

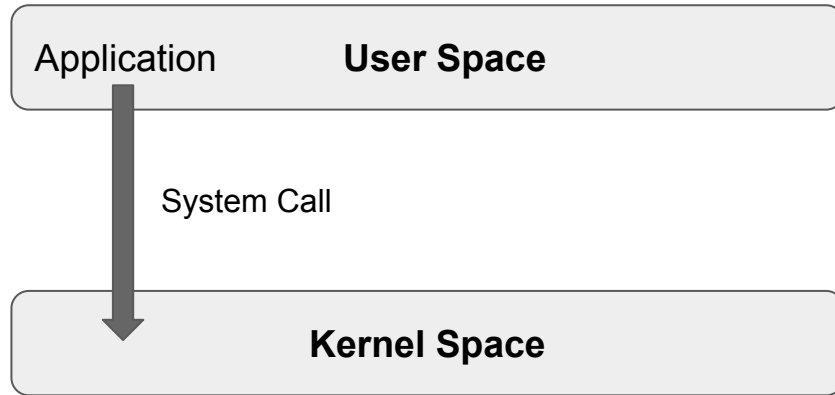
C shell

(Φροντιστήριο για την 1η σειρά)

Dimitris Deyannis
deyannis@csd.uoc.gr

System Calls

- If a **process** is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a **trap instruction** to transfer control the **operating system**



System Calls

A system call is a request for service that a program makes of the kernel. The service is generally something that only the kernel has the privilege to do, such as doing I/O

System Calls

Process Control	fork(), wait(), exec(), exit(), ...
File Manipulation	open(), close(), read(), write(), ...
Directory Management	mkdir(), rmdir(), mount(), link(), ...
Other	chdir(), chmod(), kill(), time(), ...

fork()

- Fork creates a new process (**child process**)
 - It creates an exact duplicate of the original process, including all the file descriptors, registers etc.
- The fork is called once, but returns twice!
 - After the fork, the original process and the copy (the parent and the child) go at separate ways
 - The fork call returns a value, which is zero in the child and equal to the child's process identifier (**PID**) in the parent.
- Now consider how fork is used by the shell. When a command is typed, the shell forks off a new process. This child process must execute the user command

fork() – PID (Process IDentity)

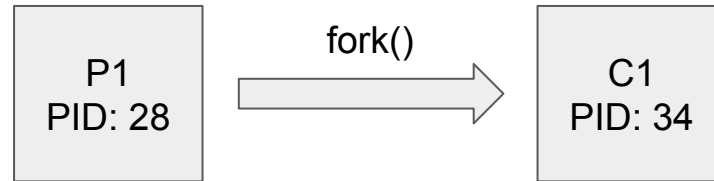
- **pid < 0** the creation of a child process was unsuccessful
- **pid == 0** the newly created child
- **pid > 0** the process ID of the child process passes to the parent

```
#include <unistd.h>
pid_t pid = fork();
printf("PID:%d\n",pid);
```

...

The parent will print:
PID:34

The child will always print:
PID:0



fork()

```
#define TRUE 1
while (TRUE) {
    type_prompt();
    read_command(command, parameters);
    if (fork() != 0) {
        /* Parent code */
        waitpid(-1, &status, 0);
    } else {
        /* Child code */
        execve(command, parameters, 0);
    }
}
```

/* repeat forever */
/* display prompt on the screen */
/* read input from terminal */
/* fork off child process */
/* wait for child to exit */
/* execute command */

exec (binary path)

- The exec() call **replaces/overwrites** a current process image with a new one (i.e. loads a new program within the current process)
- The file descriptor table remains the same as the original process
- Argument passed via exec() appear in the argv[] of the main function
- Upon success, exec() **never** returns to the caller
 - It replaces the current process image, so it cannot return anything to the program that made the call
 - If it does return, it means the call failed

exec("/bin/ls"): overwrites the memory code image with the binary from /bin/ls and executes

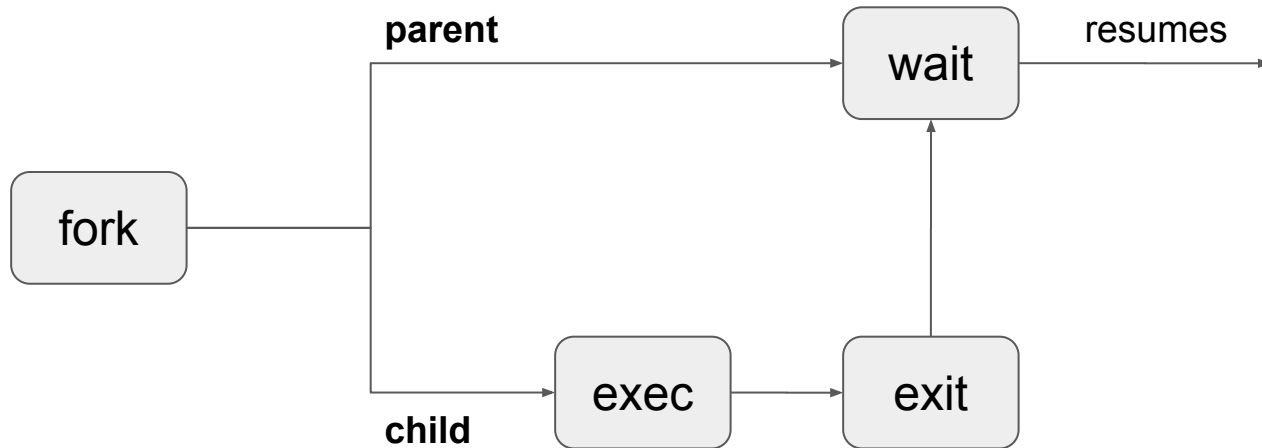
exec (binary path)

- There's not a single syscall under the same `exec()`
- By **exec()** we usually refer to a family of calls:
 - `int execl(char *path, char *arg, ...);`
 - `int execv(char *path, char *argv[]);`
 - `int execlp(char *path, char *arg, ..., char *envp[]);`
 - `int execve(char *path, char *argv[], char *envp[]);`
 - `int execlp(char *file, char *arg, ...);`
 - `int execvp(char *file, char *argv[]);`

l	argument list
v	argument vector
e	environmental vector
p	search path

fork and exec

- Often after calling `fork()` we want to load a new program into the child. E.g.: a *shell*



wait()

- Forces the parent to **suspend** execution, i.e. wait for its children or a specific child to die (terminate)
- When the child process dies, it returns an exit status to the operating system, which is then returned to the waiting parent process. The parent process then resumes execution
- A child process that dies but is never waited on by its parent becomes a **zombie process**. Such a process continues to exist as an entry in the system process table even though it is no longer an actively executing program

exit()

- This call terminates process execution gracefully. Gracefully means it does clean up and release of resources, and puts the process into the **zombie state**
- When the child process dies, an exit status is returned to the OS and a signal is sent to the parent process
- The exit status can then be retrieved by the parent process via the wait system call

fork, exec and wait

```
while (1) {
    type_prompt();
    read_command(command, parameters);
    if (fork() != 0) {
        /* Parent code */
        waitpid(-1, &status, 0);
    } else {
        /* Child code */
        execve(command, parameters, 0);
    }
}
```

/* repeat forever */
/* display prompt on the screen */
/* read input from terminal */
/* fork off child process */
/* wait for child to exit */
/* execute command */

Process state

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity

- **Orphan** is a process whose parent process has finished or terminated, though it remains running itself
- **Daemon** runs as a background process rather than being under the direct control of an interactive user
- **Zombie** is a process that has completed execution but still has an entry in the process table

Pipelines

- Pipelines (pipes) provide a unidirectional interprocess communication channel
- “|” (pipe) operator between two commands directs the stdout of the first to the stdin of the second. Any of the commands may have options or arguments
- Examples:
 - `command_1 | command_2 parameter_1 | command_3 | command_4`
 - `ls -l | grep key | more`
 - `ls -al | grep txt | wc -l`

```

void main(int argc, char *argv[]) {
    int pipefd[2];
    pid_t cpid;
    char buf;
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (cpid == 0) {
        /* Child reads from pipe */
        close(pipefd[1]); /* Close unused write end */
        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);
        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        exit(EXIT_SUCCESS);
    } else {
        /* Parent writes argv[1] to pipe */
        close(pipefd[0]); /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]); /* Reader will see EOF */
        wait(NULL); /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}

```

Redirection

- Use `dup2()`
 - `dup2(source_fd, destination_fd)`
- Standard Input “<”
 - `sort < file_list.txt`
- Standard Output “>”, “>>”
 - `ls > file_list.txt`
 - `ls >> file_list.txt` (append)
- Use `fopen()`
 - “r” for input “<”
 - “w+” for output “>”
 - “a” for append output

Assignment 1

- Implement a C shell (command interpreter) that reads and executes user commands
- Shell prompt: [cs345sh][<user>][<dir>]
- Simple command examples:
 - cd
 - exit
 -
- Complex command examples:
 - ls -al
 - cat file.txt
 - sort -r -o log.txt input.txt
 -

Assignment 1

- For this shell we replace the pipe operator with “>” instead of “|”
- Pipe examples
 - `ls -al > wc -l`
 - `ls -al > sort -r -k 6 > head 5`
 -
- For this shell we replace “<” with “|”, “>” with “||” and “>>” with “|||”
- Redirection examples
 - `cat | data.txt`
 - `ls -al || log.txt` */* overwrite */*
 - `ls -al ||| log.txt` */* append */*

Assignment 1

- Shortcuts
 - Shortcuts are key combinations, usually involving CTRL or ALT and a letter of the alphabet and trigger a specific shell operation
- CTRL-C
 - Terminates (kills) the current process
- CTRL-H
 - Deletes the character before the cursor (similar to Backspace)
- CTRL-S
 - Pauses the output
- CTRL-Q
 - Resumes the output

Useful links

- <https://linux.die.net/man/3/exec>
- <https://linux.die.net/man/2/fork>
- <https://linux.die.net/man/2/wait>
- <https://linux.die.net/man/2/pipe>
- <https://linux.die.net/man/2/dup2>
- https://www.tutorialspoint.com/c_standard_library/c_function_fopen.htm
- <http://man7.org/linux/man-pages/man2/pipe.2.html>
- <http://man7.org/linux/man-pages/man3/termios.3.html>
- <http://web.eecs.utk.edu/~huangj/cs360/360/notes/Fork/lecture.html>
- <https://kb.iu.edu/d/acar>