Processes in Linux

A process is a *software component* running under the control of the operating system kernel. Each process has a context associated with it, consisting of data area, code and stack.

The context of a process is inaccessible to other processes, and a process can only act on its own context. A process communicates with the outside world under the supervision of the operating system, thus ensuring stable operation without interference between processes.

The context of a process includes different types of control information, among which it is worth highlighting the Process Identifier (PID) and the User Identifier (UID):

The PID (*Process IDentifier*) is a positive integer that identifies a given process, it is assigned by the operating system when the process is created.

The UID (*User IDentifier*) is a positive integer that identifies a user, assigned by the operating system or by the administrator when the user is defined. A series of user rights are associated with the UID. A process acquires the UID of the user that creates it, and therefore has the same rights.

When a user creates a process by invoking a command, loading the respective executable file into memory, the process acquires the UID and rights of the invoker. However, when the owner *of* an executable file so wishes, he can allow the corresponding process to assume its UID, regardless of the user invoking it (there are obvious security risks for the owner).

Start of new processes

There is a single *system-call* that ensures this function, it is a primitive base for any multi-process operating system:

```
int fork(void);
```

When a process invokes this *system-call*, the operating system creates a copy of the current process context and assigns it a new PID. From that moment on, there are two exactly the same processes. The original process (which keeps the same PID) is called the parent process, and the new process (with a new PID) is called the child process.

Of course, the code that you want to execute after the **`fork`** is different for the parent and the child, so that the executing code knows which process it is in, just analyze the return value of the **`fork`**:

- To the parent process the *system-call* **`fork`** returns the child's PID (positive integer).
- To the child process the *system-call* **`fork`** returns the value zero.

It is typical to insert a *system-call* **`fork in an if`** function:

```
if (fork())
{
    /* continuation of parent process */
}
else
{
    /* continuation of child process */
}
```

Any process can easily obtain its own PID and also the PID of its parent. The *system-call* **`int getpid(void)`** returns to the invoking process its own PID. The *system-call* **`int getppid(void)`** returns to the invoking process the PID of its parent.

The following example illustrates the use of *fork* and these two *system-calls*:

```
void main(void)
{
    int pid ;

    printf("There is still only one process (%i)\n", getpid());
    pid = fork();
    printf("Now we are two (%i)\n", getpid());
    if (pid != 0)
    {
        printf("I am the father(%i), my son: %i\n", getpid(), pid);
        sleep(1);
    }
    else
    {
        printf("I am the son(%i), my father: %i\n", getpid(), getppid());
    }
}
```

A *system call* **`sleep`** is used to put the parent process to sleep for one second, ensuring that the parent does not terminate before the child.

Synchronization between parent and child processes

In the previous example the parent process was suspended for a second to have certain guarantees that the child finishes first. The consequence of this not happening is that if the *system-call* **`getppid()`** was called after the parent had finished, it would return an unexpected value. It would actually return the PID of the "parent's parent" since it would be the parent that would adopt the "orphaned child".

There are more powerful and expedient mechanisms to ensure synchronization between parent and children. The *system-call* **`wait`** suspends the parent process until a child terminates:

```
int wait(int *);
```

The return value of this *system-call* is the PID of the child that has terminated. If there are no more children the *system-call* **`wait`** returns the value **-1** .

A *system call* **`wait`** has as a parameter a pointer to an integer that will be used to store the *exit-status* of the child. This mechanism allows the child, when it finishes, to send the parent a numeric value, in practice it is limited to the 8 least significant bits that will have to be obtained using the "macro":

```
int WEXITSTATUS(int);
```

Children pass their exit code to the parent using the *system-call* **`exit`** which ends the process:

```
void exit(int);
```

While the invocation of *system-call* **`exit`** is not matched by *the system-call* **`wait`** in the parent process the child process remains in a state known as *zombie* in which it does not occupy resources.

The following example illustrates the use of this type of mechanism:

```
void main(void)
{
    int i, pid, ret;

    for (i = 1; i < 11; i++)
    {
        pid = fork();
        if (!pid)
        {
            printf("Sou o filho número %i\n",i);
            exit(i+10);
        }
        printf("Pai: lancei o filho %i com o PID %i\n", i, pid);
    }

    while ((pid = wait(&ret))!= -1)
    {
        ret=WEXITSTATUS(ret);
        printf("O filho com PID %i devolveu-me %i\n", pid, ret);
    }
}
```

Transferring data from the parent process to the child at the time of its creation is simple because all data defined by the parent is copied to the child. Data transfer from the child process when it terminates to the parent process is limited to 8 *bits* . This fact becomes irrelevant because there are appropriate mechanisms for interprocess communication (IPC).

exec functions

The **`exec functions`** use another basic primitive, although they vary in the form of their parameters, they all perform the same task: they load a binary file that replaces the code and data of the current process context and starts its execution.

Unless an error occurs, the execution of the code in the invoking process ends. When you want to execute a command in parallel to the main process, simply create a child that invokes an **`exec function`** .

There are two ways to specify the executable file and its arguments, in a list (suffix "l") or in the form of a vector (suffix "v"):

```
int execl (char *path, char *argv0, char *argv1, char *argv2, ..., NULL);
int execv (char * path, char * argv []);
```

In either case, the *path parameter* represents the path to the location where the binary file is located. The first argument (argv0 or argv[0]) is always the name of the binary file.

The way of indicating the arguments varies, but in any case it ends with the NULL argument ((char *)0).

The variants with the suffix "e" allow you to define environment variables for the execution of the command, the initial part of the parameters is identical to the previous ones:

```
int execle (..., char * envp []);
int execve (..., char * envp []);
```

Variants with the "p" suffix use the PATH environment variable to search for the executable file. In this case, the first parameter becomes the name of the executable file. If this file name does not contain the path, then the function searches for it in the sequence of paths specified in the PATH variable:

```
int execlp(char *filename, char *argv0, char *argv1, char *argv2, ..., NULL);
int execvp(char *filename, char *argv[]);
```

Examples:

```
1) execl ( "/ bin "," ls ","-la","~/",NULL);
2) execlp ( "/ bin / ls "," ls ","-la", "~/",NULL);
3) execlp ( " ls","ls ","-la","~/",NULL);
```

Examples 1 and 2 are completely equivalent. Example 3 will have the same effect if the path "/bin" is in the PATH.

Example with vector:

```
...
char *arg[4];

arg[0]="ls";
arg[1]="-la";
arg[2]="~/";
arg [ 3]=NULL;
execv ( "/bin", arg );
printf("EXEC function failed\n");
...
```

The error message is pertinent to any **`exec function`** , if an **`exec function`** succeeds, the following statement is never executed.

The following example creates a child process to execute the same command in parallel:

```
...
char *arg[4];

arg[0]="ls";
arg[1]="-la";
arg [2]="~/";
arg [3]=NULL;
...
If (!fork())
{
    execvp(arg[0],arg);
    printf("EXEC function failed\n");
    exit(1);
}
...
/* continuation of parent process */
```