Brief notes on memory operatins in fork(), exec() and vfork() by Dr. Beck

Background: The address space of a process is made up of a number of segments, each of which is a contiguous area of memory used for a particular purpose. In this discussion, we consider processed with these segments: code, static data, heap and stack.

`Fork()`: The `fork()` call creates a new process (the *child*) by allocating an unused task control block (TCB) and copying many of the fields from the calling process' (the *parent's*) TCB. Thus, while the child is not identical to the parent in all respects, they share many attributes in common. For example, they have the same user ID (uid) but different process IDs (pid).

`Fork()` is able to *share* read-only segments between the parent and child, allowing the child's address space to be created using the same memory as the parent. In our example, the only ready-only segment is the code segment. Since neither parent nor child can modify this segment, they can use it simultaneously with no possibility of interfering.

The other segments in our example (static data, heap and stack) are all writable, and so sharing them between a parent and child executing simultaneously could result in interference between those two processes (race conditions). Thus, the fork system call allocates new memory for those segments in the child (the same size as the segments in the parent) and copies the contents of each segment from the parent to the corresponding segment in the child. The parent and child then each have a private copy of the contents of each of these segments stored in its own private memory, and so parent and child can both execute simultaneously without the possibility of interference.

`Exec()`: `Exec()` is a system call that replaces the address space of a running process with the contents of an executable file. The code and static data segments are read from the file into newly allocated memory, the stack and heap are initialized to be empty.

It is very common for a command line interpreter (*shell*) or GUI to invoke an executable file (command or user program) by call `fork()` to create a new child process, and the child process then almost immediately calling `exec()` to start the program stored in the file. In this scenario all the work that does into copying the address space of the child (creating the static data, heap and stack segments) is almost immediately undone by the call to `exec()`.

`Vfork()`: `Vfork()` is a means for a parent to avoid the overhead of copying the parent's address space when a call to `exec()` will be made almost immediately after process creation. The call to `vfork()` is similar to `fork()` but it does not allocate new memory for the writable segments, instead allowing the child to execute using the memory segments of the parent. In order to avoid interference between parent and child, the parent is not allowed to execute after the call to `vfork()` until the child makes a call to `exec()`.

`Vfork()` avoids the need to allocate and copy memory for the child's address space, but because the child does share the parent's address space it must be very careful about what variables it modifies before calling `exec()`. If it changes anything that will cause problems in the parent once it starts running again, the parent might experience errors. Thus we can say that `vfork()` is not completely safe (because it can cause problems if the child makes error-causing modifications) but it can be much more efficient than `fork()`.