NTNU | Norwegian University of Science and Technology

Operating Systems

More Q&A for PE4 – 18.03.2021

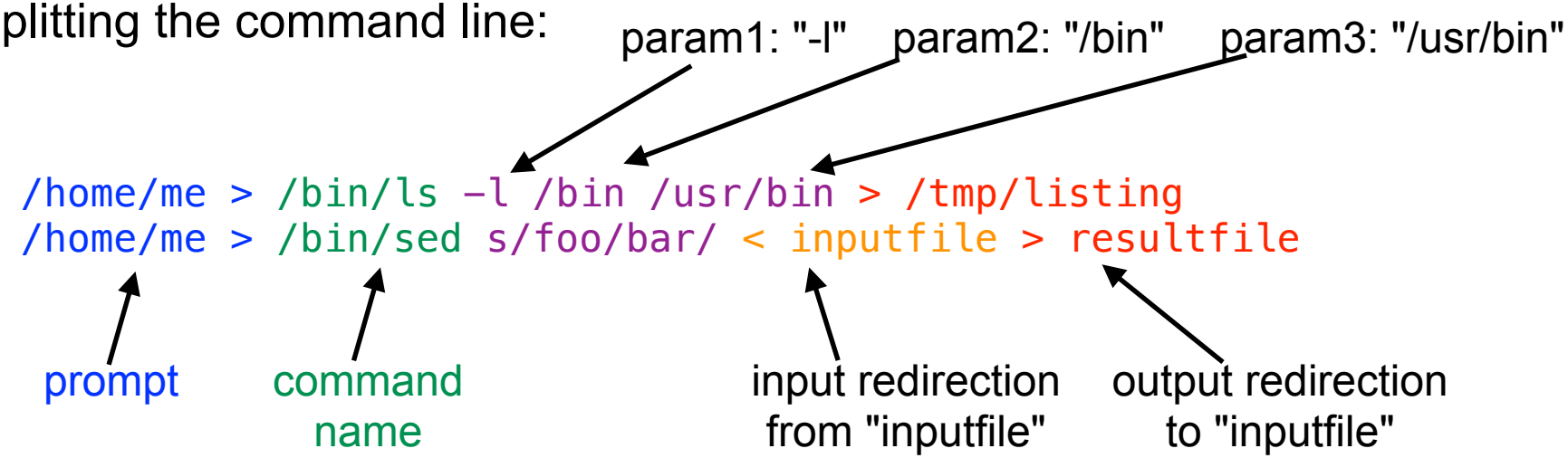
Michael Engel

PE4a Terminal I/O handling and input scanning

- A standard Unix shell…
  - reads commands from stdin (fd 0)
  - writes regular output (e.g. the prompt) to stdout (fd 1)
  - writes error messages to stderr (fd 2)
- Reading commands
  - Using `gets(3)` is not a good idea: buffer overflows!
  - `scanf(3)` might also be difficult:
    - "Scanning stops when an input character does not match such a format character"
  - `getline(3)` even can allocate memory for you!

PE4b Unix shell parsing

Splitting the command line:



- Today, this would be called a REPL – "Read-Evaluate-Print-Loop"
- Prompt: what the shell prints
- Command name: command to execute (internal or external)
- Optional: zero or more parameters
- Optional: input and output redirect (in arbitrary order)

PE4b Interpreting the command line

- What about the order on the command line?
- Would these three lines be identical?

```
/home/me > /bin/ls -l /bin /usr/bin > /tmp/listing
/home/me > /bin/ls > /tmp/listing -l /bin /usr/bin
/home/me > /bin/ls -l /bin > /tmp/listing /usr/bin
```

- on a "real" Unix shell: yes! (but most people don't expect it)
- Your shell ***does not*** have to support this
  - I/O redirection at the end of the line is perfectly fine
- However, these two lines should both work in your shell:

```
/home/me > /bin/sed s/foo/bar/ < inputfile > resultfile
/home/me > /bin/sed s/foo/bar/ > resultfile < inputfile
```

PE4a Terminal I/O handling and input scanning

Parsing by hand is lots of work and error-prone…

- Alternative: one of the strtok(3) libc functions
- From the strtok manpage on strtok_r(3):

```
char *
strtok_r(char *restrict str, const char *restrict sep, char **restrict lasts);

char line[80];
char *sep = "\\/:;=-";
char *word, *phrase, *brkt, *brkb;

strcpy(test, "This;is.a:test:of=the/string\\tokenizer-function.");

for (word = strtok_r(test, sep, &brkt); // strtok_r has an internal state machine
     word;
     word = strtok_r(NULL, sep, &brkt)) // it stores current pos in string in brkt
{
    printf("So far we're at %s:%s\n", word); // word contains ptr to current part
}
```

PE4b Unix shell parsing

Parsing by hand is lots of work and error-prone…

- Alternative: strsep(3)

```
char *
strsep(char **stringp, const char *delim);

First example:
char *token, *string, *tofree;

tofree = string = strdup("abc,def,ghi");
assert(string != NULL);

while ((token = strsep(&string, ",")) != NULL)
    printf("%s\n", token);

free(tofree);

Second example:
char **ap, *argv[10], *inputstring;

for (ap = argv; (*ap = strsep(&inputstring, " \t")) != NULL;)
    if (**ap != '\0')
        if (++ap >= &argv[10])
            break;
```

PE4b Unix shell exec calls

There are several different exec functions in libc:

```
int
execl(const char *path, const char *arg0, ... /*, (char *)0 */);

int
execle(const char *path, const char *arg0, ... /*, (char *)0, char *const envp[] */);

int
execlp(const char *file, const char *arg0, ... /*, (char *)0 */);

int
execv(const char *path, char *const argv[]);

int
execvp(const char *file, char *const argv[]);

int
execvP(const char *file, const char *search_path, char *const argv[]);
```

- Depending on your representation of the parameters you parse, some might be more appropriate than others… **execv** works for many of you!

PE4c Implement I/O redirection

- A Unix program created by `fork(2)` inherits **all file descriptors** of its parents
  - Especially stdin (0), stdout (1) and stderr (2)
- General way to do I/O redirection:
  - In the shell, use `pid = fork();` to create a child process
- In the parent process (pid returned = pid of child), just wait for termination of the child process
- In the child process (pid returned = 0)
  - If input redirection indicated:
    - open file for read & redirect input file descriptor stdin
  - If output redirection indicated:
    - open file for write (create if required) & redirect input file descriptor stdout
  - Then use exec to call the program

PE4c Unix shell I/O redirection

Redirecting I/O in Unix works uses the dup(2) or dup2(2) syscall:

```
int
dup(int fildes);

int
dup2(int fildes, int fildes2);
```

- dup copies the file descriptor passed as parameter to the first ***unused*** file descriptor

- to redirect I/O:
  - open the file you want to redirect to/from ➛ file descriptor, e.g. refd
  - then either close the fd you want to redirect (e.g. stdout = 1) and
    - and call dup with refd as parameter
  - or call dup2 with the fd you want to redirect and refd as parameters

PE4d Internal shell commands

- Why are cd and exit implemented as internal commands?
- Unix processes have the concept of a ***current directory***
- File/path names can be *relative* (to the current dir) or *absolute*
  - *absolute names* start with a "/"
    - so they always start at the root of the file system tree
  - *relative names* start with any other character
    - can include partial path, e.g. `sub/dir/file.c` refers to `/home/me/sub/dir/file.c` if current dir is `/home/me`
  - For looking up *commands*, this is not (generally) true
    - Instead, the shell searches executable files in a set of directories in an *environment variable* $PATH
    - If "." (current dir) is in $PATH: possible ***security problem***

PE4d Internal shell commands

- For looking up *commands*, this is not (generally) true
  - Instead, the shell searches executable files in a set of directories in an *environment variable* $PATH
  - If "." (current dir) is in $PATH: possible ***security problem***
- ***Why is this a security problem?***
- Imagine a $PATH such as `.:/bin:/usr/bin`
- Now if you type `ls`…
  - The shell first searches in the current directory
  - What if you typed `cd /tmp` and then `ls`?
  - … and some other user left an executable program in /tmp that deletes your home directory?

PE4d Internal shell commands

- Why are cd and exit implemented as internal commands?
- Think about exit implemented as an external command:

```
#include <stdlib.h>
int main(void) {
    exit(0);
}
```

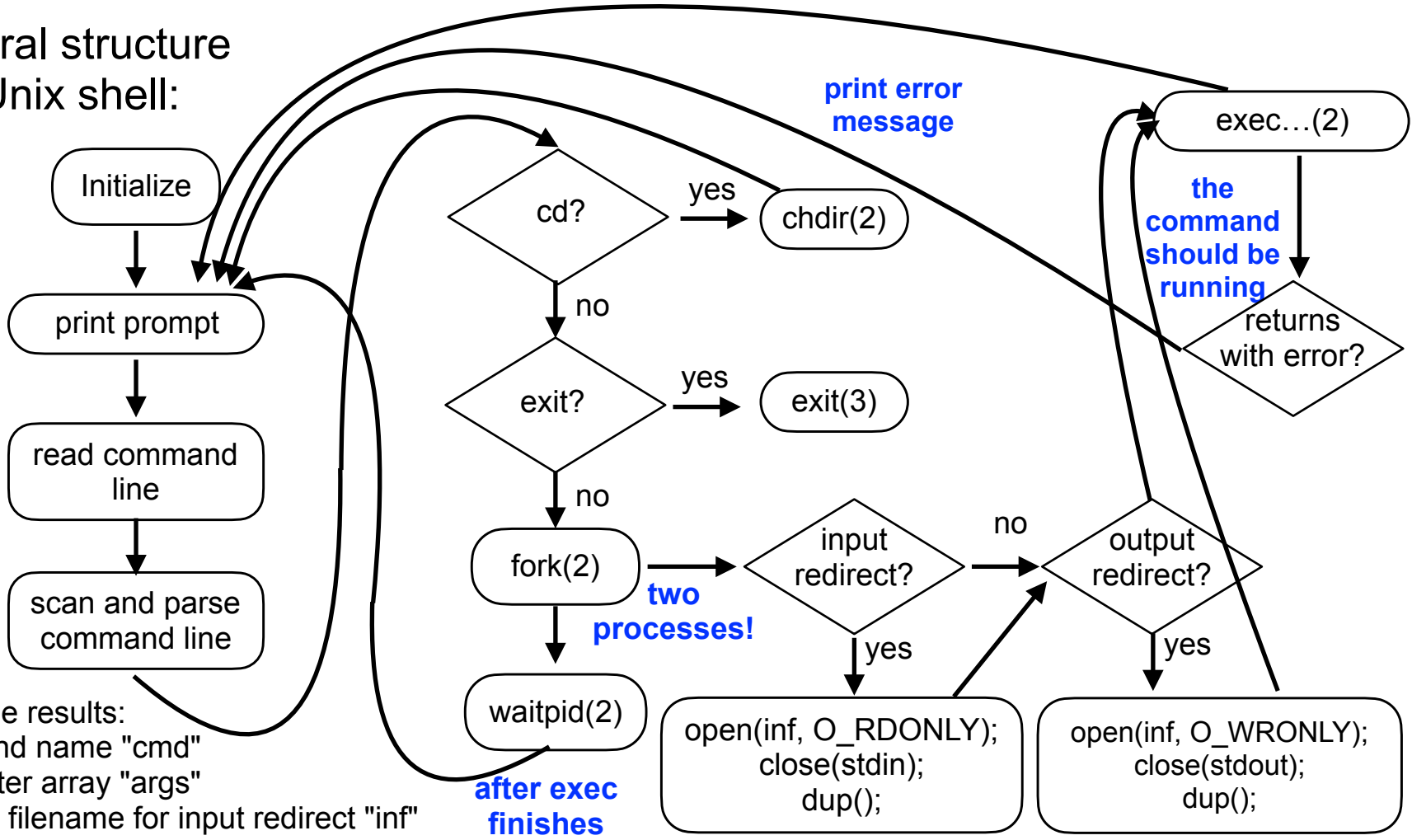
- What would this do?

PE4e Simple shell scripting

- There has been a lot of confusion about shell scripts
- A shell script is just a text file with shell commands
  - Usually one command per line
- "Real" Unix shells implement control structures
  - if/while etc.
- Your shell only has to implement sequences of commands
  - …just as if you typed them one after the other on the command line by hand
- Call it like this:
  `./wish shellscript.sh`
- Implementation is ***simple!***
  Think about what happens if you call your shell like this:
  `./wish < shellscript.sh`

PE4 Overall Unix shell structure

General structure of a Unix shell:



scan/parse results:

- command name "cmd"
- parameter array "args"
- optional filename for input redirect "inf"
- optional filename for output redirect "outf"