Processes and Inter-Process Communication

Hossein Soltanloo, University of Tehran

Processes

The `fork()` System Call

- The only way to launch a new program (process) is via `fork()` (that's what the shell or the UI does when you run a program)
- `fork()` duplicates (clones) the currently-running process: original is called "parent"; new is called "child"

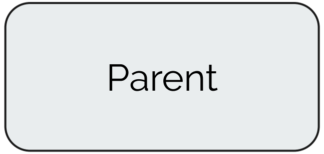
The `fork()` System Call

```
pid = fork();
if (pid == -1) {
  fprintf(stderr, "fork failed\n");
  exit(1);
}
if (pid == 0) {
  printf("This is the child\n");
  exit(0);
}
if (pid > 0) {
  printf("This is parent. The child is %d\n", pid);
  exit(0);
}
```

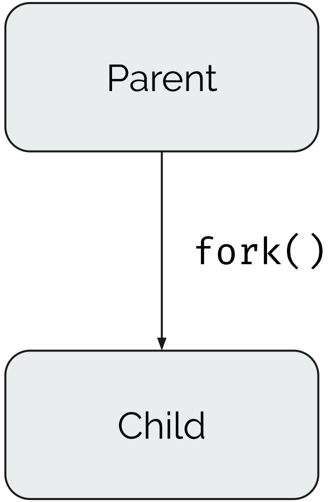
`fork()`-ing and `exec*()`-ing

- A typical use is to `fork()` the current process, then immediately run `exec*()` in the child, which replaces it with a new program/process
- `execl`, `execv`, `execle`, `execlp`, `execvp` are library (API) calls, all of which invoke `execve()`, the only actual "`exec`" system call

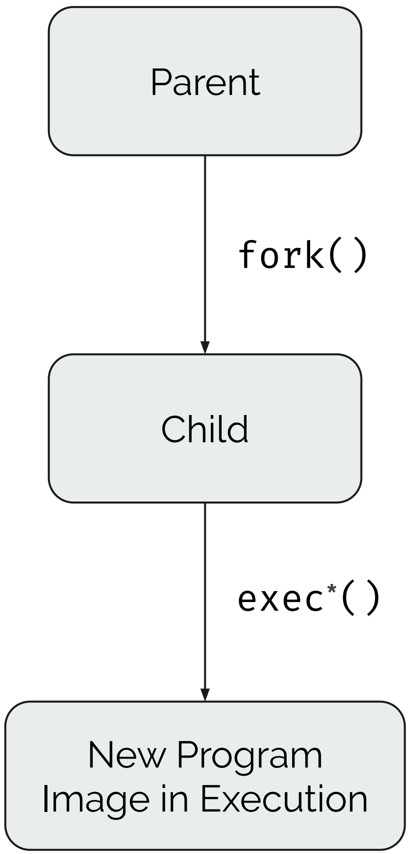
An Illustration



An Illustration



An Illustration



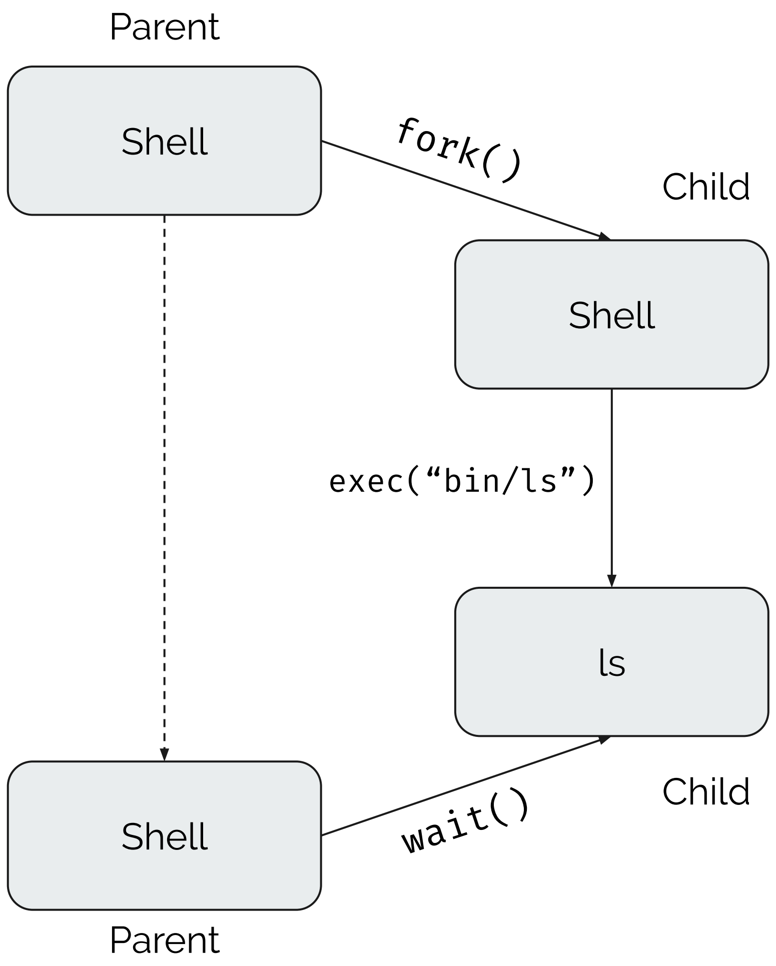
Example: Typing “`ls -l`” in Shell

- When you type ls in the shell, the shell issues a `fork()`, so now you have 2 shells (a parent shell & a child shell)
- We don’t really want 2 shells, so the child shell immediately issues an `exec*(…, “ls”, “-l”, …)`

Example: Typing "`ls -l`" in Shell

- This invokes the `execve()` system call, which replaces current process (child shell) with the "`ls`" program/process
- "Parent shell" continues to exist (as "the shell"), while child shell (now the "`ls`" program) runs to completion & terminates

An Example



Parent
Shell
fork()
Child
Shell
exec("bin/ls")
ls
Child
Shell
wait()
Parent

Kernel Implements Some `fork()` & `exec*()` Efficiencies

- After `fork()`, child & parent share same program text (marked as read-only)
  - Each process has their own instruction pointer to the text
- Copy-on-write (COW): child & parent share all data until `1` of them changes some datum's value
  - Then, kernel clones only that page of data
- If parent issues multiple `fork`s in a row, all children & parent will share the same data

COW Implementation

- Kernel marks “shared” COW pages as read-only
- If parent or child tries to write to a shared COW page, it forces a trap to the kernel
- First thing kernel checks upon this trap: Do we have a protection violation fault, or do we just need to unshare these pages?
- Kernel keeps a reference count of # of procs sharing each page of data (can unshare 1 proc at a time)
    - Unshare = copy over data & mark writable; decrease ref count on original copy; if ref count == 1, writable

What Exactly Happens Upon `exec*()`

- When issue `exec*()`, most of the "old process" resources are automatically discarded, closed, etc.
- Stack, heap, all data are wiped
- Except:
  - Process ID (`pid`) remains the same
  - All open file descriptors remain open
    - (unless you passed-in `FD_CLOEXEC` flag to `open()`)
- (That's why COW: what a waste to copy over code & data upon `fork()`, then wipe it on `exec`!)

Terminating a Process

- `exit (int status)`
  - Clean up the process (e.g close all files)
  - Tell its parent processes that he is dying (`SIGCHLD`)
  - Tell child processes that he is dying (`SIGHUP`)
  - Exit status can be accessed by the parent process.

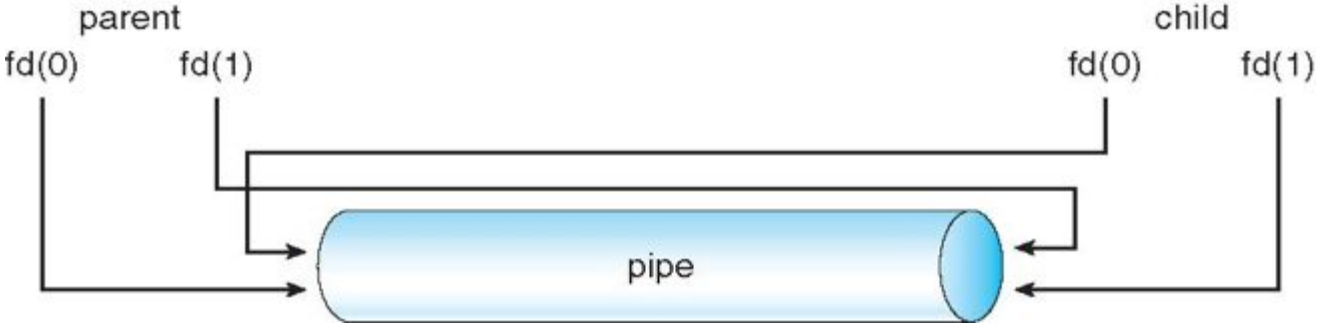
Parent-Child Synchronization

- Parent created the child, he has the responsibility to see it through:
  - check if the child is done.
    - `wait`, `waitpid`
  - check the exit status of the child
    - `pid_t wait(int *stat_loc)`
    - Some others such as whether the child was killed by an signal. etc
- A child has no responsibility for the parent
- Processes are identified by a process id (`pid`)
  - `getpid()`: find your own `pid`
  - `getppid()`: find the pid of the parent

Inter-Process Communication: Pipes

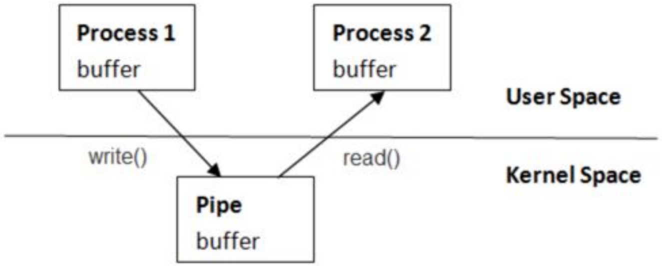
Pipes

- Pipe mechanism creates two end access points, one for read and one for write; whatever write to the pipe from one end can be read from the pipe on the other end.



Types of Pipes

- **Named pipes:**
  - like a file (create a named pipe (`mknod`), open, read/write)
  - can be shared by any process
- **Unnamed pipes:**
  - Does not associate with any physical file. In fact, a pipe is a buffer managed by the kernel. It is a temporary storage of the data to be transferred between participating cooperative processes.
  - Can only be shared by related processes (descendants of a process that creates the unnamed pipe).
  - Created using system call `pipe()`



The Pipe System Call

- Syntax
  - `int pipe(int fds[2])`
- Semantic
  - creates a pipe and returns two file descriptors `fds[0]` and `fds[1]`, both for reading and writing
  - a read from `fds[0]` accesses the data written to `fds[1]` (POSIX) and a read from `fds[1]` accesses the data written to `fds[0]` (non standard).
  - the pipe has a limited size (64K in some systems) -- cannot write to the pipe infinitely.
  - Reading from a pipe with no writer?

What You Will Implement

An Overview

- You are to implement a search system for a video games database
- The system is going to read multiple data files to extract the data and do the search among them
- For the sake of speed and performance, we need to delegate the extraction and queries to multiple processes
- The System is made up of three main components: load balancer, worker, and presenter

The Load Balancer

The load balancer process is the one responsible for fetching commands from the user and assigning parts of data to the worker processes in order for them to apply queries to chunks of data. It sends the data via an unnamed pipe for each worker process.

The Worker

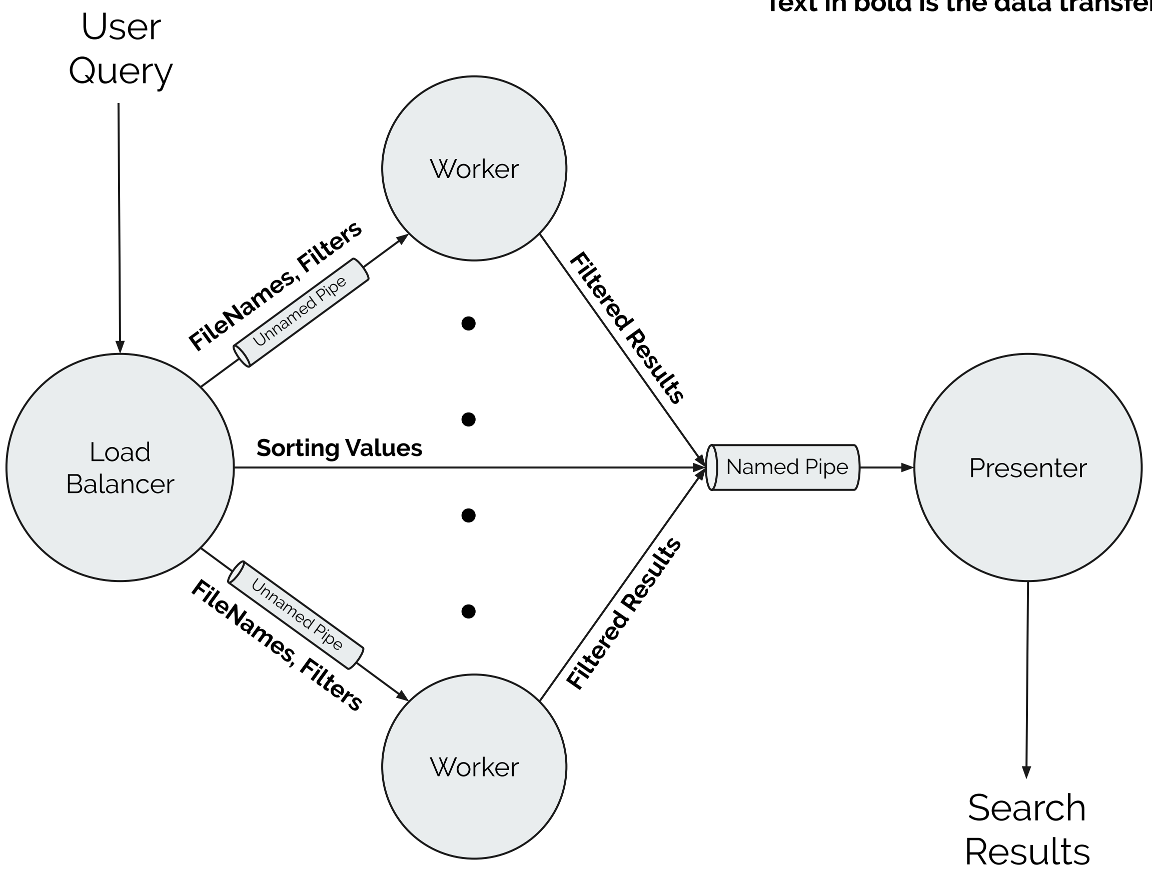
A worker process is responsible for fetching chunks of data and the filters from the load balancer. Then it searches through the chunk data assigned to it according to the filters. With parallelly running multiple worker processes, we can increase the performance and speed of the system. Finally the results from the workers will be sent to the presenter via a named pipe.

The Presenter

This process collects the data that are sent from the worker processes, sorts them according to what the user wants, and prints the sorted results to the user.

The Architecture of The System

Text in bold is the data transferred.



References:

1. http://www.cs.columbia.edu/~aho/cs6998/lectures/11-10-18_Powders_TheOSandPL&C.ppt
2. http://www.cs.fsu.edu/~xyuan/cop5570/lect3_process.ppt
3. http://user.ceng.metu.edu.tr/~ys/ceng334-os/02-processes.ppt
4. https://cse.sc.edu/~rose/311/ppt/ch03.ppt
5. http://poincare.matf.bg.ac.rs/~ivana/courses/ps/sistemi_knjige/pomocno/apue/APUE/0201433079/ch08lev1sec6.html
6. https://fr.slideshare.net/alexandruradovici/sde-tp-4-processus-135693677

Operating Systems Course
University of Tehran
Spring 2020