Exercise: List Data Structure

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

List data strucure. . .

A List is represented via a struct:

List data strucure. . .

A List is represented via a struct:

```
typedef struct List {
  int value;
  struct List *next;
} List;
```

List data strucure. . .

A List is represented via a struct:

```
typedef struct List {
  int value;
  struct List *next;
} List;
```

Remark: NULL represent the *empty list*.

Operations. . .

Empty list:

Operations. . .

Empty list:

```
List* empty();
```

Operations...

Empty list:

```
List* empty();
```

```
List* empty() {
  return NULL;
}
```

Operations...

Empty list:

```
List* empty();
```

```
List* empty() {
  return NULL;
}
```

Check if empty:

Operations...

Empty list:

```
List* empty();
```

```
List* empty() {
  return NULL;
}
```

Check if empty:

```
int isEmpty( List* );
```

Operations. . .

Empty list:

```
List* empty();
```

```
List* empty() {
  return NULL;
}
```

Check if empty:

```
int isEmpty( List* );
```

```
int isEmpty( List* list ) {
  return list == NULL;
}
```

Operations. . .

Add an element:

Operations...

Add an element:

```
List* add( List* , int );
```

Operations...

Add an element:

```
List* add( List* , int );


List* createListElement( int v, List *next ) {
  List* newList = malloc(sizeof(List));
  newList->value = v;
  newList->next = next;
  return newList;
}

List* add( List* list , int v ) {
  return createListElement( v , list );
}
```

Operations. . .

Number of elements in a list:

Operations. . .

Number of elements in a list:

```
int size( List* );
```

Operations...

Number of elements in a list:

```
int size( List* );
```

```
int size( List* list ) {
  int counter = 0;
  while (list != NULL) {
    list = list->next;
    counter++;
  }
  return counter;
}
```

Operations. . .

Check if an element occurs in the list:

Operations. . .

Check if an element occurs in the list:

```
int contains( List* , int );
```

Operations...

Check if an element occurs in the list:

```
int contains( List* , int );
```

```
int contains( List* list , int v ) {
  int result = 0;
  while ((!result)&&(list != NULL)) {
    result = (list->value==v);
    list = list->next;
  }
  return result;
}
```

Operations. . .

Remove an element from the list:

Operations...

Remove an element from the list:

```
List* remove( List* , int );
```

Operations. . .

Remove an element from the list:

```
List* remove( List* , int );


List* remove( List* list , int v ) {
  if (list == NULL) {
    return list;
  }
  if (list->value == v) {
    List* result = list->next;
    free(list);
    return result;
  }
  list->next = remove(list->next,v);
  return list;
}
```

Operations. . .

Add an element (in the correct order):

Operations...

Add an element (in the correct order):

```
List* addInOrder( List* , int );
```

Operations. . .

Add an element (in the correct order):

```
List* addInOrder( List* , int );
```

```
List* addInOrder( List* list , int v ) {
  if ((list == NULL)||(list->value>v)) {
    return createListElement(v,NULL);
  } else {
    list->next = addInOrder(list->next,v);
    return list;
  }
}
```

Operations. . .

Sort a list:

Operations. . .

Sort a list:

```
List* sort( List* list );
```

Operations...

Sort a list:

```
List* sort( List* list );
```

```
List* sort( List* list ) {
  List* result = NULL;
  while (list != NULL) {
    result = addInOrder( result , list->value );
    list = list->next;
  }
  return result;
}
```

To be continued. . .

Concepts of System Programming

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Files and the Filesystem

The file is the most basic and fundamental abstraction in Linux.

Files and the Filesystem

The file is the most basic and fundamental abstraction in Linux.

Linux follows the everything-is-a-file philosophy

Files and the Filesystem

The file is the most basic and fundamental abstraction in Linux.

Linux follows the everything-is-a-file philosophy...

... much interaction occurs via reading of and writing to files;

Files and the Filesystem

The file is the most basic and fundamental abstraction in Linux.

Linux follows the everything-is-a-file philosophy. . .

. . . much interaction occurs via reading of and writing to files;

. . . examples are stardard input (*stdin*), standard output (*stdout*) and standard error (*sdterr*).

Files and the Filesystem

The file is the most basic and fundamental abstraction in Linux.

Linux follows the everything-is-a-file philosophy. . .

. . . much interaction occurs via reading of and writing to files;

. . . examples are stardard input (*stdin*), standard output (*stdout*) and standard error (*sdterr*).

Files and the Filesystem

The file is the most basic and fundamental abstraction in Linux.

Linux follows the everything-is-a-file philosophy...

... much interaction occurs via reading of and writing to files;

... examples are stardard input (*stdin*), standard output (*stdout*) and standard error (*sdterr*).

To be accessed a file must first be opened.

Files and the Filesystem

The file is the most basic and fundamental abstraction in Linux.

Linux follows the everything-is-a-file philosophy...

... much interaction occurs via reading of and writing to files;

... examples are stardard input (*stdin*), standard output (*stdout*) and standard error (*sdterr*).

To be accessed a file must first be opened.

When a file is opened it is referenced via a file descriptor (fd). In Linux this is an integer.

File Types. . .

File Types. . .

Regular Files: A regular file contains bytes of data, organised into a linear array called a byte stream. . .

- any byte in the file can be read or written;

File Types. . .

Regular Files: A regular file contains bytes of data, organised into a linear array called a byte stream. . .

- any byte in the file can be read or written;
- operations are performed at a *location* that is file position or offset;

File Types. . .

Regular Files: A regular file contains bytes of data, organised into a linear array called a byte stream. . .

- any byte in the file can be read or written;
- operations are performed at a *location* that is file position or offset;
- the maximum position (i.e. the max number of bytes) is limited by the data type used to represent it;

File Types. . .

Regular Files: A regular file contains bytes of data, organised into a linear array called a byte stream. . .

- any byte in the file can be read or written;
- operations are performed at a *location* that is file position or offset;
- the maximum position (i.e. the max number of bytes) is limited by the data type used to represent it;
- The size of a file is measured in bytes and is called its length.

File Types...

Regular Files: A regular file contains bytes of data, organised into a linear array called a byte stream...

- any byte in the file can be read or written;
- operations are performed at a *location* that is file position or offset;
- the maximum position (i.e. the max number of bytes) is limited by the data type used to represent it;
- The size of a file is measured in bytes and is called its length.

File Types. . .

Regular Files: A regular file contains bytes of data, organised into a linear array called a byte stream. . .

- any byte in the file can be read or written;
- operations are performed at a *location* that is file position or offset;
- the maximum position (i.e. the max number of bytes) is limited by the data type used to represent it;
- The size of a file is measured in bytes and is called its length.

A file can be accessed via a filename or via an inode (*information node*).

File Types...

Regular Files: A regular file contains bytes of data, organised into a linear array called a byte stream...

- any byte in the file can be read or written;
- operations are performed at a *location* that is file position or offset;
- the maximum position (i.e. the max number of bytes) is limited by the data type used to represent it;
- The size of a file is measured in bytes and is called its length.

A file can be accessed via a filename or via an inode (*information node*).

An inode, that is identified by a inode number, stores metadata associated with a file, such as its modification timestamp, owner, type, length, and the location of the file's data-but no filename!

File Types. . .

Accessing a file via its inode number is cumbersome (and also a potential security hole), so files are always opened from user space by a name, not an inode number.

File Types. . .

Accessing a file via its inode number is cumbersome (and also a potential security hole), so files are always opened from user space by a name, not an inode number.

Directories: are used to provide the names with which to access files.

File Types. . .

Accessing a file via its inode number is cumbersome (and also a potential security hole), so files are always opened from user space by a name, not an inode number.

Directories: are used to provide the names with which to access files.

Link: is a pair name-inode.

File Types. . .

Accessing a file via its inode number is cumbersome (and also a potential security hole), so files are always opened from user space by a name, not an inode number.

Directories: are used to provide the names with which to access files.

Link: is a pair name-inode. There are two kinds of links:

- hard links;
- symbolic links.

File Types. . .

Accessing a file via its inode number is cumbersome (and also a potential security hole), so files are always opened from user space by a name, not an inode number.

Directories: are used to provide the names with which to access files.

Link: is a pair name-inode. There are two kinds of links:

- hard links;
- symbolic links.

Special files: are kernel objects that are represented as files (e.g. USB or serial ports).

Filesystems and namespaces

Linux, like all Unix systems, provides a global and unified namespace of files and directories.

Filesystems and namespaces

Linux, like all Unix systems, provides a global and unified namespace of files and directories.

A filesystem is a collection of files and directories in a formal and valid hierarchy.

Filesystems and namespaces

Linux, like all Unix systems, provides a global and unified namespace of files and directories.

A filesystem is a collection of files and directories in a formal and valid hierarchy.

Filesystems may be individually added to and removed from the global namespace of files and directories.

These operations are called mounting and unmounting.

Filesystems and namespaces

Linux, like all Unix systems, provides a global and unified namespace of files and directories.

A filesystem is a collection of files and directories in a formal and valid hierarchy.

Filesystems may be individually added to and removed from the global namespace of files and directories.

These operations are called mounting and unmounting.

Each filesystem is mounted to a specific location in the namespace, known as a mount point.

Processes

Processes are object code in execution: active, running programs. They cconsist of data, resources, state, and a virtualised computer.

[1] bss=Block Started by Symbols

Processes

Processes are object code in execution: active, running programs. They cconsist of data, resources, state, and a virtualised computer.

Processes begin life as executable object code, which is machine-runnable code in an executable format that the kernel understands. The format most common in Linux is called Executable and Linkable Format (ELF),

[1] bss=Block Started by Symbols

Processes

Processes are object code in execution: active, running programs. They cconsist of data, resources, state, and a virtualised computer.

Processes begin life as executable object code, which is machine-runnable code in an executable format that the kernel understands. The format most common in Linux is called Executable and Linkable Format (ELF),

The executable format contains metadata, and multiple sections of code and data:

- text section;
- data section;
- bss section[1];
- absolute section;
- undefined section.

[1] bss=Block Started by Symbols

Users and Groups

Authorisation in Linux is provided by users and groups.

Users and Groups

Authorisation in Linux is provided by users and groups.

Each user is associated with a unique positive integer called the user ID (*uid*).

Users and Groups

Authorisation in Linux is provided by users and groups.

Each user is associated with a unique positive integer called the user ID (*uid*).

Each user belongs to one or more groups, including a primary or login group. Each group is identified via a group id (*gid*).

Users and Groups

Authorisation in Linux is provided by users and groups.

Each user is associated with a unique positive integer called the user ID (*uid*).

Each user belongs to one or more groups, including a primary or login group. Each group is identified via a group id (*gid*).

Each process is in turn associated with exactly one *uid*, which identifies the user running the process, and is called the process's real *uid*.

Permissions

In Unix/Linux each file is associated with:

- an owning user;
- an owning group;
- and and three sets of permission bits.

Permissions

In Unix/Linux each file is associated with:

- an owning user;
- an owning group;
- and and three sets of permission bits.

The bits describe the ability of the owning user, the owning group, and everybody else to read, write, and execute the file.

Permissions

In Unix/Linux each file is associated with:

- an owning user;
- an owning group;
- and and three sets of permission bits.

The bits describe the ability of the owning user, the owning group, and everybody else to read, write, and execute the file.

The owners and the permissions are stored in the file's *inode*.

Permissions

In Unix/Linux each file is associated with:

- an owning user;
- an owning group;
- and and three sets of permission bits.

The bits describe the ability of the owning user, the owning group, and everybody else to read, write, and execute the file.

The owners and the permissions are stored in the file's *inode*.

Octal values can be used to set permissions.

Error Handling

In system programming, an error is signified via a function's return value and described via a special variable, `errno`.

Error Handling

In system programming, an error is signified via a function's return value and described via a special variable, `errno`.

glibc transparently provides `errno` support for both library and system calls.

Error Handling

In system programming, an error is signified via a function's return value and described via a special variable, errno.

glibc transparently provides errno support for both library and system calls.

This variable is declared in <errno.h> as follows:

```
extern int errno;
```

Error Handling

In system programming, an error is signified via a function's return value and described via a special variable, errno.

glibc transparently provides errno support for both library and system calls.

This variable is declared in <errno.h> as follows:

```
extern int errno;
```

The errno variable may be read or written directly.

Error Handling

The C library provides a handful of functions for translating an errno value to the corresponding textual representation:

Error Handling

The C library provides a handful of functions for translating an errno value to the corresponding textual representation:

```
#include <stdio.h>

void perror (const char *str);
```

Error Handling

The C library provides a handful of functions for translating an errno value to the corresponding textual representation:

```
#include <stdio.h>

void perror (const char *str);
```

This function prints to *stderr* (standard error) the string representation of the current error described by errno, prefixed by the string pointed at by str, followed by a colon.

To be continued. . .

Input/Output

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

File descriptors. . .

Before a file can be read from or written to, it must be opened.

File descriptors. . .

Before a file can be read from or written to, it must be opened.

The kernel maintains a per-process list of open files, called the file table.

File descriptors. . .

Before a file can be read from or written to, it must be opened.

The kernel maintains a per-process list of open files, called the file table.

Definition:A process is a running program!

File descriptors. . .

Before a file can be read from or written to, it must be opened.

The kernel maintains a per-process list of open files, called the file table.

Definition:A process is a running program!

File table is indexed via nonnegative integers known as file descriptors (often abbreviated fds).

File descriptors. . .

Before a file can be read from or written to, it must be opened.

The kernel maintains a per-process list of open files, called the file table.

Definition:A process is a running program!

File table is indexed via nonnegative integers known as file descriptors (often abbreviated fds).

Each entry in the list contains information about the file (permissions, location,. . .).

File descriptors. . .

Before a file can be read from or written to, it must be opened.

The kernel maintains a per-process list of open files, called the file table.

Definition:A process is a running program!

File table is indexed via nonnegative integers known as file descriptors (often abbreviated fds).

Each entry in the list contains information about the file (permissions, location,. . .).

File descriptors are obtained when a file is opened, and used to perform file operations..

File descriptors...

File descriptors are represented by the *C* `int` type.

File descriptors. . .

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

File descriptors...

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

- start from `0` and go up to one less than this maximum value;

File descriptors. . .

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

- start from `0` and go up to one less than this maximum value;
- by default this max value is `1024`, however can be increased up to `1048576`;

File descriptors. . .

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

- start from `0` and go up to one less than this maximum value;
- by default this max value is `1024`, however can be increased up to `1048576`;
- -1 is used to indicate an error.

File descriptors. . .

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

- start from `0` and go up to one less than this maximum value;
- by default this max value is `1024`, however can be increased up to `1048576`;
- -1 is used to indicate an error.

File descriptors. . .

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

- start from 0 and go up to one less than this maximum value;
- by default this max value is 1024, however can be increased up to 1048576;
- -1 is used to indicate an error.

Each process has at least three file descriptors:

- standard input: 0 (STDIN_FILENO);
- standard output: 1 (STDOUT_FILENO);
- standard error: 2 (STDERR_FILENO);.

Opening files. . .

A file is opened and a file descriptor is obtained with the open() system call:

Opening files. . .

A file is opened and a file descriptor is obtained with the open() system call:

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

int open (const char *name, int flags);
int open (const char *name, int flags, mode_t mode);
```

Opening files. . .

A file is opened and a file descriptor is obtained with the open() system call:

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

int open (const char *name, int flags);
int open (const char *name, int flags, mode_t mode);
```

Example:

```
int fd;
fd = open( "/home/piton/potions" , O_RDONLY );
if (fd<0) {
  // Error!
}
```

Opening flags. . .

flags argument may be bitwise-ORed with zero or more of the following values, modifying the behavior of the open request:

- O_RDONLY
- O_WRONLY
- O_RDWR
- O_APPEND
- O_ASYNC
- O_CLOEXEC
- O_CREAT
- O_DIRECT
- O_DIRECTORY
- O_EXCL
- O_LARGEFILE
- . . .

Opening modes. . .

Parameter `mode` provides the permissions of the newly created file.

Opening modes. . .

Parameter `mode` provides the permissions of the newly created file.

It is composed by an octal value with three digits represeting:

- User permission;
- Group permission;
- Others permiossion.

Opening modes. . .

Parameter `mode` provides the permissions of the newly created file.

It is composed by an octal value with three digits represeting:

- User permission;
- Group permission;
- Others permiossion.

Each digit consists of three bits *rwx* indicating *read*, *write* and *exec* permissions.

Example. . .

```
int fd;
fd = open(file, O_WRONLY | O_CREAT | O_TRUNC, 0664);
if (fd == -1) {
  /* error */
}
```

Creating a file. . .

The combination of O_WRONLY | O_CREAT | O_TRUNC is so common that a system call exists to provide just that behaviour:

Creating a file. . .

The combination of O_WRONLY | O_CREAT | O_TRUNC is so common that a system call exists to provide just that behaviour:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *name, mode_t mode);
```

Creating a file...

The combination of O_WRONLY | O_CREAT | O_TRUNC is so common that a system call exists to provide just that behaviour:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *name, mode_t mode);
```

This is not a typo!

Creating a file...

The combination of O_WRONLY | O_CREAT | O_TRUNC is so common that a system call exists to provide just that behaviour:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *name, mode_t mode);
```

This is not a typo!

```
int fd;
fd = creat (filename, 0644);
if (fd == 1) {
  /* error */
}
```

Reading from files:

The most basic mechanism used for reading is the `read()` system call:

Reading from files:

The most basic mechanism used for reading is the read() system call:

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t len);
```

Reading from files:

The most basic mechanism used for reading is the read() system call:

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t len);
```

Example:

```
unsigned long word;
ssize_t nr;

/* read a couple bytes into 'word' from 'fd' */
nr = read (fd, &word, sizeof (unsigned long));
if (nr == 1) {
  /* error */
}
```

read(): Return Values

System call read returns the number of bytes that are read from the file:

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;
- ret is 0, end-of-file has been reached;

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;
- ret is 0, end-of-file has been reached;
- ret is -1, there is an error, a code is store in variable errno:

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;
- ret is 0, end-of-file has been reached;
- ret is −1, there is an error, a code is store in variable errno:
  - EINTR (operation has been suspended, and it can be reissued);

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;
- ret is 0, end-of-file has been reached;
- ret is −1, there is an error, a code is store in variable errno:
  - EINTR (operation has been suspended, and it can be reissued);
  - EAGAIN (no data is available, operation should be reissued later);

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;
- ret is 0, end-of-file has been reached;
- ret is −1, there is an error, a code is store in variable errno:
  - EINTR (operation has been suspended, and it can be reissued);
  - EAGAIN (no data is available, operation should be reissued later);
  - ...

Example: reading all bytes

```
ssize_t ret;
while (len != 0 && (ret = read (fd, buf, len)) != 0) {
  if (ret==-1) {
    if (errno == EINTR)
      continue;
        perror ("read");
    break;
  }

  len -= ret;
  buf += ret;
}
```

Writing on files

The most basic and common system call used for writing is write():

Writing on files

The most basic and common system call used for writing is write():

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
```

Writing on files

The most basic and common system call used for writing is write():

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
```

A call to write() writes up to count bytes starting at buf to the current position of the file referenced by the file descriptor fd.

Writing on files

The most basic and common system call used for writing is write():

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
```

A call to write() writes up to count bytes starting at buf to the current position of the file referenced by the file descriptor fd.

Example:

```
const char *buf = "My ship is solid!";
ssize_t nr;
/* write the string in 'buf' to 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1) {
  /* error */
}
```

To be continued. . .