Présentation MyFind

ACU 2022 Team

Copyright

This document is for internal use only at EPITA <http://www.epita.fr>.

Copyright © 2021-2022 Assistants <assistants@tickets.assistants.epita.fr>.

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

MyFind

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

Usage

```
42sh$ ./myfind [options] [starting-point...] [expressions]
```

Examples

Simple find

```
42sh$ ./myfind
.
./exo2
./exo2/exo2
./exo2/exo2.c
./exo2/exo2.o
./exo1
./exo1/a.out
./exo1/exo1.h
./exo1/exo1-test.c
./exo1/exo1.c
./exo1/exo1.o
42sh$
```

Examples

Find by name

```
42sh$ ./myfind -name *.o
./exo2/exo2.o
./exo1/exo1.o

42sh$ ./myfind -name *.o -delete
42sh$ echo $?
0
42sh$ ./myfind -name *.o
42sh$
```

Find by name advanced

```
42sh$ ./myfind -name *.c -a ! -name *test.c
./exo2/exo2.c
./exo1/exo1.c
42sh$
```

Examples

Find by perm

```
42sh$ ./myfind -perm /1
.
./exo2
./exo2/exo2
./exo1
./exo1/a.out
42sh$ find -perm /1 -type f
./exo2/exo2
./exo1/a.out
42sh$
```

The project

Notions

To realize this project you will 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 are optional, but will help you to properly start the project.

Starting point

Starting points

- List of directories where the file searching starts
- If none, use the current directory to begin the search

Usage

```
42sh$ ./myfind [options] [starting-point...] [expressions]
```

The expression list

Expression list

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

Usage

```
42sh$ ./myfind [options] [starting-point...] [expressions]
```

Expression list

- Expressions are linked
- Each expression returns a value: `True` or `False`
- Do not evaluate the nth expression if the $n-1$th returns `False`

Example

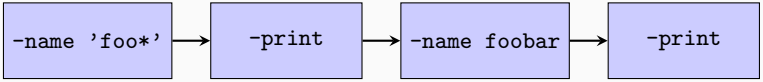

Figure 1: An expression list

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

Another example

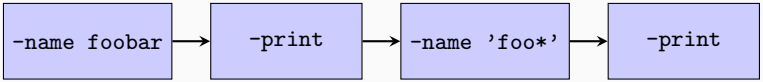

Figure 2: A similar but different expression list

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

The operators

- Modify the evaluation flow
  - AND: `-a`
  - OR: `-o`
  - NOT: `!`
  - PARENTHESES: `(` `)`

The operators - Evaluation flow alteration

- Beware the operators' priorities and their impact on the construction of your AST

Example - Without parentheses

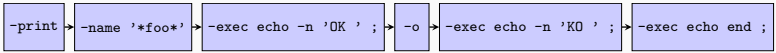

Figure 3: Without parentheses

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

Ast without parentheses

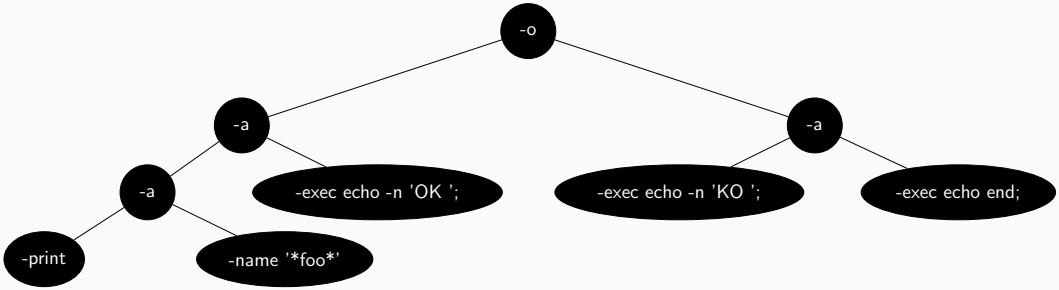

Figure 4: AST without parentheses

Example - Without parentheses

```
foobaz
OK foobaz/acu
KO end
```

Example - With parentheses

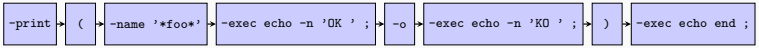

Figure 5: With parentheses

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

AST with parentheses

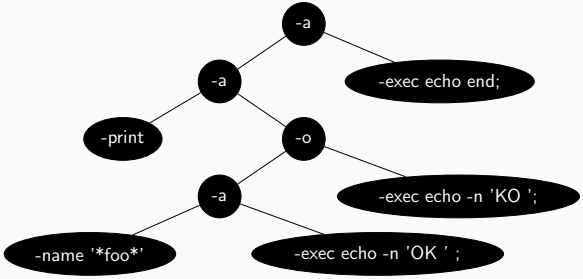

Figure 6: AST with parentheses

Example - With parentheses

```
foobaz
OK end
foobaz/acu
KO end
```

Options

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

Usage

```
42sh$ ./myfind [options] [starting-point...] [expressions]
```

File manipulation

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`)
  - ...

Useful notions

Helper struct

```c
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
    }
};

char *s = get_my_input(...);
for (int i = 0; i < 2; ++i)
    if (strcmp(s, funs[i].name) == 0)
        funs[i].fun();
```

Process Execution

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 *parent*

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("parent");
return 0;
```

Fork

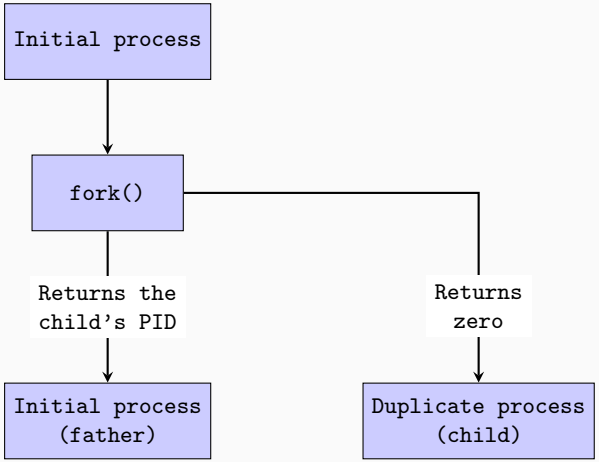

Figure 7: A fork

Execution: execve

`execve(2)`: This syscall replaces the current process by a new one with a new stack and heap.

exec

Many wrappers around `execve` exist.

`man 3 exec` for more information about the `exec()` family.

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;
}
```

Zombies

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

Wait for a process: waitpid

A process can watch for another:

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

```
man 2 waitpid
```

Waitpid

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

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 // parent
{
    int status = 0;
    waitpid(pid, &status, 0);
    exit(status);
}
```

Conclusion

Project

- 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**

Subject

- You **must** read the whole subject before starting to code!

Recap

Newsgroup `assistants.projets`

Tag `[FIND]`

Deadline November 6, 21:42

As usual:

- Your project must comply with the coding-style
- Cheating will be penalized
- You will not get help from the assistants if you do not have a `Makefile` or if you did not attempt debug

Moreover:

- This is a long project, you must have a great architecture
- You **must** do a great testsuite