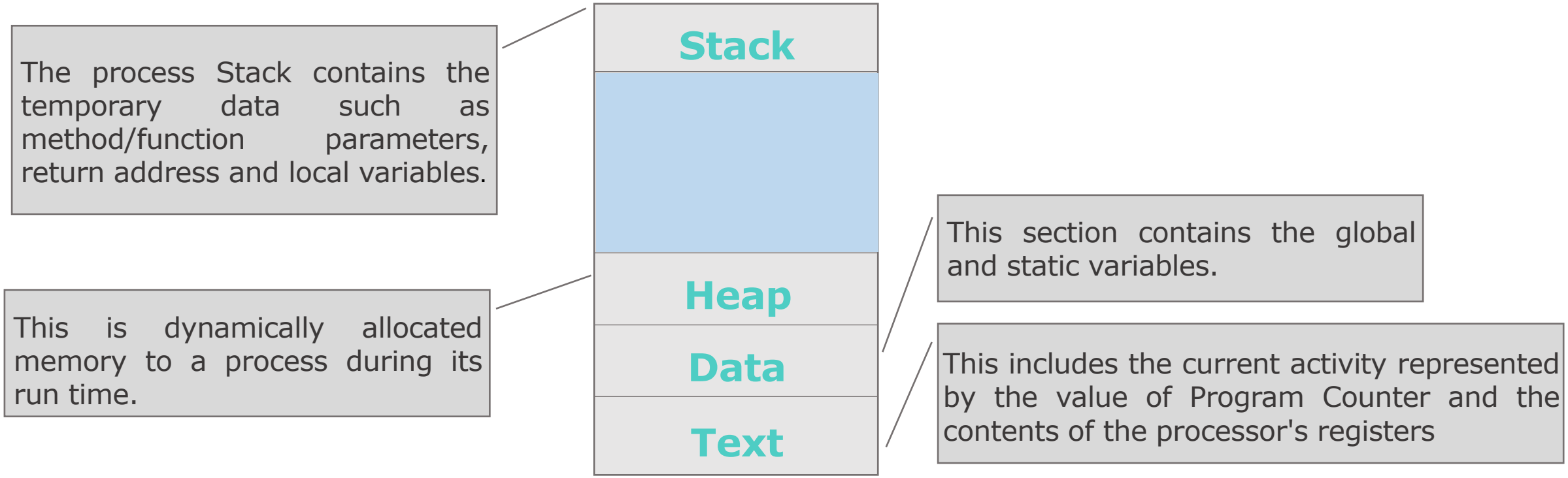
POLITECNICO
DI TORINO

Processes

Operating Systems – Sarah Azimi

Process

- A process is a program in execution.
- A process can be divided into four sections.



Process Creation

- A process may create several new processes, via a create-process system call, during the course of execution:
  - The creating process is called a **parent** process
  - The new processes are called the **children** processes of that parent.

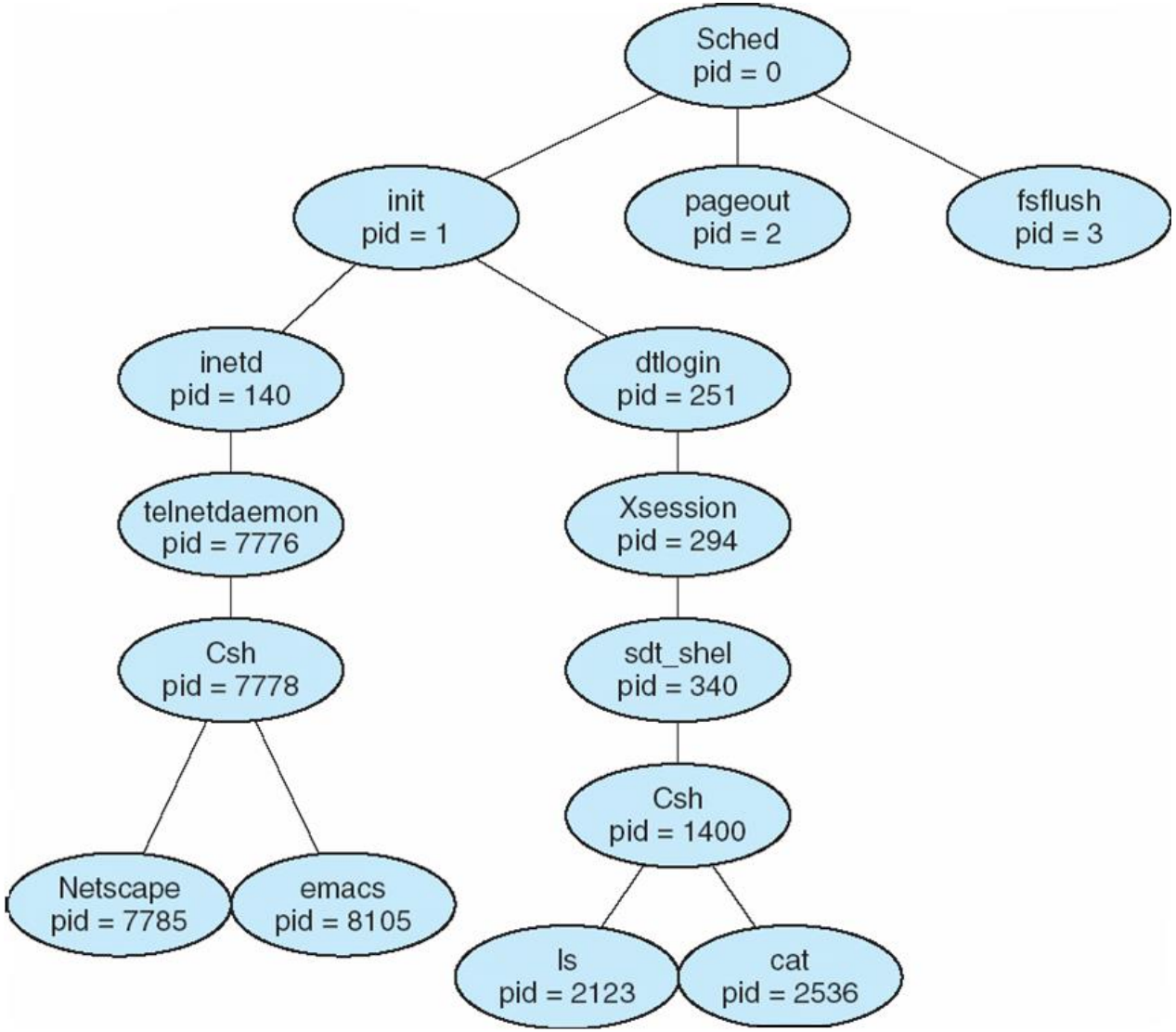
- Parent process create children processes, which, in turn create other processes, forming a **tree** of processes

Process Identifier

- Generally, process is identified and managed via a unique **process identifier (pid),** which is an integer number (a variable of type pid_t).

Process Tree

- There are some special processes:
  - Process ID 0 is usually the scheduler process and is often known as the swapper: it is part of the kernel and is known as a system process.
  - **Process ID 1** is usually the ***init process*** and is invoked by the kernel at the end of the bootstrap procedure.



Process Creation

- Process creation:
  - ***fork*** system call creates new process, called child process
- Process modification:
  - ***exec*** system call used after a fork to replace the process' memory space with a new program.

Process Creation using fork()

- Fork() system call is used for creating a new process, which is called child process.
  - The child process runs concurrently with the process that makes the fork() call.
  - Both new child process and parent will execute the next instructions following the fork().
- Syntax:
  - `int fork()`

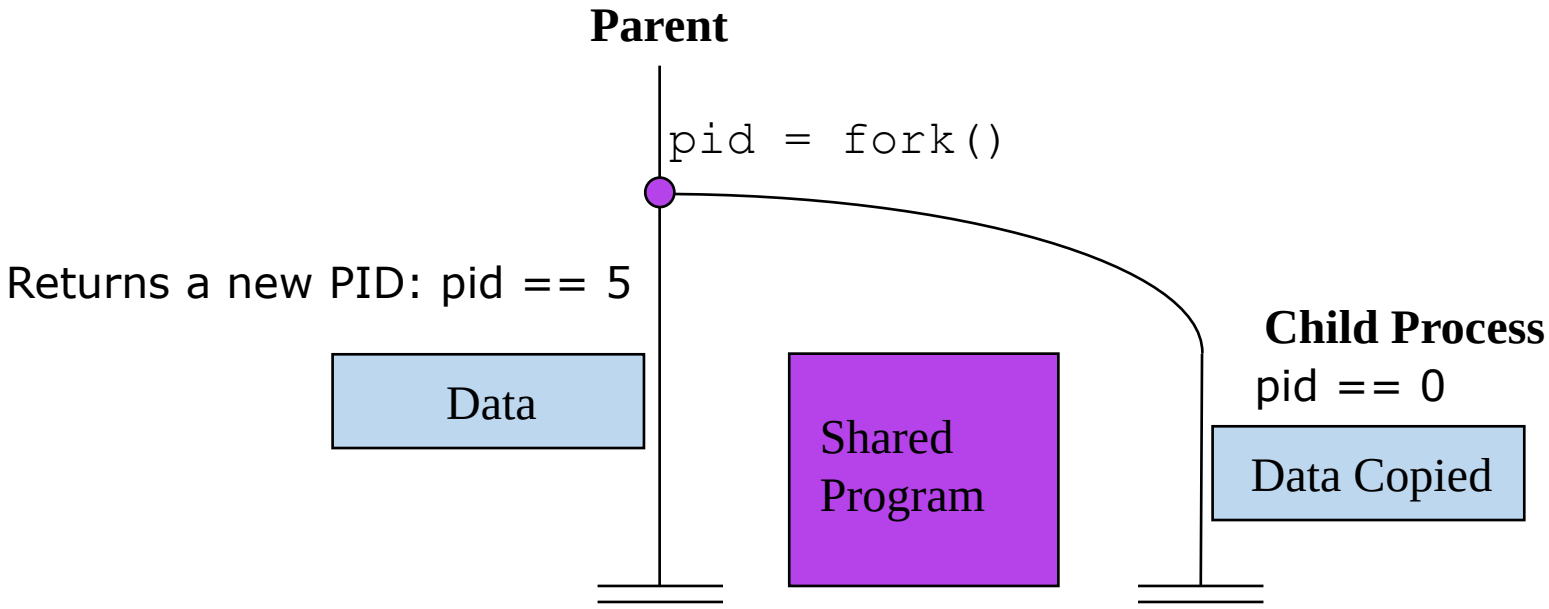
Your program should include two header files:

<unistd.h> and **<sys/types.h>**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
        pid_t pid;
        //instructions before fork()
        pid = fork();
        //instructions after fork()
        if (pid <0)
                printf("Error");
        else if (pid == 0)
                printf ("I am a child");
        else
                printf ("I am the father, my child pid is %d", pid);
}
```

Process Creation using fork()

- Syntax:
  - `int fork()`
- It takes no parameters and returns an integer value.
  - Negative value: creation of child process was not successful.
  - Zero: returned to the newly created child process.
  - Positive value: returned to the parent. The value contains process ID of newly child process.



Process Creation using fork()

- The two processes share:
  - the same program code.
  - the file descriptors (stdin, stdout and all the open files).
  - the user ID, the group ID, the root directory, the working directory.
- The new process consists of a copy of the address space of the original process
- Both processes share the same value of the Program Counter register:
  - Both processes continue the execution at the instruction after the fork().

Example

- Process creation:

```
int main()
{
pid_t  pid, fork_return;
    /* fork another process */
    fork_return = fork();
    if (fork_return < 0) { /* error occurred */
                printf("Fork Failed");
    }
    else if (fork_return == 0) { /* child process */
                {
                            printf("I'm the children!\n");
                }
    }
    else { /* parent process */
        printf("I'm the father! the child pid is %d\n", fork_return);
        }
Return 0;
}
```

Process Identification

- **getpid()** returns the process ID of the current process.

```
int main()
{
pid_t  pid, fork_return;
    /* fork another process */
    fork_return = fork();
    if (fork_return < 0) { /* error occurred */
                printf("Fork Failed");
    }
    else if (fork_return == 0) { /* child process */
                {
                pid = getpid();
                printf("I'm the children! pid: %d\n", pid);
                /* system call getpid() returns the pid of the calling process */
                }
    }
    else { /* parent process */
        printf("I'm the father! the child pid is %d\n", fork_return);
        }
Return 0:
}
```

Process Identification

- **getppid()** returns the process ID of the parent of the current process.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
        pid_t pid;
        pid = fork();
        if (pid <0)
                printf("Fork failed!");
        else if (pid == 0)
                printf ("I am a child with pid = %d\n and my parent pid = %d", getpid(), getppid());
        else
                printf ("I am the father, my child pid is %d", pid);
        return 0;
}
```

Example

- Including the initial parent process, how many processes are created by the following program?

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>


int main ()
{

1. Fork ()

…..

2. Fork()

….
}
```

F1

F2

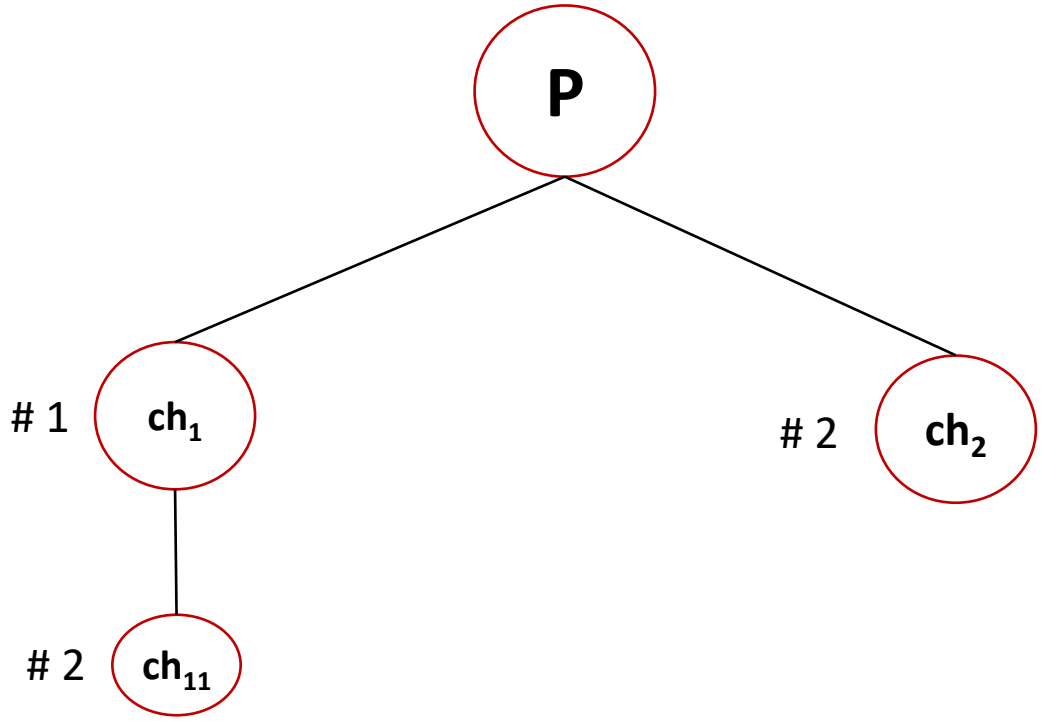
Example

- Including the initial parent process, how many processes are created by the following program?

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>


int main ()
{

1. Fork ()

…..

2. Fork()

….
}
```

F1

F2



Example

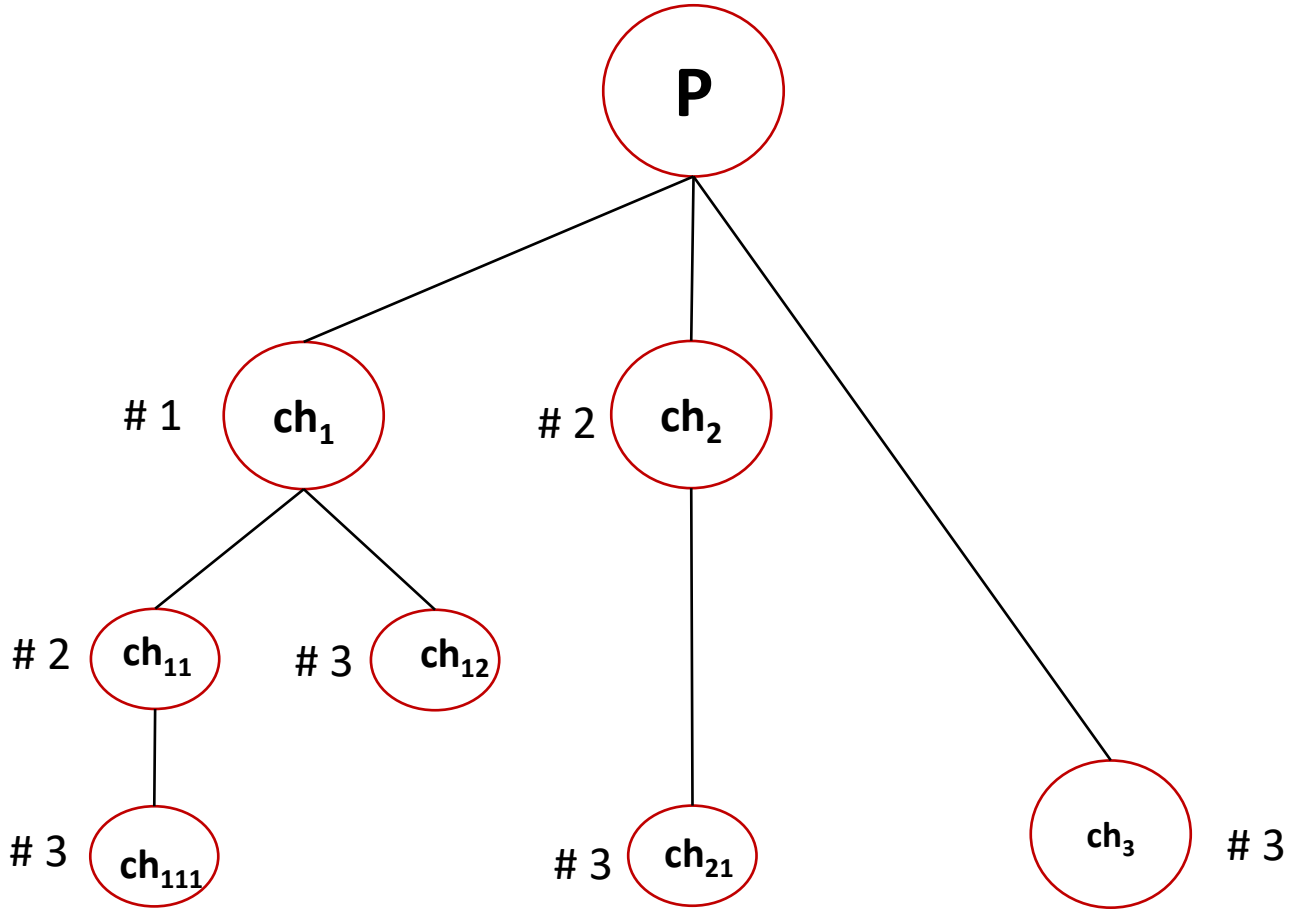
- Including the initial parent process, how many processes are created by the following program?

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
1. fork ()
…..
2. fork()
….
3. fork()
}
```

F1

F2

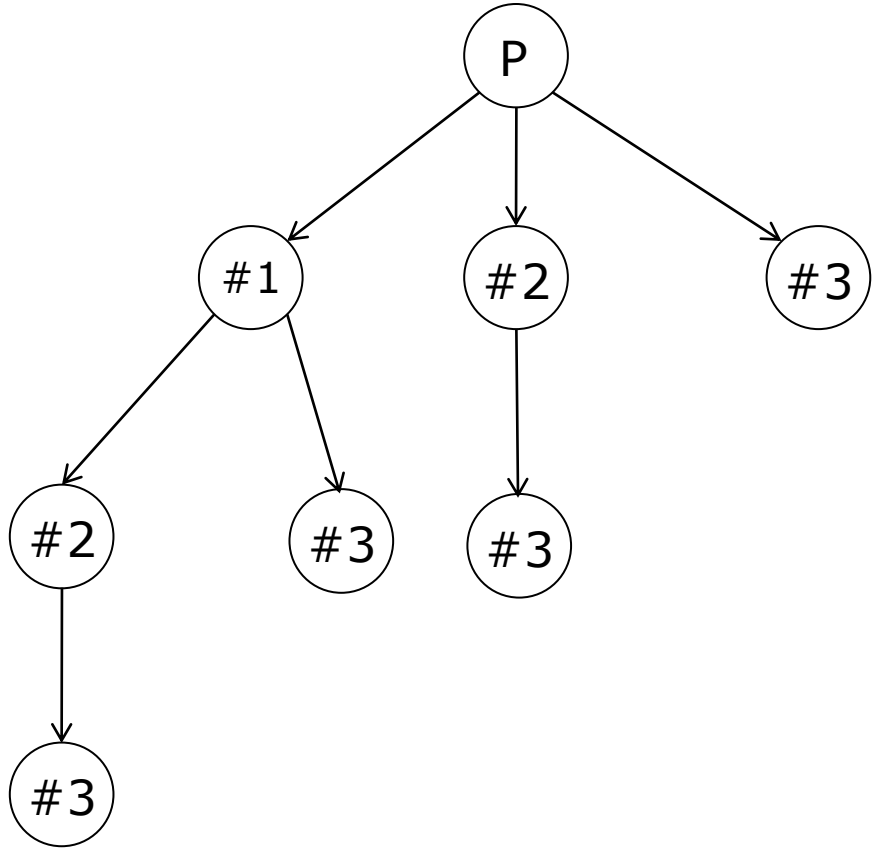
F3



Exercise

- Including the initial parent process, how many processes are created by the following program?

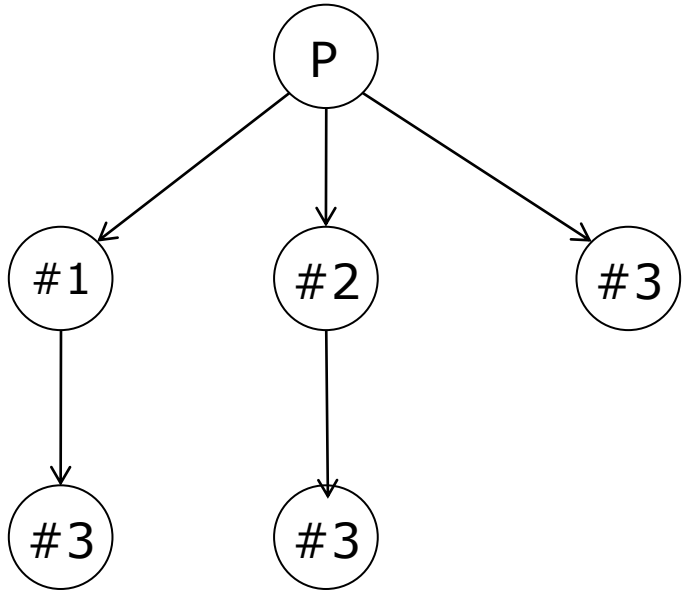
```
int main()
{
    /* fork a child process */
    fork();

    /* fork another process */
    fork();

    /* and fork another */
    fork();

}
```



Exercise

- Draw a diagram showing the parent/child relationships among the processes created by the following code fragment (showing which call to fork created each process).

```
pid = fork() /* call #1 */

if (pid != 0)
fork();  /* call #2 */

fork();  /* call #3 */
```



Exercise

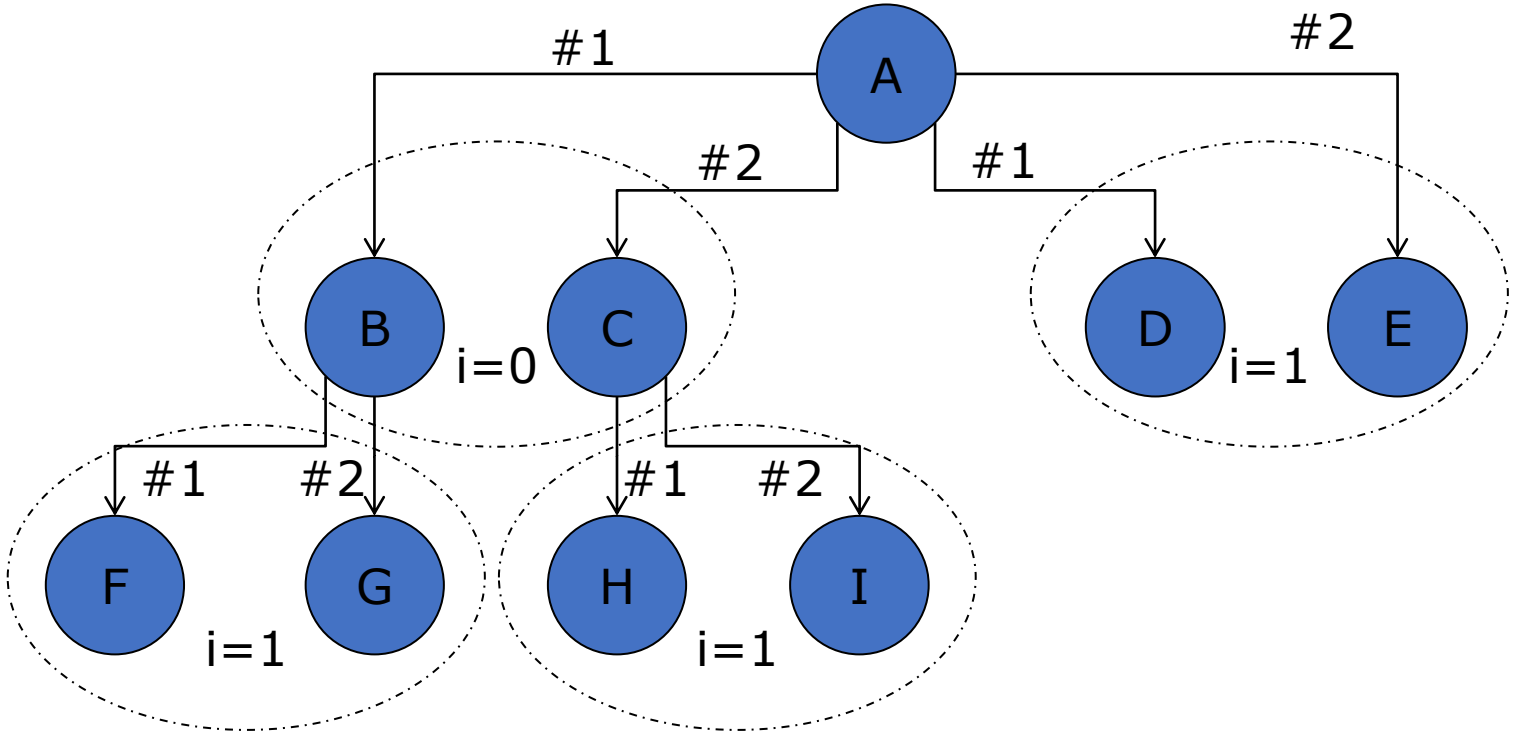
- Draw a diagram showing the parent/child relationships among the processes created by the following code fragment (showing which call to fork created each process).

```
pid = fork() /* call #1 */

fork();  /* call #2 */

if (pid != 0)
    fork();  /* call #3 */
```

Exercise

- Draw a diagram showing the parent/child relationships among the processes created by the following code fragment (showing which call to fork created each process).

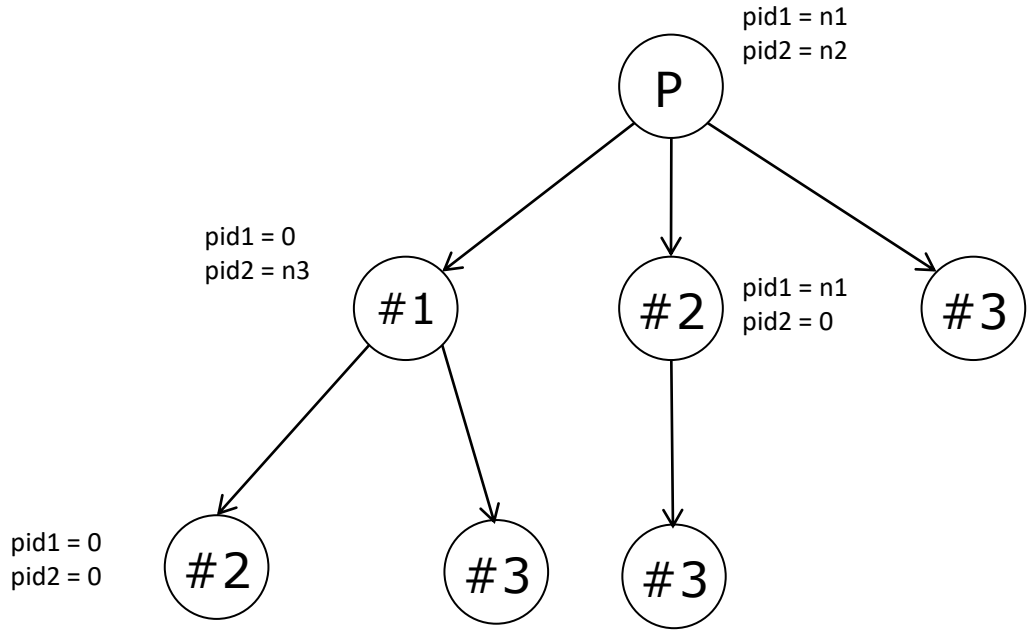
```
#include <stdio.h>
int i:
int main ()
{
for (i=0 ; i < 2 ; i++)
    {
    printf("i: %d \n", i);
    if (fork()) /* call #1 */
            fork(); /* call #2 */
    }
}
```



Exercise

- Draw a diagram showing the parent/child relationships among the processes created by the following code fragment (showing which call to fork created each process)

```
#include <stdlib.h>
main()
{
      pid1 = fork();  /* call #1 */
      pid2 = fork();  /* call #2 */
      if (pid1 != pid2)
            fork();  /* call #3 */
}
```



Exercise

- Write a program able to generate n child processes.
- Each created process has to generate its pid and terminates its execution

sleep()

- A process may suspend for a period of time using the sleep function.

- Syntax:
  - `unsigned int sleep (seconds)`

- The sleep() function cause the calling process to be suspended from execution until the number of realtime seconds specified by the argument seconds has elapsed.
- It is provided by `unistd.h` library

Process termination

- In Unix, if the parent terminates, all its children (become orphan) have assigned as their new parent the *init* process (whose PID is 1)

- Thus, the children still have a parent to collect their status and execution statistics.

Process termination

- exit() is a system call that terminates the process, asking the operating system to delete the process.
- Syntax:
  - `void exit(int status);`

  - The value of status is returned to the parent process (i.e., the lower 8 bits contain the status, and the upper 8 bits contain 0).
    - A status value of 0 or EXIT_SUCCESS represents success and any other value or EXIT_FAILURE indicates an error.
  - The exit() function does not return any value.

Value returned to the parent process

Example

```
#include <stdio.h>
main()
{
    int pid ;
    printf("I'm the original process with PID %d and PPID %d.\n", getpid(), getppid()) ;
    /* Duplicate. Child and parent continue from here */
    pid = fork ( ) ;
    if ( pid != 0 ) /* pid is non-zero, so I must be the parent*/
    {
            printf("I'm the parent with PID %d and PPID %d.\n", getpid(), getppid()) ;
            printf("My child's PID is %d\n", pid ) ;
    }
    else /* pid is zero, so I must be the child */
    {
            printf("I'm the child with PID %d and PPID %d.\n", getpid(), getppid()) ;
            sleep(4); /* make sure that the parent terminates first */
            printf("I'm the child with PID %d and PPID %d.\n", getpid(), getppid()) ;
    }
    printf ("PID %d terminates.\n", getpid()) ;
}
```

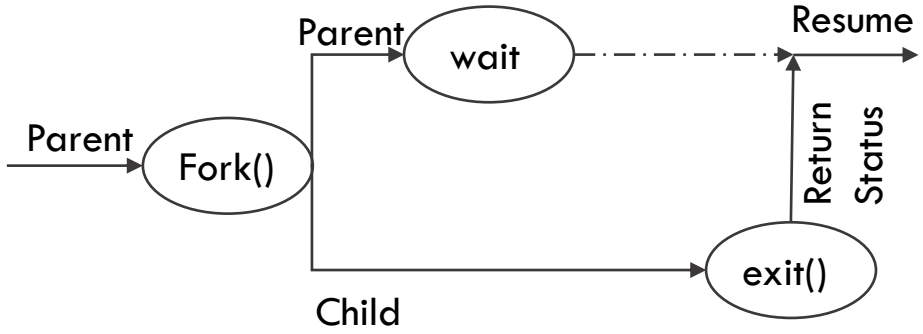
The output is:

```
I'm the original process with PID 5100 and PPID 5011.
I'm the parent process with PID 5100 and PPID 5011.
My child's PID is 5101
I'm the child process with PID 5101 and PPID 5100.
PID 5100 terminates. /* Parent dies */
I'm the child process with PID 5101 and PPID 1. /* Orphaned, whose parent process is "init" with pid 1 */
PID 5101 terminates.
```

Wait()

- wait() is a system call that suspends the execution of the parent process while the child executes.
- When the child process terminates, it returns an *exit status* to the operating system, which is then returned to the waiting parent process. The parent process then resumes execution.

- Syntax:
  - `int wait(int *status);`
  - `Status` is the address of the variable containing the exit status of the child process.
  - The parameter could be NULL.

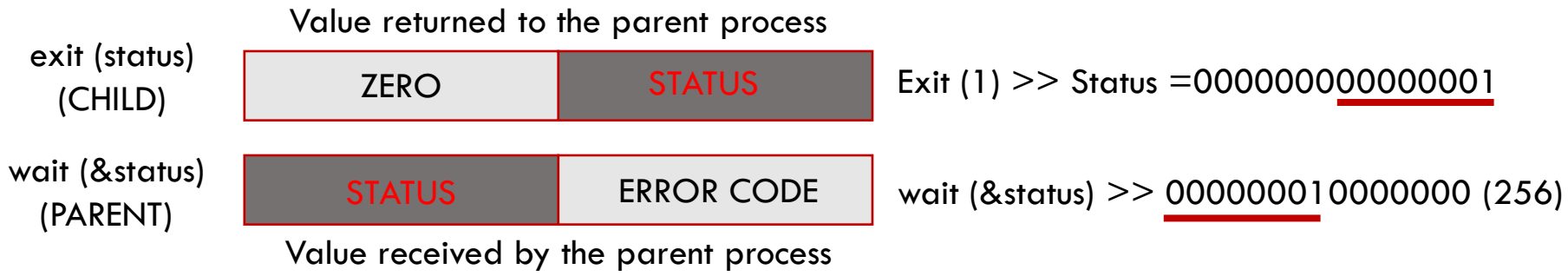


Wait()

- A process that calls `wait()` can:
  - **Suspend** (block) if all of its children are still running, or
  - **Return** immediately with the termination status of a child (if a child has already terminated)
  - **Return** immediately with an **error** if there are no child processes.
  - **Return** with the **termination** status of a child (the first one that terminates).

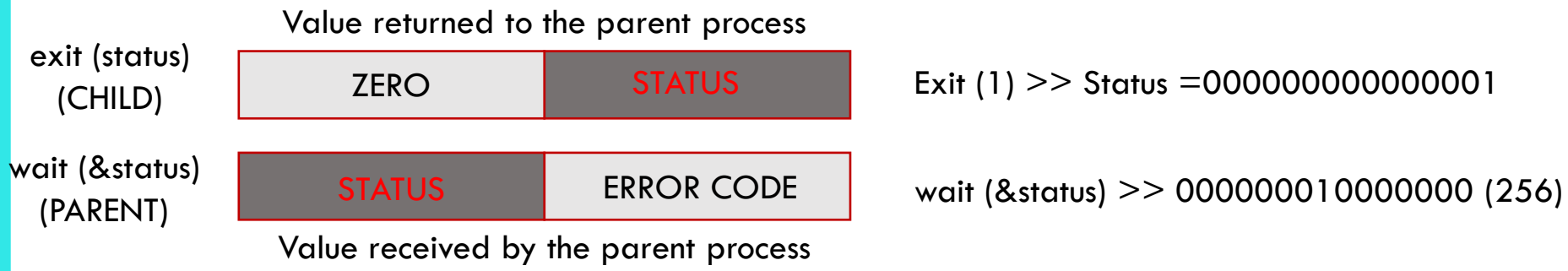
Wait()

- The argument to wait()is the address of an integer variable or the NULL pointer.
- If it's not NULL, the system writes 16 bits of status information about the terminated child.
- Among these 16 bits, the higher 8 bits contain the lower 8 bits of the argument the child passed to exit(), while the lower 8 bits are all zero if the process exited correctly and contain error information if not.



Read the exit status

Reading the **status** value received by parent from the child using macros available in **<sys/wait.h>**
1. Macro: *int* **WIFEXITED** *(int status)*
This macro returns a non zero value if the child process terminated normally with exit.
Macro: int ***WEXITSTATUS*** *(int status)*
This macro returns the low-order 8 bits of the exit status value from the child process



Read the exit status

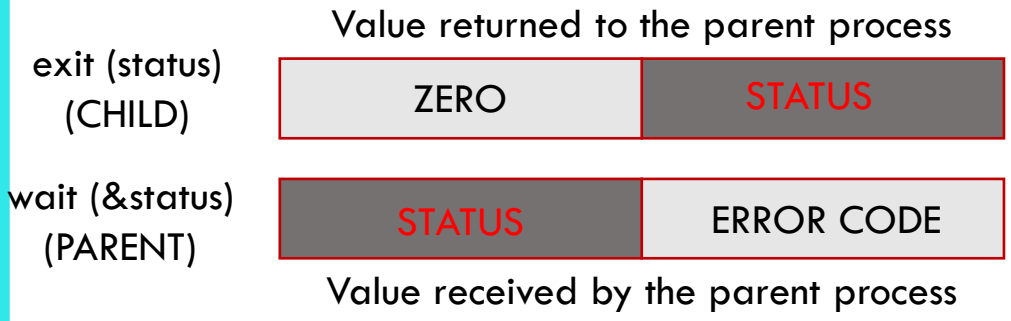
Reading the **status** value received by parent from the child using macros available in **<sys/wait.h>**

1. Macro: *int* **WIFEXITED** *(int status)*

This macro returns a non zero value if the child process terminated normally with exit.

Macro: int ***WEXITSTATUS*** *(int status)*

This macro returns the low-order 8 bits of the exit status value from the child process




```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (){
    pid_t pid, childpid;
    int status_Value;
    pid = fork ();
    switch (pid){
        case -1:
            printf ("fork failed\n");
            exit (-1);

        case 0://child
            printf ("I am child, PID: %d\n", getpid());
            sleep (5);
            exit (1);

        default://parent
            childpid = wait (&status_Value);
            printf ("Child end: PID = %d\n", childpid);
            printf ("WIFEXITED(status_Value)%d\n",WIFEXITED(status_Value));

            if (WIFEXITED (status_Value))
            //WIFEXITED : True if correctly terminated,WEXITSTATUS:select 8 LSB bit from exit())
                {printf ("Status Value using Macro: %d\n", WEXITSTATUS (status_Value));
                printf ("Status Value from the parent:%d\n",status_Value); }
            else
                printf ("Abnormal termination\n");}
    return 0;
}
```

```
Peppermint Terminal
sarah@sarah-VirtualBox ~/Desktop/OS_2020 $ gcc macros.c  -o macros
sarah@sarah-VirtualBox ~/Desktop/OS_2020 $ ./macros
I am child, PID: 2568
Child end: PID = 2568
WIFEXITED(status_Value)1
Status Value using Macro: 1
Status Value from the parent:256
sarah@sarah-VirtualBox ~/Desktop/OS_2020 $ _
```

Example

- Using the following program, what is the output at line A and line B?

```
int value = 5;
int main()
{
pid_t pid;
int status;

pid = fork();

if (pid == 0) { /* child process */
                { value += 15;
                  printf ("CHILD: value =%d\n", value); /* LINE A */
                  exit (0);
                }
else { /* parent process */
       wait (&status);
       printf ("PARENT: value =%d\n", value); /* LINE B */
     }
}
```

Example

- What might be the output when the following program is running?

```
int main(int argc, char *argv[]) {
        int x;
        pid_t pid;
        int status;
        x = 13;
        printf("x is now %d\n",x);
        pid = fork();
        if (pid == 0) {
                x = x+4;
                printf(" Add 4, and x = %d\n", x);
                exit(0);}
        else {
                x = x*4;
                printf(" Multiply by 4, and x = %d\n", x);
                wait(&status);}
        printf("In the end, x = %d\n", x);
        return 0;
}
```

Example

- Write a C program which uses Unix system calls to create a new process. The child process should create an empty file called ‘abc’ and then terminate. The parent process should wait for the child process to terminate and then output the process number of the child process.
- Don't forget to check for error conditions.

Solution

```
#include <stdio.h>
#include <sys/types.h>
main(){
    pid_t child;
    int status;
    FILE *fp;
    child=fork();
    if (child==-1)
            printf("Error creating child process");
    else if (child==0) {
            fp = fopen("abc", "w");
            if (fp==NULL)
                    printf("Error creating file");
            else
                    fclose(fp);
            exit(0);}
else {
        child=wait(&status);
        if (child==-1)
                printf("Error waiting for child process");
        else
                printf("%ld\n", (long)child);}
        }
```

Example

- If a child process makes the call exit(-1), what exit status will be seen by the parent?

Waitpid()

- wait() waits for any child.
- waitpid() waits for a specific child (the one with pid = waiting_for_pid)

- Syntax:

```
pid = waitpid(waiting_for_pid, &status, options);
```

  - Return value: on success, returns the process ID of the terminated chils whose state changed: on error, -1 is returned.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (){

    int status, i, N = 2 ;
    pid_t pid [N];
    for (i = 0 ; i < N ; i ++){
        pid[i] = fork ();
        if (pid[i]==0){
            sleep(1);
            exit (10+i);}}
    for (i= 0 ; i < N; i ++){
        waitpid (pid[i], &status, 0);
        if (WIFEXITED (status))
            printf ("The child %d terminated, exit status = %d\n", pid[i], WEXITSTATUS(status));
         else
            printf ("The child %d did not terminated correctly\n", pid[i]);}
    return 0;}
```

```
Peppermint Terminal
sarah@sarah-VirtualBox ~/Desktop/OS_2020 $ gcc waitpid.c  -o waitpid
sarah@sarah-VirtualBox ~/Desktop/OS_2020 $ ./waitpid
The child 5698 terminated correctly with exit status = 10
The child 5699 terminated correctly with exit status = 11
sarah@sarah-VirtualBox ~/Desktop/OS_2020 $
```

Example

- The parent process has to create N processes.
- The parent process has to wait the end of each of them in the same order of creation.

```
int main()
{
int status, i;
pid_t pid[N];
for (i=0; i<N ; i++)
if ((pid[i]=fork())==0)
{
sleep(1);
exit(10+i);
}
for(i=0; i<N ; i++){
waitpid(pid[i], &status, 0);
  if (WIFEXITED(status))
        printf("The child %d terminated correctly with exit status =%d\n",  pid[i], WEXITSTATUS(status));
  else
       printf("The child didn't %d terminate correctly\n", pid);
  }
exit(0);
}
```

Recap

- `wait()` and `waitpid()` are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.

- A state change is considered to be:
  - The child terminated.
  - The child was stopped by a signal.
  - The child was resumed by a signal.
- In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.

Zombie Processes

- A process that terminates cannot leave the system until its parent accepts its return code.
  - Note: the parent accepts the child's return code either via a wait() or if it terminates.
- If its parent process is already dead, it'll already have been adopted by the **"init" process**, which always accepts its children's return codes.
- However, if a process's parent is alive but never terminates, the process's return code will never be accepted, and the process will remain a zombie.

Example

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{  pid_t child_pid;
  /* Create a child process.  */
  child_pid = fork ();
  if (child_pid > 0) {
    /* This is the parent process.  Sleep for a minute.  */
    sleep (60);
  }
  else {
    /* This is the child process.  Exit immediately.  */
    exit (0);
  }
  return 0;
}
```

Zombie's Life

The output is:

> a.out & execute the program in the background

[1] 32482

> ps -h obtain process status

PID	TT	STAT	TIME	COMMAND	
31021	pts/1	S	0:01	bash	the shell
32482	pts/1	S	0:01	a.out	
32483	pts/1	Z	0:00	[a.out] <defunct>	the zombie child process
32484	pts/1	R	0:00	ps -h	

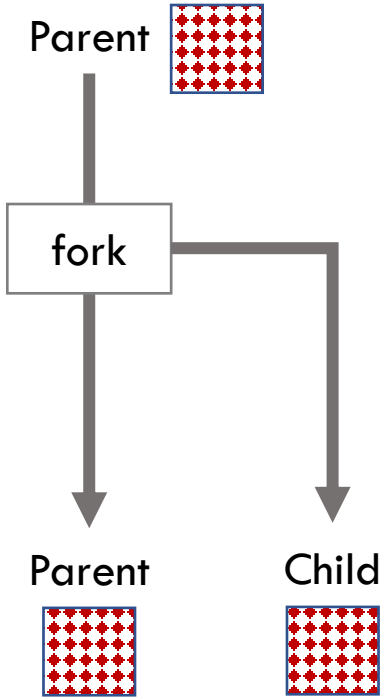
After 1 minute

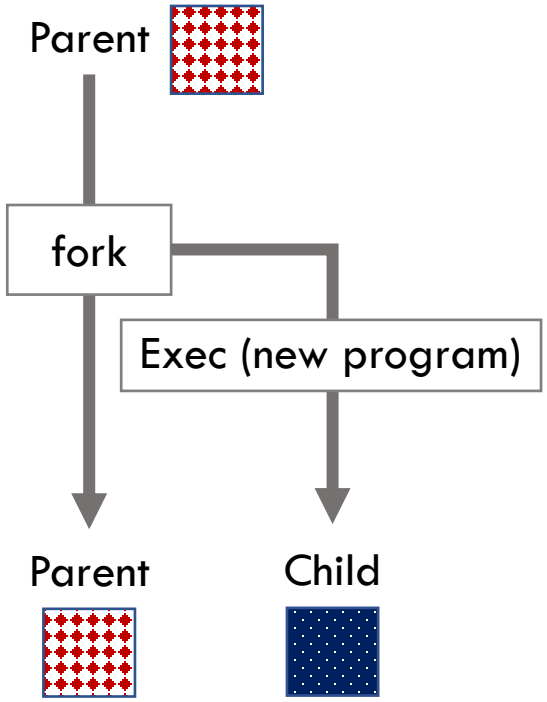
> ps -h

PID	TT	STAT	TIME	COMMAND	
31021	pts/1	S	0:01	bash	the shell
32502	pts/1	R	0:00	ps -h	

Exec() Family

- `Exec` family of function are replacing binary files of process's program with the one inside the exec().
- It comes under the header file `unistd.h`.





Exec() Family

- Family of functions for replacing process's program with the one inside the `exec()` call

- Typically the `exec()` system call is used after a `fork()` system call by one of the two processes to replace the process's memory space with a new program
- The `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts its execution
- As a new process is not created, the PID does not change across an `exec()`, but the data, heap and stack of the calling program are replaced by those of the new program

- There is a family of different exec functions: there are 6 versions of the exec function, and they all do about the same thing: they replace the current program with the text of the new program. Main difference is how parameters are passed.

Exec() Family

- There are 6 different ways of calling exec. Which one to use depends on three conditions:
  - How arguments are passed (list of parameters or array)
  - How the path is specified (pathname or filename)
  - Whether a new environment is used

*int execl(const char *pathname, const char *arg0, ...);*

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

*int execlp(const char *filename, const char *arg0, . . .);*

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

*int execle(const char *pathname, const char *arg0, ... , char *const envp[]);*

*int execve(const char *pathname, char *const argv[], char *const envp[]);*

Execl()

- Takes the path of the executable binary file as the first argument. Then the argument that you want to pass to the executable followed by NULL. The execl() system call runs the command and prints the output. If any error occurs, the execl() returns -1.
- Syntax:

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

Execl() Example

- Syntax:

  int execl (const char *path, const char *arg, ..., NULL);

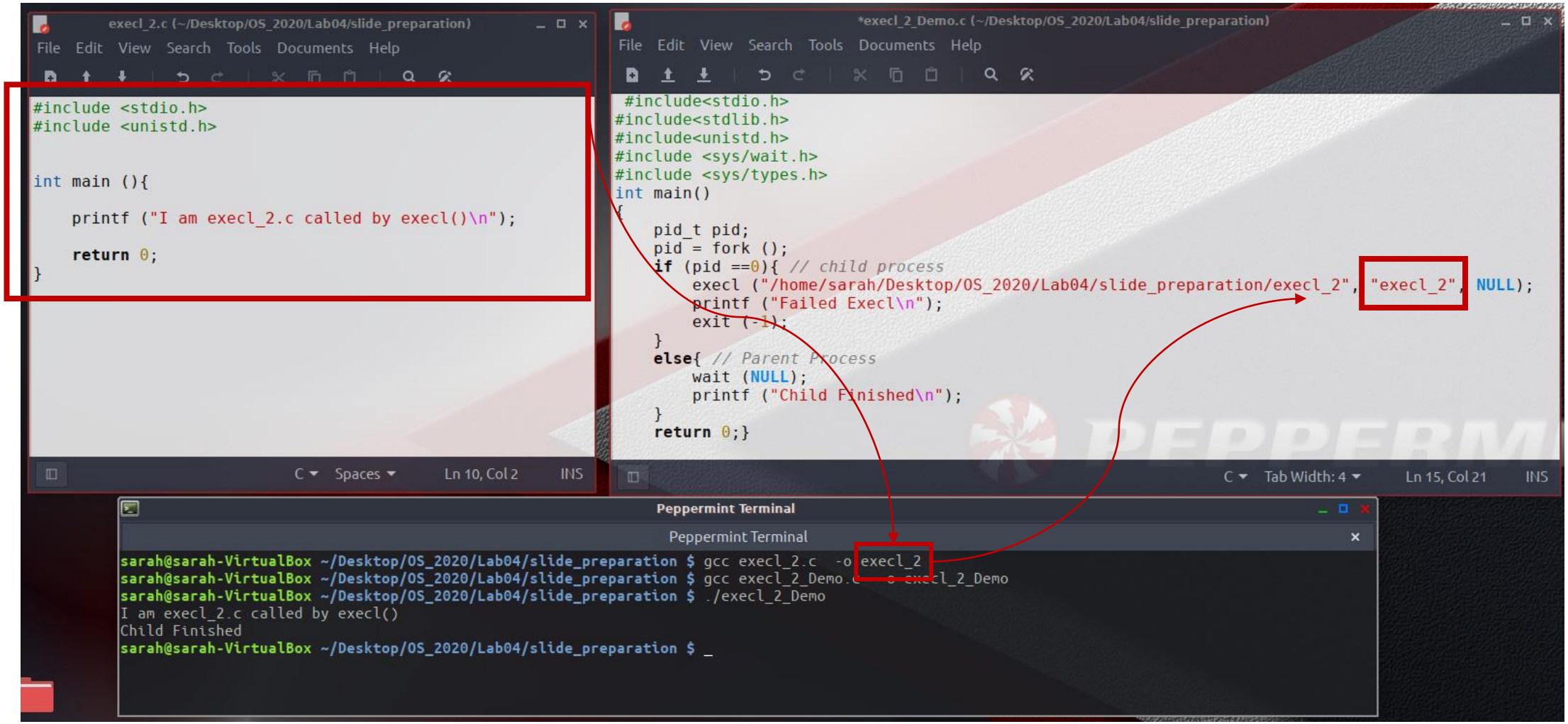
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (){

    pid_t pid;
    pid = fork ();
    if (pid ==0){ // child process

        execl ("/bin/ls", "ls", "-a", "/home/sarah/Desktop/OS_2020", NULL);

        printf ("Failed Execl\n");
        exit (-1);
    }
    else{ // Parent Process
        wait (NULL);
        printf ("Child Finished\n");
    }
    return 0;
}
```

```
Peppermint Terminal
Peppermint Terminal
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ which ls
/bin/ls
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ gcc execl.c  -o exec
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ ./exec
.  ..  Lab01  Lab02  Lab03  Lab04
Child Finished
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ _
```

Execl() Example

- Syntax:

  int execl (const char *path, const char *arg, ..., NULL);



Execlp()

- Execl () does not use the PATH environment variable. So, the full path of the executable file is required. While execlp () uses PATH environment. So, if an executable file or command is available in the PATH, then the command or the filename is enough to run it.
- Syntax:

```
int execlp (const char *file, const char *arg, ..., NULL);
```

Execlp() Example

- Syntax:

  int execlp (const char *file, const char *arg, ..., NULL);

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (){

    pid_t pid;
    pid = fork ();
    if (pid ==0){ // child process

        execlp ("ls", "-a", "/home/sarah/Desktop/OS_2020", NULL);

        printf ("Failed Execl\n");
        exit (-1);
    }
    else{ // Parent Process
        wait (NULL);
        printf ("Child Finished\n");
    }
    return 0;
}
```

```
Peppermint Terminal
Peppermint Terminal
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ gcc execlp.c  -o execlp
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ ./execlp
Lab01  Lab02  Lab03  Lab04
Child Finished
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $
```

Execv()

- In execl (), the parameters of the executable file is passed to the function as different arguments. While execv (), you can pass all the parameters in a NULL terminated array argv. The first element of the array should be the path of the executable file.

- Syntax:

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

Execv() Example

- Syntax:

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (){

    pid_t pid;

    char *binaryPath = "/bin/ls";
    char *args [] = {binaryPath, "-ls", "/home/sarah/Desktop/OS_2020", NULL};

    pid = fork ();
    if (pid ==0){ // child process

        execv (binaryPath, args);

        printf ("Failed Execl\n");
        exit (-1);
    }
    else{ // Parent Process
        wait (NULL);
        printf ("Child Finished\n");
    }
    return 0;
}
```

```
Peppermint Terminal
Peppermint Terminal
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ gcc execv.c -o execv
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ ./execv
total 16
4 drwxrwxr-x 3 sarah sarah 4096 mar 29 11:36 Lab01
4 drwxrwxr-x 2 sarah sarah 4096 apr  6 11:06 Lab02
4 drwxrwxr-x 3 sarah sarah 4096 apr 10 16:30 Lab03
4 drwxrwxr-x 4 sarah sarah 4096 apr 10 16:30 Lab04
Child Finished
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $
```

Execvp()

- This function works the same way as execv (), but the PATH environment variable is used. So, the full path of the executable file is not required as in execlp ().

- Syntax:

```
int execv (const char *file,  char  * const argv [ ], ..., NULL);
```

Execvp() Example

- Syntax:

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (){

    pid_t pid;

    char *programName= "ls";
    char *args [] = {programName, "-ls", "/home/sarah/Desktop/OS_2020", NULL};

    pid = fork ();
    if (pid ==0){ // child process

        execvp (programName, args);

        printf ("Failed Execl\n");
        exit (-1);
    }
    else{ // Parent Process
        wait (NULL);
        printf ("Child Finished\n");
    }
    return 0;
}
```

```
Peppermint Terminal
Peppermint Terminal

sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ gcc execvp.c  -o execvp
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ ./execvp
total 16
4 drwxrwxr-x 3 sarah sarah 4096 mar 29 11:36 Lab01
4 drwxrwxr-x 2 sarah sarah 4096 apr  6 11:06 Lab02
4 drwxrwxr-x 3 sarah sarah 4096 apr 10 16:30 Lab03
4 drwxrwxr-x 4 sarah sarah 4096 apr 10 16:30 Lab04
Child Finished
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $
```

Execle()

- Works like execl() but you can provide your own environment variables along with it. The environment variables are passed as an array envp. The last element of the envp array should be NULL.

- Syntax:

```
int execle (const char *path, const char *arg, ..., NULL, char * const envp[] );
```

Execle() Example

- Syntax:

  int execle (const char *path, const char *arg, ..., NULL, char * const envp[]);

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (){

    pid_t pid;

    char *binaryPath = "/bin/bash";
    char *arg1 = "-c";
    char *arg2 = "echo \"Visit $HOSTNAME:$PORT from your browser.\"";
    char *const env[] = {"HOSTNAME=www.linuxhint.com", "PORT=8080", NULL};

    pid = fork ();
    if (pid ==0){ // child process

        execle(binaryPath, binaryPath,arg1, arg2, NULL, env);

        printf ("Failed Execl\n");
        exit (-1);
    }
    else{ // Parent Process
        wait (NULL);
        printf ("Child Finished\n");
    }
    return 0;
}
```

```
Peppermint Terminal
Peppermint Terminal
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ gcc execle.c -o execle
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ ./execle
Visit www.linuxhint.com:8080 from your browser.
Child Finished
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $
```

Execve()

- Just like execle () you can provide your own environment variables along with execve(). You can also pass arguments as arrays as you did in execv().

- Syntax:

```
int execle (const char *file, const char *argv [ ], ..., NULL, char * const envp[] );
```

Execve() Example

- Syntax:

  int execle (const char *file, const char *argv [ ], ..., NULL, char * const envp[]);

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (){

    pid_t pid;
    char *binaryPath = "/bin/bash";
    char *const args[] = {binaryPath, "-c", "echo \"Visit $HOSTNAME:$PORT from your browser.\"", NULL};
    char *const env[] = {"HOSTNAME=www.linuxhint.com", "PORT=8080", NULL};

    pid = fork ();
    if (pid ==0){ // child process

        execve(binaryPath, args, env);

        printf ("Failed Execl\n");
        exit (-1);
    }
    else{ // Parent Process
        wait (NULL);
        printf ("Child Finished\n");
    }
    return 0;
}
```

```
Peppermint Terminal
Peppermint Terminal
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ gcc execve.c -o execv
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $ ./execv
Visit www.linuxhint.com:8080 from your browser.
Child Finished
sarah@sarah-VirtualBox ~/Desktop/OS_2020/Lab04/slide_preparation $
```

System()

- System() passes the command name or program name specified by command to the host environment to be executed by the command processor and returns after the command has been completed.

- Syntax:

```
int system (const char *command);
```

The function returns :

- -1 on error
- The return status of the command

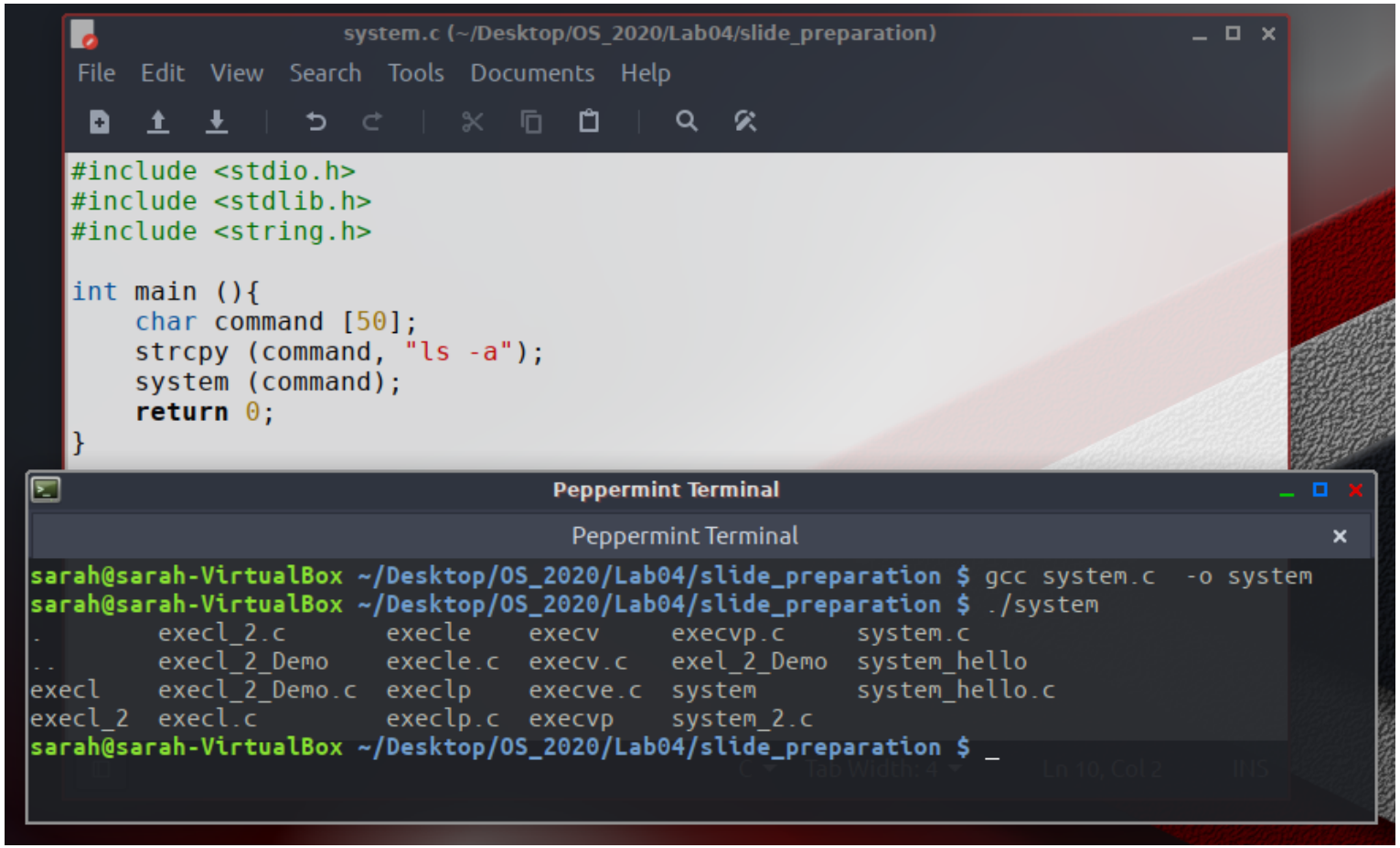
System() Example

- Syntax:

  int system (const char *command);

The function returns :

- -1 on error
- The return status of the command



System() Example

- Syntax:

  int system (const char *command);