OPERATING SYSTEMS
– Laboratory 9 –

UNIX PROCESSES

1. <u>UNIX PROCESSES</u>

- *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 (<u>parent process</u>)
- the new process is called a <u>child process</u> 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:

  - <u>0</u> – in the child process

  - <u>The identifier of the child process created (child PID)</u> – the parent process

  - <u>-1</u> – 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 <u>PID</u> by parameter `pid`
`0`	Signal is sent to all the processes with group ID (GID) <u>equal to GID of the calling process</u>
`-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) <u>equal to the absolute value of the parameter `pid`</u>

- signals:

```
man 7 signal
```

6. EXAMPLES

- template of child/parent with fork

```c
/*  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");
}
```

REFERENCES:

▪ Tutorial IPC: https://beej.us/guide/bgipc/html/single/bgipc.html

Lab 9 – Homework – Processes in C

No.	Assignment
1.	We have a file that contains the first N natural numbers ordered. Let P*i* a process that eliminates all multiples of *i* from the interval 2 to N given. Using a suitable number of P*i* processes, write a program that determines all prime numbers between 1 and N given.
2.	A natural number is perfect if it is equal with the sum of all its divisors not including the number itself. Write a program that uses an adequate number of processes to determine all the perfect numbers smaller than N given.
3.	We have a file that contains N integer numbers. Using two types of processes (one for determining the minimum and the other to determine the maximum value from an array), write a program that determines the kth element in increasing order of the integer array, without sorting the array.
4.	Given P*i* a process that checks if *i* is a divisor of a number N given, and determines the multiplicity order of this divisor. Launch a suitable structure of such processes to decompose the number N given in prime factors.
5.	Given P a process that calculates the greatest common divisor (gcd) of two numbers. Using a suitable structure of such processes, write a program that determines the gcd of N given numbers.
6.	Given P*i* the process that determines the frequency of occurrences of letter *i* in a given text. Write a program that uses such processes to print a statistic of frequency for each letter occurrence.
7.	An array of numbers is given. Calculate the sum of the cubes of these numbers such that the cube of each number is calculated by a separate process.
8.	The following arithmetic expression is given (a+b*c) / (a-b+d-c) + a*b*c*d. Write a program that determines the value of this expression, such that each arithmetical operation is executed by a separate process.
9.	An array of numbers is given x1, x2, ..., xn. Compute the partial sums x1, x1+x2, x1+x2+x3, ..., x1+x2+...+xn, using a separate process for each partial sum.
10.	An array of letters is given. Using a separate process for each vowel, delete all vowels from the array.
11.	Given the Polynomic A=(a0, a1, ...,an). Compute the value of A for an x, using Horner formula and several processes. For example: $a0+a1*x+a2*x^2$ can be computed: <table>

Table for assignment 11:

	x^2	x	x^0
	a2	a1	a0
x=v	a2=r1	a1+r1*v=r2	a0+r2*v=**r3**

12.	Compute the sum of an array of numbers using *divide et impera* method: a process splits the array in two sub-arrays and gives them to two other chid processes to compute their sums, then adds the results obtained. The child processes use the same technique (split, etc. …).
13.	Calculate the product of an array n of numbers using *divide et impera* method: a process splits the array in two sub-arrays and gives them to two other chid processes to compute their product, then multiplies the results obtained from the two. The two child processes apply the same technique (split, etc…).