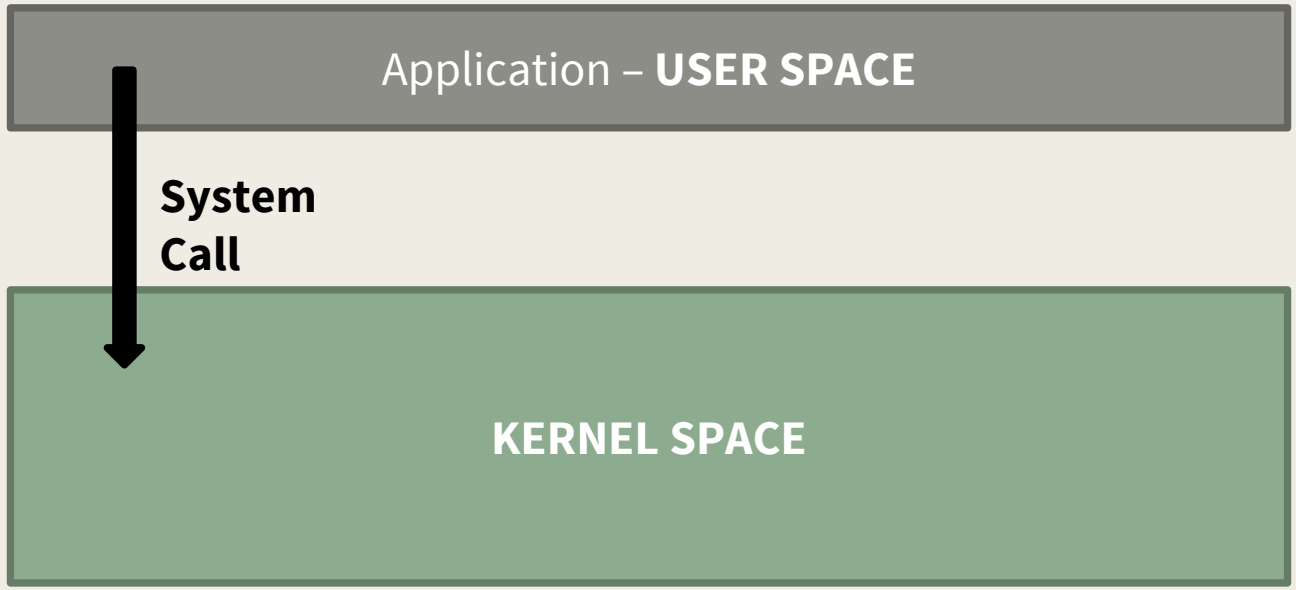
Assignment 1

Υπεύθυνοι βοηθοί:
Ειρήνη Δέγκλερη (degleri@csd.uoc.gr)
Ηλίας Παπαδόπουλος (ppapadop@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 to 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()
Directories Managment	mkdir(), rmdir(), mount(), link()
Other	chdir(), chmod(), kill(), time()

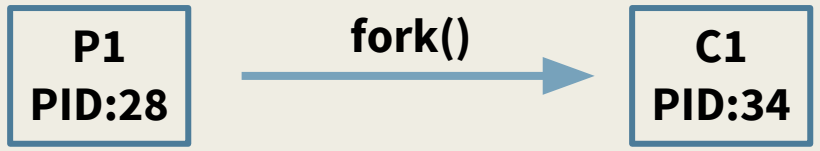
Fork()

- Fork creates a new process (**child process**).
  - It creates an exact duplicate of the original process, including all the file descriptors, registers—everything.
- **The fork is called once, but returns twice!**
  - *After the fork, the original process and the copy (the parent and child) go their separate ways.*
  - The fork call returns a value, which is zero in the child and equal to the child's process identifier or 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.



Consider a piece of program:

```
#include <unistd.h>

pid_t pid = fork();
printf("PID: %d\n", pid);
....
```

The parent will print:
PID: 34
And the child will always print:
PID: 0

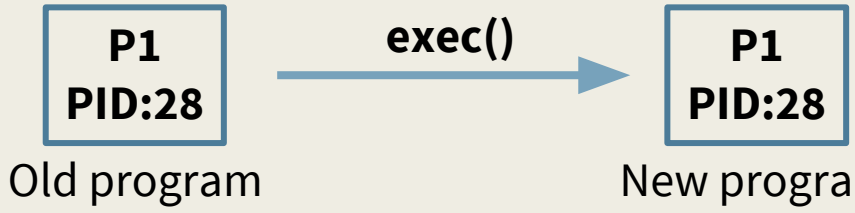
Fork()

```
#define TRUE 1
while (TRUE) {                          /* repeat forever */
        type_prompt();                          /* display prompt on the screen */
        read_command(command, parameters);      /* read input from terminal */
        if (fork() != 0) {                      /* fork off child process */
                /* Parent code. */
                waitpid(-1, &status, 0);        /* wait for child to exit */
        } else {
                /* Child code. */
                execve(command, parameters, 0);   /* 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 current process).*
- The file descriptor table remains the same as that of original process.
- Arguments 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 binary from /bin/ls and execute.



Exec (binary_path)

- There ’s not a syscall under the name exec().
- By **exec()** we usually refer to a family of calls:
  - *int execl(char *path, char *arg, ...);*
  - *int execv(char *path, char *argv[]);*
  - *int execle(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[]);*

Where: ***l*** *= argument list*

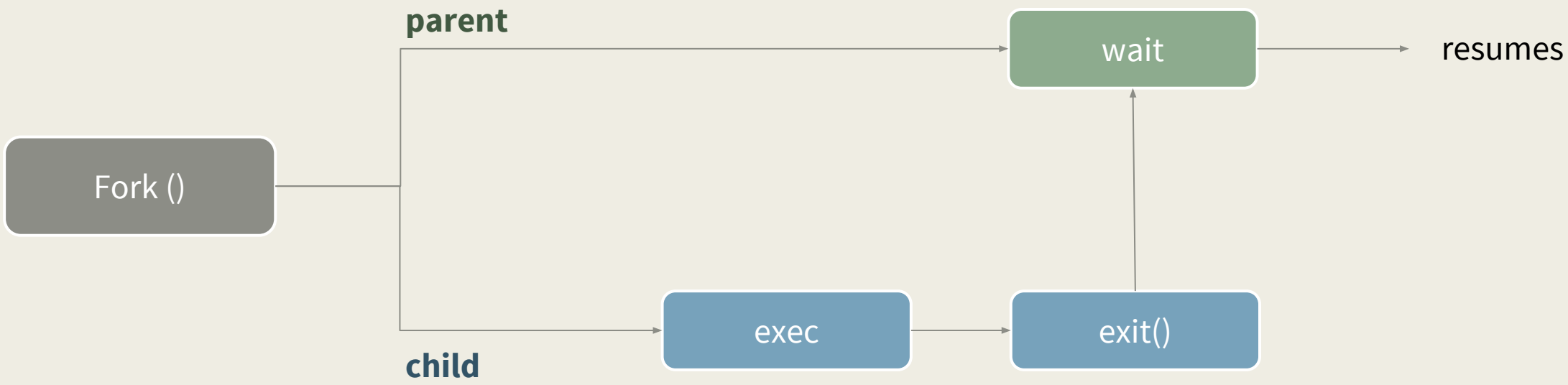
v *= argument vector*

e *= environmental vector*

p *= search path*

Fork and exec

- Often after doing 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 **gracefully** terminates process execution.
  Gracefully means it does clean up and release of resources, and puts the process into the **zombie state**.

- By calling *wait()*, the parent cleans up all its zombie children.

- When the child process dies, an exit status is returned to the operating system 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) {                                 /* repeat forever */
        type_prompt();                      /* display prompt on the screen */
        read_command(command, parameters);  /* read input from terminal */
        if (fork() != 0) {                  /* fork off child process */
                /* Parent code. */
                waitpid(-1, &status, 0);    /* wait for child to exit */
        } else {
                /* Child code. */
                execve(command, parameters, 0); /* execute command */
        }
}
```

States of a process

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

Pipes

- 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.

- e.g of pipelines:
  - command1 | command2 paramater1
  - ls -l | grep key

```
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 “<“**
  - e.g. *sort < file_list.txt*
- **Standard Output “>“, “>>”**
  - e.g. *ls > file_list.txt*
  - e.g. *ls >> file_list.txt* (*append*)
- Use **fopen()**
  - “**r**” for input “<“
  - “**w+**” for output “>”
  - “**a**” for append output “>>”

```
FILE *fp;
fp = fopen ("file.txt", "w+“);
```

Assignment 1

A C shell (command interpreter) that reads user commands and executes them.

- getlogin() (if dosen't work try: *struct passwd *pw = getpwuid(getuid());*
  printf("Username: %s\n", pw->pw_name);)

- Simple commands such as:
  - cd (see chdir())
  - set var="ls", unset var and printlvars
  - exit
  - *Also,*
    - ls, ls –l, ls -a -l, cat file.txt, sort -r -o output.txt file_to_sort.txt, ...

Assignment 1

A C shell (command interpreter) that reads user commands and executes them.

- Complex commands such as:
  - *Redirection of input and output (see dup2())*
    - ls –l > output
      cat < input
      cat < input > output
  - *Pipes (see pipe())*
    - ps axl | grep zombie
    - ps axl | grep zombie > output
    - ls | grep “.c”

Assignment 1

1. Print prompt
2. Read command
   a. Parse command // look for “-, | , >, >>, <, &”

   if command == exit // terminate shell

   else if command == set or unset //insert to/delete from local variable table

   if command == unset // check if variable was previously set

   else if command = printlvars // print local var table

   else if command == cd // use chdir()

   2.2 fork

if command has “|” // use pipe()
if command has “>, >> , < ” // use dup2()
exec(...)

if command has “&” // work in background
else // wait

go back to Step 1

Useful links

- **Shell:** http://linuxcommand.org/learning_the_shell.php
- **fork**(): https://linux.die.net/man/2/fork
- **exec**(): https://linux.die.net/man/3/exec
- **wait**(): https://linux.die.net/man/2/wait
- **pipe**(): https://linux.die.net/man/2/pipe
- **dup2**(): https://linux.die.net/man/2/dup2
- **fopen**(): https://www.tutorialspoint.com/c_standard_library/c_function_fopen.htm
- **set, unset**: http://sc.tamu.edu/help/general/unix/vars.html

Reading material

- Κλήσεις συστήματος (Κεφ. 1.6)
- Διεργασίες (Κεφ. 2.1)