OPERATING SYSTEMS
– Laboratory 9 –

UNIX PROCESSES

1. UNIX PROCESSES

- *process* = a *program that is running*
- the *kernel* of the operating system maintains permanently a tables with process IDs
- examine processes:

```
ps -e
ps -f -p 1
ps -F -u dbota
```

2. FUNCTION `fork()`

- prototype:

```
#include <unistd.h>

pid_t fork(void);
```

- creates a new process by duplicating the calling process (parent process)
- the new process is called a child process and is an exact copy of the parent process
- the two processes continue their execution with the instructions that follow the **`fork()` call**
- the function returns:
  - `0` - in the child process
  - The identifier of the child process created (child PID) - the parent process
  - `-1` - if the call failed (error)
- the **`fork()`** function call can fail if:
  - there is not enough memory for the creation of the child process
  - the number of total processes exceeds the maxim allowed limit
- examples: `fork_1.c, fork_2.c, fork_3.c`

3. FUNCTIONS `wait()`, `waitpid()`

- prototypes:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- **`wait()`** suspends the execution of the calling process until a child process terminates
- **`wait(&status)`** call is equivalent with **`waitpid(-1, &status, 0)`**

- **`waitpid()`** suspends the execution of the calling process until:
  - the child process specified with the argument **`pid`** finishes its execution
  - the child process specified with the argument **`pid`** was stopped by a signal
  - the child process specified with the argument **`pid`** was restarted by a signal
- the meaning of the values of the argument `pid`:

pid	Meaning
`< -1`	Wait all child processes to finish for those with group ID (GID) equal with the absolute value of `pid`
`-1`	Wait for all child processes to finish
`0`	Wait for all child processes to finish for those with group ID (GID) equal with the GID of the parent process
`> 0`	Wait for the process with PID specified to finish `pid`

- examples: `fork_4.c, fork_5.c`

4. FUNCTION `signal()`

- prototype:

```
#include <signal.h>

sighandler_t signal(int signum, sighandler_t handler);
```

- established the action mode (handler) of a signal sign
- signals:

  `man 7 signal`
- if the signal *`signum`* can be sent to a process, then it can choose:
  - to ignore the signal - `SIG_IGN`
  - to handle it implicitly - `SIG_DFL`
  - to specify a function that defines the actions that are executed at a signal occurrence
- prevent „zombie" processes:

  `signal(SIGCHLD, SIG_IGN)`
- example: `fork_7.c`

5. FUNCTION `kill()`

- prototype:

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

- allows sending a signal to a process or group of processes
- meaning of the values of the argument `pid`:

pid	Significance
> 0	Signal is sent to the process with the specified PID by parameter pid
0	Signal is sent to all the processes with group ID (GID) equal to GID of the calling process
-1	Signal is sent to all the processes for which the calling process has permission to deliver signals (except process init)
< -1	Signal is sent to all the processes with group ID (GID) equal to the absolute value of the parameter pid

- signals:

  man 7 signal

6. EXAMPLES

- template of child/parent with fork

```
/*  fork_1.c
    Create a child process using fork()
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
   int a = 5;
   int pid = fork();       // try to create a child process
   if (pid == -1)          // fork() has failed
   {
      perror("fork() error\n");
      exit(EXIT_FAILURE); //exit(1);
   }

   if (pid == 0)           // in the child process
   {
      printf("[In CHILD] My PID is %d. My parent PID is %d.\n", getpid(), getppid());
      exit(EXIT_SUCCESS); //exit(0);
   }
   else              // in the parent process
   {
      printf("[In PARENT] My PID is %d. My child PID is %d.\n", getpid(), pid);
      int status;
      wait(&status); // wait(0);
   }
   return 0;
}
```

- exercise. Use the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
   int p, i;
   p=fork();
   if (p == -1) {perror("fork impossible!"); exit(1);}
   if (p == 0) {
       for (i = 0; i < 10; i++)
            printf("Child: i=%d pid=%d, ppid=%d\n", i, getpid(), getppid());
            exit(0);
   } else {
       for (i = 0; i < 10; i++)
            printf("Parent: i=%d pid=%d ppid=%d\n", i, getpid(), getppid());
            wait(0);
  }
  printf("Finished; pid=%d ppid=%d\n", getpid(), getppid());
}
```

a) Run as is. How many lines are printed?

b) Comment the exit(0) call. What happens if you run the program now?

c) Comment the wait(0) call. What happens now?

d) Comment both exit and wait and change the number of iterations in the for statements to be different (e.g. 5 and 7, 20 and 7)

More examples: fork folder.

7. EXEC functions

Execute an external program.

The exec() family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for execve(2). (See the manual page for execve(2) for further details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that is to be executed. The const char *arg and subsequent ellipses in the execl(), execlp(), and execle() functions can be thought of as

arg0, arg1, ..., argn. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments must be terminated by a null pointer, and, since these are variadic functions, this pointer must be cast (char *) NULL.

The execv(), execvp(), and execvpe() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers must be terminated by a null pointer.

The execle() and execvpe() functions allow the caller to specify the environment of the executed program via the argument envp. The envp argument is an array of pointers to null-terminated strings and must be terminated by a null pointer. The other functions take the environment for the new process image from the external variable environ in the calling process.

Examples:

Program 7.1:

```
#include <stdio.h>
#include <unistd.h>
int main() {
   char* argv[3];
   argv[0] = "/bin/ls";
   argv[1] = "-l";
   argv[2] = NULL;
   execv("/bin/ls", argv);
}
```

Program 7.2:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // system
int main() {
    //uncomment each of these lines one by one - only one of the 5
    //execl("/bin/ls", "/bin/ls", "-l", NULL);
    // execlp("ls", "ls", "-l", NULL);
    // execl("/bin/ls","/bin/ls","-l","p1.c","execl.c", "fork1.c", "xx", NULL);
    // execl("/bin/ls","/bin/ls","-l","*.c", NULL); //will not be interpreted as expected
    system("ls -l *.c");
}
```