Interpreter Files:-

There's a difference between interpreter file and interpreter
#! Pathname optional arguments
Pathname is an interpreter but the whole is an interpreter file.

Interpreter files are useful:
Interpreter scripts provide an efficiency gain. Consider the previous example again. We could still hide that the program is an awk script, by wrapping it in a shell script:

awk 'BEGIN { for (i = 0; i < ARGC; i++) printf "ARGV[%d] = %s\n", i, ARGV[i] exit }' $*

The problem with this solution is that more work is required. First, the shell reads the command and tries to execlp the filename. Because the shell script is an executable file but isn't a machine executable, an error is returned and execlp assumes that the file is a shell script (which it is). Then /bin/sh is executed with the pathname of the shell script as its argument. The shell correctly runs our script, but to run the awk program, the shell does a fork, exec, and wait. Thus there is more overhead involved in replacing an interpreter script with a shell script.

- When interepreter scripts finds an executable file that isn't a machine executable, **execlp has to choose a shell to invoke, and it always uses /bin/sh, so by using interpreter script #! /bin/csh, we can make it work**

There are two uses for fork:

1. When a process wants to duplicate itself so that the parent and the child can each execute different sections of code at the same time. This is common for network servers—the parent waits for a service request from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.

2. When a process wants to execute a different program. This is common for shells. In this case, the child does an exec (which we describe in Section 8.10) right after it returns from the fork.

Difference between fork and vfork:- is that vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes. (This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)
The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls exec (or exit) right after the vfork. Instead, the child runs in the address space of the parent until

it calls either exec or exit. This is more optimized but can leads to undefined results if child is modifying any data.