P2: Exec and low-level I/O

Ion Mandoiu
Laurent Michel
Revised by M. Khan and J. Shi

Pitfall

- Anything might go wrong?

```
pid_t pid = fork();
if (pid < 0) {
    perror("fork()"); exit(1);  // exit if fork() fails
} else if (pid == 0) {
    child_tasks();
} else {
    parent_tasks();
}
more_parent_tasks();
```

Pitfall

- Anything might go wrong?

```
pid_t pid = fork();
if (pid < 0) {
    perror("fork()"); exit(1);  // exit if fork() fails
} else if (pid == 0) {
    child_tasks();
    exit(0);                    // terminate the child process
} else {
    parent_tasks();
}
more_parent_tasks();
```

Process upgrades

- Usually….
  - A fresh clone wants to run *different code*
- This is done by
  - Loading another executable into the process address space
    - [picked up from the file system of course]

- Note
  - **Opened files are NOT AFFECTED by the upgrade operation**

The `exec` family

- The act of ‘upgrading’ is done by the child with a system call
  - Many variants. "man -S3 execl" for all details

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ...
          /*, (char *) NULL */ );
```

  - The path to the executable to load inside our own address space
  - A list of arguments to be passed to the new executable
  - A final NULL pointer to give the “end of argument list”
  - If successful, `execl()` does not return! Started a new process

Exec example

- We will turn the child process into the following exectuable

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

int main(int argc,char* argv[]) {
   int i, sum=0;
   for(i=1;i<argc;i++)
      sum += atoi(argv[i]);
   printf("sum is: %d\n",sum);
   return 0;
}
```

This is a simple "adder" program that computes the sum of its integer arguments

Parent Program

```
int main() {  // complete code is in demo repo
  char *cmd1 = "./adder", *cmd2 = "expr";
  pid_t child = fork();
  if (child == 0) {
    printf("In child!\n");
    execl(cmd1,cmd1,"1","2","3","10",NULL);
    printf("Oops.... something went really wrong!\n");
    perror(cmd1);
    return -1;
  } else {
    printf("In parent!\n");
    execl(cmd2,cmd2,"100","+","300",NULL);
    printf("Oops.... something went really wrong!\n");
    perror(cmd2);
    return -1;
  }
}
```

How is executable found?

- Specify a path, like `/bin/ls`
- Specify a file, and the system searches in directories listed in PATH
  - echo $PATH in bash to see directories separated by ':'

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

// execlp() searches paths for file
int execlp(const char *file, const char *arg0, ...
          /*, (char *) NULL */ );
```

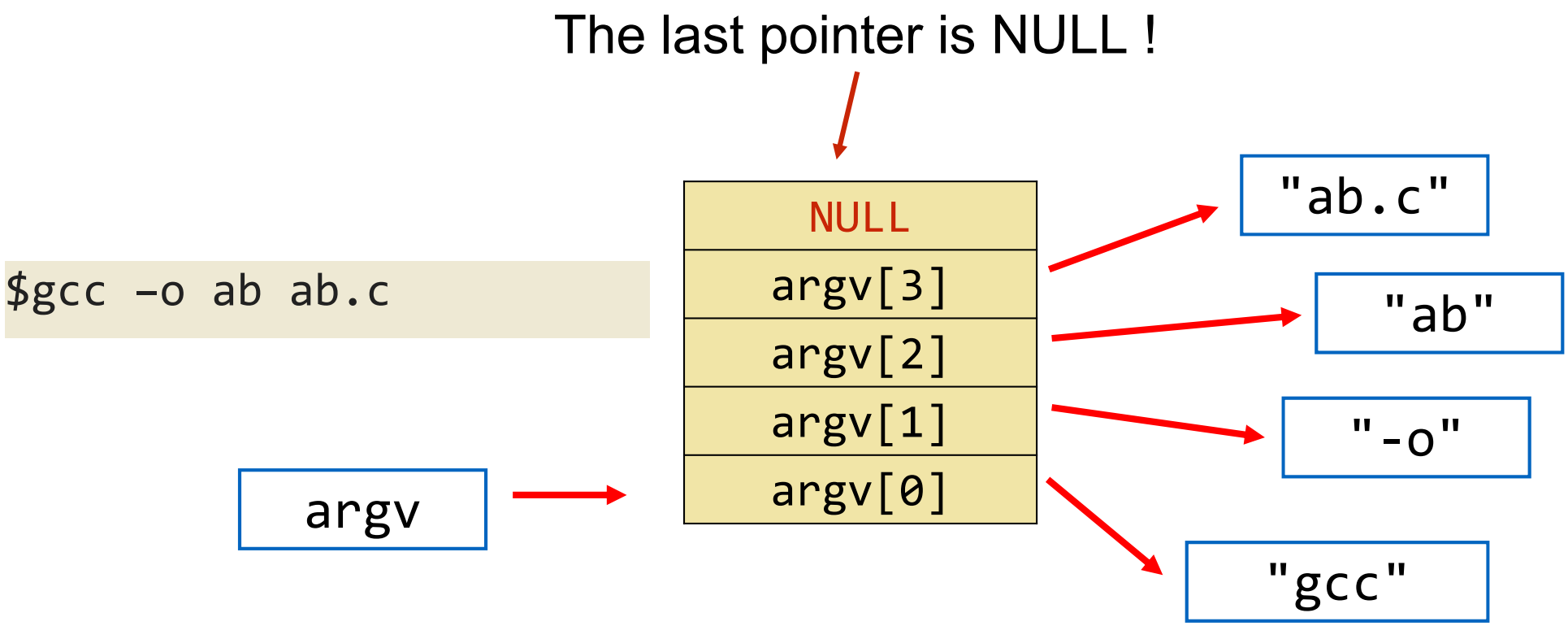
`execv` family

```
// If the number of arguments is unknown at compile time
#include <unistd.h>
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- The arguments in execl in are placed in an array
  - argv is the argv you see in the main function!
- execv needs a path while execvp can search file in PATH
- Start a new process if successful. Similar to execl

argv to execv and execvp

- Note the NULL pointer at the end
- Why?



10

Question

- What might go wrong?

```
pid_t pid = fork();
if (pid < 0) {
    perror("fork()"); exit(1);  // exit if fork() fails
}
else if (pid == 0) {
    // in child process
    execlp("genie", "genie", "clean the house", NULL);
}
// in parent process
online_shopping();
```

File APIs

- Remember the (C standard library) IO APIs
  - The "f" family (fopen,fclose, fread, fgetc, fscanf, fprintf,…)
  - All these use a FILE* abstraction to represent a file
    - Additional features: user-space buffering, line-ending translation, formatted I/O, etc.
- UNIX has lower-level APIs for file handling
  - Directly mapped to **system calls**
    - open, close, read, …
  - Use *file descriptors* **[which are just integers]**
  - Deal with bytes only

Some low level file APIs

- Read the man pages (man –s2 …) for more functions

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

int open(const char *path, int oflag);
int open(const char *path, int oflag, int mode);
int close(int fd);

ssize_t read(int fd, void *buf, size_t nbyte);
ssize_t write(int fd, const void *buf, size_t nbyte);

off_t lseek(int fd, off_t offset, int whence);
```

Open a file

```
#include <fcntl.h>
#include <unistd.h>

int open(const char *path, int oflag);
```

- Parameters
  - path: the path to the file to be opened/created
  - oflag: read, write, or read and write, and more (on the next slide)

- The function returns a file descriptor, a small, nonnegative integer
  - Return -1 on error

Flags in open()

- Must include one of the following:
  O_RDONLY (read only), O_WRONLY (write only), or O_RDWR (read and write)
- And or-ed (|) with many optional flags, for example,
  - O_TRUNC: Truncate the file (remove existing contents) if opening a file for write
  - O_CREAT: Create a file if it does not exist.

Example:

```
// remember open() returns -1 on error
fd1 = open("a.txt", O_RDONLY); // open for read
fd1 = open("a.txt", O_RDWR); // open for read and write
fd1 = open("a.txt", O_RDWR|O_TRUNC); // read, write, truncate the file
```

Create a file with open()

```
// a mode must be provided if O_CREAT or O_TMPFILE is set

int open(const char *path, int oflag, int mode);
```

mode: specify permissions when a new, or temporary, file is created.

leading 0 is important
Octal numbers!

```
open("b.txt", O_WRONLY|O_TRUNC|O_CREAT, 0600);
```

// open b.txt for write. If the file exists, clear (truncate) the contents.

// if the file does not exist, create one, and set the permission so that the owner of the file can read and write, but other people cannot.

File descriptor vs stream

```
#include <stdio.h>
int fileno(FILE *stream);
// returns a file descriptor for a stream
```

FD	FILE *
0	stdin
1	stdout
2	stderr

File descriptors after fork and exec

- Opened files are NOT AFFECTED by the upgrade operation

```
pid_t pid = fork();
assert(pid >= 0);
if (pid == 0) {
    // Child process can access FDs 0, 1, and 2
    // if execl() is successful, gcc can access FDs 0, 1, and 2
    execlp("gcc", "gcc", "a.c", NULL);
    // If control gets here, execlp() failed.
    // Remember to terminate the child process!
    return 1;
}
```