# PERL - PROCESS MANAGEMENT

You can use Perl in various ways to create new processes as per your requirements. This tutorial will list down few important and most frequently used methods of creating and managing Perl processes.

- You can use special variables **$$** or **$PROCESS_ID** to get current process ID.
- Every process created using any of the mentioned methods, maintains its own virtual environment with-in **%ENV** variable.
- The **exit** function always exits just the child process which executes this function and the main process as a whole will not exit unless all running child-processes have exited.
- All open handles are dup-ed in child-processes, so that closing any handles in one process does not affect the others.

## Backstick Operator

This simplest way of executing any Unix command is by using backstick operator. You simply put your command inside the backstick operator, which will result in execution of the command and returns its result which can be stored as follows −

```
#!/usr/bin/perl

@files = `ls -l`;

foreach $file (@files){
   print $file;
}

1;
```

When the above code is executed, it lists down all the files and directories available in the current directory −

```
drwxr-xr-x 3 root root 4096 Sep 14 06:46 9-14
drwxr-xr-x 4 root root 4096 Sep 13 07:54 android
-rw-r--r-- 1 root root  574 Sep 17 15:16 index.htm
drwxr-xr-x 3  544  401 4096 Jul  6 16:49 MIME-Lite-3.01
-rw-r--r-- 1 root root   71 Sep 17 15:16 test.pl
drwx------ 2 root root 4096 Sep 17 15:11 vAtrJdy
```

## The system Function

You can also use **system** function to execute any Unix command, whose output will go to the output of the perl script. By default, it is the screen, i.e., STDOUT, but you can redirect it to any file by using redirection operator > −

```
#!/usr/bin/perl

system( "ls -l")

1;
```

When above code is executed, it lists down all the files and directories available in the current directory −

```
drwxr-xr-x 3 root root 4096 Sep 14 06:46 9-14
drwxr-xr-x 4 root root 4096 Sep 13 07:54 android
-rw-r--r-- 1 root root  574 Sep 17 15:16 index.htm
drwxr-xr-x 3  544  401 4096 Jul  6 16:49 MIME-Lite-3.01
```

```
-rw-r--r-- 1 root root   71 Sep 17 15:16 test.pl
drwx------ 2 root root 4096 Sep 17 15:11 vAtrJdy
```

Be careful when your command contains shell environmental variables like *PATHor*HOME. Try following three scenarios −

```
#!/usr/bin/perl

$PATH = "I am Perl Variable";

system('echo $PATH');  # Treats $PATH as shell variable
system("echo $PATH");  # Treats $PATH as Perl variable
system("echo \$PATH"); # Escaping $ works.

1;
```

When above code is executed, it produces the following result depending on what is set in shell variable $PATH.

```
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin
I am Perl Variable
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin
```

## The fork Function

Perl provides a **fork** function that corresponds to the Unix system call of the same name. On most Unix-like platforms where the fork system call is available, Perl's fork simply calls it. On some platforms such as Windows where the fork system call is not available, Perl can be built to emulate fork at the interpreter level.

The fork function is used to clone a current process. This call create a new process running the same program at the same point. It returns the child pid to the parent process, 0 to the child process, or undef if the fork is unsuccessful.

You can use **exec** function within a process to launch the requested executable, which will be executed in a separate process area and exec will wait for it to complete before exiting with the same exit status as that process.

```
#!/usr/bin/perl

if(!defined($pid = fork())) {
   # fork returned undef, so unsuccessful
   die "Cannot fork a child: $!";
}elsif ($pid == 0) {
   print "Printed by child process\n";
   exec("date") || die "can't exec date: $!";

} else {
   # fork returned 0 nor undef
   # so this branch is parent
   print "Printed by parent process\n";
   $ret = waitpid($pid, 0);
   print "Completed process id: $ret\n";

}

1;
```

When above code is executed, it produces the following result −

```
Printed by parent process
Printed by child process
Tue Sep 17 15:41:08 CDT 2013
Completed process id: 17777
```

The **wait** and **waitpid** can be passed as a pseudo-process ID returned by fork. These calls will properly wait for the termination of the pseudo-process and return its status. If you fork without ever waiting on your children using **waitpid** function, you will accumulate zombies. On Unix systems, you can avoid this by setting $SIG{CHLD} to "IGNORE" as follows −

```
#!/usr/bin/perl

local $SIG{CHLD} = "IGNORE";

if(!defined($pid = fork())) {
   # fork returned undef, so unsuccessful
   die "Cannot fork a child: $!";
}elsif ($pid == 0) {
   print "Printed by child process\n";
   exec("date") || die "can't exec date: $!";

} else {
   # fork returned 0 nor undef
   # so this branch is parent
   print "Printed by parent process\n";
   $ret = waitpid($pid, 0);
   print "Completed process id: $ret\n";

}

1;
```

When above code is executed, it produces the following result −

```
Printed by parent process
Printed by child process
Tue Sep 17 15:44:07 CDT 2013
Completed process id: -1
```

## The kill Function

Perl **kill**$\boldsymbol{'KILL', (ProcessList)}$ function can be used to terminate a pseudo-process by passing it the ID returned by fork.

Note that using kill$'KILL', (ProcessList)$ on a pseudo-process may typically cause memory leaks, because the thread that implements the pseudo-process does not get a chance to clean up its resources.

You can use **kill** function to send any other signal to target processes, for example following will send SIGINT to a process IDs 104 and 102 −

```
#!/usr/bin/perl

kill('INT', 104, 102);

1;
```

Loading [MathJax]/jax/output/HTML-CSS/jax.js