CS 333: Operating Systems Lab
Autumn 2018

Lab 5: inheritance and pigeons

Goal

In this lab we explore xv6 does memory management and implement a simple signal handling service.

Before you begin

- Download the source code for xv6 from the following url:
  `http://www.cse.iitb.ac.in/~puru/courses/autumn18/labs/xv6-public.tar.gz`
  Make sure you read the README to understand how to boot into a xv6-based system.
- Download, read and use as reference the xv6 source code companion book.
  url: `http://www.cse.iitb.ac.in/~puru/courses/autumn18/labs/xv6-rev10.pdf`
- Implement all custom system calls in the file `sysproc.c`

1. no init yet!

In this exercise, you will change a default functionality of xv6. An *orphan* (child) process in one whose parent process has finished or terminated, but the (child) process itself continues to execute. In xv6 (and other Unix-like operating systems), an orphaned process will be immediately adopted by a special process (usually the `init` process, with `pid` 1). We will change this feature such that an orphaned process will be adopted by its closest ancestor instead of the `init` process. As a special case, the closest ancestor can be the init process as well.

The source file `ancestor.c`, as part of the lab archive, can be used to test implementation of this feature. Add this files/program to xv6 and also check sample outputs for more details. You will need to implement a new system call `getppid()` to get pid of the parent process.

Hint: *All that exits, does not just exit.*

2. process pigeons

Signals, as their name implies, used to signal something. Signals are user-level mechanisms for processes to communicate via events (signals). e.g., `SIGINT` tells your program that someone tries to interrupt it with `CTRL-C`. In this exercise we will add a feature to xv6 that enables inter-process communication through signals. This might be useful as a way to interrupt other processes and create a higher layer protocol for data transfer. More generally, we will implement a primitive form of user-level signal handlers.

1. Add a new `signal_process(pid, type)` system call, where *pid* is the process id to which signal is to be sent and *type* is the type of the signal being sent which can be either of `{PAUSE, KILL, CONTINUE}` If an user-level program calls `signal_process(pid, type)`, then a signal is *delivered* to the process *pid* immediately and the corresponding process handles it as soon as it gets scheduled next time. The type of the signal determines the action to be taken, i.e., *PAUSE* does not execute/schedule the process, *KILL* terminates the process, and *CONTINUE* release a paused processes to continue execution. Note that a paused process make no progress (is never scheduled to execute) and a process continues from where it left off after a pause.
2. A program `signal-test.c` (provided in the archive) spawns child processes and use the above system call for testing. Go through the sample outputs and code to understand its behaviour. Add this file to xv6 commands to validate your code. Make changes to xv6 files so that it runs with the `signal-test.c`.
3. Multiple signals to a process get overridden by the last signal sent. You do not have to handle a queue of signals. (At least in this lab).

4. **Hints:** Signals are set in the system call and are to handled when is process is scheduled or to be scheduled.

3. wait a sec!

This part is an addition to the previous task. Add a duration parameter to *PAUSE* signal we can specify duration to pause a process. Implement a system call `pause(pid, duration)` that pauses process with id `pid` for `duration` ticks (tick is the unit of time that after which a timer interrupts the CPU, periodically). A program `timed_pause.c` is provided to test this feature. Add this to xv6 and see sample outputs for more details.

Hints: Look at the system call `uptime` and also try to understand happenings in the `trap()` function in the file `trap.c`.

4.

For each of the above exercises a sample test program is given for testing. Setup additional test cases by extending/modifying them for better/improved testing. e.g., KILL signal to a terminated process, signal overriding, terminate signal to parent process, effect of different pause intervals etc.

Submission Guidelines

- All submissions via moodle. Name your submission as: **<rollno_lab5>.tar.gz** item The tar should contain the following files in the following directory structure:

  <rollno_lab5>/
  |___ sysproc.c
  |___ <other modified files in xv6>
  |___ ancestor.c
  |___ ancestor1.c \\ other test cases
  |___ ancestor2.c \\ with description
  |___ ...
  |___ signal-test.c
  |___ signal-test1.c \\ other test cases
  |___ signal-test2.c \\ with description
  |___ ...
  |___ timed_pause.c
  |___ timed_pause1.c \\ other test cases
  |___ timed_pause2.c \\ with description
  |___ ...

- Submissions will be evaluated based on code submitted and correctness across different types of test cases. Make sure all files updated and required for compilation are part of the submission.
- Submissions via moodle.
- **Deadline: Monday, 20^{st} August 2018 - 05:00 PM.**