UNIX PROCESSES

PROCESS LIFECYCLE, CONTROL AND SIGNALS

By

0xLeo (github.com/0xleo)

June 22, 2019

DRAFT 0.9

Missing: reentrant functions and more on signal handlers.

Contents

1 Creating a process – the `fork()` call

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.

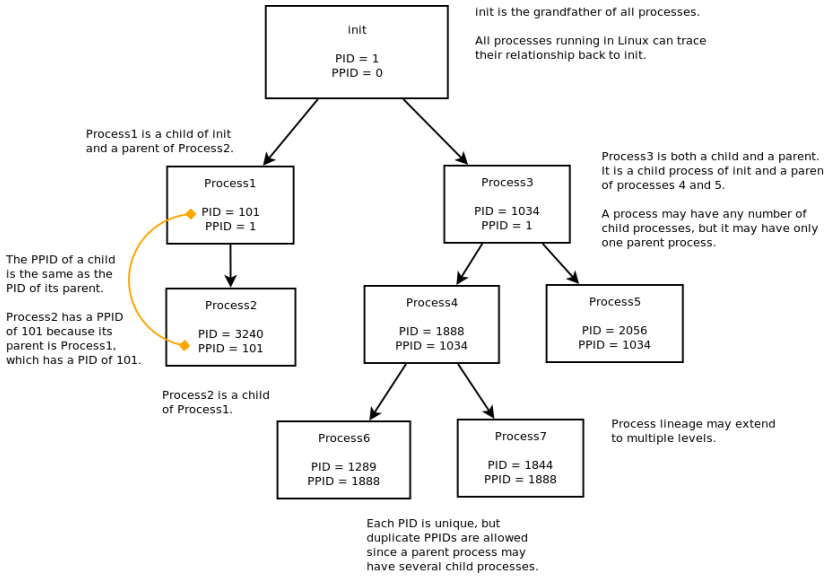


Fig. 1. An example process tree with init as the parent [2].

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).

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("The process ID is: %d\n", getpid());
    printf("The parent process ID is: %d\n", getppid());
    return 0;
}
```

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 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 it replaced with a new program. `exec` loads the program into the current process space and runs it from the entry point [3].

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

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].

1.3 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"

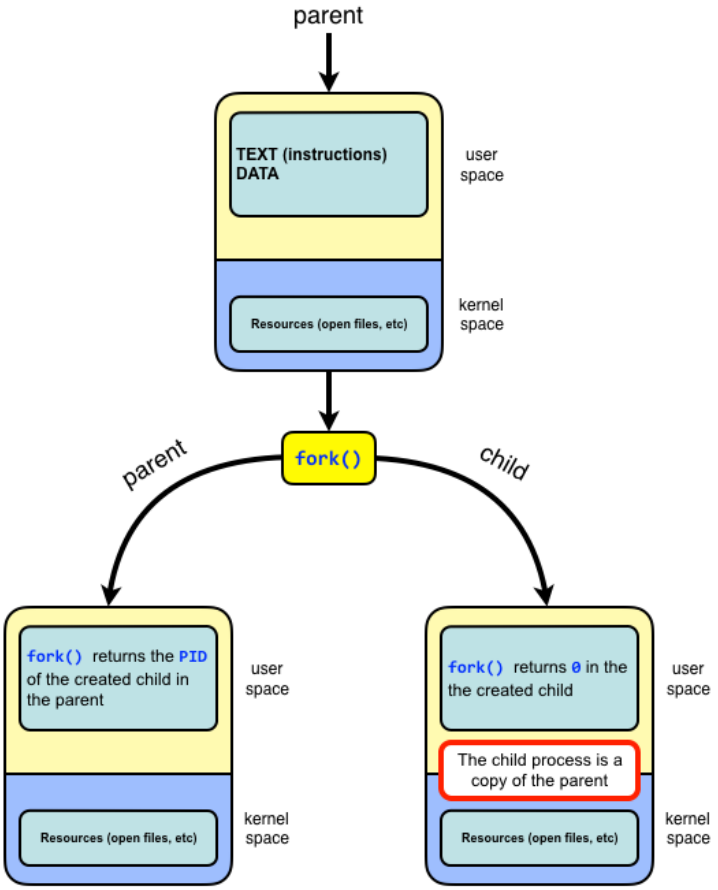
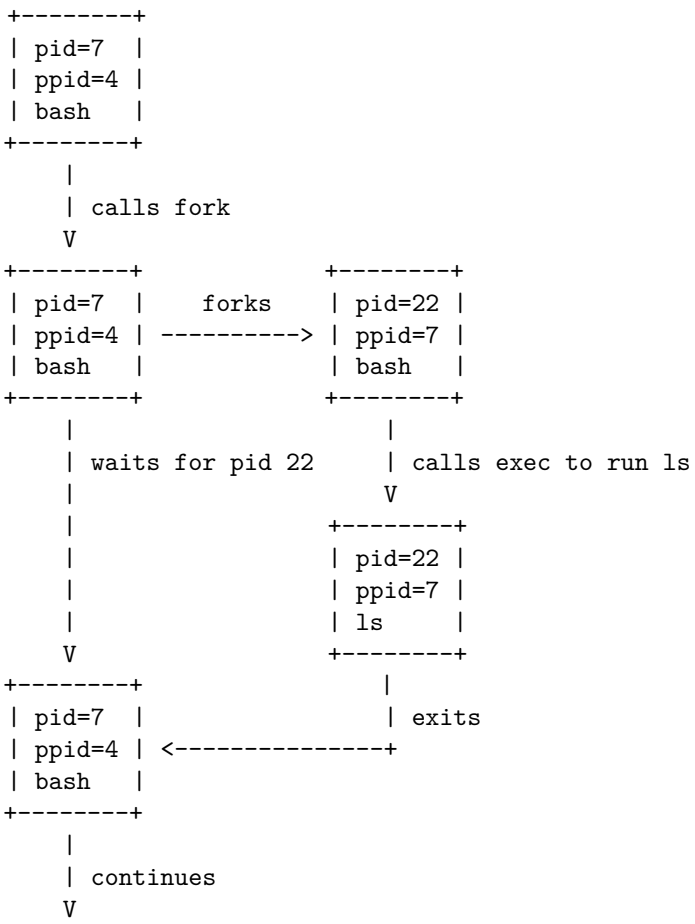


Fig. 2. How fork works in user space and kernel space.

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



1.4 `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 [4]:

Fork's return indicates failure or not.

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

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).

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    pid_t child_pid;
    printf("The main program process ID is %d\n", (int) getpid());

    child_pid = fork();
```

```
    if (child_pid != 0){
        printf("This is the parent process,  with id %d\n", (int) getpid());
        printf("The child's process ID is %d\n", (int) child_pid);
    }
    else
        printf("This is the child process,  with ID %d\n", (int) getpid());

    return 0;
}
```

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
```

1.5 (Optional) More fork() examples

Predict the output of the following examples.

EXAMPLE 1.2.

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

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    printf("Hello world!\n");
    // make two process which run same
    // program after this instruction
    fork();

    printf("Bye world!\n");
    return 0;
}
```

```
Hello world!
Bye world!
Bye world!
```

EXAMPLE 1.3. [4]

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

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    // Each fork runs recusrively the rest of the program
    fork();
    fork();
    fork();

    printf("Hello world!\n");
    return 0;
}
```

```
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
```

EXAMPLE 1.4. *In this example, although the virtual address of fork() is the same for the parent and child, it maps to different physical addresses therefore parent and child hold different "instances" of the variable* `x`.

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

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void fork_values(){
    // forked process copies the same virtual address for vars
    // BUT different physical address
    int x = 1;
    printf("Initially, x = %d at address 0x%x\n",
            x, &x);
    pid_t fork_ret = fork();
    if (fork_ret == 0)
        printf("Child has x = %d at 0x%x\n",
                ++x, &x);
    else
        printf("Parent has x = %d at 0x%x\n",
                --x, &x);
}

int main()
{
    fork_values();
    return 0;
}
```

```
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).

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    if (fork() || fork())
        fork();
    printf("1 ");
    return 0;
}
```

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
```

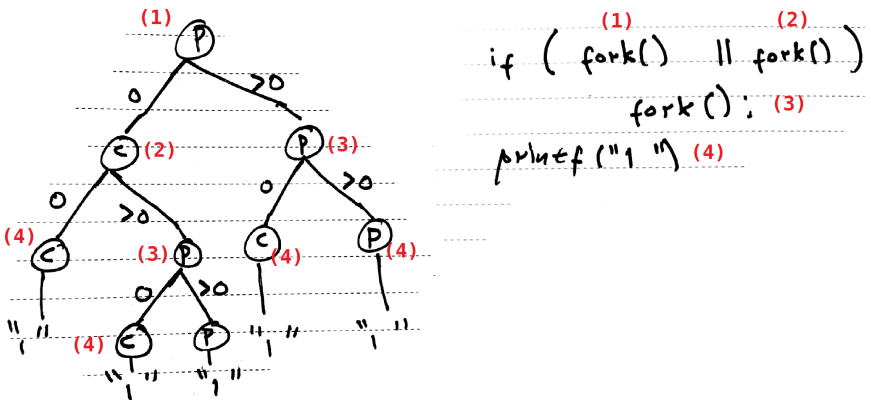


Fig. 3. Solution to forked condition.

2 Executing a process – the `exec()` call

2.1 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 [1].

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 execle( 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, ..., argm 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`). `l`: list
- `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`. `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`: environment

```
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 /bin/, /usr/bin/, etc.

p: PATH

Their differences are summarised in the table below.

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

Fig. 4. The 6 exec functions

2.2 exec basic examples

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

execl.c

```c
#include <unistd.h>     // exec*

int main () {
    /* Executes ls -l / */
    char* cmd = "/bin/ls";// executable
    char* argv0 = "ls";    // name to use
    char* argv1 = "-l";    // cmd arg
    char* argv2 = "/";     // cmd arg
    char* argv3 = NULL;    // terminator

    return execl (cmd, argv0,  argv1,
    argv2, NULL);
}
```

execv.c

```c
#include <unistd.h>     // exec*

int main () {
    /* Executes ls -l / */
    char* cmd = "/bin/ls";
    char* argv[4];
    argv[0] = "ls";
    argv[1] = "-l";
    argv[2] = "/";
    argv[3] = NULL;

    return execv(cmd,  argv);
}
```

execlp.c

```c
#include <unistd.h>     // exec*

int main () {
    /* Executes ls -l / */
    char* cmd = "ls";      // executable
    char* argv0 = "ls";    // name to use
    char* argv1 = "-l";    // cmd arg
    char* argv2 = "/";     // cmd arg
    char* argv3 = NULL;    // terminator

    return execlp (cmd, argv0,  argv1,
    argv2, NULL);
}
```

execvp.c

```c
#include <unistd.h>     // exec*

int main () {
    /* Executes ls -l / */
    char* cmd = "ls";
    char* argv[4];
    argv[0] = "ls";
    argv[1] = "-l";
    argv[2] = "/";
    argv[3] = NULL;

    return execvp (cmd,  argv);
}
```

execle.c

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    /* Executes env given some arguments
    in  envp */
    char* fpath = "/bin/sh";
    char* arg[] = { fpath, "-c", "env",
                    NULL };
    char* envp[] =
    {
        "HOME=/",
        "PATH=/bin:/usr/bin",
        "USER=leo",
        "TERM=xterm",
        NULL
    };
    execle(fpath, arg[0], arg[1],
        arg[2], NULL, envp);
    fprintf(stderr, "Oops!\n");
    return -1;
}
```

execve.c

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    /* Executes env given some arguments
    in  envp */
    char* fpath = "/bin/sh";
    char* arg[] = { fpath, "-c", "env",
                    NULL };
    char* envp[] =
    {
        "HOME=/",
        "PATH=/bin:/usr/bin",
        "USER=leo",
        "TERM=xterm",
        NULL
    };
    execve(fpath, arg, envp);
    fprintf(stderr, "Oops!\n");
    return -1;
}
```

In the `exece` examples, we do not expect to hit the "Oops!" message as the program image will be replace 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
```

2.3 A commot `exec` pitfall

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

EXAMPLE 2.1. *What's wrong with this code?*

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

```c
#include <unistd.h>
#define HELLO_NUMBER 10

int main(){
    pid_t children[HELLO_NUMBER];
    int i;
    for(i = 0; i < HELLO_NUMBER; i++){
        pid_t child = fork();
        if(child == -1){
            break;
        }
        if(child == 0){ //I am the child
             execlp("ehco", "echo", "hello", NULL);
        }
        else{
            children[i] = child;
        }
    }

    int j;
    for(j = 0; j < i; j++){
        waitpid(children[j], NULL, 0);
    }
    return 0;
}
```

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,...,...2^{10} processes, "fork bombing" our machine. How could we prevent this? Put an `exit` (`exit(int status)` 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.

2.4 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).

```c
/* from Advanced Linux Programming (page 51) */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

/* Spawn a child process running a new program. PROGRAM is the name
   of the program to run; the path will be searched for this program.
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list. Returns  the process ID of
   the spawned process.  */
int spawn (char* program, char** arg_list) {
  pid_t child_pid;

  /* Duplicate this process. */
```

```
  child_pid = fork ();

  if (child_pid != 0){
    /* This is the parent process. */
    return child_pid;
  }else {
    /* Now execute PROGRAM, searching for it in the path.  */
    execvp (program,  arg_list);
    /* The execvp  function returns only if an error occurs.  */
    fprintf (stderr,  "an error occurred in execvp\n");
    abort ();
  }
}

int main () {
  /*  The argument list to pass to the "ls" command.  */
  char* arg_list[] = {
    "ls",     /* argv[0], the name of the program.  */
    "-l",
    "/",
    NULL      /* The argument list must end with a NULL.  */
  };

  /* Spawn a child process running the "ls" command. Ignore the
     returned child process ID.  */
  spawn ("ls", arg_list);
  printf ("done with main program\n");
  return 0;
}
```

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.

3 Controlling processes with signals

NOTE: Before reading this chapter, remember that signal handlers should do as little as they can and operations should be atomic (1 cycle). The code used is only for API demonstration purposes and should not be used in practice. Signal handlers should also not print anything.

3.1 What are Unix signals?

> **DEFINITION 3.1.** *Signal is a notification, a message sent by either operating system or some application to our program. Signals are a software interrupt mechanism for one-way asynchronous notifications. A signal may be sent from the kernel to a process, from a process to another process, or from a process to itself.*

Linux kernel ≥ 3.2.0 implements 31 signals (http://poincare.matf.bg.ac.rs/~ivana/courses/ps/sistemi_knjige/pomocno/apue/APUE/0201433079/ch10lev1sec2.html#ch10fig01). Every signal has a name starting from the characters `SIG`. Signals don't carry any argument and their names are mostly self explanatory. For example, `SIGABRT` is the abort signal that is generated when a process calls the `abort` function. Signal names are all defined by positive integer constants (the signal number) in the header `<signal.h>`. Each one is identified by a number from 1 to 31.

We can tell the kernel to do one of three things when a signal occurs. We call this the `disposition` of the signal, or the action associated with a signal.

1. *Ignore* the signal. This works for most signals, but two signals can never be ignored: SIGKILL and SIGSTOP.
2. *Catch* (handle) the signal.To do this, we tell the kernel to call a function of ours whenever the signal occurs. For example, if the process has created temporary files, we may want to write a signal-catching function for the `SIGTERM` signal (the termination signal that is the default signal sent by the `kill` command) to clean up the temporary files.
3. Let the *default action* apply. Every signal has a default action – for most signals it is to terminate the process.

3.2 Catch signal – the `signal` function

The simplest interface to the signal features of the UNIX System is the signal function.

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
// Returns: previous disposition of signal if OK, SIG_ERR on error
```

- `sig` – This is the signal number (from 1 to 31) to which a handling function is set. All signal numbers are explained in table **TODO!** and declared in `signal.h`.
- func – This is a pointer to a function. This can be a function defined by the programmer or one of the following predefined functions:
  1. `SIG_DFL` – Default handling. The signal is handled by the default action for that particular signal.
  2. `SIG_IGN` – Ignore Signal. The signal is ignored.
- `return` – returns the previous value of the signal handler, or `SIG_ERR` on error.

EXAMPLE 3.1. *This example demonstrates how to catch a `SIGINT` signal. `SIGINT` is generated e.g. when the user presses `Ctr-C`.*

Listing 9: Catching and handling a `SIGINT` using the `signal` function (src/signal_catch.c).

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void sighandler(int);

int main () {
   signal(SIGINT, sighandler);

   while(1) {
      printf("Going to sleep for a second...\n");
      sleep(1);
   }
   return(0);
}

void sighandler(int signum) {
   printf("Caught signal %d, coming out...\n", signum);
   exit(1);
}
```

After waiting for a few seconds and pressing `Ctr-C`, the output is the following.

```
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
^CCaught signal 2, coming out...

shell returned 1
```

3.2.1 User-defined signals

Linux also provides two user-defined signals, namely `SIGUSR1` and `SIGUSR2`. `SIGUSR1` for instance can be emitted from the terminal with the `kill` command:

```
$ kill -USR1 PID_NUMBER
```

EXAMPLE 3.2. *This example demonstrates how to catch `SIGUSR` signals. Note that the `pause` function simply suspends the calling process until a signal is received.*

Listing 10: Handling user signal (src/signal_user.c).

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h> // pause()


/* one handler for both signals */
static void sig_usr(int);

int main(void)
{
    printf("my PID is %d, send signal\n", getpid());
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        fprintf(stderr, "can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        fprintf(stderr, "can't catch SIGUSR2");
    for ( ; ; )
        pause();
}

/* argument is signal number */
static void sig_usr(int signo)
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
        fprintf(stderr, "received signal %d\n", signo);
}
```

If we run this program, note its PID:

```
my PID is 1557, send signal
```

Then open another terminal and type

```
$ kill -USR1 1557
$ kill -USR2 1557
$ kill 1557
```

then the output is

```
received SIGUSR1
received SIGUSR2
Terminated
```

Note that by default `kill` sends `SIGTERM`.

3.3 Send signal – the `kill` function

The kill() function sends a signal to a process or process group specified by pid. The signal to be sent is specified by sig and is either 0 or one of the signals from the list in the <sys/signal.h> header file.

The process sending the signal must have appropriate authority to the receiving process or processes. The kill() function is successful if the process has permission to send the signal sig to any of the processes specified by pid. If kill() is not successful, no signal is sent.

```
#include <signal.h>

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

The `kill()` function shall send a signal to a process or a group of processes specified by pid. The signal to be sent is specified by `sig` and is either one from the list given in `<signal.h>` or 0. If `sig` is 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of pid.

If `pid` is

- > 0, `sig` shall be sent to the process whose process ID is equal to pid.
- 0, `sig` shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the process group ID of the sender, and for which the process has permission to send a signal.
- -1, `sig` shall be sent to all processes (excluding an unspecified set of system processes) for which the process has permission to send that signal.

`int sig`: As already mentioned, `sig` is the signal number to send with value from 1 to 31 for modern Linux kernels, defined in `<signal.h>`.

`return`: Upon successful completion, `0` shall be returned. Otherwise, `-1` shall be returned and `errno` set to indicate the error.

Below are some typical `kill()` examples to control the child.

```
kill(child, SIGUSR1); // Send a user-defined signal
kill(child, SIGSTOP); // Stop the child process (the child cannot prevent this)
kill(child, SIGTERM); // Terminate the child process (the child can prevent this)
kill(child, SIGINT); // Equivalent to CTRL-C (by default closes the process)
```

3.3.1 `kill()` examples

EXAMPLE 3.3. *The listing below demonstrates how a process can send SIGINT to itself, terminating itself.*

Listing 11: A process sending SIGINT to itself (src/kill_suicide.c).

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <unistd.h>
4  #include <sys/types.h> // pid_t
5  #include <time.h>
6
7  void suicide(){
8      pid_t myPid = getpid();
9      printf("My PID is %d\n", myPid);
10     /* Look it up on a terminal */
11     sleep(30);
12     kill(myPid, SIGINT);
13     /* Look up again */
14 }
15
16 int main()
```

```
{
    suicide();
    return 0;
}
```

4 Waiting for process termination

4.1 Why wait for children to complete?

Going back to fork-exec, 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?

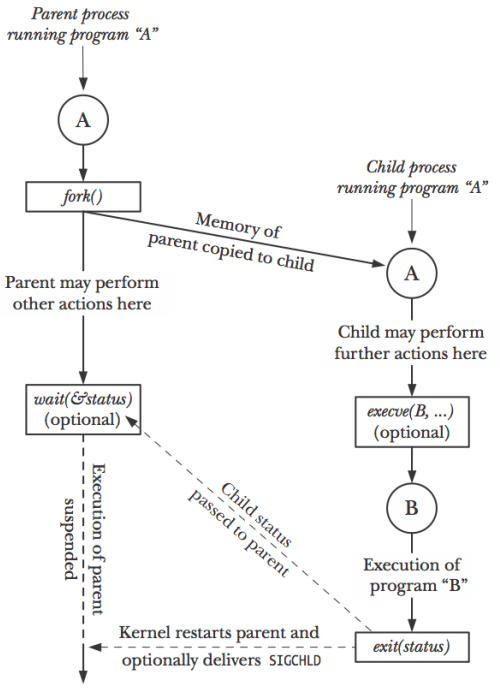
Another reason why wait should be used by the parent is because When a process ends via `exit` or `return`, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains. When a process terminates, 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.



cycle.png cycle.png **Figure 24-1:** Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

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

4.2 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 breif 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
```

4.3 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.

4.3.1 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);
```

`wait` and `waitpid` calls

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

- < -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`.

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

Return value (same for both):

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

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 is allows a process to be made non-blocking via `options` and can select which child process to wait via `pid`.

4.4 Practical examples of the `wait()` call

EXAMPLE 4.1.

Listing 12: Waiting for child and getting its status (src/fork_exec_wait.c).

```
/* from Advanced Linux Programming (page 57) */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include<sys/wait.h>

/* Spawn a child process running a new program. PROGRAM is the name
   of the program to run; the path will be searched for this program.
```

```
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list. Returns  the process ID of
   the spawned process.  */
int spawn (char* program, char** arg_list) {
    pid_t child_pid;

    /* Duplicate this process. */
    child_pid = fork ();

    if (child_pid != 0){
        /* This is the parent process. */
        return child_pid;
    }else {
        /* Now execute PROGRAM, searching for it in the path.  */
        execvp (program,  arg_list);
        /* The execvp  function returns only if an error occurs.  */
        fprintf (stderr,  "an error occurred in execvp\n");
        abort ();
    }
}

int main () {
    int child_status;

    /*  The argument list to pass to the "ls" command.  */
    char* arg_list[] = {
        "ls",     /* argv[0], the name of the program.  */
        "-l",
        "/",
        NULL      /* The argument list must end with a NULL.  */
    };

    /* Spawn a child process running the "ls" command. Ignore the
       returned child process ID.  */
    spawn ("ls", arg_list);
    /* Wait for the child process to complete. */
    wait (&child_status);
    if (WIFEXITED (child_status))
        printf ("the child process exited normally, with exit code %d\n",
                WEXITSTATUS (child_status));
    else
        printf ("the child process exited abnormally\n");
    return 0;
}
```

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 4.2.

Listing 13: Status code of multiple children (src/wait_pid_array.c).

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

void waitexample()
```

```
{
    int i, stat;
    pid_t pid[5];
    for (i=0; i<5; i++)
    {
        if ((pid[i] = fork()) == 0)
        {
            sleep(1);
            exit(100 + i);
        }
    }

    // Using waitpid() and printing exit status
    // of children.
    for (i=0; i<5; i++)
    {
        pid_t cpid = waitpid(pid[i], &stat, 0);
        if (WIFEXITED(stat))
            printf("Child %d terminated with status: %d\n",
                   cpid, WEXITSTATUS(stat));
    }
}

// Driver code
int main()
{
    waitexample();
    return 0;
}
```

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 4.3. *When the parent hits a `wait()`, it pauses its execution and waits for the other forked branch, the child to finish. Then it continues. Such a flow is demonstrated below.*

Listing 14: Waiting to synchronise with child (src/fork_sync.c).

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

void fork9() {
    int child_status;
    if (fork() == 0) {
        printf("HC: hello from child\n");
    }
    else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    exit(0);
}

int main()
{
    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.

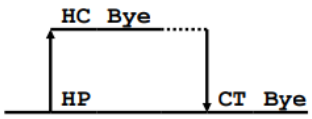


Fig. 6. Synchronising child and parent in the example.

EXAMPLE 4.4. *This example shows a principle similar to the one in Listing 5. Although the child inherits the same virtual address space, the actual physical address of variable `var` is different between then so each one holds a different instance of the variable.*

Listing 15: Modifying the same variable in a child and in a parent using a signal handler (src/sig_wait_val.c).

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <wait.h>
4  #include <stdlib.h> // exit, NULL
5  #include <unistd.h> // fork
6
7  int val = 10;
8  void handler(int sig) {
9      val += 5;
10 }
11
12 int main() {
13     pid_t pid;
14     signal(SIGCHLD, handler);
15     if ((pid = fork()) == 0) {
16         val -= 3;
17         printf("[child] val = %d\n", val);
18         exit(0);
19     }
20     waitpid(pid, NULL, 0);
21     printf("[parent] val = %d\n", val);
22     exit(0);
23 }
```

Output:

```
[child] val = 7
[parent] val = 15
```

4.5 More advanced examples

EXAMPLE 4.5. *What's the output of the following program?*

Listing 16: Fork followed by waiting for status (src/fork_waitpid.c).

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
  int status;
  printf("%s\n", "Hello");
  printf("%d\n", !fork());
  // waitpid(-1, ... -> wait for any child process
  if(waitpid(-1, &status, 0) != -1) {
    printf("%d\n", WEXITSTATUS(status));
  }
  printf("%s\n", "Bye");
  exit(2);
}
```

The order of 0 (for parent) and 1 (for child) may change. The diagram below explains the flow.

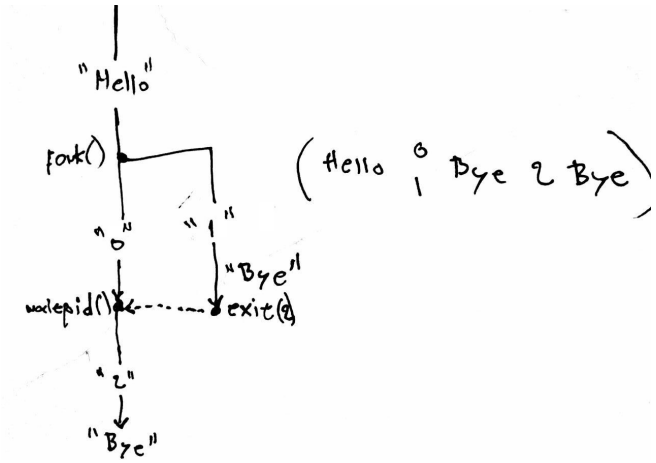


Fig. 7. Flow diagram for the example.

```
Hello
0
1
Bye
2
Bye

shell returned 2
```

EXAMPLE 4.6.

Listing 17: Forking, killing and waiting for status of multiple children (src/fork_kill_wait.c).

```c
#define _POSIX_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

#define N 5

/*
 * int_handler - SIGINT handler
 */

void fork12()
```

```c
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++){
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
                ;
        }
    }
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}


int main(int argc, char *argv[])
{
    fork12();
    return 0;
}
```

Output:

```
Killing process 29280
Killing process 29281
Killing process 29282
Killing process 29283
Killing process 29284
Child 29280 terminated abnormally
Child 29281 terminated abnormally
Child 29282 terminated abnormally
Child 29283 terminated abnormally
Child 29284 terminated abnormally
```

EXAMPLE 4.7. *This example is almost the same as the previous but additionally includes a signal handler attached to signal SIGINT.*

Listing 18: Fork children, send them SIGINT, use a signal handler and wait (src/fork_kill_signal_wait.c).

```c
#include <stdio.h>
#include <signal.h>
#include <wait.h>
#include <stdlib.h> // exit, NULL
#include <unistd.h> // fork

#define N 3
/*
int_handler - SIGINT handler
 */
void int_handler(int sig)
{
```

```
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

/*
 * fork13 - Simple signal handler example
 */
void fork13()
{
    pid_t pid[N];
    int i;
    int child_status;

    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
                ;
        }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}

int main()
{
    fork13();
    return 0;
}
```

Output:

```
Killing process 1269
Killing process 1270
Killing process 1271
Process 1271 received signal 2
Process 1269 received signal 2
Process 1270 received signal 2
Child 1271 terminated with exit status 0
Child 1269 terminated with exit status 0
Child 1270 terminated with exit status 0
```

EXAMPLE 4.8. *The program in the listing below forks and waits the parent for its children to exit and generate a SIGCHLD (`pause()` command, not `wait()`). Upon receiving SIGCHLD, the signal handler checks whether there are process to wait and as long as this is true reaps them (`wait()`). Not the random order of the output order because of the `fork()` call.*

Listing 19: Catching and handling a `SIGINT` using the `signal` function (src/signal_wait_all_children.c).

```
/*
 * adapted from: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f05/code/ecf
```

```c
   /forks.c
 */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

#define N 5
int ccount = N;

/*
 * child_handler2 - SIGCHLD handler that reaps all terminated children
 */
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    /* reap the zombie processes */
    while ((pid = wait(&child_status)) > 0) {
        ccount--;
        /* NOT recommended to use printf here - for debug only */
        printf("Received signal %d from process %d\n", sig, pid);
    }
}

/*
 * fork15 - Using a handler that reaps multiple children
 */
void fork15()
{
    pid_t pid[N];
    int i;

    /* call child_handler2 when SIGCHLD is received */
    signal(SIGCHLD, child_handler2);

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            printf("Created child with PID %d\n", getpid());
            /* Child: Exit (and emit SIGCHLD) */
            exit(0);
        }
    /*
     * pause() - waits for any signal.
     * Without this parent may exit before waiting.
     */
    while (ccount > 0) {
        pause();
    }
}

int main()
{
    fork15();
    return 0;
}
```

The output could look like:

```
Created child with PID 3652
Created child with PID 3653
Created child with PID 3654
```

```
Received signal 17 from process 3652
Received signal 17 from process 3653
Received signal 17 from process 3654
Created child with PID 3656
Created child with PID 3655
Received signal 17 from process 3655
Received signal 17 from process 3656
```

Note the randomness in the output order as `fork()` does not determine whether the child or parent will execute first.

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://delightlylinux.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.

[4] GeeksForGeeks, *Fork() in c*, https://www.geeksforgeeks.org/fork-system-call/.