```
1  #include <errno.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <time.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <dirent.h>
8  #include <string.h>
9  #include <unistd.h>
10
11 char* concat(const char *s1, const char *s2);
12 int bytecomp(char *path1, char *path2){
13   FILE *fp1, *fp2;
14   fp1 = fopen(path1, "r");
15   if (fp1 == NULL){
16     printf(stderr,"Warning: in opening file in read mode %s:%s\n",path1,strerror(errno));
17     printf("%s not compared due to error",path1);
18     return(EXIT_FAILURE); /*notifing user the file not get compared*/
19   }
20   fp2 = fopen(path2, "r");
21   if (fp2 == NULL){
22     printf("Critial Error in opening target in read mode:%s:%s\n", path2,strerror(errno));
23     exit(EXIT_FAILURE); /*target cant be opened, no point continuing*/
24   }
25   if ((fp1 != NULL) && (fp2 != NULL)){
26     char ch1, ch2;
27     while (((ch1 = fgetc(fp1)) != EOF)&&((ch2 = fgetc(fp2)) != EOF)){
28         if (ch1 == ch2){continue;
29         }else{return 1;}
30     }
31     if(fclose(fp1)==EOF) printf("Warning:Error in closing file %s:%s\n",path1,strerror(errno
       ));
32     if(fclose(fp2)==EOF) printf("Warning:Error in closing file %s:%s\n",path2,strerror(errno
       ));
33     return EXIT_SUCCESS;
34   }
35   return(EXIT_SUCCESS);
36 }
37 char* READSYMLINK(char* currentpath,char* dname, const struct stat* STAT); /*declaration*/
38 int hunt(const struct stat target,char* targetpath,const char* searchpath,unsigned int
     call_source);
39 int processstat(char* dname,const struct stat target,char* targetpath,unsigned int
     walk_permission,char* searchpath,unsigned int call_source){
40    /* return 1 if found*/
41    char* path=concat(concat(searchpath,"/"),dname);
42    unsigned int read_permission=0;
43    struct stat sb;
44    if (lstat(path, &sb) == -1) {
45      printf(stderr,"Warning:lstat() failure for item(%s)/nsearchpath(%s): %s",path,
           searchpath,strerror(errno));
46      return(-1);}    /*leave this path or item*/
47    switch (sb.st_mode & S_IFMT) {
48     case S_IFDIR:{
49         hunt(target,targetpath,path,call_source); /*recursive call*/
50         break;}
51     case S_IFLNK:{
52         char* resolvedlink= READSYMLINK(searchpath,dname,&sb);
53         struct stat symlink;
54         if (lstat(path, &symlink) == -1) {
55              printf(stderr,"Warning:lstat() failure for symlink(%s)/nsearchpath(%s): %s",
                   path,searchpath,strerror(errno));
56              break;}
57             /* unsigned int link_permission=1;
58              * if(symlink.st_mode&S_IXOTH) {link_permission=0;}//follow symlink solution
                   with permission check attempt
59          * //an array here keep track of attempted softlinks to check and avoid loop
```

```
60          * if(processstat(resolvedlink,target,targetpath,link_permission&&walk_permission,
                searchpath,soft_link)==1) printf("Original symlink name:%s under path[%s]\n\n\n
                ",dname,searchpath);
61          */
62          unsigned int soft_link=1;
63          if(symlink.st_mode&S_IFREG){     /*proceed if type is a file*/
64                  if(processstat(resolvedlink,target,targetpath,walk_permission,searchpath,
                        soft_link)==1) printf("Original␣symlink␣name:%s␣under␣path[%s]\n\n\n",
                        dname,searchpath);
65                  /*walk_permission not accureat, irrelevent either*/
66          }
67          break;}
68      case S_IFREG: /*regular file ::lets comp\n*/
69          {long long currentfile_size=(long long) sb.st_size;
70          if(sb.st_mode & S_IROTH) read_permission=1;
71          if(currentfile_size==target.st_size){
72            if(!bytecomp(path,targetpath)){
73              if(!call_source) {fputs("Matching␣file␣path:",stdout);
74              }else{printf("symlink␣content:%s\nWith␣resolved␣path:",dname);}
75              puts(path);
76              if(sb.st_ino==target.st_ino&&sb.st_dev==target.st_dev){
77                if (!call_source){printf("HARD␣LINK␣TO␣TARGET\n");
78                }else{printf("SYMLINK␣RESOLVED␣TO␣ORIGINAL␣TARGET\n");}
79              }else{
80                if (!call_source){printf("Duplicate:␣file␣link␣count:␣%ld\n", (long) sb.
                      st_nlink);
81                  }else{printf("SYMLINK␣Duplicate:␣file␣link␣count:␣%ld\n", (long) sb.st_nlink
                        );}
82              }
83              if(!call_source){ /*this condition removed for symlink follow solution attempt*/
84              if(walk_permission&read_permission){
85                puts("OK␣READ␣BY␣OTHERS\n\n\n");
86              }else{
87                puts("No␣READ␣BY␣OTHERS\n\n\n");
88              }
89              }return 1; /*found!*/
90            }
91          }break;}
92      default:
93        printf("Warning:␣(%s)␣under␣path␣[%s]␣links␣to␣something␣not␣a␣file/dir/symlink,
              skipping",dname,searchpath);
94      break;
95      }
96     return 0;
97  }
98  static int filter (const struct dirent *unfiltered);
99  int hunt(const struct stat target,char* targetpath,const char* searchpath,unsigned int
        call_source){
100   /*get cwd's permmision for current dir: assuming other user have permission to call this
        program and is under current directory*/
101   unsigned int walk_permission=0;
102     /*// use if assumption changed to non-user/non-group call from root/dir
103      *char *realPath = realpath (searchpath, NULL);
104      *if (realPath == NULL){
105     printf(stderr,"Warning:realpath() failure for searchpath(%s): %s",searchpath,strerror (
          errno));
106   }*/
107   struct stat walk;
108   if (lstat(searchpath, &walk) == -1) {
109     printf(stderr,"lstat()␣failure␣for␣searchpath(%s):␣%s",searchpath,strerror(errno));
110     exit(EXIT_FAILURE);
111   }
112   if(walk.st_mode&S_IXOTH) walk_permission=1; /* no x for others starting for this serch
        path*/
113   struct dirent **eps;
114   int n = scandir (searchpath, &eps, filter, alphasort);
115   if (n >= 0){
116     for (int cnt = 0; cnt < n; ++cnt){
```

```
117         processstat(eps[cnt]->d_name,target,targetpath,walk_permission,searchpath,call_source)
                ;}
118       }else{
119         fprintf(stderr, "␣the␣directory␣could␣not␣be␣opened␣␣or␣reading␣or␣the␣malloc␣call␣
                failed␣for␣path:␣%s:␣%s\n", searchpath,strerror(errno));
120       }
121     return EXIT_SUCCESS;
122   }
123
124   int main (int argc, char *argv[]){
125     if (argc != 3){
126       for (int i = 1; i < argc; i++)
127         printf("Invaid␣argument:%s␣\n", argv[i]);
128       fprintf(stderr,"Usage:␣%s␣filename␣starting_path\n",argv[0]);
129       exit(-1);}
130     /*program assume the target is relative to cwd*/
131     char* filename=argv[1];       /*target file name info acquiring */
132     struct stat target;
133     char* targetpath=filename;
134     if (stat(targetpath, &target) == -1) {
135       fprintf(stderr, "CRITICAL␣ERROR-->Uable␣to␣read␣stat␣of␣target␣file(%s):%s\n",filename,
              strerror(errno));
136       exit(EXIT_FAILURE);
137     }
138     if((target.st_mode&S_IRWXU)<S_IRUSR){   /*check readibility */
139        fprintf(stderr, "Warning-->Uable␣to␣read␣target␣file(%s):No␣read␣permmision␣on␣target␣
               file␣on␣User␣group[root␣please␣ignore]\n",filename);
140     }
141     struct stat search;
142     if (lstat(argv[2], &search) == -1) {
143       fprintf(stderr, "CRITICAL␣ERROR-->Uable␣to␣read␣search␣directory(%s):%s\n",argv[2],
              strerror(errno));
144       exit(EXIT_FAILURE);
145     }
146     if ((search.st_mode & S_IFMT)==S_IFLNK){
147       hunt(target,targetpath,argv[2],1);   /*search start with softlink*/
148     }else if( (search.st_mode & S_IFMT)==S_IFDIR){
149       hunt(target,targetpath,argv[2],0);   /*search start with hardlink*/
150     }else{
151       printf("CRITICAL␣ERROR-->:Searchpath␣identified␣as␣filetype(%s)\n",argv[2]);
152       exit(EXIT_FAILURE);
153     }
154     return EXIT_SUCCESS;
155   }
156   char* concat(const char *s1, const char *s2){
157       char *result = malloc(strlen(s1)+strlen(s2)+1);   /* +1 for \0 */
158       if (result == NULL) {
159           fprintf(stderr, "CRITICAL␣ERROR-->Insufficient␣memory␣for␣concatenating␣strings(%s␣
                  and␣%s):%s\n",s1,s2,strerror(errno));
160           exit(EXIT_FAILURE);
161         }
162       strcpy(result, s1);
163       strcat(result, s2);
164       return result;
165   }
166   static int filter (const struct dirent *unfiltered){
167     if(unfiltered->d_type==DT_DIR||unfiltered->d_type==DT_REG||unfiltered->d_type==DT_LNK){/*
          filtering data type*/
168         const char *name = unfiltered->d_name;
169       if (unfiltered->d_type==DT_DIR && (!strcmp(name, ".") || !strcmp(name, ".."))) return 0;
            /*ignoring parent and self directory*/
170       return 1;
171     }else{
172       return 0;
173     }
174   }
175   char* READSYMLINK(char* currentpath,char* dname, const struct stat* STAT){/*passing path,
        name and stat struct of symlink*/
```

```
176     ssize_t resolved;
177     char* linkname = malloc(STAT->st_size + 1);
178     if (linkname == NULL) {
179       fprintf(stderr, "CRITICAL␣ERROR-->Insufficient␣memory␣for␣creating␣buffer␣for␣symlink(%s
            )under␣path:%s\nError␣messege:%s\n",
180           dname,currentpath,strerror(errno));
181       return NULL;
182     }
183     resolved = readlink(dname, linkname, STAT->st_size + 1);
184     if (resolved < 0) {
185       fprintf(stderr, "CRITICAL␣ERROR-->Unable␣to␣read␣symlink(%s)under␣path:%s\nError␣messege
            :%s\n",
186               dname,currentpath,strerror(errno));
187       return NULL;
188     }
189     if (resolved > STAT->st_size) {
190       fprintf(stderr, "CRITICAL␣ERROR-->symlink␣increased␣in␣size␣between␣lstat()␣and␣readlink
            ()\n");
191       return NULL;
192     }
193     linkname[STAT->st_size] = '\0';
194     return linkname;
195 }
```