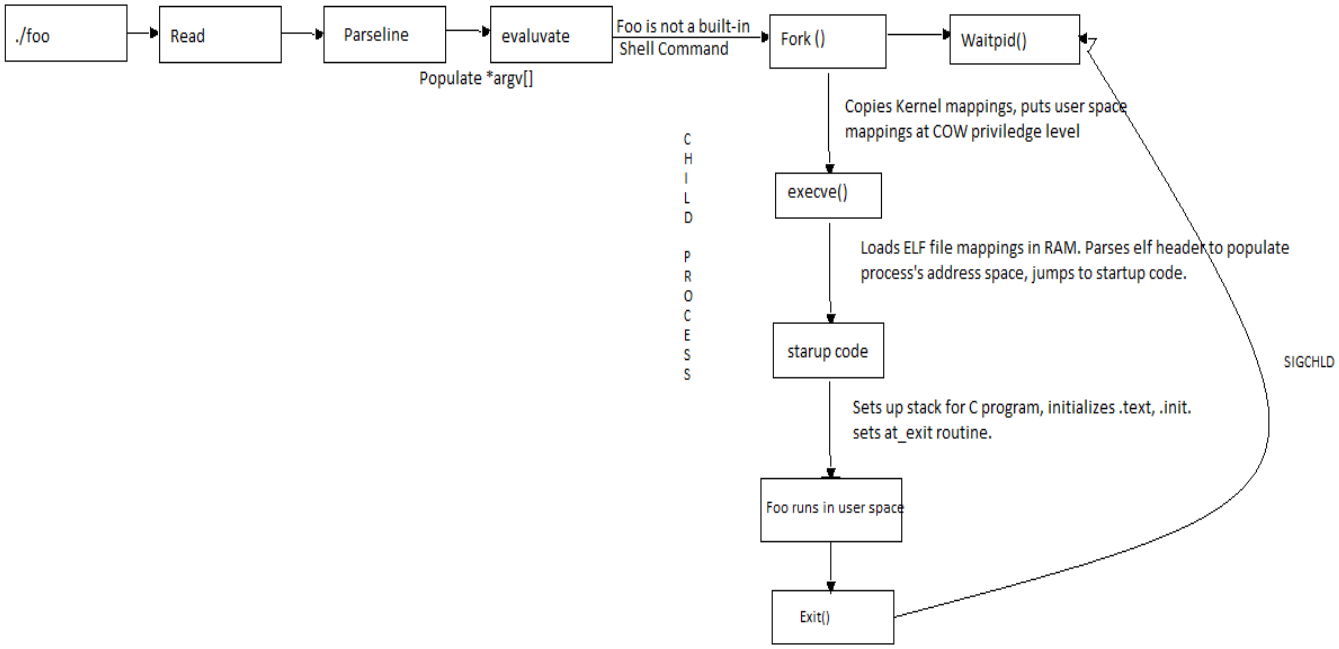
Flow chart: The below flow-chat shows the over view of various events programmed to take place while executing "./foo" in the command line.



Index: These are the sections we shall discuss in some detail.

(I) The shell

(II) Fork → Process creation details.

→Copy on write fork and its benefits.

→ waitpid for the shell

(III) execve → Loader

→ Parsing elf binary and populating process address space.

(IV) Startup code

(V) Program execution

(V) Exit function

(VI) Process termination

→ Sigchld, process remains as zombie in the terminated state

→ Returns to waitpid of the shell

→ Another command

Section 1: The Shell

The shell is an interactive application level program that helps the user interact with the kernel and run other application level programs. Generally a shell works in a series of read and evaluate steps.
When we type ./foo in the command line. The "." signifies that the executable we are trying to run is in the current directory. So any command typed in the shell goes through 3 phases. They are as follows.

(i) **Read:** The shell reads the input commands

(ii**) Parseline:** The shell parses the space separated command and populates the *argv[] vector. Generally argv is a null terminated array of pointers. Which has the name of the file "foo" in argv[0].
This argv vector will be given as input to the execve function about with we shall discuss in the sections to come. The argv vector will also eventually be passed to the "main" function of the "foo" executable.
If last argument of argv is "&", the process that shall run "foo" will be in the background. As it has not been given, we shall proceed the discussion in the assumption that such an input has not been given.

(iii) **Evaluate function:** The evaluate function of the shell will tell us that "foo" is not an inbuilt command of the shell like "cd". It is an executable, hence we have to ask the kernel to spwan a process and load it and run it in the user space.

Section 2: The Fork()

At this juncture, the shell calls the "Fork()" system call. Fork is an original UNIX system call which creates a new process and copies the whole address space of the parent process into the newly spawned child process. This technique is not found to be very optimal. Consider we run an executable or a utility, we most likely wont use any of the parent's user level address space mappings. But we end up copying it anyway. To address this problem we have various versions of "fork()" system call like, vfork() or clone(). "vfork()" does not replicate the parent address space. Whereas "clone()" creates a new process with shared mappings with the parent process. "Clone()" is used in many multi-threaded libraries like "Pthreads".
Another noteworthy aspect to discuss at this juncture is the Copy-on-write fork optimization. With experience we know that most of the "fork()" call will be followed by an "execve" call which is used to load and run an executable. In this case we can place all the parent process's user level address space mappings in shared state with a new privilege level. The idea is if either of the process attempt to write to these shared pages, we shall interrupt the modifying write and make separate copy of that specific page that was previously in shared state. This optimization aims at reducing the unnecessary copies of parent address space into the child process.
Finally, in Linux the "Fork()" call is a special function call that is called once and returned twice, once in the parent process and once in the child process. In the parent process it returns the process id(PID) of the child and in the child process it returns the pid of the parent.

General events that are programmed to occur during process creation are as follows:

a) Allocate and create page directory for this process.

- Allocate page directory.
- Initialize kernel portion in the page directory.

b) Generate pid for this process.

b) Initialize status variables for this process (parent id, process type and scheduling status).

c) Setup appropriate values for segment registers, including Requested Priviledge level

d) Enable interrupts in user mode.

e) Set the %eax to 0 (inside register data structure for this process).

f) Copy address space mappings of parent process into child process.

- If a page is writable, then mark it as copy-on-write for both parent and child.

g) Return pid of the child to the parent process.

Section 3: The execve()

Coming back to our example, the shell has now spawned a new process and populated the necessary kernel level data structures for process management and the shell is waiting for the child process to terminate.

At this juncture we shall call the “execve()” system call who's handler is the loader that loads the executable from the disk into the address space of the newly spawned process. The execve call takes in argv list that the shell populates as the input. It also takes in a list of environment variables as an input. The loader part in the execve() system call parses the ELF header of the executable to access various sectional offsets in the executable file. It copies the data mappings of sections (mainly code section and data section) into the respective locations of the address space of the newly spawned process. The address space structure in most of the monolithic kernel designs is same for all the processes unless the kernel employs address space randomization as part of its security feature. The execve function does not return if it can locate the specified file. If there is an error in locating the file it returns with the appropriate error value. It is also to be noted that apart from the header information a lot of “actual” copying is not done during the loading phase. As Linux follows the demand paging paradigm, the virtual mappings of the executable is copied to the child process's address space and the actual copying of data from the disk is done while the program executes. This copying is abstracted by the virtual memory infrastructure's page fault handler. It is done so to maintain the responsiveness of the system.

Section 4: Startup code.

After the loading process is complete the loader jumps to the programs entry point, which is pointed by the “_start” label. For any C program to function we must set up a stack, in other words the starup code sets up the user stack for the main function of “foo”. It initializes all the initializations in the .text and .init section. The startup code also initializes the “at_exit” function which gives a list of routines to be called when the application terminates.

Section 5: User program execution.

Now “foo” can execute safely in the user space, it has it's stack and heap initialized. If “foo” wants to use any of the system resources it typically does a library call, beneath which we perform system calls. While a system call is invoked, “foo” is halted. The state of the process running “foo” is saved and the system call handler is executed in kernel space. This is implemented using protected control transfer between user space to kernel space and back to user space to continue the execution of “foo”.

Section 6: exit() function.

After “foo” executes, it can explicitly from the main call the “exit()” system call. If it doesn't libc calls it implicitly. The exit function calls all the routines specified by the “at_exit()” in the reverse order in of their registration. By this system call we return the control back to the kernel that shall start the process termination routine.

Section 7: Process termination.

The kernel ideally destroys all the objects it had created for this process to run and sets the state of the process as “NOT_RUNNABLE”, it does not have an address space to run upon anymore. At this point in time the process is in “Zombie” state. It remains in this state until the parent process reaps, the terminated child process. This process now sends a “SIGCHLD” signal to the parent process, in our case the Shell.

The Shell which was waiting on the “Waitpid” call receives the SIGCHLD signal and now the process struct and its adjoining fields are removed from the process control block as the child process has been appropriately reaped.

After receiving the SIGCHLD, the shell which ideally will be in a while loop goes back to take the next command from the user.