καταλόγου ή έχει συμβεί κάποιο σφάλμα. Μέσα λοιπόν από έναν βρόχο επανάληψης που **εκτελείται για όσο χρονικό**

διάστημα η συνάρτηση readdir δεν επιστρέφει τιμή NULL, η εφαρμογή ανακτά το όνομα της **επόμενης καταχώρησης** του καταλόγου που είναι αποθηκευμένο στη μεταβλητή **d_name** (θυμηθείτε από τη C πως ο τελεστής → χρησιμοποιείται για την προσπέλαση πεδίων δομής η οποία προσπελαύνεται μέσω **δείκτη**) και το εκτυπώνει στην οθόνη. Με τον τρόπο αυτό, στο τέλος της διαδικασίας, θα έχουν εκτυπωθεί στην οθόνη τα περιεχόμενα του καταλόγου που έχει ορίσει ο .

```
printf ("List of contents of directory %s\n", argv[1]);
while ((dit = readdir(dip))!=NULL) {
      files++;
      printf("%s\n", dit->d_name); }
printf ("Directory %s contains %d file(s)\n", argv[1], files);
return (0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Το πρόγραμμα εκτυπώνει τα ονόματα των περιεχομένων του καταλόγου (αρχεία και κατάλογοι), ένα όνομα σε κάθε γραμμή συμπεριλαμβανομένων και των κρυφών αντικειμένων.

```
amarg@amarg-vbox:~$ ./myLs .
List of contents of directory .
.bashrc
months.txt
filestat.o
file2.zip
Templates
forkex5
lex
myCopyc.o
forkex4
pid.c
Downloads
file1.txt
forkex5.c
testChmod
flength
file3.zip
.bash_logout
.vboxclient-draganddrop.pid
.sudo_as_admin_successful
.bash_history
rfile
forkex7
```