Linux and Windows File Systems

Permission Rights and System Calls

Adrian Coleșa

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

March 25, 2020

The purpose of today's lecture

- Presents and compare *permission rights* in Linux and Windows
- Presents and compare *system calls for files* in Linux and Windows

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 10. Case Study 1: Unix and Linux, pg. 732 - 744, p. 753 - 757
- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 11. Case Study 2: Windows 2000, pg. 830 - 833, p. 844 - 847
- Lab texts related to Linux's and Windows' file system and their system calls.
- From `http://msdn.microsoft.com/en-us/library/aa364407(VS.85).aspx` about File management and Directory Management (following the lecture slides)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Outline

Outline

1. Permission Rights
   - Linux Permission Rights
   - Windows' Permission Rights

2. Basic System Calls on Linux and Windows
   - Linux
   - Windows

3. Conclusions

Basic Permission Rights

- defined for three classes of users
  - owner or user (`u`)
  - groups (`g`) the owner belongs to
  - others (`o`)
- operations (types)
  - *read* (`r`)
  - *write* (`w`)
  - *execute* (`x`)
- See examples running `ls -l` command on different files and directories

Basic Permission Rights

- defined for three classes of users
  - owner or user (`u`)
  - groups (`g`) the owner belongs to
  - others (`o`)
- operations (types)
  - *read* (`r`)
  - *write* (`w`)
  - *execute* (`x`)
- See examples running `ls -l` command on different files and directories

Basic Permission Rights

- defined for three classes of users
  - owner or user (`u`)
  - groups (`g`) the owner belongs to
  - others (`o`)
- operations (types)
  - *read* (`r`)
  - *write* (`w`)
  - *execute* (`x`)
- See examples running `ls -l` command on different files and directories

Basic Permission Rights

- defined for three classes of users
  - owner or user (`u`)
  - groups (`g`) the owner belongs to
  - others (`o`)
- operations (types)
  - *read* (`r`)
  - *write* (`w`)
  - *execute* (`x`)
- See examples running `ls -l` command on different files and directories

Basic Permission Rights

- defined for three classes of users
  - owner or user (`u`)
  - groups (`g`) the owner belongs to
  - others (`o`)
- operations (types)
  - *read* (`r`)
  - *write* (`w`)
  - *execute* (`x`)
- See examples running `ls -l` command on different files and directories

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Basic Permission Rights

- defined for three classes of users
  - owner or user (`u`)
  - groups (`g`) the owner belongs to
  - others (`o`)
- operations (types)
  - *read* (`r`)
  - *write* (`w`)
  - *execute* (`x`)
- See examples running `ls -l` command on different files and directories

Basic Permission Rights

- defined for three classes of users
  - owner or user (`u`)
  - groups (`g`) the owner belongs to
  - others (`o`)
- operations (types)
  - *read* (`r`)
  - *write* (`w`)
  - *execute* (`x`)
- See examples running `ls -l` command on different files and directories

Basic Permission Rights

- defined for three classes of users
  - owner or user (`u`)
  - groups (`g`) the owner belongs to
  - others (`o`)
- operations (types)
  - *read* (`r`)
  - *write* (`w`)
  - *execute* (`x`)
- See examples running `ls -l` command on different files and directories

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Basic Permission Rights

- defined for three classes of users
  - owner or user (`u`)
  - groups (`g`) the owner belongs to
  - others (`o`)
- operations (types)
  - *read* (`r`)
  - *write* (`w`)
  - *execute* (`x`)
- See examples running `ls -l` command on different files and directories

I-node Field Structure

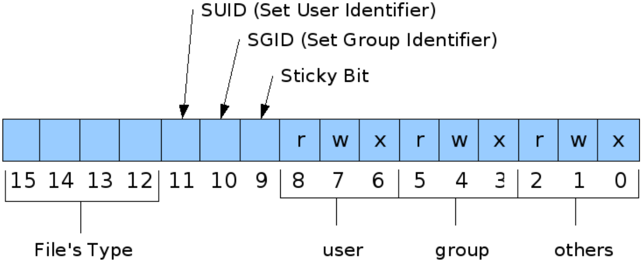


Figure: The way permission rights are stored

- bit value: 0 / 1 → denied / allowed
- example
  - string: `rw-r--r--`
  - binary: 110100100
  - octal: 0644

Permissions on Regular Files

- `read`: read file's contents
- `write`: write (modify, append to, truncate) file's contents
- `execute`: execute file

Permissions on Regular Files

- `read`: read file's contents
- `write`: write (modify, append to, truncate) file's contents
- `execute`: execute file

Permissions on Regular Files

- `read`: read file's contents
- `write`: write (modify, append to, truncate) file's contents
- `execute`: execute file

Permissions on Directories

- `read`: read (list) directory's contents
- `write`: write (modify, add and remove elements) file's contents
  - confusing and too limited
- `execute`: **traverse directory**, i.e. search for an element in the directory
- $\Rightarrow$ **Read and/or Write without Execute not so useful**
  - but ... **Execute without Read and/or Write makes sense**
  - when we want a directory to be traversed, but its contents not be visible
  - commonly used in practice for the `/home` directory

Permissions on Directories

- `read`: read (list) directory's contents
- `write`: write (modify, add and remove elements) file's contents
  - confusing and too limited
- `execute`: **traverse directory**, i.e. search for an element in the directory
- ⇒ **Read and/or Write without Execute not so useful**
  - but … **Execute without Read and/or Write makes sense**
  - when we want a directory to be traversed, but its contents not be visible
  - commonly used in practice for the `/home` directory

Permissions on Directories

- `read`: read (list) directory's contents
- `write`: write (modify, add and remove elements) file's contents
  - confusing and too limited
- `execute`: **traverse directory**, i.e. search for an element in the directory
- ⇒ **Read and/or Write without Execute not so useful**
  - but ... **Execute without Read and/or Write makes sense**
  - when we want a directory to be traversed, but its contents not be visible
  - commonly used in practice for the `/home` directory

Permissions on Directories

- `read`: read (list) directory's contents
- `write`: write (modify, add and remove elements) file's contents
  - confusing and too limited
- `execute`: **traverse directory**, i.e. search for an element in the directory
- ⇒ **Read and/or Write without Execute not so useful**
  - but ... **Execute without Read and/or Write makes sense**
  - when we want a directory to be traversed, but its contents not be visible
  - commonly used in practice for the `/home` directory

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Permissions on Directories

- `read`: read (list) directory's contents
- `write`: write (modify, add and remove elements) file's contents
  - confusing and too limited
- `execute`: **traverse directory**, i.e. search for an element in the directory
- ⇒ **Read and/or Write without Execute not so useful**
  - but ... **Execute without Read and/or Write makes sense**
  - when we want a directory to be traversed, but its contents not be visible
  - commonly used in practice for the `/home` directory

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Permissions on Directories

- `read`: read (list) directory's contents
- `write`: write (modify, add and remove elements) file's contents
  - confusing and too limited
- `execute`: **traverse directory**, i.e. search for an element in the directory
- $\Rightarrow$ **Read and/or Write without Execute not so useful**
  - but . . . **Execute without Read and/or Write makes sense**
  - when we want a directory to be traversed, but its contents not be visible
  - commonly used in practice for the `/home` directory

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Permissions on Directories

- `read`: read (list) directory's contents
- `write`: write (modify, add and remove elements) file's contents
  - confusing and too limited
- `execute`: **traverse directory**, i.e. search for an element in the directory
- ⇒ **Read and/or Write without Execute not so useful**
  - but . . . **Execute without Read and/or Write makes sense**
  - when we want a directory to be traversed, but its contents not be visible
  - commonly used in practice for the `/home` directory

Permissions on Directories

- `read`: read (list) directory's contents
- `write`: write (modify, add and remove elements) file's contents
  - confusing and too limited
- `execute`: **traverse directory**, i.e. search for an element in the directory
- $\Rightarrow$ **Read and/or Write without Execute not so useful**
  - but ... **Execute without Read and/or Write makes sense**
  - when we want a directory to be traversed, but its contents not be visible
  - commonly used in practice for the `/home` directory

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Basic permission rights for FS elements in Linux are `r`, `w`, `x` for `u`, `g`, `o`

`rwxrwxrwx`

Questions (1)

Give the equivalent permission right representation for the following cases?

1. `rwxr-xr--`
2. `r--r--r--`
3. `0765`

Questions (2)

Which of the following operations

- O1 `ls /home/os`
- O2 `cat /home/os/file.txt`
- O3 `rm /home/os/file.txt`

could be performed by the "os" user, supposing the directory "`/home/os`" is its home directory and has the following permission rights (all the file in "`/home/os`" have `r--r--r--` permissions)?

- P1 `r-xr--r--`
- P2 `--x--x--x`
- P3 `rw-r--r--`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Special Techniques: `SUID` and `SGID` Bits

- SUID
  - has effect only for executable files
  - the process resulting from the corresponding executable file will have the effective UID that of the owner of the file
  - see the classical example of `/usr/bin/passwd` executable file, that can be run by any *non-privileged user*, modifying the `/etc/passwd` file belonging to *root*
- SGID
  - similar to SUID but applies to files's GID

Special Techniques: SUID and SGID Bits

- SUID
  - has effect only for executable files
  - the process resulting from the corresponding executable file will have the effective UID that of the owner of the file
  - see the classical example of `/usr/bin/passwd` executable file, that can be run by any *non-privileged user*, modifying the `/etc/passwd` file belonging to *root*
- SGID
  - similar to SUID but applies to files's GID

Special Techniques: `SUID` and `SGID` Bits

- SUID
  - has effect only for executable files
  - the process resulting from the corresponding executable file will have the effective UID that of the owner of the file
  - see the classical example of `/usr/bin/passwd` executable file, that can be run by any *non-privileged user*, modifying the `/etc/passwd` file belonging to *root*
- SGID
  - similar to SUID but applies to files's GID

Special Techniques: SUID and SGID Bits

- SUID
  - has effect only for executable files
  - the process resulting from the corresponding executable file will have the effective UID that of the owner of the file
  - see the classical example of `/usr/bin/passwd` executable file, that can be run by any *non-privileged user*, modifying the `/etc/passwd` file belonging to *root*
- SGID
  - similar to SUID but applies to files's GID

Special Techniques: SUID and SGID Bits

- SUID
  - has effect only for executable files
  - the process resulting from the corresponding executable file will have the effective UID that of the owner of the file
  - see the classical example of `/usr/bin/passwd` executable file, that can be run by any *non-privileged user*, modifying the `/etc/passwd` file belonging to *root*
- SGID
  - similar to SUID but applies to files's GID

Special Techniques: `SUID` and `SGID` Bits

- SUID
  - has effect only for executable files
  - the process resulting from the corresponding executable file will have the effective UID that of the owner of the file
  - see the classical example of `/usr/bin/passwd` executable file, that can be run by any *non-privileged user*, modifying the `/etc/passwd` file belonging to *root*
- SGID
  - similar to SUID but applies to files's GID

Special Techniques: Sticky Bit

- has effect only for directories
- allows elements to be removed only by their owner
- see `/tmp` directory

Special Techniques: Sticky Bit

- has effect only for directories
- allows elements to be removed only by their owner
- see `/tmp` directory

Special Techniques: Sticky Bit

- has effect only for directories
- allows elements to be removed only by their owner
- see `/tmp` directory

Extended Attributes and ACL-like Permission Rights

- extended attributes
  - extra attributes like "append-only"
  - see "man chattr"
- Access Control List (ACL)
  - list of complementary permission rights per user / group
  - see "man getfacl"

Extended Attributes and ACL-like Permission Rights

- extended attributes
  - extra attributes like “append-only”
  - see “man chattr”
- Access Control List (ACL)
  - list of complementary permission rights per user / group
  - see “man getfacl”

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Extended Attributes and ACL-like Permission Rights

- extended attributes
  - extra attributes like "append-only"
  - see "man chattr"
- Access Control List (ACL)
  - list of complementary permission rights per user / group
  - see "man getfacl"

Extended Attributes and ACL-like Permission Rights

- extended attributes
  - extra attributes like "append-only"
  - see "man chattr"
- Access Control List (ACL)
  - list of complementary permission rights per user / group
  - see "man getfacl"

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Extended Attributes and ACL-like Permission Rights

- extended attributes
  - extra attributes like "append-only"
  - see "man chattr"
- Access Control List (ACL)
  - list of complementary permission rights per user / group
  - see "man getfacl"

Extended Attributes and ACL-like Permission Rights

- extended attributes
  - extra attributes like "append-only"
  - see "man chattr"
- Access Control List (ACL)
  - list of complementary permission rights per user / group
  - see "man getfacl"

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "/etc/shadow" file allows for brute-force password guess
  - writable "/etc/passwd" allows for creation of new users
  - writable "/etc/sudoers" allows getting root (admin) permissions
  - writable "/sbin" allows replacement of system executables
  - ...

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "/etc/shadow" file allows for brute-force password guess
  - writable "/etc/passwd" allows for creation of new users
  - writable "/etc/sudoers" allows getting root (admin) permissions
  - writable "/sbin" allows replacement of system executables
  - ...

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

Security Considerations: Study Case (Bad Code)

Consider the following C program "fopen-permissions.c"

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

int main( void)
{
    FILE *f;

    umask(0000);

    if ((f=fopen("TEST", "w+")) == NULL) {
        perror("File creation error");
        exit (1);
    }

    printf("File TEST created with the following permission rights\n");

    system("stat --format=%A TEST");

    unlink("TEST");

    return 0;
}
```

Security Considerations: Study Case (Bad Code) (cont.)

When run, the program displays

```
$ gcc -Wall fopen-permissions.c -o fopen-permissions

$ ./fopen-permissions
File TEST created with the following permission rights
-rw-rw-rw-
```

Security Considerations: Study Case (Good Code)

Consider the following C program "`fopen-permissions.c`"

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

int main( void)
{
    FILE *f;

    umask(0022);

    if ((f=fopen("TEST", "w+")) == NULL) {
        perror("File creation error");
        exit (1);
    }

    printf("File TEST created with the following permission rights\n");

    system("stat --format=%A TEST");

    unlink("TEST");

    return 0;
}
```

Security Considerations: Study Case (Good Code) (cont.)

When run, the program displays

```
$ gcc -Wall fopen-permissions.c -o fopen-permissions
```

```
$ ./fopen-permissions
File TEST created with the following permission rights
-rw-r--r--
```

Never trust the users!
Protect files using the appropriate permission rights!

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

1. Permission Rights
   - Linux Permission Rights
   - Windows' Permission Rights
2. Basic System Calls on Linux and Windows
   - Linux
   - Windows
3. Conclusions

Strategy

- Permission rights are defined for each user using ACLs (*Access Control Lists*)
- Basically, they are: *read* (`r`), *write* (`w`), *execute* (`x`)

Simplified (Synthesized) View

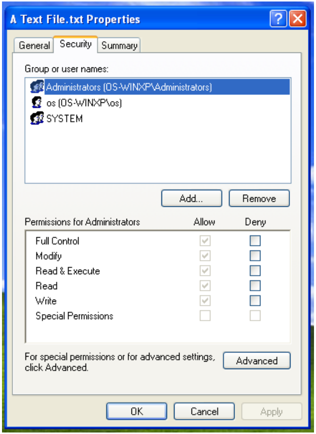


Figure: The Simplified View of Permission Rights

Simplified (Synthesized) View (cont.)

- Files
  - `read`: permits viewing or accessing of the file's contents
  - `write`: permits writing to a file
  - `read&execute`: permits viewing and accessing of the file's contents as well as executing of the file
  - `modify`: permits reading and writing of the file; allows deletion of the file
  - `full control`:permits reading, writing, changing and deleting of the file

Simplified (Synthesized) View (cont.)

- Directories
  - `read`: permits viewing and listing of files and subdirectories
  - `write`: permits adding of files and subdirectories
  - `read&execute`: permits viewing and listing of files and subdirectories, as well as executing of files
  - `modify`: permits reading and writing of files and subdirectories; allows deletion of the directory
  - `full control`: permits reading, writing, changing, and deleting of files and subdirectories

Advanced View

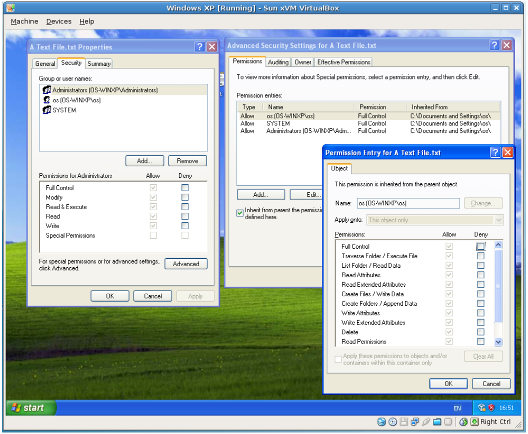


Figure: The Way of Setting Detailed Permission Rights (Windows XP)

Advanced View (cont.)

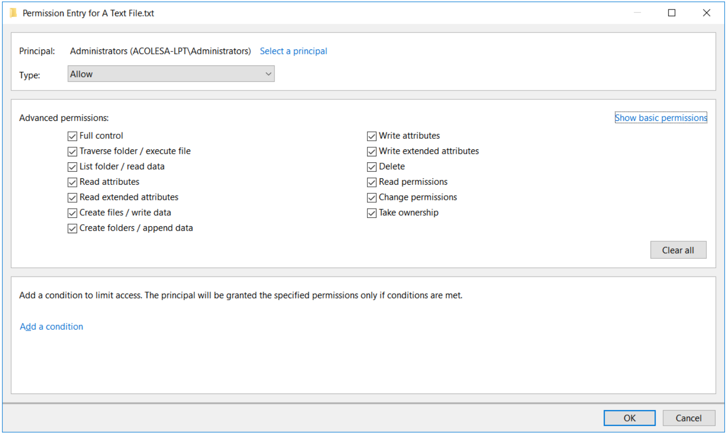


Figure: The Advanced View of Permission Rights (Windows 10)

Advanced View (cont.)

- Traverse Folder/Execute File
- List Folder/Read Data
- Read Attributes
- Read Extended Attributes
- Create Files/Write Data
- Create Folders/Append Data
- Write Attributes
- Write Extended Attributes
- Delete Subfolders and Files
- Delete
- Read Permissions
- Change Permissions
- Take Ownership

Relationship Between Synthesized and Advanced Permission Rights

	Full control	Modify	Read & Execute	List folder contents	Read	Write	Special permissions
Full control	X						
Traverse folder/Execute file	X	X	X	X			
List folder / Read data	X	X	X	X	X		
Read Attributes	X	X	X	X	X		
Read extended attributes	X	X	X	X	X		
Create files/Write data	X	X				X	
Create folders/Append data	X	X				X	
Write attributes	X	X				X	
Write extended attributes	X	X				X	
Delete subfolders and files	X	X					X
Delete	X	X					
Read permissions	X	X	X	X	X		
Change permissions	X						X
Take ownership	X						X

Outline

Outline

Access File Data

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int close(int fd);

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Manipulate Files

```
int creat(const char *pathname, mode_t mode);
int rename(const char *oldpath, const char *newpath);
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

Manipulate Directories

```
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);
int symlink(const char *oldpath, const char *newpath);
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
off_t telldir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
int closedir(DIR *dir);
```

Create and Remove a File

```
int fd;

  // create a new file
fd = creat("/home/os/file", 0600);
    // file size = 0 (no space allocated)
    // only the i-node (metadata) allocated
    // if file exists, it is truncated
    // the new file is opened for WRONLY
    // note permissions: rw-------

  // remove the file (remove a link to the file)
unlink("/home/os/file");
```

File Structure (Format): What Is The File Provided Like?

- no (special) structure
- just a sequence (stream) of bytes
  - each byte has its fixed position (offset)
- **no special byte**(s) inside the file to mark the end of file
  - every byte in the file could have any possible (user-provided) value
  - the file size kept as an i-node field

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

File Structure (Format): What Is The File Provided Like?

- no (special) structure
- just a sequence (stream) of bytes
  - each byte has its fixed position (offset)
- **no special byte**(s) inside the file to mark the end of file
  - every byte in the file could have any possible (user-provided) value
  - the file size kept as an i-node field

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

File Structure (Format): What Is The File Provided Like?

- no (special) structure
- just a sequence (stream) of bytes
  - each byte has its fixed position (offset)
- **no special byte**(s) inside the file to mark the end of file
  - every byte in the file could have any possible (user-provided) value
  - the file size kept as an i-node field

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

File Structure (Format): What Is The File Provided Like?

- no (special) structure
- just a sequence (stream) of bytes
  - each byte has its fixed position (offset)
- **no special byte**(s) inside the file to mark the end of file
  - every byte in the file could have any possible (user-provided) value
  - the file size kept as an i-node field

File Structure (Format): What Is The File Provided Like?

- no (special) structure
- just a sequence (stream) of bytes
  - each byte has its fixed position (offset)
- **no special byte**(s) inside the file to mark the end of file
  - every byte in the file could have any possible (user-provided) value
  - the file size kept as an i-node field

File Structure (Format): What Is The File Provided Like?

- no (special) structure
- just a sequence (stream) of bytes
  - each byte has its fixed position (offset)
- **no special byte**(s) inside the file to mark the end of file
  - every byte in the file could have any possible (user-provided) value
  - the file size kept as an i-node field

Relationship Between the OS and User Views of A File

- user (application) view
  - unstructured sequence of bytes
  - $\Rightarrow$ a logically contiguous area
- OS logical view
  - sequence of blocks
  - $\Rightarrow$ a logically contiguous area
- OS physical view
  - collection of blocks
  - $\Rightarrow$ a collection of more physical contiguous areas

Relationship Between the OS and User Views of A File

- user (application) view
  - unstructured sequence of bytes
  - ⇒ a logically contiguous area
- OS logical view
  - sequence of blocks
  - ⇒ a logically contiguous area
- OS physical view
  - collection of blocks
  - ⇒ a collection of more physical contiguous areas

Relationship Between the OS and User Views of A File

- user (application) view
  - unstructured sequence of bytes
  - ⇒ a logically contiguous area
- OS logical view
  - sequence of blocks
  - ⇒ a logically contiguous area
- OS physical view
  - collection of blocks
  - ⇒ a collection of more physical contiguous areas

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Relationship Between the OS and User Views of A File

- user (application) view
  - unstructured sequence of bytes
  - ⇒ a logically contiguous area
- OS logical view
  - sequence of blocks
  - ⇒ a logically contiguous area
- OS physical view
  - collection of blocks
  - ⇒ a collection of more physical contiguous areas

Relationship Between the OS and User Views of A File

- user (application) view
  - unstructured sequence of bytes
  - $\Rightarrow$ a logically contiguous area
- OS logical view
  - sequence of blocks
  - $\Rightarrow$ a logically contiguous area
- OS physical view
  - collection of blocks
  - $\Rightarrow$ a collection of more physical contiguous areas

Relationship Between the OS and User Views of A File

- user (application) view
  - unstructured sequence of bytes
  - $\Rightarrow$ a logically contiguous area
- OS logical view
  - sequence of blocks
  - $\Rightarrow$ a logically contiguous area
- OS physical view
  - collection of blocks
  - $\Rightarrow$ a collection of more physical contiguous areas

Relationship Between the OS and User Views of A File

- user (application) view
  - unstructured sequence of bytes
  - $\Rightarrow$ a logically contiguous area
- OS logical view
  - sequence of blocks
  - $\Rightarrow$ a logically contiguous area
- OS physical view
  - collection of blocks
  - $\Rightarrow$ a collection of more physical contiguous areas

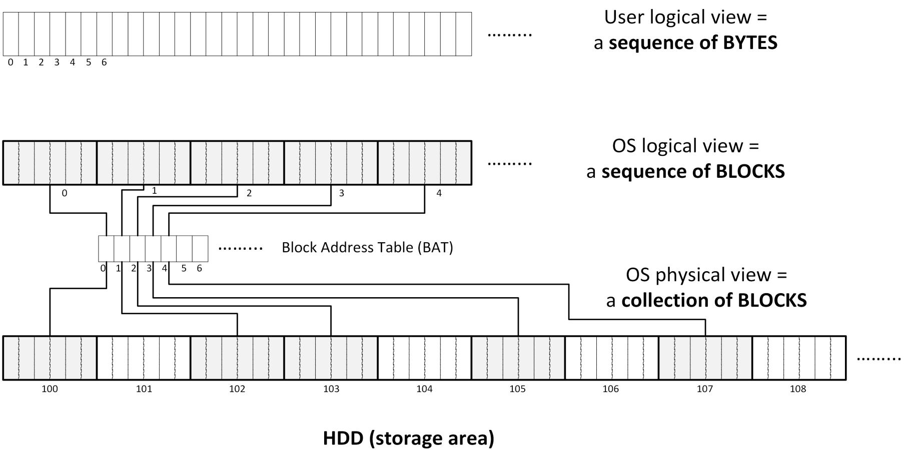
Relationship Between the OS and User Views of A File

- user (application) view
  - unstructured sequence of bytes
  - ⇒ a logically contiguous area
- OS logical view
  - sequence of blocks
  - ⇒ a logically contiguous area
- OS physical view
  - collection of blocks
  - ⇒ a collection of more physical contiguous areas

Relationship Between the OS and User Views of A File

- user (application) view
  - unstructured sequence of bytes
  - $\Rightarrow$ a logically contiguous area
- OS logical view
  - sequence of blocks
  - $\Rightarrow$ a logically contiguous area
- OS physical view
  - collection of blocks
  - $\Rightarrow$ a collection of more physical contiguous areas

Illustration of Different File Views



Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every** `open` $\Rightarrow$ a new *open file* $\Rightarrow$ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every** `open` ⇒ a new *open file* ⇒ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every** `open` ⇒ a new *open file* ⇒ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every** `open` ⇒ a new *open file* ⇒ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every** open ⇒ a new *open file* ⇒ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every** open ⇒ a new *open file* ⇒ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every** open $\Rightarrow$ a new *open file* $\Rightarrow$ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every `open`** ⇒ a new *open file* ⇒ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every `open`** ⇒ a new *open file* ⇒ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every `open`** ⇒ a new *open file* ⇒ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every `open`** ⇒ a new *open file* ⇒ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

Open a File

```
int fd;

  // open an existing file
fd = open("/home/os/file_1", O_RDWR);
    // file must exist
    // opened for both RD and WR (if allowed by persmission rights)

  // ALWAYS CHECK return values of I/O operations!
if (fd < 0) {
    // -1 returned in case of error
    // e.g. file does not exists (wrong filepath)
    // e.g. permission denied
    perror("File cannot be opened");  // display the system err msg.
    exit(1);                          // terminate program
}

  // create a file with "open"
fd = open("/home/os/file_2", O_CREAT | O_EXCL | O_RDWR, 0600);
    // O_EXCL check if file does not already exist
```

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being read)
    - taken from (to be written)
  - return the number of bytes successfully read or written
- ***end of file (EOF)* detected when `read` returns 0 (zero)**

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being read)
    - taken from (to be written)
  - return the number of bytes successfully read or written
- ***end of file (EOF)* detected when `read` returns 0 (zero)**

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being *read*)
    - taken from (to be *written*)
  - return the number of bytes successfully read or written
- ***end of file (EOF)*** **detected when `read` returns 0 (zero)**

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being *read*)
    - taken from (to be *written*)
  - return the number of bytes successfully read or written
- ***end of file (EOF)*** **detected when `read` returns 0 (zero)**

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being read)
    - taken from (to be written)
  - return the number of bytes successfully read or written
- ***end of file (EOF)* detected when `read` returns 0 (zero)**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being *read*)
    - taken from (to be *written*)
  - return the number of bytes successfully read or written
- ***end of file (EOF)* detected when `read` returns 0 (zero)**

Reading From/Writing To A File

- operations are performed relative to the *current position*
    - most of the cases at beginning of file after open
- current position is advanced (increased)
    - with the number of bytes successfully read or written
    - by each `read` and `write`, respectively
- both `read` and `write` syscalls
    - use *memory address* where bytes are
        - written to (after being *read*)
        - taken from (to be *written*)
    - return the number of bytes successfully read or written
- ***end of file (EOF)*** **detected when `read` returns 0 (zero)**

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being *read*)
    - taken from (to be *written*)
  - return the number of bytes successfully read or written
- ***end of file (EOF)*** **detected when `read` returns 0 (zero)**

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being *read*)
    - taken from (to be *written*)
  - return the number of bytes successfully read or written
- ***end of file (EOF)*** **detected when** `read` **returns 0 (zero)**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being *read*)
    - taken from (to be *written*)
  - return the number of bytes successfully read or written
- *end of file (EOF)* detected when `read` returns 0 (zero)

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being *read*)
    - taken from (to be *written*)
  - return the number of bytes successfully read or written
- ***end of file (EOF)*** **detected when `read` returns 0 (zero)**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Example: Opening Files, Reading From Them, Creating Duplicates

```c
// Process 1
int fd1, fd2;
char buf[100]="1234567890";

fd1=open("file1", O_RDWR);        // fd1 = 3
fd2=open("file1", O_RDONLY);      // fd2 = 4
write(fd1, buf, 10);              // write "1234567890"
read(fd2, buf, 5);                // read "12345"

// Process 2
int fd1, fd2, fd3;
char buf[100];

fd1=open("file1", O_RDWR);        // fd1 = 3
fd2=open("file2", O_RDWR);        // fd2 = 4
fd3=dup(fd2);                     // fd3 = 5
write(fd2, buf, 100);             // write first 100 bytes
write(fd3, buf, 100);             // write next 100 bytes
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Example: Open File Tables Illustration

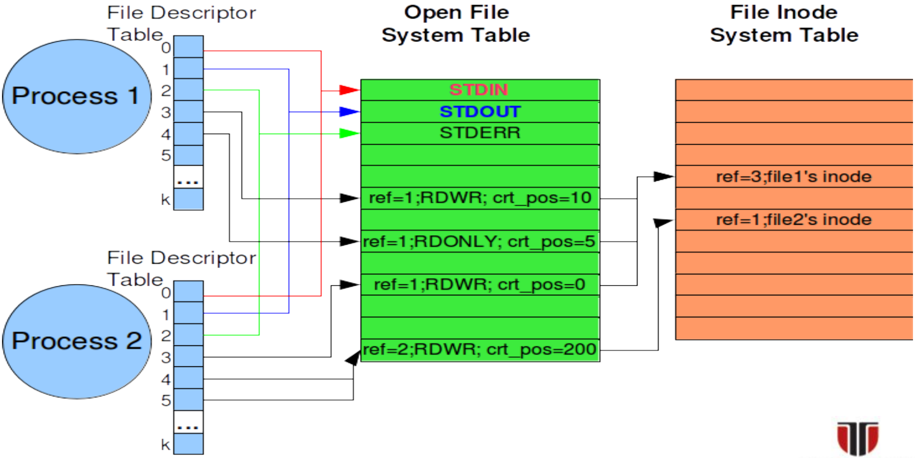


Figure: The Effect of More Open and Read Operations

Each "`open()`" leads to a new, independent open file, i.e. a new entry in OFT and FDT.

File Format: Text Files

- each byte has its own interpretation
  - contains the code of a printable character
- special characters: new line (`0x0A`)
- OS knows nothing about (ANY) file's format
  - ⇒ `read` has no notion of reading until some special char
  - i.e. `read` cannot read a text line
  - **end of line has to be detected by application**

File Format: Text Files

- each byte has its own interpretation
  - contains the code of a printable character
- special characters: new line (`0x0A`)
- OS knows nothing about (ANY) file's format
  - ⇒ `read` has no notion of reading until some special char
  - i.e. `read` cannot read a text line
  - **end of line has to be detected by application**

File Format: Text Files

- each byte has its own interpretation
  - contains the code of a printable character
- special characters: new line (`0x0A`)
- OS knows nothing about (ANY) file's format
  - ⇒ `read` has no notion of reading until some special char
  - i.e. `read` cannot read a text line
  - **end of line has to be detected by application**

File Format: Text Files

- each byte has its own interpretation
  - contains the code of a printable character
- special characters: new line (`0x0A`)
- OS knows nothing about (ANY) file's format
  - ⇒ `read` has no notion of reading until some special char
  - i.e. `read` cannot read a text line
  - **end of line has to be detected by application**

File Format: Text Files

- each byte has its own interpretation
  - contains the code of a printable character
- special characters: new line (`0x0A`)
- OS knows nothing about (ANY) file's format
  - ⇒ `read` has no notion of reading until some special char
  - i.e. `read` cannot read a text line
  - **end of line has to be detected by application**

File Format: Text Files

- each byte has its own interpretation
  - contains the code of a printable character
- special characters: new line (`0x0A`)
- OS knows nothing about (ANY) file's format
  - ⇒ `read` has no notion of reading until some special char
  - i.e. `read` cannot read a text line
  - end of line has to be detected by application

File Format: Text Files

- each byte has its own interpretation
  - contains the code of a printable character
- special characters: new line (`0x0A`)
- OS knows nothing about (ANY) file's format
  - ⇒ `read` has no notion of reading until some special char
  - i.e. `read` cannot read a text line
  - **end of line has to be detected by application**

Example: Read the First Text Line

```c
#define MAX_LINE 1024

int fd, i;
char c;
char line[MAX_LINE+1];

fd = open("file.txt", O_RDONLY);
if (fd < 0) {
    perror("Cannot open the file");
    exit(1);
}

i=0;
while ( (i < MAX_LINE) && (read(fd, &c, 1) > 0) && (c != '\n')) {
    line[i] = c;
    i++;
}

line[i] = '\0';
printf("The read line is: %s\n", line);
```

File Format: Binary Files

- any non-text file is a binary file
- actually, **any file is a binary file**
  - just a stream of bytes
- $\Rightarrow$ **applications have to know**
  - formats of files they work with (e.g. pdf, docx etc.)
  - i.e. **the way the bytes must be grouped** for good interpretation
  - i.e. the offsets real information (interpretable groups of bytes) is placed

File Format: Binary Files

- any non-text file is a binary file
- actually, **any file is a binary file**
  - just a stream of bytes
- ⇒ **applications have to know**
  - formats of files they work with (e.g. pdf, docx etc.)
  - i.e. **the way the bytes must be grouped** for good interpretation
  - i.e. the offsets real information (interpretable groups of bytes) is placed

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

File Format: Binary Files

- any non-text file is a binary file
- actually, **any file is a binary file**
  - just a stream of bytes
- $\Rightarrow$ **applications have to know**
  - formats of files they work with (e.g. pdf, docx etc.)
  - i.e. **the way the bytes must be grouped** for good interpretation
  - i.e. the offsets real information (interpretable groups of bytes) is placed

File Format: Binary Files

- any non-text file is a binary file
- actually, **any file is a binary file**
  - just a stream of bytes
- ⇒ **applications have to know**
  - formats of files they work with (e.g. `pdf`, `docx` etc.)
  - i.e. **the way the bytes must be grouped** for good interpretation
  - i.e. the offsets real information (interpretable groups of bytes) is placed

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

File Format: Binary Files

- any non-text file is a binary file
- actually, **any file is a binary file**
  - just a stream of bytes
- $\Rightarrow$ **applications have to know**
  - formats of files they work with (e.g. `pdf`, `docx` etc.)
  - i.e. **the way the bytes must be grouped** for good interpretation
  - i.e. the offsets real information (interpretable groups of bytes) is placed

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

File Format: Binary Files

- any non-text file is a binary file
- actually, **any file is a binary file**
  - just a stream of bytes
- $\Rightarrow$ **applications have to know**
  - formats of files they work with (e.g. `pdf`, `docx` etc.)
  - i.e. **the way the bytes must be grouped** for good interpretation
  - i.e. the offsets real information (interpretable groups of bytes) is placed

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

File Format: Binary Files

- any non-text file is a binary file
- actually, **any file is a binary file**
  - just a stream of bytes
- $\Rightarrow$ **applications have to know**
  - formats of files they work with (e.g. `pdf`, `docx` etc.)
  - i.e. **the way the bytes must be grouped** for good interpretation
  - i.e. the offsets real information (interpretable groups of bytes) is placed

Example: Write / Read into / from Binary Files

- process 1

```
int fd;
int number = 10;
char c = 'A';

if ((fd = creat("intfile.bin", 0644)) < 0) {
    perror("Cannot create the file");
    exit(1);
}

// write a char on the first byte
write(fd, &c, sizeof(c));

// write an integer's representation on the next four bytes
write(fd, &number, sizeof(number));
```

Example: Write / Read into / from Binary Files (cont.)

- process 2

```
int fd;
int number;

if ((fd = open("intfile.bin", O_RDONLY)) < 0) {
    perror("Cannot open the file");
    exit(1);
}

// position where WE (MUST) KNOW the integer is
// i.e. one byte after beginning of file
lseek(fd, sizeof(char), SEEK_SET);

// read four bytes from crt position
// i.e. an integer's representation
read(fd, &number, sizeof(number));
```

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
  - **a gap results in file**
  - i.e. unwritten space
- Linux does not allocate physical space for gaps
- `read` **returns zeros from a gap**
  - there is no difference between reading previously written zeros or reading from a gap
  - it is the application's responsibility to remember where gaps are (manages them)
- usage
  - core dumps for crashed processes
  - virtual HDD files (dynamically allocated)

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
  - **a gap results in file**
  - i.e. unwritten space
- Linux does not allocate physical space for gaps
- **`read` returns zeros from a gap**
  - there is no difference between reading previously written zeros or reading from a gap
  - it is the application's responsibility to remember where gaps are (manages them)
- usage
  - core dumps for crashed processes
  - virtual HDD files (dynamically allocated)

Files With Holes (Gaps). Description

- `lseek` **is allowed to position after the end of file**
- **when write** to that position
  - **a gap results in file**
  - i.e. unwritten space
- Linux does not allocate physical space for gaps
- `read` **returns zeros from a gap**
  - there is no difference between reading previously written zeros or reading from a gap
  - it is the application's responsibility to remember where gaps are (manages them)
- usage
  - core dumps for crashed processes
  - virtual HDD files (dynamically allocated)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
  - **a gap results in file**
  - i.e. unwritten space
- Linux does not allocate physical space for gaps
- `read` **returns zeros from a gap**
  - there is no difference between reading previously written zeros or reading from a gap
  - it is the application's responsibility to remember where gaps are (manages them)
- usage
  - core dumps for crashed processes
  - virtual HDD files (dynamically allocated)

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
  - **a gap results in file**
  - i.e. unwritten space
- Linux does not allocate physical space for gaps
- **`read` returns zeros from a gap**
  - there is no difference between reading previously written zeros or reading from a gap
  - it is the application's responsibility to remember where gaps are (manages them)
- usage
  - core dumps for crashed processes
  - virtual HDD files (dynamically allocated)

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
  - **a gap results in file**
  - i.e. unwritten space
- Linux does not allocate physical space for gaps
- **`read` returns zeros from a gap**
  - there is no difference between reading previously written zeros or reading from a gap
  - it is the application's responsibility to remember where gaps are (manages them)
- usage
  - core dumps for crashed processes
  - virtual HDD files (dynamically allocated)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
    - **a gap results in file**
    - i.e. unwritten space
- Linux does not allocate physical space for gaps
- **`read` returns zeros from a gap**
    - there is no difference between reading previously written zeros or reading from a gap
    - it is the application's responsibility to remember where gaps are (manages them)
- usage
    - core dumps for crashed processes
    - virtual HDD files (dynamically allocated)

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
  - **a gap results in file**
  - i.e. unwritten space
- Linux does not allocate physical space for gaps
- **`read` returns zeros from a gap**
  - there is no difference between reading previously written zeros or reading from a gap
  - it is the application's responsibility to remember where gaps are (manages them)
- usage
  - core dumps for crashed processes
  - virtual HDD files (dynamically allocated)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
  - **a gap results in file**
  - i.e. unwritten space
- Linux does not allocate physical space for gaps
- **`read` returns zeros from a gap**
  - there is no difference between reading previously written zeros or reading from a gap
  - it is the application's responsibility to remember where gaps are (manages them)
- usage
  - core dumps for crashed processes
  - virtual HDD files (dynamically allocated)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
  - **a gap results in file**
  - i.e. unwritten space
- Linux does not allocate physical space for gaps
- **`read` returns zeros from a gap**
  - there is no difference between reading previously written zeros or reading from a gap
  - it is the application's responsibility to remember where gaps are (manages them)
- usage
  - core dumps for crashed processes
  - virtual HDD files (dynamically allocated)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
    - **a gap results in file**
    - i.e. unwritten space
- Linux does not allocate physical space for gaps
- **`read` returns zeros from a gap**
    - there is no difference between reading previously written zeros or reading from a gap
    - it is the application's responsibility to remember where gaps are (manages them)
- usage
    - core dumps for crashed processes
    - virtual HDD files (dynamically allocated)

Files With Holes (Gaps). Example

- Example 1: Command line

```
dd if=/dev/zero of=file bs=1024 count=1 seek=1024
```

- Example 2: C Program

```
int fd;
  // create a zero-sized file
fd = creat("file.with.gaps.bin", 0644);
if (fd < 0) {
    perror("Cannot create file");
    exit(1);
}

  // position 4KiB after the end of file
lseek(fd, 4096, SEEK_END);
        // has no effect: only modify the crt position

  // write at crt position
write (fd, "END", 3);
        // create a gap (unwritten bytes) of 4KB,
        // followed by 3 written bytes
        // file's size is now 4096 + 3 = 4099
```

File format (text, binary, holes etc.) is the business of user applications — OS is not involved!

Input/Output Redirection. Example

```
int fd1, fd2, n1, n2;

fd1 = open("input.txt", O_RDOLNY);
fd2 = creat("output.txt", O_WRONLY);

  // reads an integer from STDIN
scanf("%d", &n1);    // calls read(0, ...);

  // redirects STDIN
close(0);            // breaks the initial association between 0 and STDIN
dup(fd1);            // associates 0 with the same open file like fd1

  // reads an integer from STDIN
scanf("%d", &n2);    // calls read(0, ...);
    // actually reads from "input.txt"

// writes an integer to STDOUT
printf("%d\n", n1); // calls write(1, ...);

// redirects STDOUT
dup2(fd2, 1);        // makes 1 a duplicate for fd2

// writes an integer to STDOUT
printf("%d\n", n2); // calls write(1, ...);
    // actually writes to "output.txt"
```

Input/Output Redirection. Illustration on Open File Tables

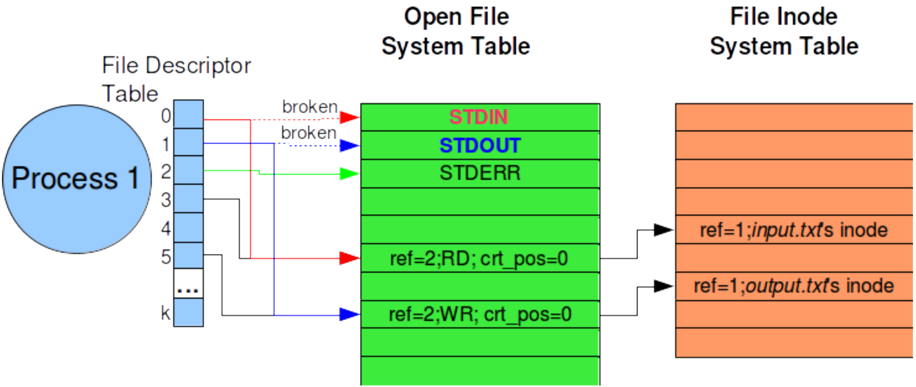


Figure: The effect of redirecting STDIN and STDOUT

Standard channel (STDIN, STDOUT, STDERR) redirection is possible because any resource in the system is modeled as a file!

Changing Permission Rights

- specify permission rights for all three groups of users
  - user: `rwx` $\rightarrow$ 111 (7)
  - group: `r-x` $\rightarrow$ 101 (5)
  - others: `--x` $\rightarrow$ 001 (1)
- example

```
chmod("file", 0751);
```

Getting Information (Metadata) About a File

- Linux metadata = **i-node (information node)**
- **each file and directory has its own, unique i-node**
- all i-nodes
  - have the same fixed size
  - placed together in one HDD area (inode area)
- ⇒ **i-node numbers** used as an index to identify an i-node
- i-node contents
  - file type, size, owner, group, permissions etc.

Getting Information (Metadata) About a File

- Linux metadata = **i-node (information node)**
- **each file and directory has its own, unique i-node**
- all i-nodes
  - have the same fixed size
  - placed together in one HDD area (inode area)
- ⇒ **i-node numbers** used as an index to identify an i-node
- i-node contents
  - file type, size, owner, group, permissions etc.

Getting Information (Metadata) About a File

- Linux metadata = **i-node (information node)**
- **each file and directory has its own, unique i-node**
- all i-nodes
  - have the same fixed size
  - placed together in one HDD area (inode area)
- ⇒ **i-node numbers** used as an index to identify an i-node
- i-node contents
  - file type, size, owner, group, permissions etc.

Getting Information (Metadata) About a File

- Linux metadata = **i-node (information node)**
- **each file and directory has its own, unique i-node**
- all i-nodes
  - have the same fixed size
  - placed together in one HDD area (inode area)
- ⇒ **i-node numbers** used as an index to identify an i-node
- i-node contents
  - file type, size, owner, group, permissions etc.

Getting Information (Metadata) About a File

- Linux metadata = **i-node (information node)**
- **each file and directory has its own, unique i-node**
- all i-nodes
  - have the same fixed size
  - placed together in one HDD area (inode area)
- ⇒ **i-node numbers** used as an index to identify an i-node
- i-node contents
  - file type, size, owner, group, permissions etc.

Getting Information (Metadata) About a File

- Linux metadata = **i-node (information node)**
- **each file and directory has its own, unique i-node**
- all i-nodes
  - have the same fixed size
  - placed together in one HDD area (inode area)
- ⇒ **i-node numbers** used as an index to identify an i-node
- i-node contents
  - file type, size, owner, group, permissions etc.

Getting Information (Metadata) About a File

- Linux metadata = **i-node (information node)**
- **each file and directory has its own, unique i-node**
- all i-nodes
  - have the same fixed size
  - placed together in one HDD area (inode area)
- ⇒ **i-node numbers** used as an index to identify an i-node
- i-node contents
  - file type, size, owner, group, permissions etc.

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Getting Information (Metadata) About a File

- Linux metadata = **i-node (information node)**
- **each file and directory has its own, unique i-node**
- all i-nodes
  - have the same fixed size
  - placed together in one HDD area (inode area)
- $\Rightarrow$ **i-node numbers** used as an index to identify an i-node
- i-node contents
  - file type, size, owner, group, permissions etc.

Example: Get a File's I-Node

```
int res;
struct stat inode;

  // gets file's inode
res = lstat("/home/os/input.txt", &inode);
if (res < 0) {
    perror("Cannot get file inode");
}

  // identify file's type
if (S_ISREG(inode.st_mode)) {
    printf("It is a file\n");
    printf("File's size [bytes]: %d\n", inode.st_size);
}

if (S_ISDIR(inode.st_mode))
    printf("It is a directory\n");

if (S_ISLNK(inode.st_mode))
    printf("It is a symbolic link\n");
```

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - ⇒ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - ⇒ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - $\Rightarrow$ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - ⇒ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - $\Rightarrow$ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - $\Rightarrow$ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - ⇒ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - $\Rightarrow$ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - $\Rightarrow$ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - $\Rightarrow$ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - $\Rightarrow$ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - $\Rightarrow$ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - $\Rightarrow$ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

Reading Directory Contents

```
DIR* dir;
struct dirent *entry;
char path[MAX_PATH];
struct stat file_info;

  // open directory
if ((dir = opendir("/home/os")) == NULL) {
    perror("Cannot open the directory");
    exit(1);
}

  // read one-by-one dir entries until NULL returned
while ( (entry = readdir(dir)) != NULL) {
      // avoid "." and ".." as they are not useful
    if (strcmp(entry->d_name, ".") &&  strcmp(entry->d_name, "..")) {

          // build the complete path = dirpath + direntry's name
        sprintf(path, "%s/%s", "/home/os", entry->d_name);

          // get element's inode
        stat(path, &file_info);

          // identify type
        if (S_ISREG(file_info.st_mode))
            printf("%s is a file\n", path);
        else
            if (S_ISDIR(file_info.st_mode))
                printf("%s is a dir\n", path);
    }
}
```

Searching for an Element in a Directory

- naive, inefficient way (does not benefit from the specialized directory structure)

```
DIR* dir;
struct dirent *entry;
char path[MAX_PATH];
struct stat file_info;

  // open directory
if ((dir = opendir("/home/os")) == NULL) {
    perror("Cannot open the directory");
    exit(1);
}

  // read one-by-one dir entries until NULL returned
while ( (entry = readdir(dir)) != NULL) {
    if (strcmp(entry->d_name, SEARCHED_NAME) == 0) {
          // build the complete path = dirpath + direntry's name
        sprintf(path, "%s/%s", "/home/os", entry->d_name);
        printf("Found %s\n", path);
        break;
    }
}
```

Searching for an Element in a Directory (cont.)

- efficient way (benefit from the specialized directory structure)

```
char path[MAX_PATH];
struct stat file_info;

// build the complete path = dirpath + SEARCHED_NAME
sprintf(path, "%s/%s", "/home/os", SEARCHED_NAME);

// try getting file info
// succeds if file exists, fails otherwise
if (stat(path, &file_info) > 0) {
    printf("Found %s\n", path);
} else {
    perror("Cannot get file's inode");
    exit(1);
}
```

A directory is presented and interacted with as a collection of elements!
Searching, traversing, creating, changing could be performed only at directory element level.

Security Considerations: File-Path Traversal

- context
  - an application wants to **confine access** (of its users) **to a subdirectory** (subtree)
  - very common to Web applications
- problem
  - when user controls (specifies) parts of the file path
  - ".." **could be used to evade** the restricted directory
- solution
  - check for ".." in user-controlled file paths

Security Considerations: File-Path Traversal

- context
  - an application wants to **confine access** (of its users) **to a subdirectory** (subtree)
  - very common to Web applications
- problem
  - when user controls (specifies) parts of the file path
  - ".." **could be used to evade** the restricted directory
- solution
  - check for ".." in user-controlled file paths

Security Considerations: File-Path Traversal

- context
  - an application wants to **confine access** (of its users) **to a subdirectory** (subtree)
  - very common to Web applications
- problem
  - when user controls (specifies) parts of the file path
  - ".." **could be used to evade** the restricted directory
- solution
  - check for ".." in user-controlled file paths

Security Considerations: File-Path Traversal

- context
  - an application wants to **confine access** (of its users) **to a subdirectory** (subtree)
  - very common to Web applications
- problem
  - when user controls (specifies) parts of the file path
  - **“..” could be used to evade** the restricted directory
- solution
  - check for “..” in user-controlled file paths

Security Considerations: File-Path Traversal

- context
  - an application wants to **confine access** (of its users) **to a subdirectory** (subtree)
  - very common to Web applications
- problem
  - when user controls (specifies) parts of the file path
  - **“..” could be used to evade** the restricted directory
- solution
  - check for “..” in user-controlled file paths

Security Considerations: File-Path Traversal

- context
  - an application wants to **confine access** (of its users) **to a subdirectory** (subtree)
  - very common to Web applications
- problem
  - when user controls (specifies) parts of the file path
  - **“..” could be used to evade** the restricted directory
- solution
  - check for “..” in user-controlled file paths

Security Considerations: File-Path Traversal

- context
  - an application wants to **confine access** (of its users) **to a subdirectory** (subtree)
  - very common to Web applications
- problem
  - when user controls (specifies) parts of the file path
  - **".." could be used to evade** the restricted directory
- solution
  - check for ".." in user-controlled file paths

Security Considerations: File-Path Traversal

- context
  - an application wants to **confine access** (of its users) **to a subdirectory** (subtree)
  - very common to Web applications
- problem
  - when user controls (specifies) parts of the file path
  - **“..” could be used to evade** the restricted directory
- solution
  - check for “..” in user-controlled file paths

File-Path Traversal Illustration

- vulnerable code

```
char filepath[MAX_PATH];

scanf("%s", filename);  // <--- provided by the user (a possible attacker)!!!

snprintf(filepath, MAX_PATH, "/home/restricted_user/%s", filename);

fd = open(filepath, O_RDONLY);
...  // display the file's contents
```

- malicious value for "`filename`"

```
"../../../../../../etc/passwd"
```

- secure code

```
char filepath[MAX_PATH];

scanf("%s", filename);
if (strstr(filename, "..") != NULL)
    return;
snprintf(filepath, MAX_PATH, "/home/restricted_user/%s", filename);

fd = open(filepath, O_RDONLY);
```

Never trust the users! Check for ".." in the given file paths, if want avoiding path traversal!

Trace System Calls of An Application

- Can be done using
  - the `strace` command
  - the `ptrace` system call
- Example
  - create a file of 8 KB

    ```
    dd if=/dev/zero of=file1 bs=4096 count=2
    ```

  - trace the `cp` command's execution

    ```
    strace cp file1 file2
    ```

  - main part of the output

Trace System Calls of An Application (cont.)

```
stat64("file1", {st_mode=S_IFREG|0644, st_size=8192, ...}) = 0
open("file1", O_RDONLY|O_LARGEFILE)     = 3
stat64("file2", 0xbf850e94)             = -1 ENOENT (No such file or directory)
open("file2", O_WRONLY|O_CREAT|O_LARGEFILE, 0100644) = 4
fstat64(4, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
fstat64(3, {st_mode=S_IFREG|0644, st_size=8192, ...}) = 0
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\"..., 4096) = 4096
write(4, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\"..., 4096) = 4096
write(4, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
read(3, "", 4096)                       = 0
close(4)                                = 0
close(3)                                = 0
```

Questions (3)

- Show the **current position** for each open file (i.e. file descriptor) and the **contents of "buf"** after the execution of **each instruction** from the code below, supposing they are ALL executed successfully.

```
char buf[100];

int fd1 = open("file1", O_RDONLY);
int fd2 = open("file1", O_RDWR);
int fd3 = open("file2", O_RDWR);
int fd4 = dup(fd3);
write(fd2, "This is funny, isn't it?", 10);
read(fd1, buf, 4);
write(fd3, "1234567890", 10);
lseek(fd3, 0, SEEK_SET);
read(fd3, buf, 5);
read(fd4, buf, 5);
```

Outline

Access File Data

```
HANDLE WINAPI CreateFile(LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode,
          LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDisposition,
          DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);

BOOL WINAPI ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead,
          LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);

BOOL WINAPI WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite,
          LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped);

DWORD WINAPI SetFilePointer(HANDLE hFile, LONG lDistanceToMove,
                          PLONG lpDistanceToMoveHigh, DWORD dwMoveMethod);
```

Manipulate Files

```
BOOL WINAPI MoveFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName);

BOOL WINAPI SetEndOfFile(HANDLE hFile);

DWORD WINAPI GetFileAttributes(LPCTSTR lpFileName);

BOOL WINAPI GetFileInformationByHandle(HANDLE hFile,
          LPBY_HANDLE_FILE_INFORMATION lpFileInformation);

BOOL WINAPI GetFileSecurity(LPCTSTR lpFileName, SECURITY_INFORMATION RequestedInformation,
          PSECURITY_DESCRIPTOR pSecurityDescriptor, DWORD nLength,LPDWORD lpnLengthNeeded);

BOOL WINAPI SetFileSecurity(LPCTSTR lpFileName, SECURITY_INFORMATION SecurityInformation,
          PSECURITY_DESCRIPTOR pSecurityDescriptor);
```

Manipulate Directories

```
BOOL WINAPI CreateDirectory(LPCTSTR lpPathName, LPSECURITY_ATTRIBUTES lpSecurityAttributes);

BOOL WINAPI CreateHardLink(LPCTSTR lpFileName, LPCTSTR lpExistingFileName,
          LPSECURITY_ATTRIBUTES lpSecurityAttributes);

BOOLEAN WINAPI CreateSymbolicLink(LPTSTR lpSymlinkFileName, LPTSTR lpTargetFileName, DWORD dwFlags);

BOOL WINAPI DeleteFile(LPCTSTR lpFileName);

HANDLE WINAPI FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA lpFindFileData);

BOOL WINAPI FindNextFile(HANDLE hFindFile, LPWIN32_FIND_DATA lpFindFileData);

BOOL WINAPI RemoveDirectory(LPCTSTR lpPathName);
```

Sparse Files

```
LARGE_INTEGER   FileSize;


FileSize.QuadPart = 8 * 1024 * 1024 * 1024;

FileHandle = CreateFile("file",
                GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE,
                NULL, CREATE_NEW, FILE_FLAG_NO_BUFFERING, NULL);


DeviceIoControl(FileHandle, FSCTL_SET_SPARSE, NULL, 0,
                NULL, 0, &BytesReturned, NULL);

SetFilePointerEx(FileHandle, FileSize, 0, FILE_BEGIN);

SetEndOfFile(FileHandle);
```

Alternate Data Streams. Command Line Examples

- Example 1

```
echo hello > file.txt:alternatestream.txt

more < file.txt:alternatestream.txt

notepad file.txt:alternatestream.txt
```

- Example 2

```
type c:\windows\systems32\calc.exe > file.txt:calc.exe

start .\file.txt:calc.exe
```

- Getting alternate streams

```
http://www.microsoft.com/technet/sysinternals/default.mspx

streams [-s] [-d] <file\_name>
```

Alternate Data Streams. C Program Example

```
HANDLE inhandle, outhandle;
char buffer[BUF_SIZE];
int count, s;
DWORD ocnt;

inhandle = CreateFile("sursa.txt", GENERIC_READ, 0,
                      NULL, OPEN_EXISTING, 0, NULL);

outhandle = CreateFile("dest.txt:file.txt", GENERIC_WRITE,
                       0, NULL, CREATE_ALWAYS,
                       FILE_ATTRIBUTE_NORMAL, NULL);

/* copy the file  */
do {
      s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);

      if (s && count > 0)
          WriteFile(outhandle, buffer, count, &ocnt, NULL);

} while (s>0 && count>0);
```

Outline

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

Lessons Learned

- the “file” is a sequence of bytes
  - each application should manage its own files’ format
- the “directory” is a collection of elements
  - traverse it element by element
  - search for elements
- do not trust users!
  - set strong permission rights!
  - check for path traversal elements!

Lessons Learned

- the “file” is a sequence of bytes
  - each application should manage its own files’ format
- the “directory” is a collection of elements
  - traverse it element by element
  - search for elements
- do not trust users!
  - set strong permission rights!
  - check for path traversal elements!

Lessons Learned

- the “file” is a sequence of bytes
  - each application should manage its own files’ format
- the “directory” is a collection of elements
  - traverse it element by element
  - search for elements
- do not trust users!
  - set strong permission rights!
  - check for path traversal elements!

Lessons Learned

- the "file" is a sequence of bytes
  - each application should manage its own files' format
- the "directory" is a collection of elements
  - traverse it element by element
  - search for elements
- do not trust users!
  - set strong permission rights!
  - check for path traversal elements!

Lessons Learned

- the “file” is a sequence of bytes
  - each application should manage its own files’ format
- the “directory” is a collection of elements
  - traverse it element by element
  - search for elements
- do not trust users!
  - set strong permission rights!
  - check for path traversal elements!

Lessons Learned

- the "file" is a sequence of bytes
  - each application should manage its own files' format
- the "directory" is a collection of elements
  - traverse it element by element
  - search for elements
- do not trust users!
  - set strong permission rights!
  - check for path traversal elements!

Lessons Learned

- the “file” is a sequence of bytes
  - each application should manage its own files’ format
- the “directory” is a collection of elements
  - traverse it element by element
  - search for elements
- do not trust users!
  - set strong permission rights!
  - check for path traversal elements!

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- the “file” is a sequence of bytes
  - each application should manage its own files’ format
- the “directory” is a collection of elements
  - traverse it element by element
  - search for elements
- do not trust users!
  - set strong permission rights!
  - check for path traversal elements!