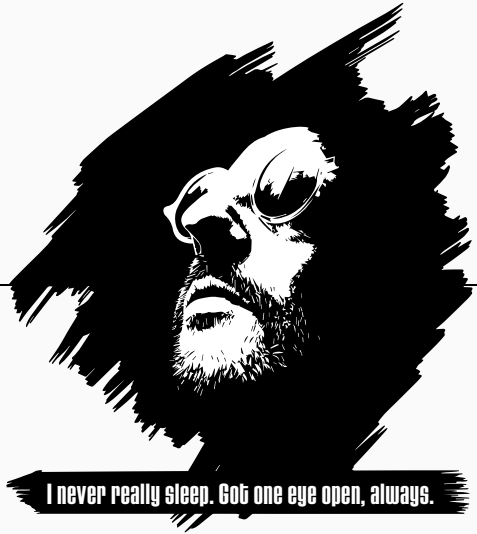
myFind - Apping - Presentation

ACU 2020 Team



Copyright

Rules

- You must have downloaded your copy from the Assistants' Intranet <https://intra.assistants.epita.fr>.
- This document is strictly personal and must **not** be passed on to someone else.
- Non-compliance with these rules can lead to severe sanctions.

MyFind

- Implement a simplified version of the `find (1)` command.
- Implement some actions / tests / operators of the original `find`.
- Implement a test suite.

Usage

```
./myfind [options] [starting-points] [expression]
```

Example

```
42sh$ ls foo
bar baz

42sh$ ./myfind foo
foo
foo/bar
foo/baz

42sh$ ./myfind foo -name bar
foo/bar
```

Notions

To realize this project you'll have to handle:

- Command line parsing
- Abstract Syntax Tree (AST)
- File manipulation
- Process execution

Optional exercises

Before starting this project, we suggest you to complete these basic exercises:

- Simple `ls`
- Simple `stat`
- AST evaluation

They're optional, but will help you to properly start the `my_find` project.

Starting points

- List of directories where the searching files start.
- If none use the current directory to begin the search.

Usage

- `./myfind [options] [starting-points] [expression]`

Expression

- An expression is a list of tests or actions to apply to each file.
- Different types of expression elements:
  - Tests
  - Actions
  - Operators
- Each test or action may be separated with operators.

Usage

- `./myfind [options] [starting-points] [expression]`

Example

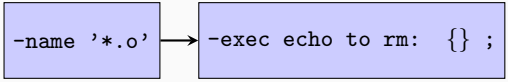


Figure 1: An expression list

```
42sh$ ./myfind . -name '*.o' -exec echo to rm: {} \;
to rm: ./d.o
to rm: ./c.o
to rm: ./b.o
to rm: ./a.o
```

Options

- `-P` myfind never follows symbolic links (default behaviour)
- `-L` myfind follows symbolic links
- `-H` myfind does not follow symbolic links, except if it is given in the command line
- `-d` myfind shall traverse the file system with a post-order processing.

The expression list

The expression list

- The expression list begins after the starting point.
- Series of tests or action with possibly arguments and separated by operators
- Examples:
  - Test: `-name <name>` `-newer` ...
  - Action: `-print` `-delete` ...
  - Operators: `-o` `-a`

```
./myfind [options] [starting-points] [expression]
```

Expression list

- Expressions are linked.
- Each expression return a value: `True` or `False`.
- Don't evaluate the nth expression if the $n-1$th return `False`.

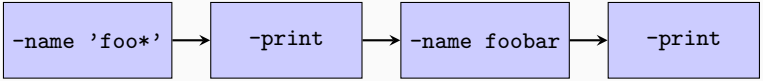


Figure 2: An expression list

Example

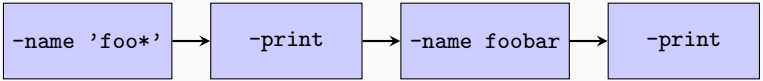


Figure 3: An expression list

```
42sh$ ls qux
foo  foobar
42sh$ ./myfind qux -name 'foo*' -print -name 'foobar' -print
qux/foobar
qux/foobar
qux/foo
```

The operators

- Modify the evaluation flow.
- And, Or, Not: `-a -o !`
- Parentheses: `( )`

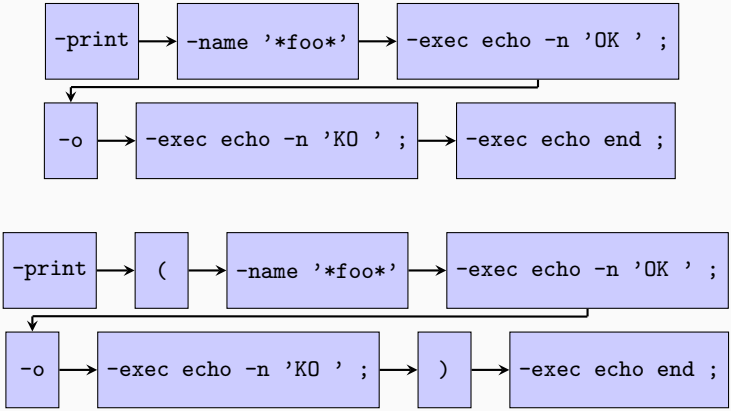


Figure 4: Operators have an importance

Example - Without parentheses

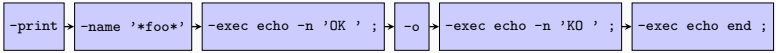


Figure 5: Without parentheses

```
42sh$ ls qux
acu  chair  foo  foobar  megafoobar
42sh$ ./myfind qux -print -name '*foo*' -exec echo -n 'OK ' \; \
                -o -exec echo -n 'KO ' \; -exec echo end \;
qux
KO end
qux/acu
KO end
qux/chair
KO end
qux/megafoobar
OK qux/foobar
OK qux/foo
OK 42sh$
```

Example - With parentheses

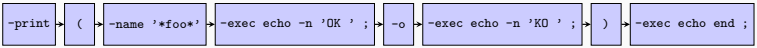


Figure 6: With parentheses

```
42sh$ ls qux
acu  chair  foo  foobar  megafoobar
42sh$ ./myfind qux -print \( -name '*foo*' -exec echo -n 'OK ' \; -o \
               -exec echo -n 'KO ' \; \) -exec echo end \;
```

Example - With parentheses

```
qux
KO end
qux/acu
KO end
qux/chair
KO end
qux/megafoobar
OK end
qux/foobar
OK end
qux/foo
OK end
42sh$
```

Useful notions

Definition

```
struct function
{
    char *name;
    int (*fun)(...);
};
```

- Stored in array
- Useful to avoid “if machines”

Example

```
struct function funs[2] =
{
    {
        .name = "print",
        .fun = print_fun
    },
    {
        .name = "delete",
        .fun = delete_fun
    }
};

for (int i = 0; i < 2; ++i)
    if (strcmp("print", funs[i].name) == 0)
        funs[i].fun();
```

Opendir

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
```

- Allows you to manipulate directory entries
- You can iterate over those entries

Opendir - Example

```c
DIR *dir = opendir("/home/acu");
struct dirent *entry = readdir(dir);
for (; entry; entry = readdir(dir))
    printf("%s\n", entry->d_name);
```

Stat

- `man 2 stat`
- Information about the file:
  - Type / Protection (`st_mode`)
  - User owner (`st_uid`)
  - Group owner (`st_gid`)
  - Size (`st_size`)
  - ...

Fd handling

Many of the file-related syscalls allow you to use file descriptors instead of strings.

While you have the fd, you can manipulate the resources.

- `stat(2)`: `fstat(2)`, `fstatat(2)`
- `open(2)`: `openat(2)`
- `opendir(3)`: `fdopendir(3)`
- ...

Read the man of all allowed functions/syscalls, it will be helpful.

Execution: execve

Syscall that replaces the memory image of a process and executes a new program: `execve(2)`.

exec

```
man 3 exec
```

Many wrappers around execve exist.

Example

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    char *cmd[] = {"xeyes", "-center", "blue", NULL};
    if (execvp(cmd[0], cmd) == -1)
        return 1;

    puts("This will never be seen");
    return 0;
}
```

Unix process

- Program currently running
- Processes have a **PID** (*Process IDentifier*), a unique identifier in the system.
- The only way to create a process in Unix: *duplicate* the current process (memory, stack ...).
- When duplicating a process, some resources are shared, like opened file descriptors.
- Hierarchical organization: each process has a *father*.

Duplicate a process: fork

Syscall that duplicates the current process: `fork(2)`.

```
pid_t pid = fork();

if (pid == -1)
    puts("error");
else if (pid == 0)
    puts("child");
else
    puts("father");
return 0;
}
```

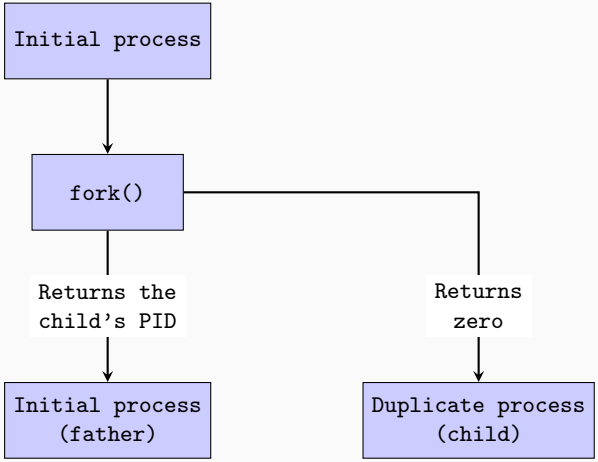


Figure 7: A fork

Wait for a process: waitpid

A process can watch for another:

- Wait for the end of execution.
- Get the return code.
- ...

Waitpid

```
pid_t waitpid(pid_t pid, int *status, int options);
```

```
man 2 waitpid
```

Zombies

- When a process terminates, its father must read its exit status (`waitpid(2)`).
- Until the father picks up the child's exit status, the terminated process remains a zombie.
- You must not leave zombies.

Final example

```c
pid_t pid = fork();

if (pid == -1) // error
{
    printf("An error occured\n");
    exit(1);
}
if (pid == 0) // child
{
    char *args[3] = {"echo", "foo", NULL};
    execvp(args[0], args); // foo should appear on stdout
    printf("An error occured\n");
    exit(1);
}
else // father
{
    int status = 0;
    waitpid(pid, &status, 0);
    exit(status);
}
```

Leaks

- Many resources are shared when a fork happens.
- Including files descriptors
- In our case, we do not want sharing fd related to files or directories manipulation.
- `man 2 open`, `O_CLOEXEC`
- It will be checked.

Conclusion

- Optional exercises before starting the project
- Core features and Additional features (Mandatory)
- We suggest you to follow the subject's order
- You must not have any leaks in your program (**1 leak = 0% to the test**)
- You must read the whole subject **before starting to code**

Newsgroup `assistants.projets`, `[FIND]` tag.
DeadLine November 12, 11:42 am

As usual:

- Your project must comply with the coding-style.
- Cheating will be penalized.
- You will not be helped if you don't have a Makefile or you didn't attempt debug.

Moreover:

- This is a long project, you must have a great architecture.
- You **must** do a great test suite.

- **Any questions?**