OSN Tutorial 2

BUILDING YOUR OWN SHELL

PROCESSES

- As we covered previous time, processes have their own unique ID with which they can be identifies.
- Try typing "ps" on your terminal to find out the processes that our currently running and their pid.
- But the question is : Are all these processes running in a similar fashion?

…

…

They are not.

CONTROLLING TERMINAL

- To understand the difference between background and foreground processes, it is important for one to understand what a controlling terminal is.
- A controlling terminal is a terminal in Linux where a process starts. If you enter a command from the shell to "sleep", the terminal window where this command is entered is the controlling terminal.
- Boiling it down, it's the terminal window that you are using.
- All input to the process is given through this terminal only, and all commands entered are run as processes in this terminal.
- Now, getting to foreground and background processes..

FOREGROUND AND BACKGROUND

- Processes can be divided in two categories widely :
  I. Foreground Processes
  II. Background Processes
- By default, processes run in the foreground. To run any command in the background, type an ampersand (&) at the end of the command.
- A foreground process has **access to the controlling terminal**, and the shell waits for it to end before it can resume its operation.
- However, a background process as the name suggests runs in the background and has no user interaction and does not have access to the controlling terminal.
- It cannot read from stdin, but it CAN output to stdout.

SWITCHING BETWEEN FG AND BG

- So.. a process switching from background process to a foreground process is basically gaining access to the controlling terminal.
- However, a background process cannot change the access of a controlling terminal normally or write to it.
- There is some signal handling that needs to happen here when a background process must access these functionalities.

MORE PROCESSES

- We discussed fork() last time which creates a new process.
- Doing Mini Project 0, you might have noticed that the parent and child process go on to execute simultaneously after this call.
- However, having the parent wait before this child completes can be beneficial in some cases.
- So how do you make the parent wait for it’s child process to end?

MAKE IT WAIT

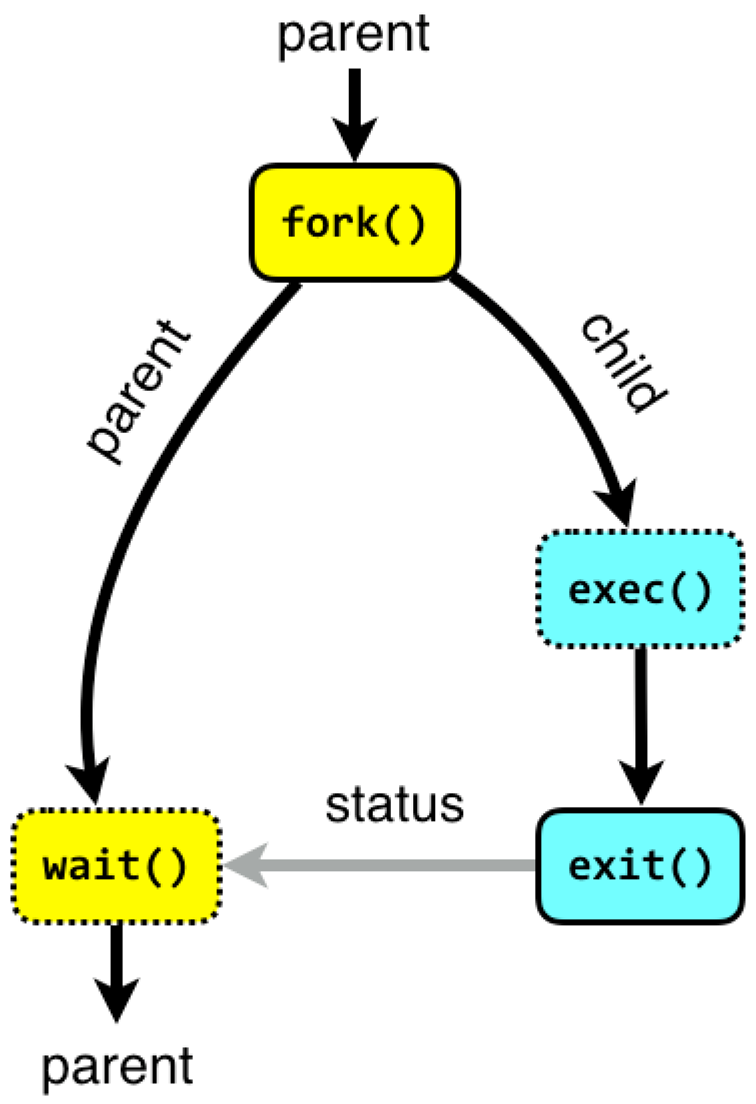
- Wait() is the system call that you are looking for here. This is a blocking command, and it makes the parent wait until one of its child terminates.
- Waitpid(pid_t pid) is another system call that can be used for this purpose which waits for the child with process id pid to terminate.
- Waitpid can take several more arguments and you should probably man it for further information (Hint : You will need it for Mini Project).

REAL POTENTIAL OF FORK - EXECVP

Execvp replaces the current process image with a new process image that is specified in it's arguments.

How is it useful?

Fork a new process from within a process. This process is duplicate of what you were running but then run Execvp and BOOM! You have a totally new process with a different functionality at hand.



FILE DESCRIPTORS

- We discussed about it before. It's just a number that signifies a file.
- Some are already assigned : 0 (stdin), 1 (stdout) and 2 (stderr).
- Kernel maintains a table of open file descriptors for each process (which is also changed with context switch).
- This table maps this small integer to a file descriptor structure (struct fd in Linux) which has the required info about an open file.
- This structure also contains a pointer to an open file description,

FILE DESCRIPTORS ALONG WITH FORK()

- One may ask this question, "What happens to a file descriptor after a fork() call?"
- While some may think that since the process created afterwards is an entirely different process from the original one, their will be no effect of one or the other.
- That's wrong, however.
- An important detail of the design of Unix is that the open file description and the file buffer are kernel data structures.
- Therefore, these are NOT duplicated by a fork() call.
- Hence, whenever one of the processes reads from or writes to its local file descriptor, the effects in the open file description and file buffer are seen by the other process.

FILES

- Now that we know about file descriptors, we can get to files. Fun fact : Everything in Linux is a file.
- There can be a file a.txt on your system, and it can have multiple open file description in the kernel which can be pointed by multiple file descriptors.
- It’s important to understand that a file descriptor will only affect another file descriptor if they share the same open file description.
- By default, using open creates a new open file description.


Always has been
Wait it's all Files

DUP SYSTEM CALL

- Basically, dupes a file descriptor that is passed as an argument.
- On success, returns a new file descriptor (-1 on error).
- It assigns the lowest available file descriptors.
- These two file descriptors can be used interchangeably, as they have the same open file description and share the same offset etc.
- Closing one does not affect the other.

DUP2 SYSTEM CALL

- Dup2 system call is very similar in its functionality. It takes two arguments the oldfd and newfd and copies oldfd into newfd.
- If newfd was previously open, it closes it before reusing it, automatically.
- Dup2(oldfd, newfd)
- You can use these system calls for implementing I/O redirection.
- How? Figure it out.

USING PIPES

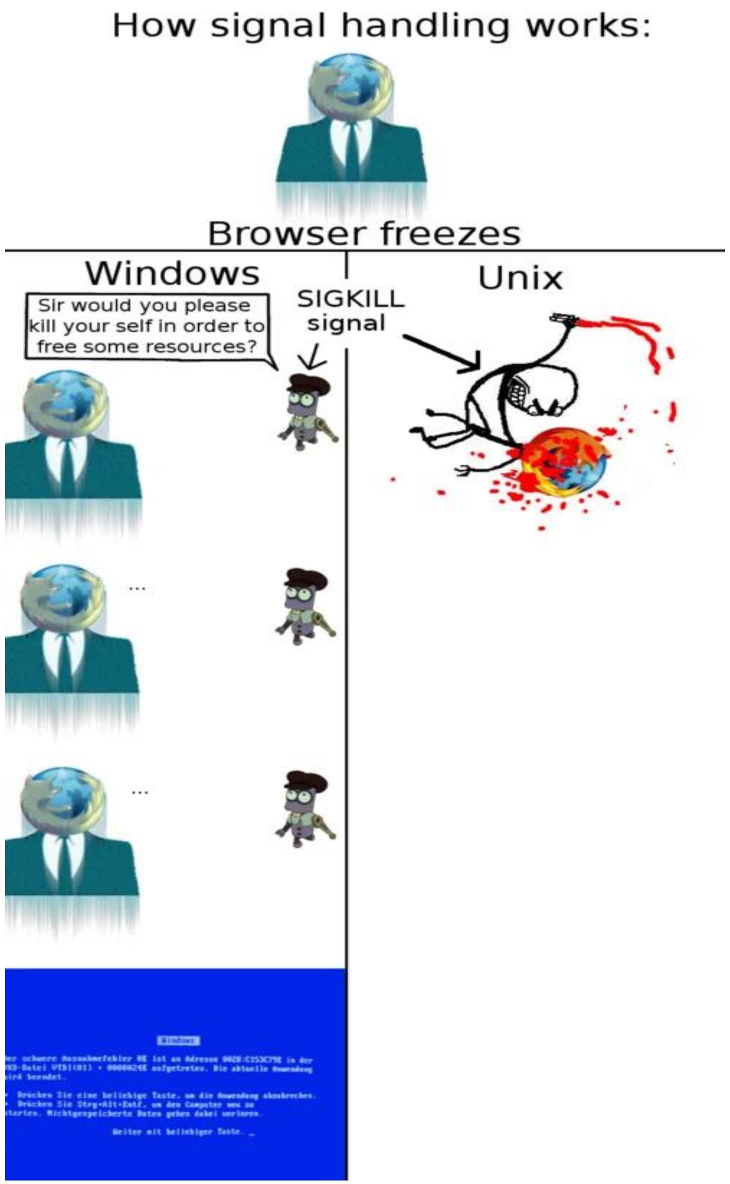
- A pipe is a connection between two processes.
- Takes output of one and gives as input for another.
- Try it out yourself.
- This is on a crude level though; true piping goes a bit deeper than this.

PIPE SYSTEM CALL

- Opens a pipe. Basically, a virtual file which can be written to and from.
- This pipe can be used by the process and its descendants since they get a copy of it.
- This system call basically returns two file descriptors, one pointing to the read end of the pipe and the other to the write-end.
- The read-end of the pipe can be used for reading from the pipe and write can be used to write to the pipe.

SIGNAL()

- There is quite a bit of signals involved when multiple processes are trying to get their job done.
- These can be used to terminate a process, stop it etc.
- Using signal system call, you can change how a process reacts upon receiving a signal.
- You can create your own custom signal handler, choose to ignore the signal or choose to receive it.
- It's very useful here since you can ignore signals using this.

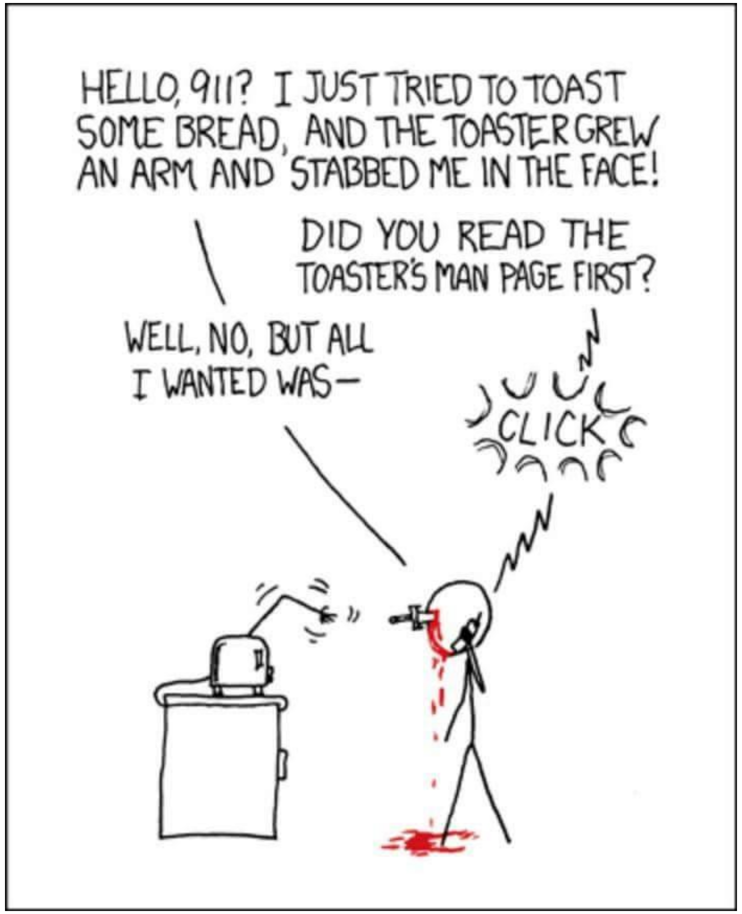


SOME GENERAL ADVICE (SERIOUSLY THIS TIME)

Read MAN pages.

Write modular code.

Start on time.



Any Doubts?