What's the difference between fork() and vfork()?

Some systems have a system call `vfork()`, which was originally designed as a lower-overhead version of `fork()`. Since `fork()` involved copying the entire address space of the process, and was therefore quite expensive, the `vfork()` function was introduced (in 3.0BSD).

However, since `vfork()` was introduced, the implementation of `fork()` has improved drastically, most notably with the introduction of `copy-on-write', where the copying of the process address space is transparently faked by allowing both processes to refer to the same physical memory until either of them modify it. This largely removes the justification for `vfork()`; indeed, a large proportion of systems now lack the original functionality of `vfork()` completely. For compatibility, though, there may still be a `vfork()` call present, that simply calls `fork()`without attempting to emulate all of the `vfork()` semantics.

As a result, it is *very* unwise to actually make use of any of the differences between `fork()` and `vfork()`. Indeed, it is probably unwise to use `vfork()` at all, unless you know exactly *why* you want to.

The basic difference between the two is that when a new process is created with `vfork()`, the parent process is temporarily suspended, and the child process might borrow the parent's address space. This strange state of affairs continues until the child process either exits, or calls `execve()`, at which point the parent process continues.

This means that the child process of a `vfork()` must be careful to avoid unexpectedly modifying variables of the parent process. In particular, the child process must **not** return from the function containing the `vfork()` call, and it must **not** call `exit()` (if it needs to exit, it should use `_exit()`; actually, this is also true for the child of a normal `fork()`).