A note on the vfork

Decades ago, the BSD Unix developers came up with an efficient special case system call— the `vfork(2)`. The idea at the time, was to perform some optimizations where you performed a fork and almost immediately an exec in the child (the fork-exec, in other words). As we know, using the fork-exec is quite a common and useful semantic (the shell and network servers use it heavily). When the vfork is called instead of the fork, the kernel does not go through the heavy copying operations usually required; it optimizes things.

The bottom line is this: At the time, `vfork(2)` was useful on Unix; but today's Linux `fork(2)` is as optimized as can be, rendering the `vfork` to the back door. It's still there, for perhaps two reasons:

- Compatibility—to aid the porting of BSD apps to Linux
- It is apparently useful on some arcane special Linuxes that run on MMU-less processors (like uClinux)

On today's regular Linux platforms, it is not recommended to use the `vfork(2)`; just stick to the `fork(2)`.

More Unix weirdness

From fork rule #3, we understand that the parent and child processes run in parallel. What if one of them terminate? Will the other die too? Well, no, of course not; they are independent entities. However, there are side effects.

Orphans

Consider this scenario: A process forks, the parent and child are alive and running their individual code paths in parallel. Let's say the parent's PID is 100 and the child's is 102, implying the child's PPID is 100 of course.

The parent process, for whatever reason, dies. The child continues on without any trouble, except for a side effect: The moment the parent (PID 100) dies, the child's PPID (100) is now invalid! Thus, the kernel intervenes, setting the child's PPID to the overall mothership—the ancestor of all user space tasks, the root of the process tree—the init, or on recent Linux, the systemd, process! It's PID is, by venerable Unix convention, always the number `1`.

Terminology: the child that lost its immediate parent is now said to be re-parented by systemd (or init), and its PPID will thus be `1`; this child is now an orphan.

There is a possibility that the overall ancestor process (init or systemd) does *not* have PID 1, and thus the orphan's PPID may not be 1; this can occur, for example, on Linux containers or custom namespaces.

We notice that the child's PPID value abruptly changed; thus, the systems programmer must ensure that they do *not* depend on the PPID value being the same (which can always be queried via the `getppid(2)` system call) for any reason!

Zombies

The orphaned process does not pose any problem; there is another scenario with the distinct possibility of a nasty problem arising out of it.

Consider this scenario: a process forks, the parent and child are alive and running their individual code paths in parallel. Let's say the parent's PID is 100 and the child's is 102, implying the child's PPID is 100 of course.

Now we delve into a further level of detail: the parent process is supposed to wait upon the termination of its children (via any of the available `wait*(2)` APIs of course); what if it does not? Ah, this is really the bad case.

Imagine this scenario: the child process terminates, but the parent is not waiting (blocking) upon it; thus it continues to execute its code. The kernel, however, is not pleased: The Unix rule is that the parent process must block upon its children! As the parent isn't, the kernel cannot afford to completely clean up the just-dead child; it does release the entire VAS freeing up all the memory, it does flush and close all open files, as well as other data structures, but it does not clear the child's entry in the kernel's process table. Thus, the dead child still has a perfectly valid PID and some miscellaneous information (it's exit status, exit bitmask, and so on). The kernel keeps these details as this is the Unix way: the parent must wait upon its children and reap them, that is, fetch their termination status information, when they die. How does the parent process reap the child(ren)? Simple: by performing the wait!

So, think about it: The child has died; the parent has not bothered to *wait* for it; the kernel has cleaned up, to some extent, the child process. But it technically exists, as it's half dead and half alive; it's what we call a ***zombie process***. In fact, this is a process state on Unix: Z for zombie (you can see this in the output of `ps -l`; additionally, the process is marked as *defunct*).

So why not just kill off the zombie(s)? Well, they're already dead; we cannot kill them. The reader might then query, well, so what? let them be. OK, there are two reasons that zombies cause real headaches on production systems:

- They take up a precious PID
- The amount of kernel memory taken up by the zombie is not insignificant (and essentially is a waste)

So, the bottom line is this: a couple of zombies might be OK, but dozens and hundreds, and more, are certainly not. You could reach a point where the system is so clogged with zombies that no other process can run—the `fork(2)` fails with `errno` set to `EAGAIN` (try again later) as no PIDs are available! It's a dangerous situation.

The Linux kernel developers had the insight to provide a quick fix: if you notice zombies on the system, you can, at least temporarily, get rid of them by killing their parent process! (Once the parent is dead, of what use is it to have the zombies? The point was, they remained so that the parent could reap them by doing a wait). Note that this is merely a bandage, not a solution; the solution is to fix the code (see the following rule).

This is a key point; in fact, what we call the wait scenario #4: the wait gets unblocked with children that already terminated, in effect, the zombies. In other words, you not only should, you must, wait upon all children; otherwise, zombies will occur (Note that the zombie is a valid process state on the Unix/Linux OS; every process, on the 'way' to death will pass through the **zombie** (Z) state. For most it's transient; it should not remain in this state for any significant length of time).

Fork rule #7

All of this neatly brings us to our next rule of fork.

Fork rule #7: *The parent process must wait (block) upon the termination (death) of every child, directly or indirectly.*

The fact is that, just like the `malloc-free`, the `fork-wait` go together. There will be situations in real-world projects where it might look impossible for us to force the parent process to block on the wait after the fork; we shall address how these seemingly difficult situations can be easily addressed (that's why we refer to an indirect method as well; hint: it's to do with signaling, the topic of the next chapter).

The rules of fork – a summary

For your convenience, this table summarizes the fork rules we have encoded in this chapter:

Rule	The rule of fork
1	After a successful fork, execution in both the parent and child process continues at the instruction following the fork
2	To determine whether you are running in the parent or child process, use the fork return value: it's always `0` in the child, and the PID of the child in the parent
3	After a successful fork, both the parent and child process execute code in parallel
4	Data is copied across the fork, not shared
5	After the fork, the order of execution between the parent and child process is indeterminate
6	Open files are (loosely) shared across the fork
7	The parent process must wait (block) upon the termination (death) of every child, directly or indirectly

Summary

A core area of Unix/Linux systems programming is learning how to correctly handle the all-important `fork(2)` system call, to create a new process on the system. Using the `fork(2)` correctly takes a lot of deep insights. This chapter helped the systems developer by providing several key rules of fork. The concepts learned—the rules, working with data, open files, security issues, and so on—were revealed via several code examples. A lot of details on how to wait for your children processes correctly were discussed. What exactly are orphans and zombie processes, and why and how we should avoid zombies was dealt with too.

We shall first enumerate the `sa_flags` possible values in this table and then proceed to work with them:

`sa_flag`	Behavior or semantic it provides (from the man page on `sigaction(2)`).
`SA_NOCLDSTOP`	If `signum` is `SIGCHLD`, do not generate `SIGCHLD` when children stop or stopped children continue.
`SA_NOCLDWAIT`	(Linux 2.6 and later) If `signum` is `SIGCHLD`, do not transform children into zombies when they terminate.
`SA_RESTART`	Provide behavior compatible with BSD signal semantics by making certain system calls restartable across signals.
`SA_RESETHAND`	Restore the signal action to the default upon entry to the signal handler.
`SA_NODEFER`	Do not prevent the signal from being received from within its own signal handler.
`SA_ONSTACK`	Call the signal handler on an alternate signal stack provided by `sigaltstack(2)`. If an alternate stack is not available, the default (process) stack will be used.
`SA_SIGINFO`	The signal handler takes three arguments, not one. In this case, `sa_sigaction` should be set instead of `sa_handler`.

Keep in mind that `sa_flags` is an integer value interpreted by the OS as a bitmask; bitwise-ORing several flags together to imply their combined behavior is indeed common practice.

Zombies not invited

Let's get started with the flag `SA_NOCLDWAIT`. First, a quick digression:

As we learned in `Chapter 10`, *Process Creation*, a process can fork, resulting in an act of creation: a new child process is born! From that chapter, it is now relevant to recall our Fork **Rule #7**: The parent process must wait (block) upon the termination (death) of every child, directly or indirectly.

The parent process can wait (block) upon the child's termination via the wait system call API set. As we learned earlier, this is essential: if the child dies and the parent has not waited upon it, the child becomes a zombie—an undesirable state to be in, at best. At worst, it can terribly clog system resources.

However, blocking upon the death of the child (or children) via the wait API(s) causes the parent to become synchronous; it blocks, and thus, in a sense, it defeats the whole purpose of multiprocessing, to be parallelized. Can we not be asynchronously notified when our children die? This way, the parent can continue to perform processing, running in parallel with its children.

Ah! Signals to the rescue: the OS will deliver the `SIGCHLD` signal to the parent process whenever any of its children terminate or enter the stopped state.

Pay attention to the last detail: the `SIGCHLD` will be delivered even if a child process stops (and is thus not dead). What if we do not want that? In other words, we only want the signal sent to us when our children die. That is precisely what the `SA_NOCLDSTOP` flag performs: no child death on stop. So, if you do not want to get spoofed by the stopping of the children into thinking they're dead, use this flag. (This also applies when a stopped child is subsequently continued, via the `SIGCONT`).

No zombies! – the classic way

The previous discussion should also make you realize that, hey, we now have a neat asynchronous way in which to get rid of any pesky zombies: trap the `SIGCHLD`, and in its signal handler, issue the wait call (using any of the wait APIs covered in `Chapter 9`, *Process Execution*), preferably with the `WNOHANG` option parameter such that we perform a non-blocking wait; thus, we do not block upon any live children and just succeed in clearing any zombies.

Here is the classic Unix way to clear zombies:

```
static void child_dies(int signum)
{
    while((pid = wait3(0, WNOHANG, 0)) != -1);
}
```

Delving into depth here would be of academic interest only on modern Linux (modern Linux, in your author's opinion, being the 2.6.0 Linux kernel and beyond, which, by the way, was released on December 18, 2003).

No zombies! – the modern way

So, with modern Linux, avoiding zombies became vastly easier: just trap the `SIGCHLD` signal using `sigaction(2)`, specifying the `SA_NOCLDWAIT` bit in the signal flags bitmask. That's it: zombie worries banished forever! On the Linux platform, the `SIGCHLD` signal is still delivered to the parent process—you can use it to keep track of children, or whatever accounting purposes you may dream up.

By the way, the `POSIX.1` standard also specifies another way to get rid of the pesky zombie: just ignore the `SIGCHLD` signal (with the `SIG_IGN`). Well, you can use this approach, with the caveat that then you will never know when a child does indeed die (or stop).

So, useful stuff: let's put our new knowledge to the test: we rig up a small multiprocess application that generates zombies, but also clears them in the modern way as follows (`ch11/zombies_clear_linux26.c`):

For readability, only the relevant parts of the code are displayed; to view and run it, the entire source code is available here: `https://github.com/PacktPublishing/Hands-on-System-Programming-with-Linux`.

```
int main(int argc, char **argv)
{
  struct sigaction act;
  int opt=0;

  if (argc != 2)
      usage(argv[0]);

  opt = atoi(argv[1]);
  if (opt != 1 && opt != 2)
      usage(argv[0]);

  memset(&act, 0, sizeof(act));
  if (opt == 1) {
        act.sa_handler = child_dies;
        /* 2.6 Linux: prevent zombie on termination of child(ren)! */
        act.sa_flags = SA_NOCLDWAIT;
  }
  if (opt == 2)
        act.sa_handler = SIG_IGN;
  act.sa_flags |= SA_RESTART | SA_NOCLDSTOP; /* no SIGCHLD on stop of
child(ren) */

  if (sigaction(SIGCHLD, &act, 0) == -1)
```

```
        FATAL("sigaction failed");
    printf("parent: %d\n", getpid());
    switch (fork()) {
    case -1:
        FATAL("fork failed");
    case 0: // Child
        printf("child: %d\n", getpid());
        DELAY_LOOP('c', 25);
        exit(0);
    default: // Parent
        while (1)
            pause();
    }
    exit(0);
}
```

(For now, ignore the `SA_RESTART` flag in the code; we shall explain it shortly). Here is the signal handler for `SIGCHLD`:

```
#define DEBUG
//#undef DEBUG
/* SIGCHLD handler */
static void child_dies(int signum)
{
#ifdef DEBUG
   printf("\n*** Child dies! ***\n");
#endif
}
```

Notice how we only emit a `printf(3)` within the signal handler when in debug mode (as it's async-signal unsafe).

Let's try it out:

```
$ ./zombies_clear_linux26
Usage: ./zombies_clear_linux26 {option-to-prevent-zombies}
 1 : (2.6 Linux) using the SA_NOCLDWAIT flag with sigaction(2)
 2 : just ignore the signal SIGCHLD
$
```

OK, first we try it with option `1`; that is, using the `SA_NOCLDWAIT` flag:

```
$ ./zombies_clear_linux26 1 &
[1] 10239
parent: 10239
child: 10241
c $ ccccccccccccccccccccccc
*** Child dies! ***
```

```
$ ps
  PID TTY TIME CMD
 9490 pts/1 00:00:00 bash
10239 pts/1 00:00:00 zombies_clear_l
10249 pts/1 00:00:00 ps
$
```

Importantly, checking with `ps(1)` reveals there is no zombie.
Now run it with option `2`:

```
$ ./zombies_clear_linux26 2
parent: 10354
child: 10355
ccccccccccccccccccccccccc
^C
$
```

Notice that the `*** Child dies! ***` message (that we did get in the previous run) does not appear, proving that we never enter the signal handler for `SIGCHLD`. Of course not; we ignored the signal. While that does prevent the zombie, it also prevents us from knowing that a child has died.

The SA_NOCLDSTOP flag

Regarding the `SIGCHLD` signal, there is an important point to realize: The default behavior is that, whether a process dies or stops, or a stopped child continues execution (typically via the `SIGCONT` signal being sent to it), the kernel posts the `SIGCHLD` signal to its parent.

Perhaps this is useful. The parent is informed of all these events—the child's death, stop-page, or continuation. On the other hand, perhaps we do not want to be spoofed into thinking that our child process has died, when in reality it has just been stopped (or continued).

For such cases, use the `SA_NOCLDSTOP` flag; it literally means no `SIGCHLD` on child stop (or resume). Now you will only get the `SIGCHLD` upon child death.

Interrupted system calls and how to fix them with the SA_RESTART

Traditional (older) Unix OSes suffered from an issue regarding the handling of signals while processing blocking system calls.

When a thread is newly created (either via the `fork(2)`, `pthread_create(3)` or `clone(2)` APIs), and once the OS determines that the thread is fully born, it informs the scheduler of its existence by putting the thread into a runnable state. A thread in the **R** state is either actually running on a CPU core or is in the ready-to-run state. What we need to understand is that in both cases, the thread is enqueued on a data structure within the OS called a **run queue** (**RQ**). The threads in the run queue are the valid candidates to run; no thread can possibly run unless it is enqueued on an OS run queue. (For your information, Linux from version 2.6 onward best exploits all possible CPU cores by setting up one RQ per CPU core, thus obtaining perfect SMP scalability.) Linux does not explicitly distinguish between the ready-to-run and running states; it merely marks the thread in either state as **R**.

The sleep states

Once a thread is running its code, it obviously keeps doing so, until, typically, one of a few things (mentioned as follows) happen:

- It blocks on I/O, thus sleeping—entering state of **S** or **D**, depending (see the following paragraph).
- It is preempted; there's no state change, and it remains in a ready-to-run state **R** on a run queue.
- It is sent a signal that causes it to stop, thus entering state **T**.
- It is sent a signal (typically SIGSTOP or SIGTSTP) that causes it to terminate, thus first entering state **Z** (zombie is a transient state on the way to death), and then actually dying (state X).

Often, a thread will encounter in its code path a blocking API—one that will cause it to enter a sleep state, waiting on an event. While blocked, it is removed (or dequeued) from the run queue it was on, and instead added (enqueued) onto what's called a **wait queue** (**WQ**). When the event it was waiting upon arises, the OS will issue it a wakeup, causing it to become runnable (dequeued from its wait queue and enqueued onto a run queue) again. Note that the thread won't run instantaneously; it will become runnable (**Rr** in *Figure 1*, Linux state machine), and a candidate for the scheduler; soon enough, it will get a chance and actually run on the CPU (**Rcpu**).

A common misconception is to think that the OS maintains one run queue and one wait queue. No—the Linux kernel maintains one run queue per CPU. Wait queues are often created and used by device drivers (as well as the kernel); thus, there can be any number of them.

The depth of the sleep determines precisely which state the thread is put into. If a thread issues a blocking call and the underlying kernel code (or device driver code) puts it into an interruptible sleep, the state is marked as **S**. An interruptible sleep state implies that the thread will be awoken when any signal destined for it is delivered; then, it will run the signal handler code, and if not terminated (or stopped), will resume the sleep (recall the `SA_RESTART` flag to `sigaction(2)` from `Chapter 11`, *Signaling - Part I*). This interruptible sleep state **S** is indeed very commonly seen.

On the other hand, the OS (or driver) could put the blocking thread into a deeper uninterruptible sleep, in which case the state is marked as **D**. An uninterruptible sleep state implies that the thread will not respond to signals (none; not even a SIGKILL from root!). This is done when the kernel determines that the sleep is critical and the thread must await the pending event, blocking upon at any cost. (A common example is a `read(2)` from a file—while data is being actually read, the thread is placed into an uninterruptible sleep state; another is the mounting and unmounting of a filesystem.)

Performance issues are often caused by very high I/O bottlenecks; high CPU usage is not always a major problem, but continually high I/O will make the system feel very slow. A quick way to determine which application(s) (processes and threads, really) are causing the heavy I/O is to filter the `ps(1)` output looking for processes (or threads) in the **D**, uninterruptible sleep state. As an example, refer to the following:

```
$ ps -LA -o state,pid,cmd | grep "^D"
D 10243 /usr/bin/gnome-shell
D 13337 [kworker/0:2+eve]
D 22545 /home/<user>/.dropbox-dist/dropbox-
lnx.x86_64-58.4.92/dropbox
$
```

Notice that we use `ps -LA`; the `-L` switch shows all threads alive as well. (FYI, the thread shown in the preceding square brackets, `[kworker/...]`, is a kernel thread.)

The following diagram represents the Linux state machine for any process or thread:

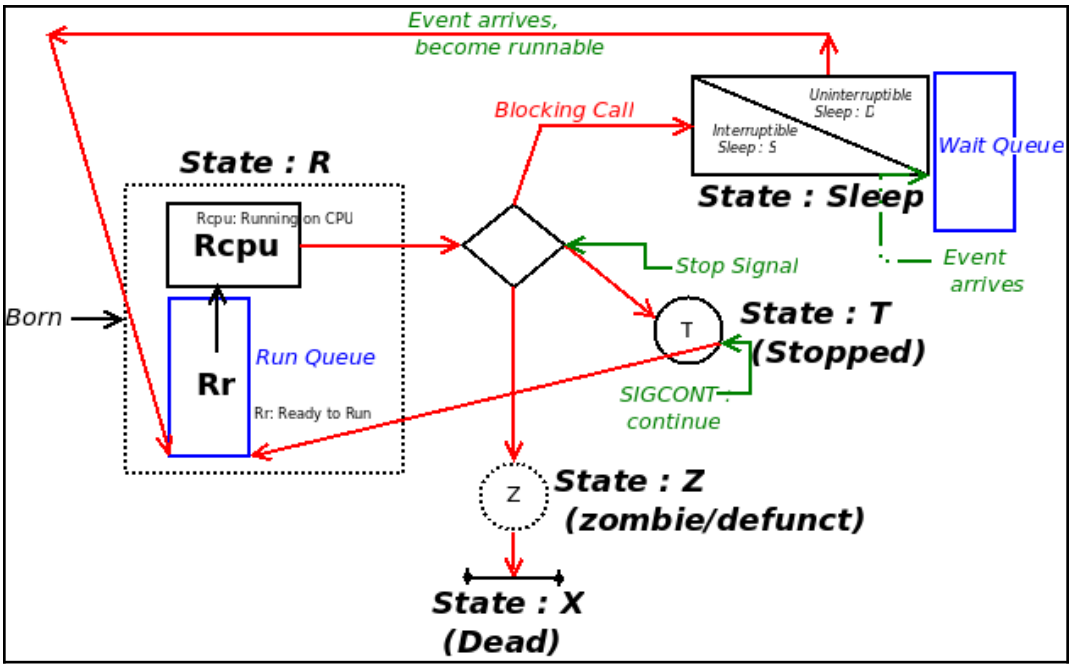

Figure 1: Linux state machine

The preceding diagram shows transitions between states via red arrows. Do note that for clarity, some transitions (for example, a thread, can be killed while asleep or stopped) are not explicitly shown in the preceding diagram.

What is real time?

Many misconceptions exist regarding the meaning of real time (in application programming and OS contexts). Real time essentially means that not only do the real-time thread (or threads) perform their work correctly, but they must perform within a given worst-case deadline. Actually, the key factor in a real time system is called determinism. Deterministic systems have a guaranteed worst-case response time to real-world (or artificially generated) events; they will process them within a bounded time constraint. Determinism leads to predictable response, under any conditions—even extreme load. One way in which computer scientists classify algorithms is via their time complexity: the big-O notation. O(1) algorithms are deterministic; they guarantee that they will complete within a certain worst-case time, no matter the input load. True real-time systems require O(1) algorithms for implementing their performance-sensitive code paths.