
UNIX PROCESSES

PROCESS LIFECYCLE, CONTROL AND SIGNALS

By

0xLeo (github.com/0xleo)

JUNE 13, 2019

DRAFT 0.7

MISSING: SIGNALS, FORK BOMB

Contents

1 Processes in Linux	2
1.1 Process ID and parent ID	2
1.2 Creating processes	3
1.2.1 The fork and exec method	3
1.2.2 The fork() system call	3
1.2.3 fork() code and example	4
1.2.4 (Optional) More fork() examples	5
1.3 The exec call	7
1.3.1 The exec family	7
1.3.2 exec basic examples	8
1.3.3 The fork-exec pattern	10
1.4 Controlling processes with signals	11
1.5 Waiting for process termination	11
1.5.1 Why wait for children to complete?	11
1.5.2 Creating and observing a zombie process	12
1.5.3 The wait system call	13
1.5.4 The wait family of functions	13
1.5.5 Practical examples of wait() call	14

1 Processes in Linux

1.1 Process ID and parent ID

DEFINITION 1.1 (pid). Each process in a Linux system is identified by its unique process ID, sometimes referred to as pid. The pid does not change during the lifetime of a process [1].

Process IDs are 32-bit or 64-bit integers that are assigned sequentially by Linux as new processes are created. Each process own a unique pid.

Every process also has a parent process except the special init process. init is the grandfather of all processes on the system because all other processes run under it. Every process can be traced back to init, and it always has a pid of 1. The kernel itself has a pid of 0. A process tree is shown below. Modern Linux systems have replaced init process with systemd (System Management Daemon) [2].

The init process.

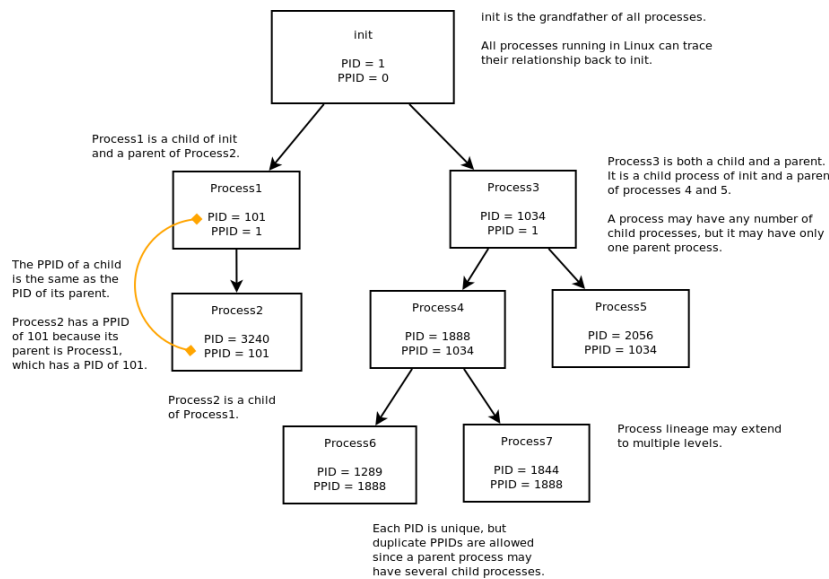


Fig. 1. An example process tree with init as the parent [2].

Thus, you can think of the processes on a Linux system as arranged in a *tree*, with the `init` process at its root. The parent process ID, or `ppid`, is simply the process ID of the process's parent. When referring to process IDs in a C or C++ program, always use the `pid_t` typedef, which is defined in `<sys/types.h>`. A program can obtain the process ID of the process it's running in with the `getpid()` system call, and it can obtain the process ID of its parent process with the `getppid()` system call.

Listing 1: Get process and parent ID on Linux (src/get_pid.c).

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     printf("The process ID is: %d\n", getpid());
7     printf("The parent process ID is: %d\n", getppid());
8     return 0;
9 }
```

If we run the program multiple times under the same shell, the with the pid will change but the ppid will remain the same as it will be the shell's (parent) pid.

Processing in Linux can be viewed during using the `ps` command and killed using the `kill` or `killall`. See manual for more details.

1.2 Creating processes

1.2.1 The fork and exec method

The preferred method to create process in Unix is the fork and exec. This method takes more than one step. fork makes a child process that is *almost* an exact copy of its parent. The new process gets a different PID and has as its PPID the PID of the process that created it.

Because the two processes are now running exactly the same code, they can tell which is which by the return code of fork - the child gets 0, the parent gets the PID of the child. Then using one of the exec family calls, the current process is replaced with a new program. exec loads the program into the current process space and runs it from the entry point [3].

The two calls are not required to be used together. It's perfectly acceptable for a program to fork itself without execing if, for example, the program contains both parent and child code [3].

Child and parent are distinguished by the return value of fork().

1.2.2 The fork() system call

Fork system call creates a new process, called *child process*, which is a copy of the current process (except of their return values) and runs *concurrently* with it. The program starts running concurrently at the code location where fork was called. The parent process waits for the child to finish and then resumes.

fork() "returns twice"

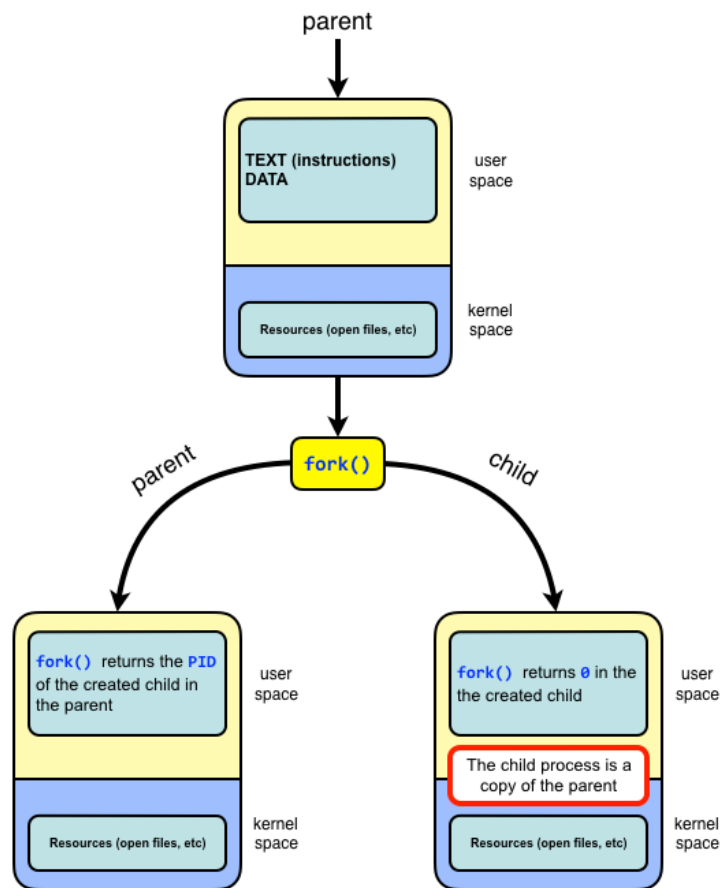
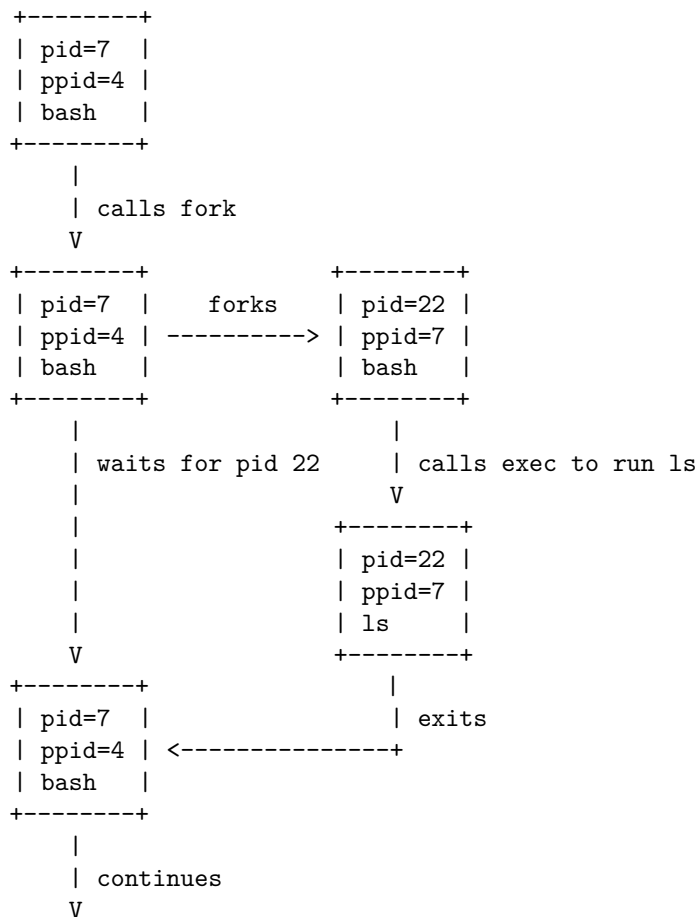


Fig. 2. How fork works in user space and kernel space.

The following diagram illustrates the typical fork/exec operation where the bash shell is used to list a directory with the ls command [3]:



1.2.3 fork() code and example

The fork system call does not take an argument. As previously mentioned, the process that invokes the fork() is known as the parent and the new process is called the child. It returns an integer.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

To differentiate whether its called for the parent or child, it returns different values for each, particularly:

1. fork_return == -1: fork() failed and there is no child.
2. fork_return == 0 returned to the newly created child process.
3. fork_return > 0: returned to parent or caller. The value contains process ID of newly created child process

Fork's return indicates failure or not.

EXAMPLE 1.1. Predict the output of the following snippet of code.

Listing 2: How process IDs are assigned to child and parent (src/fork_child_parent.c).

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main(int argc, char *argv[])
6 {
7     pid_t child_pid;
8     printf("The main program process ID is %d\n", (int) getpid());
9
10    child_pid = fork();
11    if (child_pid != 0){
12        printf("This is the parent process, with id %d\n", (int) getpid());

```

```

13     printf("The child's process ID is %d\n", (int) child_pid);
14 }
15 else
16     printf("This is the child process, with ID %d\n", (int) getpid());
17
18     return 0;
19 }

```

The output is:

```

The main program process ID is 5000
This is the parent process, with id 5000
The child's process ID is 5001
This is the child process, with ID 5001

```

1.2.4 (Optional) More fork() examples

Predict the output of the following examples.

EXAMPLE 1.2.

Listing 3: Forked hello world (src/fork_hello.c).

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main() {
6     printf("Hello world!\n");
7     // make two process which run same
8     // program after this instruction
9     fork();
10
11     printf("Bye world!\n");
12     return 0;
13 }

```

```

Hello world!
Bye world!
Bye world!

```

EXAMPLE 1.3.

Listing 4: Forked hello world multiple times (src/fork_hellox3.c).

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main() {
6     // Each fork runs recursively the rest of the program
7     fork();
8     fork();
9     fork();
10
11     printf("Hello world!\n");
12     return 0;
13 }

```

```

Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!

```

Hello world!

EXAMPLE 1.4.

Listing 5: Accessing the same variable with two processes (src/fork_values2.c).

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 void fork_values(){
6     // forked process copies the same virtual address for vars
7     // BUT different physical address
8     int x = 1;
9     printf("Initially, x = %d at address 0x%x\n",
10           x, &x);
11     pid_t fork_ret = fork();
12     if (fork_ret == 0)
13         printf("Child has x = %d at 0x%x\n",
14               ++x, &x);
15     else
16         printf("Parent has x = %d at 0x%x\n",
17               --x, &x);
18 }
19
20 int main()
21 {
22     fork_values();
23     return 0;
24 }
```

Initially, x = 1 at address 0xa7d99060

Parent has x = 0 at 0xa7d99060

Child has x = 2 at 0xa7d99060

EXAMPLE 1.5.

Listing 6: Forked condition (src/fork_tree.c).

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     if (fork() || fork())
7         fork();
8     printf("1 ");
9     return 0;
10 }
```

Fig. 3 illustrates the execution flow. The first `fork()` in the `if` statement creates one parent (return value positive) and a child (return 0). For the parent, the OR is true, so it directly enters the block, where it forks again. The child needs to evaluate the second `fork` and enters the block only for the parent as only the parent satisfies the OR. The final printed output is

1 1 1 1 1

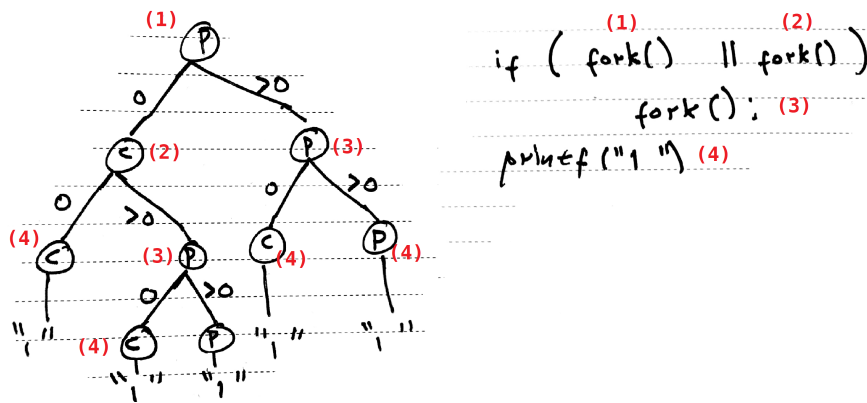


Fig. 3. Solution to forked condition.

1.3 The exec call

1.3.1 The exec family

The exec functions replace the program running in a process with another program. When a program calls an exec function, that process immediately ceases executing that program and begins executing a new program from the beginning, assuming that the exec call doesn't encounter an error.

There are several variations of the exec function. The name base is exec followed by one of the following letters.

- `int execl (const char *path, const char *arg, ...);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execl(const char *path, const char *arg, ..., char *const envp[]);`
- `int execv (const char *path, char *const argv[]);`
- `int execvp(const char *file, char *const argv[]);`
- `int execve(const char *file, char *const argv[], char *const envp[]);`

All definitions are found in `<unistd.h>`. The `exec()` functions only return if an error has occurred. The return value is `-1`, and `errno` is set to indicate the error. Each system call is the word `exec` followed by either `l` or `v` and then possibly followed by either `e` or `p`. If `exec` is followed by

`exec()` returns `(-1)` only if it fails.

- `l`, then it expects the arguments as a NULL-terminated *list*, e.g. `path, arg0, arg1, ..., argn`. The first argument (`path` or `file`) describes the program path or file (depending on the `p` option). The second, `arg0` described how the newly spawned process should be named. For example, if `path = "/bin/ls"`, it would make sense to have `arg0 = "ls"` but we could also set `arg0 = "potato"`. `arg1, arg2, ..., argn` are in the form of pointers to `char`. They are the command line options to the program, e.g. in this case they could be `"-l", "/"`. The last argument must always be `NULL ((char*) 0)`.
- `v`, it is similar but expects the arguments as an *array* of `char*` (vector). Again, the first argument is the path to the program to execute or the file (depending on the `p` option) and the second the array of `char*`. In this case, if we wanted to execute the same `ls -l /` command, we would pass `{"/bin/ls", "ls", "-l", "/", NULL}` to `execv`.

`l`: list

`v`: vector

If it is further followed by

- `e`, then it expects `char* envp` after the NULL-terminated arguments to the program. `envp` points to an array of pointers that in turn point to strings that define environment variables. These strings usually have the form: `ENVVAR=value` where `ENVVAR` is the name of the environment variable and `value` is the string value to set it to. The `envp` array is *also* terminated by a `NULL` pointer. If `envp` is `NULL`, then the child process acquires the environment of the calling process. For example, we could have

`e`: environ
ment


```
char * environment [4];
environment [0]="SHELL=/bin/csh";
environment [1]="LOGNAME=heino";
environment [2]="OSTYPE=LiNuX";
environment [3]=NULL;
```

- p, it will search for the file using the current environment variable PATH, which usually includes p: PATH /bin/, /usr/bin/, etc.

Their differences are summarised in the table below.

Function	<i>pathname</i>	<i>filename</i>	Arg list	<i>argv[]</i>	<i>environ</i>	<i>envp[]</i>
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(letter in name)		p	l	v		e

Fig. 4. The 6 exec functions

1.3.2 exec basic examples

Some basic examples that show the usage of the exec family are listed below.

execl.c

```
1 #include <unistd.h>    // exec*
2
3 int main () {
4     /* Executes ls -l / */
5     char* cmd = "/bin/ls"; // executable
6     char* argv0 = "ls";    // name to use
7     char* argv1 = "-l";    // cmd arg
8     char* argv2 = "/";     // cmd arg
9     char* argv3 = NULL;    // terminator
10
11     return execl (cmd, argv0, argv1,
12                  argv2, NULL);
13 }
```

execlp.c

```
1 #include <unistd.h>    // exec*
2
3 int main () {
4     /* Executes ls -l / */
5     char* cmd = "ls";     // executable
6     char* argv0 = "ls";   // name to use
7     char* argv1 = "-l";   // cmd arg
8     char* argv2 = "/";    // cmd arg
9     char* argv3 = NULL;   // terminator
10
11     return execlp (cmd, argv0, argv1,
12                   argv2, NULL);
13 }
```

execle.c

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 int main() {
5     /* Executes env given some arguments
6     in envp */
7     char* fpath = "/bin/sh";
8     char* arg[] = { fpath, "-c", "env",
9                     NULL };
10
11     char* envp[] =
12     {
13         "HOME=/",
14         "PATH=/bin:/usr/bin",
15         "USER=leo",
16         "TERM=xterm",
17         NULL
18     };
19     execl(fpath, arg[0], arg[1],
20          arg[2], NULL, envp);
21     fprintf(stderr, "Oops!\n");
22     return -1;
23 }
```

execv.c

```
1 #include <unistd.h>    // exec*
2
3 int main () {
4     /* Executes ls -l / */
5     char* cmd = "/bin/ls";
6     char* argv[4];
7     argv[0] = "ls";
8     argv[1] = "-l";
9     argv[2] = "/";
10    argv[3] = NULL;
11
12    return execv(cmd, argv);
13 }
```

execvp.c

```
1 #include <unistd.h>    // exec*
2
3 int main () {
4     /* Executes ls -l / */
5     char* cmd = "ls";
6     char* argv[4];
7     argv[0] = "ls";
8     argv[1] = "-l";
9     argv[2] = "/";
10    argv[3] = NULL;
11
12    return execvp (cmd, argv);
13 }
```

execve.c

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 int main() {
5     /* Executes env given some arguments
6     in envp */
7     char* fpath = "/bin/sh";
8     char* arg[] = { fpath, "-c", "env",
9                     NULL };
10
11     char* envp[] =
12     {
13         "HOME=/",
14         "PATH=/bin:/usr/bin",
15         "USER=leo",
16         "TERM=xterm",
17         NULL
18     };
19     execve(fpath, arg, envp);
20     fprintf(stderr, "Oops!\n");
21     return -1;
22 }
```

In the exece examples, we do not expect to hit the “Oops!” message as the program image will be replaced with the one called by exece unless an error occurs. Also, the system will append some additional environment variables to the defined ones, so the output looks something like:

```
PWD=/tmp
HOME=/
TERM=xterm
USER=leo
SHLVL=0
PATH=/bin:/usr/bin
_=/bin/env
```

The bug below demonstrates why it's always good to check if exec executed without errors.

EXAMPLE 1.6. *What's wrong with this code?*

Listing 7: How many processes are created? (src/exec_fork_bug.c).

```
1 #include <unistd.h>
2 #define HELLO_NUMBER 10
3
4 int main(){
5     pid_t children[HELLO_NUMBER];
6     int i;
7     for(i = 0; i < HELLO_NUMBER; i++){
8         pid_t child = fork();
9         if(child == -1){
10             break;
11         }
12         if(child == 0){ //I am the child
13             execlp("ehco", "echo", "hello", NULL);
14         }
15         else{
16             children[i] = child;
17         }
18     }
19
20     int j;
21     for(j = 0; j < i; j++){
22         waitpid(children[j], NULL, 0);
23     }
24     return 0;
25 }
```

We misspelled ehco, so we can't exec it. The first time, instead of being replaced with another program, the child will continue and get forked in 2. Then the two children will do the same and get forked in 4,...,...2¹⁰ processes, "fork bombing" out machine. How could we prevent this? Put an exit right after exec so in case exec fails we won't end up fork bombing our machine.

It's a good practice to exit after exec in case it fails.

1.3.3 The fork-exec pattern

A common pattern to run a subprogram within a program is first to fork the process and then exec the subprogram. This allows the calling program to continue execution in the parent process while the calling program is replaced by the subprogram in the child process

The code below demonstrates how to replace the child with another program which runs the `ls -l /` command.

Listing 8: Using fork followed by exec (src/fork-exec.c).

```
1 /* from Advanced Linux Programming (page 51) */
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6
7 /* Spawn a child process running a new program. PROGRAM is the name
8    of the program to run; the path will be searched for this program.
9    ARG_LIST is a NULL-terminated list of character strings to be
10    passed as the program's argument list. Returns the process ID of
11    the spawned process. */
12 int spawn(char* program, char** arg_list) {
13     pid_t child_pid;
14
15     /* Duplicate this process. */
16     child_pid = fork ();
17
18     if (child_pid != 0){
```

```

19  /* This is the parent process. */
20  return child_pid;
21 }else {
22     /* Now execute PROGRAM, searching for it in the path. */
23     execvp (program, arg_list);
24     /* The execvp function returns only if an error occurs. */
25     fprintf (stderr, "an error occurred in execvp\n");
26     abort ();
27 }
28 }
29
30 int main () {
31     /* The argument list to pass to the "ls" command. */
32     char* arg_list[] = {
33         "ls",          /* argv[0], the name of the program. */
34         "-l",
35         "/",
36         NULL          /* The argument list must end with a NULL. */
37     };
38
39     /* Spawn a child process running the "ls" command. Ignore the
40        returned child process ID. */
41     spawn ("ls", arg_list);
42     printf ("done with main program\n");
43     return 0;
44 }

```

The output of this looks something like:

```

done with main program
total 52
lrwxrwxrwx   1 root root    7 Dec  6 2018 bin -> usr/bin
drwxr-xr-x   3 root root 4096 Apr  4 22:30 boot
drwxr-xr-x  21 root root 3580 Jun  5 17:53 dev
<-- omitted -->

```

Notice that the output of ls (child) appears *after* the parent has terminated.

1.4 Controlling processes with signals

TODO!

1.5 Waiting for process termination

1.5.1 Why wait for children to complete?

If we run the code in Listing 8, it is not certain whether the child or parent will finish first and in general, this cannot be predicted when we use the fork-exec model. That's because the child process, in which ls is run, is scheduled independently of the parent process.

In some situations, though, it is desirable for the parent process to wait until one or more child processes have completed. This can be done with the wait family of system calls. These functions allow you to wait for a process to finish executing, and enable the parent process to retrieve information about its child's termination.

Why use the wait functions?

Another reason why wait should be used by the parent is because When a process ends via exit or return, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains. When a process terminates, the kernel asynchronously sends a SIGCHLD signal (software interrupt) to the parent. The parent can read the child's exit status by executing the wait system call, which reads the SIGCHLD signal. Parent, on receipt of SIGCHLD reaps the status of the child from the process table.

When we use the fork-exec model and a child process runs, Unix allocates it memory (for opened files etc.) and a PID in the process table. When the child terminates, the system flushes and closes all opened files

and data structures and sends an asynchronous SIGCHLD signal to the parent but does not remove the child's PID entry from the process table. Without using wait(), the parent fails to catch the SIGCHLD hence does not remove the terminated child's PID from the process table. That PID corresponds to a dead process (zombie process). In the shell, zombie processes can be listed with the ps Z command.

Zombie processes shouldn't exist because the amount of kernel memory although they take insignificant memory, they reserve (dead) PID entries. A way to kill a zombie process is by killing its parent.

On the other hand, if the parent terminates while its children are running, then its "zombie" children (if any) are adopted by init. init automatically performs a wait to remove the zombies. This type of zombie processes are called orphans.

The figure below shows how a healthy fork-exec-wait cycle should look like.

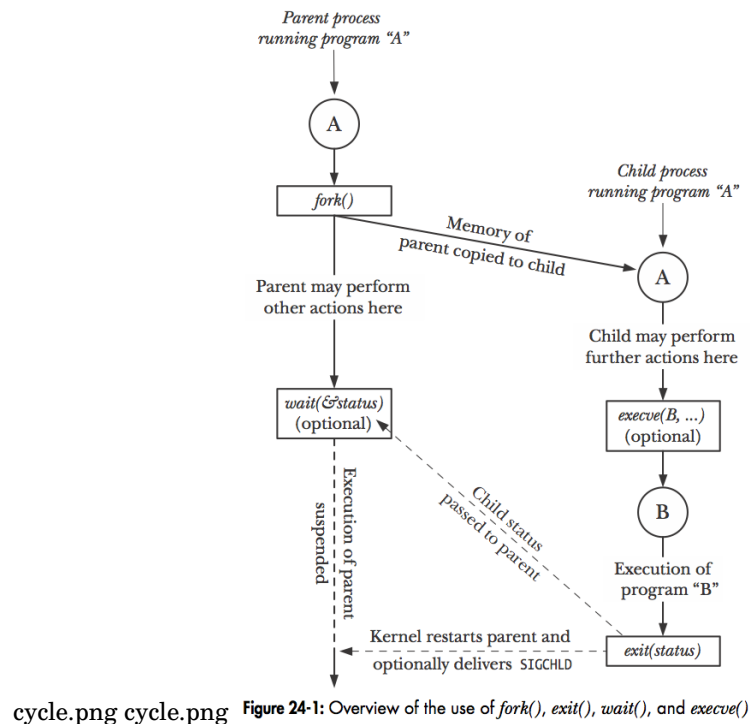


Fig. 5. fork-exec-wait model overview.

1.5.2 Creating and observing a zombie process

In Listing 2, the parent doesn't call wait. As a result, a zombie child process is created for a brief amount of time from the moment the child terminates till the parent returns. To observe, add the header

```
#include <wait.h>
```

add the line

```
sleep(100);
```

after the if-else statement. Then for the next 100 seconds, a zombie will be created. Run the program and in another terminal get the child's PID:

```
<-- omitted -->
```

This is the child process, with ID 8630

Run ps Z in another terminal to list the zombie processes and 8630 should be in them:

```
-                8630 pts/0    S+      0:00 ./a.out
```

1.5.3 The wait system call

The `wait()` system call is used by a parent process to wait for the status of the child to change. `wait()` suspends the execution of the calling process until one of its child processes exits or a signal is received. A child process may terminate due to any of these:

- It calls `exit()`;
- It returns (an `int`) from `main`
- It receives a signal (from the OS or another process) whose default action is to terminate.

1.5.4 The wait family of functions

There are the `wait`, `waitpid`, `waitid`, `wait3`, `wait4` in the wait family, but we will only concern ourselves with the first two:

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

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

Regarding the arguments, the values they can take are for `pid_t pid`

`wait` and
`waitpid`
calls

- `< -1`, meaning wait for any child process whose process group ID is equal to the absolute value of `pid`.
- `-1`, meaning wait for any child process.
- `0`, meaning wait for any child process whose process group ID is equal to that of the calling process.
- `> 0`, meaning wait for the child whose process ID is equal to the value of `pid`.

`int *wstatus`:

- Points to a location where `waitpid()` can store a status value. This status value is zero if the child process explicitly returns zero status. Otherwise, it is a value that can be analysed with the certain *status analysis macros* described later.

`int options`:

- `WNOHANG`: Return immediately if no child has exited.
- `WUNTRACED`: Also return if a child has stopped. Status for traced children which have stopped is provided even if this option is not specified.
- `WCONTINUED`: Also return if a stopped child has been resumed by delivery of `SIGCONT`.

Return value (same for both):

- process ID if OK,
- if non-blocking option and no zombies around,
- or `-1` on error.

If an error
had occurred
after waiting,
we know it
from the
return value

As mentioned before the return status of the child in `waitpid` is stored in `int& wstatus` and does not directly provide useful information but can be analysed with the `WIF*` macros. The top row shows what the macro means if it evaluates to true and the bottom how its exit code is interpreted.

1. `WIFEXITED(status) == true` then the child exited normally and `WEXITSTATUS(status)` is the return code when child exits.
2. `WIFSIGNALED(status) == true` then the child exited because a signal was not caught and `WTERMSIG(status)` gives the number of the terminating signal.
3. `WIFSTOPPED(status) == true` then the child is stopped and `WSTOPSIG(status)` is gives the number of the stop signal

To summarise, `waitpid` is more flexible as it allows a process to be made non-blocking via options and can select which child process to wait via `pid`.

1.5.5 Practical examples of wait() call

EXAMPLE 1.7.

Listing 9: Waiting for child and getting its status (src/fork_exec_wait.c).

```
1 /* from Advanced Linux Programming (page 57) */
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6 #include <sys/wait.h>
7
8 /* Spawn a child process running a new program. PROGRAM is the name
9    of the program to run; the path will be searched for this program.
10   ARG_LIST is a NULL-terminated list of character strings to be
11   passed as the program's argument list. Returns the process ID of
12   the spawned process. */
13 int spawn (char* program, char** arg_list) {
14     pid_t child_pid;
15
16     /* Duplicate this process. */
17     child_pid = fork ();
18
19     if (child_pid != 0){
20         /* This is the parent process. */
21         return child_pid;
22     }else {
23         /* Now execute PROGRAM, searching for it in the path. */
24         execvp (program, arg_list);
25         /* The execvp function returns only if an error occurs. */
26         fprintf (stderr, "an error occurred in execvp\n");
27         abort ();
28     }
29 }
30
31 int main () {
32     int child_status;
33
34     /* The argument list to pass to the "ls" command. */
35     char* arg_list[] = {
36         "ls",          /* argv[0], the name of the program. */
37         "-l",
38         "/",
39         NULL           /* The argument list must end with a NULL. */
40     };
41
42     /* Spawn a child process running the "ls" command. Ignore the
43        returned child process ID. */
44     spawn ("ls", arg_list);
45     /* Wait for the child process to complete. */
46     wait (&child_status);
47     if (WIFEXITED (child_status))
48         printf ("the child process exited normally, with exit code %d\n",
49                WEXITSTATUS (child_status));
50     else
51         printf ("the child process exited abnormally\n");
52     return 0;
53 }
```

Output:

```
total 2080564
drwxr-xr-x  2 root root      4096 Mai 13 11:48 bin
drwxr-xr-x  3 root root      4096 Mai 23 10:06 boot
```

```
drwxrwxr-x  2 root root      4096 Aug 30  2018 cdrom
drwxr-xr-x 18 root root      3960 Jun 13 09:06 dev
<-- omitted -->
the child process exited normally, with exit code 0
```

EXAMPLE 1.8.

Listing 10: Status code of multiple children (src/wait_pid_array.c).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/wait.h>
4 #include <unistd.h>
5
6 void waitexample()
7 {
8     int i, stat;
9     pid_t pid[5];
10    for (i=0; i<5; i++)
11    {
12        if ((pid[i] = fork()) == 0)
13        {
14            sleep(1);
15            exit(100 + i);
16        }
17    }
18
19    // Using waitpid() and printing exit status
20    // of children.
21    for (i=0; i<5; i++)
22    {
23        pid_t cpid = waitpid(pid[i], &stat, 0);
24        if (WIFEXITED(stat))
25            printf("Child %d terminated with status: %d\n",
26                  cpid, WEXITSTATUS(stat));
27    }
28 }
29
30 // Driver code
31 int main()
32 {
33     waitexample();
34     return 0;
35 }
```

Output:

```
Child 5315 terminated with status: 100
Child 5316 terminated with status: 101
Child 5317 terminated with status: 102
Child 5318 terminated with status: 103
Child 5319 terminated with status: 104
```

EXAMPLE 1.9.

Listing 11: Status code of multiple children (src/fork_sync.c).

```
1 #include <stdio.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5
6 void fork9() {
7     int child_status;
8     if (fork() == 0) {
9         printf("HC: hello from child\n");
```



```

10     }
11     else {
12         printf("HP: hello from parent\n");
13         wait(&child_status);
14         printf("CT: child has terminated\n");
15     }
16     printf("Bye\n");
17     exit(0);
18 }
19
20 int main()
21 {
22     fork9();
23     return 0;
24 }

```

Output:

```

HP: hello from parent
HC: hello from child
Bye
CT: child has terminated
Bye

```

In this case, when the parent sees the `wait()`, it pauses its execution and wait's for the child's if branch to finish. The following diagram explains the flow.

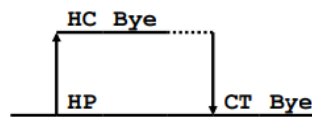


Fig. 6. Synchronising child and parent in the example.

References

- [1] C. LLC, *Advanced Linux Programming*. Sams Publishing, Jun. 2001, ISBN: 0735710430. [Online]. Available: <https://www.xarg.org/ref/a/0735710430/>.
- [2] *What are pid and ppid?* <https://delightfullylinux.wordpress.com/2012/06/25/what-is-pid-and-ppid/>, Accessed: 2019-02-06.
- [3] paxdiablo, *Differences between fork and exec*, <https://stackoverflow.com/a/1653415>.