Spark Session: minishell

updated: 07/04/2021

Project description:

> *Create a simple shell*

Topics

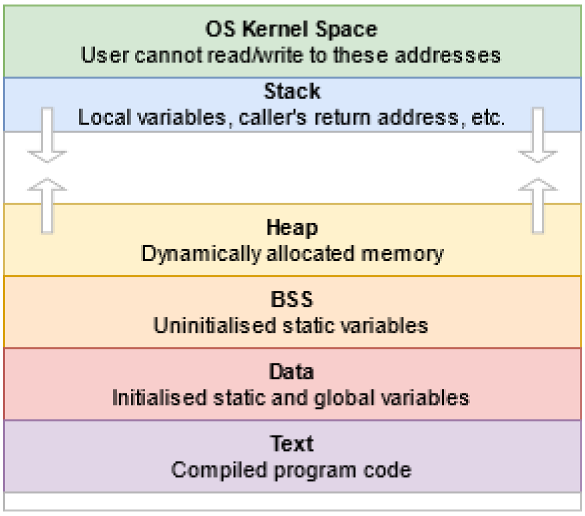
1. Processes
2. fork
3. wait
4. execve
5. dup & dup2
6. pipe

Processes

Before we get into how to work with processes, it's handy to understand what we actually mean by "process".

1. What is a process? (5 mins)

A process is its own separate entity with its own defined memory space. This memory space is what is duplicated by `fork` and rewritten by `exec`, which we'll get to in a bit.
Here's a diagram showing how this memory is divided:



To put it in really simple terms, you can think of a process like a struct — a collection of information bound to an entity. This information includes the process ID, open files, its status, etc. You can read more about that [here](#) later.

fork

`fork()` creates a new process — called the **child process** — by duplicating the calling process (**the parent process**).

1. What is the prototype of `fork()`? What does the function return? (10 mins)

    - How could you use the function return to identify if you are in the child or parent process?
    - Is `fork`'s return the same as the child's PID?
2. Which of the following is copied from the parent process to the child process? Which are not? (10 mins)
    - Data (the content of the process' memory space)
    - Location in memory
    - Process ID
    - Open file descriptors
3. Let's see some of these characteristics in action. (15 mins)
    - Write a program that:
        - initialises an int `x` to **5**;
        - calls `fork()` and then prints its return value in a statement `"fork returned: %d\n"`;
        - checks for failed forks;
        - if in the child process: **decrements** `x` by 1, prints `"This line is from child, x is %d\n"`, and then returns 0;
        - else if in the parent process: **increments** `x` by 1 and then prints `"This line is from parent, x is %d\n"`.
    - You should see how the data (the variable `x` in this case) starts with the same initial value in both processes, but that changes to this variable in one process does not affect the variable in another process.

      ```
      > ./fork_test
      fork returned: 18895
      This line is from parent, x is 6
      fork returned: 0
      This line is from child, x is 4
      ```

      example output
    - Here we've specified that child should `return` when it's done. What happens if we comment that out? Try putting another `"x is %d"` statement at the end of your main to see.
        - You should see how the child and parent processes then both execute the code that follows, returning to a common point in the program. Whether or not you want that depends on the program's purpose.

In this case, the parent and the child process execute concurrently. The order of your output might also be jumbled between child and parent, depending on how your OS handles the processes.

wait

It's also possible to have your parent process wait on its child processes to terminate. You do this by calling `wait()` in the parent process. This [synchronises](#) the parent and child process.

1. What is the prototype of `wait()`? What information is stored in the `wstatus` argument? (5 mins)

2. Calling `wait()` (or `waitpid()`) in the parent process prevents what's called "zombie processes". What does this mean? (10 mins)
3. Let's add `wait()` to the code you wrote earlier. (10 mins)
    - Create an int variable, for example `w_status`, to be passed to `wait()`.
    - In the parent process code block, call `wait()` before anything else. *Remember to pass it the address of your `w_status` int.*
    - Make sure the child process is calling `return` when it's done.
    - Use one of the macros to check if the child process terminated normally. If so, print `"Child process exited with status: %d\n"`. Use one of the other macros to get the exit status.
    - Try tweaking the argument you pass to the `return()` call in your child process. Does the output change accordingly?

    ```
    > ./wait_test
    fork returned: 17025
    fork returned: 0
    This line is from child, x is 4
    This line is from parent, x is 6
    Child process exited with status: 42
    x is 6
    ```

    example output

Here's a fun short explanation about zombie processes for later: [understanding zombie processes]()

Break (5 mins)

execve

The `exec()` family of functions allows us to replace the current process with a new program.
No new process is created; the PID remains the same. The functions simply have the existing process execute a new program.

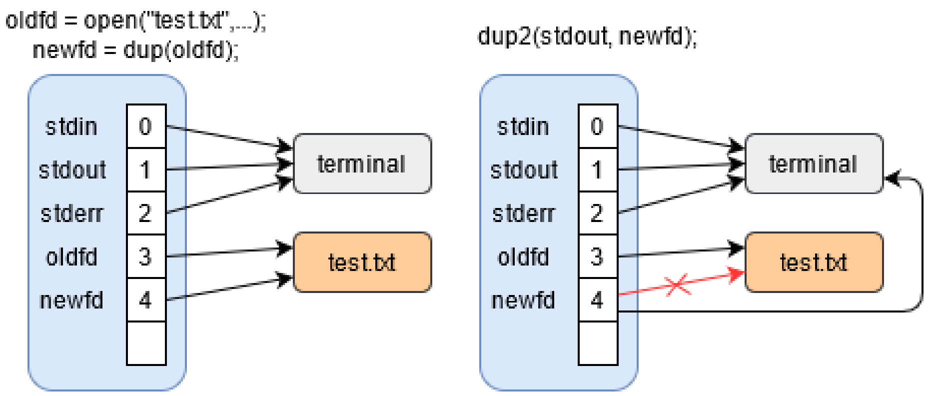
1. What is the prototype of `execve()`? (10 mins)
    - Break down each of function parameters. What does each mean?
2. What does `execve()` return? (5 mins)
3. To see how to execute a new program from within our child process, let's try using `execv()` (i.e. `execve` without the "e" environment option). (15 mins)
    - **Note**: for the purposes of keeping the exercise simple, we won't pass a specific environment. Also we'll hard-code our program arguments. You won't do this in your actual minishell of course.
    - At the beginning of your main, declare a `char *argv[2]`.
        - Initialize `argv[0]` to `"/bin/ls"`.
        - Initialize `argv[1]` to `"-l"`.
        - Initialize `argv[2]` to `NULL`.
        - *Do you know why these arguments are made in this order?*
    - In your child process, below the `"This line is from child"` statement, **instead of return()** we'll call `execv()`, passing it `"/bin/ls"` and your `argv` array.
    - Below the `execv` call, place another print statement, `"This line is from child after execv"`.

- Does your program execute `ls` with the `-l` list option when you run it? Does the "after execv" statement print?

dup & dup2

`dup()` and `dup2()` create a copy of a file descriptor.

1. What is the prototype for `dup()`? What about `dup2()`? What do both return? (5 mins)
2. What are some differences between `dup` and `dup2` with regards to the new file descriptor? (5 mins)
   - Here's a diagram to help you visualise the functions better:



3. What do the new and old file descriptor share? (5 mins)
4. Now let's write a program that: (15 mins)
   - opens a `test.txt` file with the following flags:
     - create if not existant;
     - truncates the file contents if file already exists;
     - allows reading and writing;
     - gives user read and write [permissions](), and read permissions to group and other.
     - *Hint: this is one of the flag combinations you'll also use in minishell*
   - saves the `open` return in an int `fd`;
   - creates another int, for example `dup_fd`;
   - calls `dup()`, giving it `fd` as argument and saving its return in `dup_fd`;
   - passes `fd` as the 1st argument to `write()`, with the string `"This will be written to the test file\n"`.
   - passes `dup_fd` as the 1st argument to `write()`, with the string `"This will also be written to the test file\n"`.

   ```
   > cat test.txt
   This will be written to the test file.
   This will also be written to the test file.
   ```

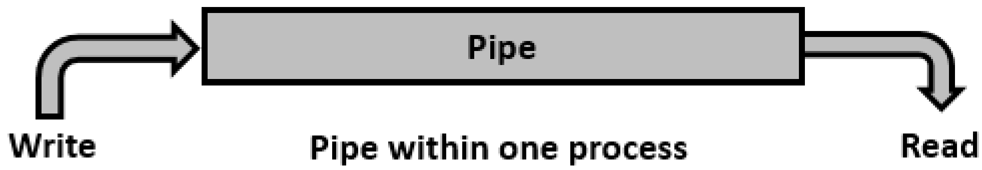
5. The real significance of `dup` and `dup2` to minishell is when we use them to **redirect** our output and/or input. For example, when the output of a command is redirected into a file or into a **pipe** (more on that later). (10 mins)

- Now, using `dup2`, turn the fd `1` (that is, stdout) into a copy of our test.txt `fd`.
- Call `write()` again, outputting `"This isn't being printed on stdout\n"` onto stdout (1).
- Does anything get printed onto your terminal when you run the program?

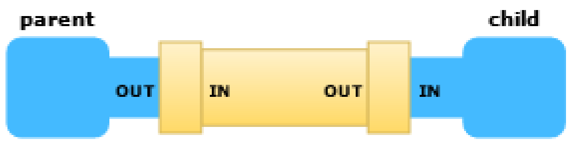
Break (5 mins)

pipe

`pipe()` allows data to be passed from one process to another.
This "pipeline" between processes is unidirectional, meaning data flows in one direction.
Therefore, you have one end of the pipe that reads data and one end of the pipe that writes data.



1. What is the prototype of `pipe()`? What is being stored in the int array you're passing it? (10 mins)
2. If you're using a pipe to pass data from one process to a second process, which `pipefd` would the 1st process **write** to? Which would the 2nd process **read** from? (10 mins)

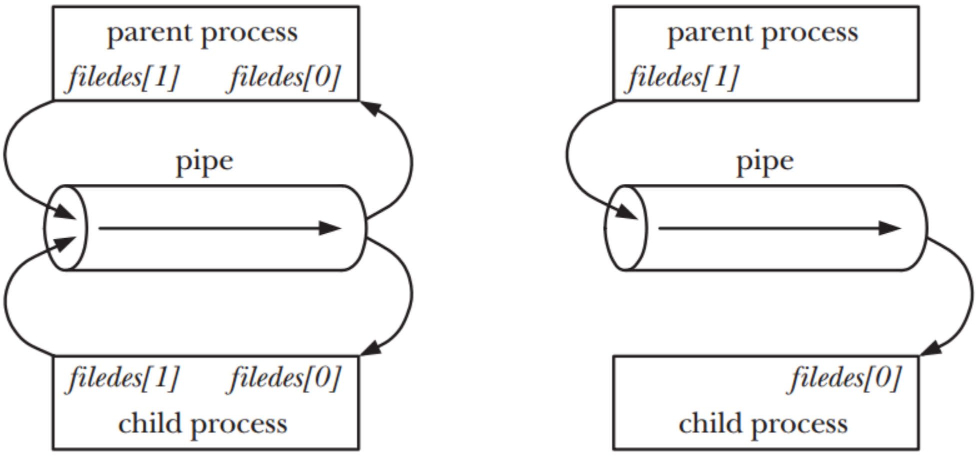


3. Let's try using `pipe` in combination with `fork` and `wait`! Write a program that: (15 mins)
    - takes command-line arguments (i.e. `int argc, char **argv`);
    - creates a **pipe**;
    - then **fork**s to create a child process;
    - the parent process:
        - should **close** the pipe end that it doesn't need;
        - **writes** argv[1] to the correct end of the pipe;
        - **close** the remaining pipe end.
    - the child process:
        - should **close** the pipe end that it doesn't need;
        - in a loop, **reads** the string from the pipe, one byte at a time;
        - calls `toupper()` (include `ctype.h`) on the read char;
        - **writes** to converted char to stdout;
        - writes a newline to stdout;

- **closes** the remaining pipe end.

```
./pipetest hello
HELLO
Child process exited with status: 0
```

4. Bonus question: why do we close the pipe ends we don't use at the start? Why do we close the pipe end we used after we're done?



a) After *fork()* **b) After closing unused descriptors**

That was a simple exercise to show you how data can be passed through pipes and interacted with within child processes.

Things get even more mind-blowing when you throw `dup` / `dup2` into the mix.

Here's a more detailed explanation about data flows through pipes, with handy diagrams: [pipes, forks, & dups]

Things that we couldn't cover today but that would be helpful to look into for your project:

- abstract syntax tree
- finite state machines