Preview

- link(), unlink() System Call
- remove(), rename() System Call
- Symbolic Links
  - Symbolic link to directory
  - Symbolic link to a executable file
- symlink() System Call
- File Times
  - utime() System Call
- mkdir() and rmdir() System Call
- chdir() Systemcall
- Standard I/O Libaries
- System Data Files and Information
- Password File

The `link()` System Call

- Any file can have multiple directory entries pointing to its i-node.
- The way we can create a link to an existing file is with the link system call.
- Prototype:

```
#include <unistd.h>
int link (const char *existingpath, const char *newpath)

                        Returns: 0 if OK,  return -1 on error
```

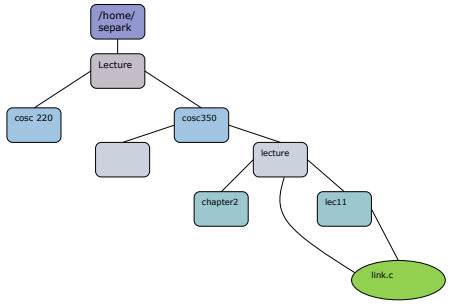
- These functions create a new directory entry, newpath, that references the existing file existingpath.

The `link()` System Call

```
#include <unistd.h>
int link (const char *existingpath, const char *newpath)

                        Returns: 0 if OK, return -1 on error
```

- The link system call create a new directory entry *newpath* that references the existing file *existingpath*. If the *newpath* already exists, an error is returned.
- The creation and increment of the link count be done automatically by kernel.
- Only a super-user process can create a new link that points to a directory.
- Because link system calls to a directory can cause loops in the file system.

The `link()` System Call



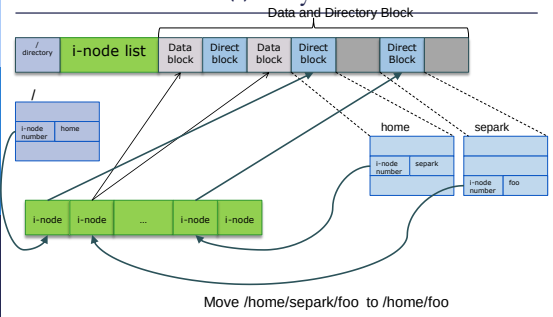

```
/* link.c */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
void err_sys(char *str)
{
   printf ("%s\n",str);
   exit (1);
}
int main ()
{
          struct stat buff;
          /* get attributes for the file link.c*/
          if (stat ("link.c", &buff) < 0)
                    err_sys("stat error for foo");
          printf ("link count for a file link.c was %d \n", buff.st_nlink);
          /* increase the number of link  by one */
          if (link ("/home/separk/Lecture/cosc350/lecture/lec11/link.c",
"/home/separk/Lecture/cosc350/lecture/link.c") <0)
           err_sys("Link Error");
          /* get attributes for the file link.c*/
          if (stat ("link.c", &buff) < 0)
                    err_sys("stat error for foo");
          printf ("link count for a file link.c is now %d \n", buff.st_nlink);

   exit(0);
}
```


The `unlink()` System Call

- By using the unlink system call, we can remove an existing directory entry.
- Prototype:

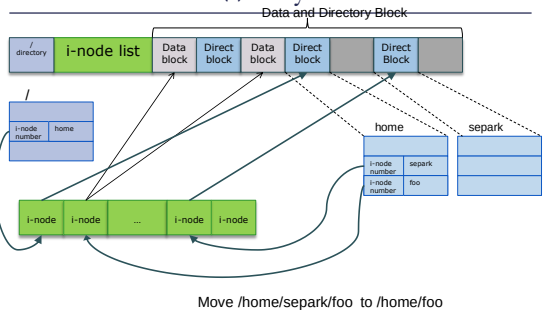
```
#include <unistd.h>
int unlink (const char *pathname)

                        Returns: 0 if OK, return -1 on error
```

The unlink() System Call



Move /home/separk/foo to /home/foo

The unlink() System Call

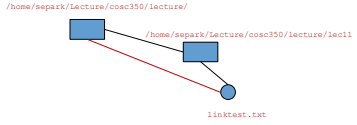


Move /home/separk/foo to /home/foo

The unlink() System Call

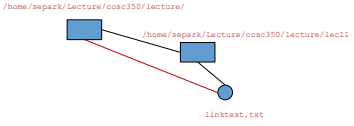
- The `unlink()` system call removes the directory entry and decrement the link count of the file referenced by pathname.
- If there are other links to the file (link count is greater than 1 before unlink), the data in the file is still accessible through the other links.
- To unlink a file, we must have write permission and execute permission in the directory containing the directory entry.
- The superuer can call unlink with pathname specifying a directory, but `rmdir()` system call should be used instead of `unlink()`

The unlink() System Call




```
link("/home/separk/Lecture/cosc350/lecture/lec11/linktest.txt", "/home/separk/Lecture/cosc350/lecture/linktest.txt"
unlink("/home/separk/Lecture/cosc350/lecture/lec11/linktest.txt")
```


The unlink() System Call




```
link("/home/separk/Lecture/cosc350/lecture/lec11/linktest.txt", "/home/separk/Lecture/cosc350/lecture/linktest.txt"
unlink("/home/separk/Lecture/cosc350/lecture/lec11/linktest.txt")
link ("/home/separk/Lecture/cosc350/lecture/linktest.txt", "/home/separk/Lecture/cosc350/lecture/lec11/linktest.txt")
unlink("/home/separk/Lecture/cosc350/lecture/linktest.txt")
```



```
/* linkunlink.c */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
void err_sys(char *str)
{
   printf ("%s\n",str);
   exit (1);
}

int main ()
{
            /* increase the number of link  by one */
            if (link ("/home/separk/Lecture/cosc350/lecture/lec11/linktest.txt",
"/home/separk/Lecture/cosc350/lecture/linktest.txt") <0)
                        err_sys("Link Error");
            printf ("create new link\n");
            sleep (30);

            /* unlink existing file */
            if (unlink("/home/separk/Lecture/cosc350/lecture/lec11/linktest.txt")<0)
                        err_sys("unlink Error");
            printf("file unlinked\n");
            sleep(30);

    /* bring back to before modification*/
            if (link ("/home/separk/Lecture/cosc350/lecture/linktest.txt",
"/home/separk/Lecture/cosc350/lecture/lec11/linktest.txt") <0)
                        err_sys("Link Error");
            printf ("link back\n");
            if (unlink("/home/separk/Lecture/cosc350/lecture/linktest.txt")<0)
                        err_sys("unlink Error");
            printf("file unlinked\n");
   exit(0);
}
```

The remove() System Call

- We can unlink a file or directory with the `remove()` system call.

```
#include <stdio.h>
int remove(const char *pathname);

                                Returns: 0 if ok, -1 on a error
```

  - For a file `remove()` is identical to `unlink()`
  - For a directory `remove()` is identical to `rmdir()`

The rename() System Call

- A file or directory is renamed with rename

```
#include <stdio.h>
int rename (const char * oldname, const char* newname);

                                Returns: 0 if ok, -1 on a error
```

- If oldname specifies a file that is not a directory, then we are renaming a file or a symbolic link.
- If *newname* exists and is not a directory, it is removed, and *oldname* is renamed to *newname*.
- If *oldname* specifies a directory, then we are renaming a directory. If *newname* exists, it must refer to a directory, and that directory must be empty.

The rename() System Call

```
/* removeEX.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int fd; /* file descriptors */
    if ((fd = open(argv[1], O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR|S_IRGRP| S_IWGRP|S_IROTH|S_IWOTH))<1)
    {
        printf("No file\n");
        exit (1);
    }
    printf("A file %s is created\n", argv[1]);
    printf("10 seconds later %s will be renamed to newname\n", argv[1]);
    sleep(10);
    rename(argv[1], "newname");
    printf ("Renamed ...");

    printf("10 seconds later newname will be removed\n");
    sleep(10);
    printf ("Removed ...");
    close(fd);
    remove ("newname");
    return 0;
}
```


Symbolic Links

- Hard link is directly point to the i-node of the file.
- A symbolic link is an indirect pointer to a file or directory.
- Symbolic link were introduced to get around the limitation of hard link.
  - Hard link require that link and the file reside in the same file system.
  - Only the super user can create a hard link to a directory

Symbolic Links

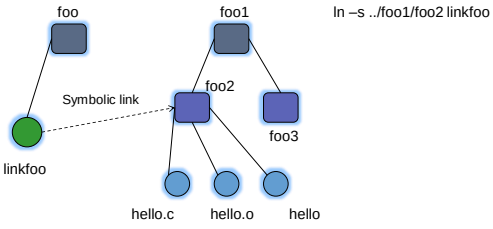
```
/* linktodir.c */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
void err_sys(char *str)
{
   printf ("%s\n",str);
   exit (1);
}

int main ()
{
 /* increase the number of link  by one */

 /* since I am not super user, system create symbolic link instead hard link */
 if (link ("/home/separk/Lecture/cosc350/lecture/lec11", "/home/separk/Lecture/cosc350/lec11") <0)
     err_sys("Link Error");
   exit(0);
}
```


Symbolic Links

- Anyone can create a symbolic link.
- It can be used to move a file or an entire directory hierarch to some other location on a system.
- It can also be used to execute a file

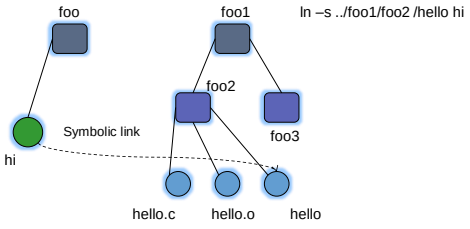
Symbolic Links

Create a symbolic link **linkfoo** from /foo to /foo1/foo2 directory



Symbolic Links

Create a symbolic link **hi** from /foo to /foo1/foo2 /hello executable code



The symlink() System Call

- A symbolic link can created with the symlink system call.
- Prototype:

```
#include <unistd.h>
int symlink (const char *actualpath, const char *sympath)

                                    Returns: 0 if OK,  -1 on a error
```

- A new directory entry, `sympath`, is created that points to `actualpath`.

The symlink() System Call

```
/* symlink.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
void err_sys(char *str)
{
   printf ("%s\n",str);
   exit (1);
}

int main()
{
   /* create a symbolic link to an executable file */
   if ( symlink("/home/separk/Lecture/cosc350/lecture/lec11/linkunlink", "/home/separk/Lecture/cosc350/lecture/lnkunlnk") <0)
      err_sys("Symbolic Link Creation Error \n");
   /* create a symbolic link to a directory */
   if (symlink("/home/separk/Lecture/cosc350/lecture/lec11", "/home/separk/Lecture/cosc350/lecture/lec9/tolec11") <0)
      err_sys("Symbolic Link Creation Error \n");

   printf("A symbolic link is created \n");
   return 0;
}
```


File Times

- Three time fields are maintained for each file.
  - st_atime – last access time of file data– i.e. read content of a file
  - st_mtime –last modification of file data– i.e. write data to a file
  - st_ctime – last change time of i-node status – i.e. change
    - Access permission
    - User ID
    - Number of links,…


```
struct stat {
        mode_t st_mode;    /*file type & mode (permissions) */
        ino_t st_ino;      /* i-node number */
        dev_t st_dev;      /* device number (file system) */
        dev_t st_rdev;     /* device number for special files */
        nlink_t st_nlink; /* number of links */
        uid_t st_uid;      /* user ID of owner */
        gid_t st_gid;      /* group ID of owner */
        off_t st_size;     /* size in bytes, for regular files */
        time_t st_atime;  /* time of last access */
        time_t st_mtime;  /* time of last modification */
        time_t st_ctime;  /* time of last file status change */
        blksize_t st_blksize; /* best I/O block size */
        blkcnt_t st_blocks; /*number of 512 byte blocks allocated */
        mode_t st_attr;    /* The DOS-style attributes for this file */
};
```

The utime() System Call

- The access time and the modification time of a file can be changed with the utime system call.
- Prototype:

```
#include <sys/types.h>
#include <utime.h>
int utime(const chr *pathname, const construct utimebuf *times);

                                Returns: 0 if OK,  -1 on a error
```

- The structure used in utime system call is:

```
struct utimebuf{
        time_t actime;
        time_t modtime;
}
```



```
/* timeexchange.c: exchange times between two files */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <utime.h>

void err_sys(char *str)
{
   printf ("%s\n",str);
   exit (1);
}
int main (int argc, const char *argv[])
{
    struct stat statbuf0, statbuf1;
    struct utimbuf timebuf0, timebuf1;
    if (argc < 3)
       err_sys ("Argument Error");
    /* get first file information */
    if (stat(argv[1], &statbuf0) <0)
       err_sys("Stat Error");
    timebuf0.actime = statbuf0.st_atime;
    timebuf0.modtime = statbuf0.st_mtime;
    /* get second file information */
    if (stat (argv[2], &statbuf1) <0)
       err_sys("Stat Error");
    timebuf1.actime = statbuf1.st_atime;
    timebuf1.modtime = statbuf1.st_mtime;
    /* change time information between two file */
    if (utime (argv[1], &timebuf1) <0)
       err_sys("Time change Error1");
    if (utime (argv[2],&timebuf0) <0)
       err_sys("Time Change Error");

    exit (0);
}
```

mkdir and rmdir System Call

- The entries for . and .. are automatically created.

Prototype:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode)

                        Returns: 0 if OK,  -1 on a error
```

```
#include <unistd.h>
int rmdir (const char *pathname)

                        Returns: 0 if OK,  -1 on a error
```


chdir System Call

- When a user logs in to a Linux or Unix system, the user normally starts at the directory specified by the file /etc/passwd file.
- We can change working directory by using the chdir system call
- Prototype

```
# include <unistd.h>
int chdir (const char *pathname);

                        Returns: 0 if OK,  -1 on a error
```


chdir System Call

```
/*****************************************
  MyChdir.c
*****************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

void err_sys(char *str)
{
   printf ("Access Error for %s\n",str);
   exit (1);
}

int main()
{
    if (chdir ("/home/separk/cosc220")< 0)
       err_sys ("change directory ");
    printf("change directory successded\n");

    exit (0);
}
```


Reading Directories

- Directories can be read by anyone who has access permission to read the directory.
- But only the kernel can write to a directory, to preserve file system.
- The actual format of a directory depends on the UNIX like system implementation and the design of the file system.
- Many implementations prevent applications from using the read function to access the contents of directories, thereby further isolating applications from the implementation-specific details of directory formats.
- `opendir()`, `rdaddir()`, `closedir()`, `rewinddir()`, `seekdir()` are system calls used for directory.

Reading Directories

- Opens a directory stream corresponding to the directory named by dirname. The directory stream is positioned at the first entry. opendir() returns a pointer to an object of type DIR. Otherwise, a null pointer is returned

```
# include <dirent.h>
DIR *opendir (const char *dirname);

      Returns: pointer to an object type DIR, NULL on a error
```

- Closes the directory stream referred to by dirp. Closes the directory stream referred to by dirp.

```
# include <dirent.h>
int closedir (DIR *dirp);

                              Returns: 0 if OK,  -1 on a error
```


Reading Directories

```
# include <dirent.h>
struct dirent* readdir (Dir* dirp);

                              Returns: pointer to dirent structure
```

- `readdir()` returns a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument dirp.

```
# include <dirent.h>
void rewinddir (Dir* dirp);

                              Returns: pointer to dirent structure
```

- `rewinddir()` resets the position of the directory stream to which dirp refers to the beginning of the directory.

Reading Directories

```
# include <dirent.h>
long int telldir(Dir* dirp);
```

- `telldir()` Obtains the current location associated with the directory stream specified by dirp.

```
# include <dirent.h>
void seekdir (Dir* dirp, long int loc);
```

- `seekdir()` sets the position of the next readdir() operation on the directory stream specified by dirp to the position specified by loc..

Reading Directories

- The `dirent` structure defined in <dirent.h> is implementation dependent on Unix like system
- Implementations define the structure to contain at least the following two members:

```
# include <dirent.h>
struct dirent{
              ino_t d_ino; /* i-node number */
              char d_name[]; /* null-terminated filename */
              …
              …
}
```



```
/*************************************************************
 * myls.c demonstrate read directory by using opendir() readdir() closedir()
 *************************************************************/
#include <stdio.h>
#include <dirent.h>


int listdir(const char *path)
{
  struct dirent *entry;
  DIR *dp;
  dp = opendir(path);
  if (dp == NULL)
  {
    perror("opendir");
    return -1;
  }
  while((entry = readdir(dp)))
     puts(entry->d_name);
  closedir(dp);
  return 0;
}
int main(int argc, char **argv) {
  int counter = 1;
  if (argc == 1)
            listdir(".");
  while (++counter <= argc) {
    printf("\nListing %s...\n", argv[counter-1]);
    listdir(argv[counter-1]);
  }
  return 0;
}
```


Standard I/O Library

- When we open a file with system call, kernel returns file descriptor which can be used for read and write to the file
- When we open a file with standard I/O functions, we say that we have associated a stream with the file.
- When we open a stream with standard I/O library fopen, it returns a pointer to FILE object.
- The file object is a structure that contains all the information required by the standard I/O library to manage the stream.

Standard I/O Library

- Three streams are predefined and automatically available to a process: standard input, standard output and standard error.
- These three standard I/O streams are referred through the predefined file pointers **stdin**, **stdout**, and **stderr**.
- These pointers are defined in <stdio.h>

Standard I/O Libraries

- fopen, fclose
- fread, fwrite
- fflush
- fseek
- fgetc getc, getchar
- fputc putc, putchar
- fgets, gets
- printf, fprintf and sprintf
- scanf, fscanf and sscanf

System Data Files and Information

- There are several data files required for normal operation:
  - the password file /etc/passwd
  - The group file /etc/group
- Two files are used as a system data.

Ex) /etc/passwd is used every time a user log in to the Linux system.

System Data File (Password File)

- The Linux password file contains the fields. It is defined in <pwd.h>
- Historically the password file has been stored in /etc/passwd and has been an ASCII file.
- Each line contains the seven fields separated by colons.

System Data File (Password File)

char *pw_name User's login name.
uid_t pw_uid Numerical user ID.
gid_t pw_gid Numerical group ID.
char *pw_dir Initial working directory.
char *pw_shell Program to use as shell.