CS 24: Introduction to Computing Systems

System Calls and Processes on Linux

Last time, we discussed exceptional control flow which enables us to work with multiple processes "simultaneously". This time we will look at system calls (which are a type of trap) more closely. Specifically, we will look at the system calls that control the process life-cycle (e.g., how processes are created or killed).

System Calls

A *system call* is a trap (an intentional exception) that a user program can use to ask the kernel to do something. There are a bunch of system calls that you use regularly. For example, whenever you print something to the console or interact with a file, you're using system calls. Higher level constructs like `printf` are built on top of lower level syscalls like `write`.

There are generally "syscall wrappers" (i.e., C functions that call syscalls) defined by the standard library to make using the syscalls easier, but you can always drop down to assembly to call them as well. For example, we can invoke the `write` syscall in either of the following ways:

- Via a C syscall wrapper:

  ```
  #include <unistd.h>
  int main() {
      write(STDOUT_FILENO, "Hello World", strlen("Hello World"));
  }
  ```

- Via the `syscall` instruction:

  ```
  mov $1, %rax    ; use the write syscall
  mov $1, %rdi    ; write to stdout
  mov $msg, %rsi  ; use string "Hello World"
  mov $12, %rdx   ; write 12 characters
  syscall         ; make syscall
  ```

Notably, the assembly way uses the "number" of the syscall (which, in the case of `write`, is 1) to look up which syscall to make. There is a giant table of these defined in the kernel.

Error Handling

Unfortunately, most system calls can fail. This means (like calls to `malloc`) you must check the return value to see if it is exceptional. Most system calls return -1 on failure. Unless the system call returns `void`, you *must* check its return value and appropriately handle the outcome.

Processes

As we saw last lecture, a *process* is an instance of a running program. On modern systems, processes context switch via the kernel to give the illusion of them all running at the same time. This form of concurrency is called *process-level concurrency*. This time, we wiill explore how Linux implements the process model. In particular, we will see that there are several key syscalls at the root of give managing processes.

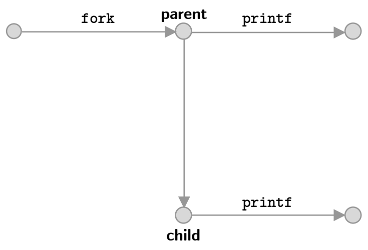
Spawning a New Process

When your system starts, the "init" process is automatically started by the kernel. "init" is the parent of all other processes (on `compute-cpu2` this process is called `systemd`). Many processes are spawned by a *shell* (like `bash`) which is then responsible for creating and eventually "reaping" processes it spawns. As I write this, `pstree` shows the following (partial) hierarchy on `compute-cpu2`:

```
systemd---usr/bin/python---/usr/bin/python---/usr/bin/python
        |                                  |-3*[{/usr/bin/python}]
       ...
        |-sshd---sshd---sshd---bash---vim
        |      |-6*[sshd---sshd---bash]
        |      |-6*[sshd---sshd---sftp-server]
        |      |-sshd---sshd---bash---pstree
       ...
```

But how does a process start a child process? This can be done with the syscall `fork` which creates a new child process that is *almost* identical to the parent. `fork` is **not** a normal function, though! The most important thing to understand about `fork` is that it returns **twice**: once in the parent (with a return value of the child's process ID) and once in the child (with a return value of 0). This is horribly confusing; so, we'll just jump in with an example:

```c
int main(int argc, char* argv[]) {
   pid_t pid = fork();
   if (pid == 0) {
      printf("Child!\n");
   }
   else {
      printf("Parent!\n");
   }
}
```



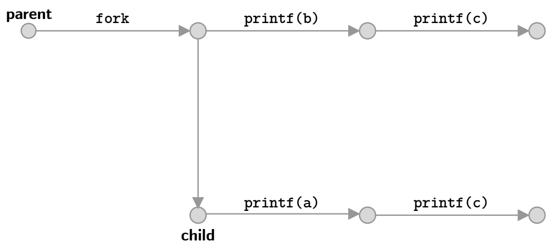
So, what does this code print out? Well...it depends. For this example, there are two possible outputs:

Option 1:	Option 2:
>> Child!	>> Parent!
>> Parent!	>> Child!

The key to this is what we learned last lecture: the kernel is responsible for choosing which process runs and the currently running process might get interrupted at any time! Because of this, the scheduler (a part of the kernel) could choose either the parent or the child to run first after the `fork`. This output is *non-deterministic* which means if you run it multiple times you might get multiple different answers.

```c
int main(int argc, char* argv[]) {
   pid_t pid = fork();
   if (pid == 0) {
      printf("a\n");
   }
   else {
      printf("b\n");
   }

   printf("c\n");
}
```



What does *this* code print out? It still depends! This time, there are actually many different possible outputs:

Option 1:	Option 2:	Option 3:	Option 4:
>> a	>> a	>> b	>> b
>> c	>> b	>> c	>> a
>> b	>> c	>> a	>> c
>> c	>> c	>> c	>> c

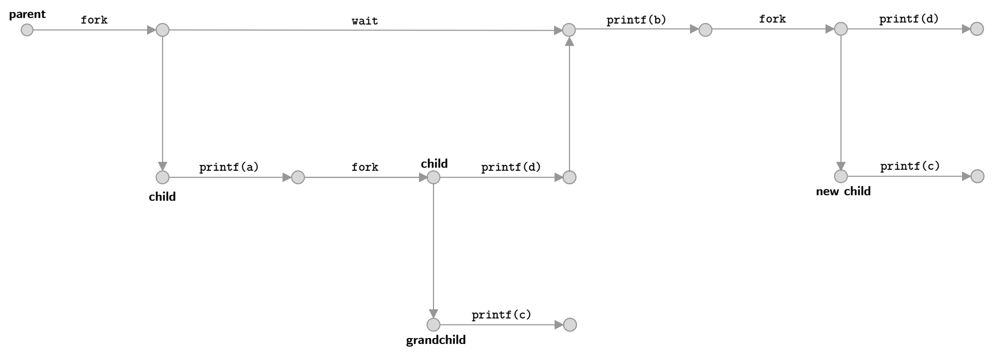
There are also a bunch of impossible orderings. For example, none of the possible outputs starts with "c", because both c's are output after a `printf` has already occurred.

Reaping a Process

Sometimes, it can be useful to stop execution until a child process is finished executing. For example, in a shell, if we `fork` a child, we want to wait for the result for the user before giving another prompt. The `wait` function helps us accomplish exactly this. It suspends execution until one of its children terminates. This forces fewer orderings to be possible, because it imposes further constraints on the order processes run. Consider another example:

```c
int main(int argc, char* argv[]) {
   pid_t pid = fork();
   if (pid == 0) {
      printf("a\n");
   }
   else {
      wait(NULL);
      printf("b\n");
   }

   pid_t pid2 = fork();
   if (pid2 == 0) {
      printf("c\n");
   }
   else {
      printf("d\n");
   }
}
```



Loading a Program

Once a new process has been `fork`ed, we need to load the program we *actually* want to run. This is done with the `exec` family of functions which take a variety of parameters usually including the `argv` for the program to load. One of the more useful places to use `exec` is in a shell like `bash`. Consider the following *very simple* shell as an example:

```c
int main() {
   while (1) {
      // Print prompt and get user input
      printf("mysh> ");
      fgets(cmdline, MAXLINE, stdin);

      // Parse user input into argv
      char *argv[MAXARGS];
      parse_line(cmdline, argv);

      // Spawn a child process and exec the program
      // specified by the user
      pid_t pid = fork();
      if (pid == 0) {
         execvp(argv[0], argv);
      }

      // Wait until the child program is done.
      wait(NULL);
   }
}
```

Notably, we are not checking error codes (and we really should be!). Also, we have not provided the implementation of `parse_line` which splits a string by spaces.