EXIT, WAIT and WAITPID indepth explanation

There are two *exit* calls *(exit* or *_exit)* that are used to terminate a process. Once a process exits, the parent process can determine its exit status by using one of the *wait* functions. The *_exit* function is defined by POSIX.1, whereas *exit* was defined by ANSI C. Both of these functions will have the desired effect of terminating your process. In general, you want to use the ANSI function because it cleans up more than the POSIX function does. Usually, but not always, *exit* is implemented as a sequence of cleanup steps, followed by a call to *_exit.* Neither call ever returns.

The *wait* calls are more than just nice for determining the manner of passing of child processes. Once a process has exited, most of its memory and other resources will usually be returned to the system (although POSIX is silent on the subject). Some resources, however, remain occupied after the process is gone. Notably, the supervisor stack of a process is often not freed at the time of *exit* (because it is difficult to free a stack the process is currently using), and, in addition, some subset of the process control data structures will also be left out to store the exit status of the deceased process.

The *status* value passed to *exit* and *_exit* becomes the exit status of the process. This value can be retrieved by the parent process if it so desires, by use of *wait* or *waitpid.* In the POSIX documents, *wait* is described as an interface that can be used to retrieve the exit status of deceased processes. It's more than that. The *wait* functions serve the important purpose of freeing the leftover resources associated with a deceased process. Historically, a process is called a "zombie" from the time it calls *exit* until the time the parent process *wait s* for it. During the zombie period, the process is still occupying system resources. If the parent process never calls *wait,* then it is possible that the system will clog up with zombies and be unable to proceed. So, it is important that parent processes *wait* (or *waitpid)* for their children, even though POSIX doesn't say they have to.
* Not all systems require a *wait* for each terminated child, but most do, so you should program with the assumption that *wait* calls are necessary.

Alternatively, the parent process itself can exit. In this case, the child processes are cleaned up by the system. Zombie processes are only a problem when a parent process remains alive, creating child processes indefinitely but not waiting for their demise.
The simple *wait* function blocks until a child process terminates. It returns the process ID of the deceased child and status information encoded in the parameter passed to it. *waitpid* is the version of *wait* with the racing stripes, chrome hubcaps, and the dingle balls in the window. You can set it to wait for only one child, or any. You can cause it to block, or not, by setting flags in the *options*

Here is a function that uses the *waitpid* call:

```
void check_for_exited_children (void)
{
      pid_t terminated;
      /* pid_t is a datatype identical to integer, Deal with terminated
      children */
```

```
        terminated = waitpid(-1, &status, WNOHANG);

        if (terminated > 0) {
              if (WIFEXITED(status)) {
                    printf("Child %d exit(%d)\n",
                    terminated, WEXITSTATUS(status));

              } else if (WIFSIGNALED(status)) {
                    printf("Child %d got signal %d\n",
                    terminated, WTERMSIG(status));

              } else if (WIFSTOPPED(status)) {
                    printf("Child %d stopped by signal %d\n",
                    terminated, WSTOPSIG(status));
              }
        }
  }
```

The first argument to *waitpid*, called *which*, specifies the set of processes you want to wait for.

- If *which* is -1, then *waitpid* will wait for any process that is a child of the calling process, just like *wait* does.
- If *which* is 0, then *waitpid* will wait for any process that is a child of the calling process that has the same process group ID as the calling process.*
- If *whih* is positive, then w*aitpid* will wait only for that process, which had better be a child of the calling process (or else an errno, ECHILD, will be returned).
- Finally, if *which* is less than -1, then the absolute value of *which* specifies a process group ID. In this case, w*aitpid* will wait for any process that is a child of the calling process that has that process group ID.
  The last argument, *options*, can have two flags set in it: WNOHANG and WUNTRACED.

  WNOHANG, which we are using, tells *waitpid* not to wait for processes to exit. If there is a dead process handy, then WNOHANG returns its status. Otherwise, an error is returned, indicating that no processes had exited when *waitpid* was called. If WUNTRACED is set, and if _POSIX_JOB_CONTROL is supported, then *waitpid* will return the status for stopped, as well as terminated, child processes. _POSIX_JOB_CONTROL does not really concern us here. The status information for the exited process can be decoded using the macros you see in *check_for_exited_children.* These macros are:

  WIFEXITED(status_info)
  This call returns a true value if the child process exited normally (via a call to *exit*, *_exit*, or by falling off the end of
  its *main* routine).

  WIFSIGNALED(status_info)

This macro returns true if the child was killed by a signal. Usually this means the process got a segmentation fault, bus error, floating point exception, or some other processor fault. It may also mean that a signal was explicitly sent
to the child process. Signals are a leading cause of process death, in addition to their other uses (like stopping processes, *faux* asynchrony and low-bandwidth, asynchronous interprocess communication). We'll discuss signals below.

WIFSTOPPED(status_info)
This macro returns a true value if the process has not terminated at all, but is, instead, just stopped (this only happens if *waitpid* is used, with WUNTRACED set in the *options* parameter). Stopped processes are usually in that state because they received a stop signal. That implies the application is exercising some sort of job control of its own (like a shell does), or is being debugged. The POSIX-standard job control features are supported only if _POSIX_JOB_CONTROL is defined in <*unistd.h*>. Job control does not concern us here.

WEXITSTATUS(status_info)
If *WIFEXITED(status_info)* returned true, then this macro returns the exit status for the process. Only the least significant 8 bits of the exit status are returned by this macro. The rest are lost.

WTERMSIG(status_info)
If *WIFSIGNALED(status_info)* returned true, then this macro tells you which signal was responsible for the child's termination.

WSTOPSIG(status_info)
If *WIFSTOPPED(status_info)* returned true, then this macro tells you which signal was responsible for the child's being stopped.