How does a shell work?

What is a shell? It is a user program that acts as an interface b/w the user and the OS kernel.

How the shell is launched?

Step 1: Power on → Bootloader
Initialize the system and loads the OS kernel

Step 2: OS kernel is loaded
OS is in control now.

Step 3: Kernel launches init (or systemd)
This is the first user process (PID 1) and responsible for starting all other system processes

Step 4: init/systemd starts login services

Step 5: User logs in - You enter your username/password at a text based login prompt.

Step 6: shell is launched for logged-in user.
- after authentication, login program launches the shell.
Different kinds of shells; bash, zsh, etc.

Once the shell is launched succesfully, it shows you prompt and then wait for you to type something. Once you type a command, shell reads it, then forks a child, execs the command executable waits for it to finish, and finally reads the next command.

For eg:- suppose you typed prompt > ls → command to list files in the current directory.
Then, first shell read the command ls and find its executable (in PATH) to run. Then shell calls fork(). This creates a child program.

parent : shell waits
child : will run the new program.

On the child process → ls is executed using exec()
i.e., child calls, exec("ls", (char *) NULL);

This replaces the child process image with ls program.

- all current file descriptors (stdin, stdout, stderr) are preserved
- ls writes to stdout, which points to your terminal.

In the parent (shell): wait for the child to finish.
- i.e., parent calls wait()
it waits until ls finishes.

After ls finishes, the shell displays the next prompt and waits for new user input.

Remark: Note that shell waits only for those child process that you launched from the shell (i.e., shell is the parent of that process).
But, if the parent of a process exits before the child, it becomes an orphan and init process adopts it and wait.

eg:-
```
int main() {
    if (fork() == 0)
        sleep(10)
        printf("child done\n"); }      //child lives longer
    else
        {
        exit(0);                       //parent exits immediately.

    } }
```
Here, the parent exits, the child is orphaned.
Then init (or systemd) becomes its new parent and will clean up.

Question: why do we need a complicated step like fork() + exec() for a simple act of creating a new process?

- Because, the separation of fork() and exec() allows the shell to do many interesting things easily.

* Redirecting output from a command to a file (instead of standard output, terminal) screen

eg. prompt > wc p3.c > newfile.txt

(here, when the child is created, before calling exec(), the shell closes standard output and opens the file newfile.txt

Consequently, any o/p from the replacing program (wc) is sent to the file instead of the screen (note that the open file descriptors of the child process are kept open across the exec() call)

Why the re-direction is done by the shell after fork()?

- so that the re-direction affects only the child, not the parent shell. If the shell did it before fork(), then its own stdout would be redirected, which we do not want.

p4.c

p4.c > No Selection

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

    // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
    }
    return 0;
}
```

close (STDOUT_FILENO) – what it does?

↳ it closes file descriptor 1, which is the standard output (stdout) of the current process.

↳ upon closing, it releases the file descriptor associated with stdout

Now, if a new file is opened, open() will return the lowest available FD, which will now be 1 (since it is free).
So, now FD1 points to that file, instead of the terminal.

The following is the output of the program.

Result

```
prompt> ./p4
prompt> cat p4.output
32 109 846 p4.c
prompt>
```

// cat → displaying file contents

* Piping: suppose we want to count how many times the word foo appears in a file called input.txt.

```
prompt> grep -o foo input.txt | wc -l
```

what happens here?

* grep searches for the word foo in input.txt
* -o flag makes grep print each match on a new line (ie., if foo appears 5 times, you get 5 lines

```
foo
foo
foo
foo
foo
```

* The pipe | sends the output of grep into wc-l
* wc-l counts how many lines it received (in this case, how many times foo appeared)

What happens underhood? * shell creates a pipe
* shell fork()s two processes:

- one runs grep -o foo input.txt, its stdout goes into pipe's write end

- another runs wc -l, it stdin comes from the pipe's read end

- shell connects the two before exec().

Remark: without pipe, we need an intermediate file to do the same.

Further reading: OSTEP Chapter-5