CS 333: Operating Systems Lab
Autumn 2018

Lab #10: trapped in the trapframe

Goal

In this xv6-based lab we will implement support for handling signals in user-space and also toy around with the CPU scheduler.

1. Support for (custom) user space signal handlers

Extend xv6 kernel to add support for handling signals via user space functions.

- Each process should be able to register its own signal handler, and should be able to send signals to any process. For simplicity, assume a process can register a single signal handler function and a signal type is passed as an argument for per-signal type handling. Once a user space handler is registered, it is responsible to handle all signals to the process. A default signal handler exists for each signal in the kernel and is executed if a process has not registered a custom handler.
- When a signal is pending for a process, the operating system should execute the signal handler. If a signal handler is registered by the process, it should schedule the user space function and then return to the kernel to setup return to the original return address in the user space (unless the process decides to exit as part of signal handling).
- The default signal handler prints `PID` of the process and signal type of the pending signal. If the pending signal is of the type '1', it should kill the process. Otherwise, it allows the process to continue the execution.
- Signals should be handled when a process is interrupted from user space to kernel space and is being switched back to user space. The source code location for this is just after system call and timer interrupt processing in the trap handler.

  Note that the timer interrupt may occur during system call processing (nested interrupts) in which the signal should not be handled.

Implement the following system calls to build this functionality, along with additions to `struct proc` to store signal related information.

- `int signal(char *handler)`, register a signal handler. Argument to system call is address of the handler function of the process.
- `int signal_send(int pid, int sig_type)`, sends a signal of type, `sig_type`, to the process identified by `pid`.
- `int signal_ret(void)`, the system call to intimate end of user space handler. The kernel restores state of process and returns to execution from the original interruption point in user space of the process when it was interrupted.

  This system call should be executed as the last statement in the signal handler, unless the handler calls `exit` to terminate the process.

Hints:

- To execute the signal handler when a process returns to user space, setup the `trapframe` to start execution at the user space signal handler on return to user space.
- Passing the signal type argument to the signal handler function requires modifying the user space stack of the corresponding process. Need to understand how arguments and return value are passed to a function, and where they are stored on the stack.
- When the control comes back to the kernel after executing the signal handler (when `signal_ret()` is invoked), recreate the `trapframe` to start execution of the process from its original return address location.

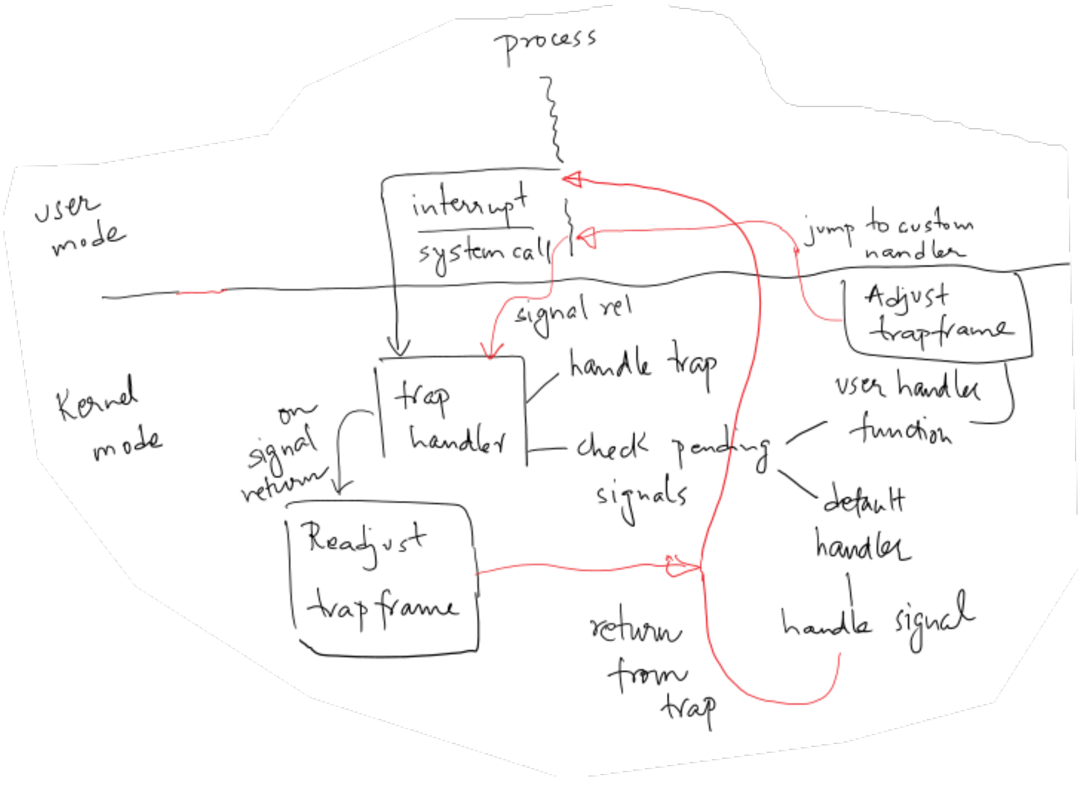

Figure 1: A sequence of operations for implementing support for user space signal handlers.

2. Game of processes

Time logging

1. Add the following variables related to time logging in `struct proc` and update them at appropriate places. Look up the `uptime` system call implementation to understand how to use system clock ticks. For the purpose of this lab, time is measured in terms of ticks.
   (a) `arrival_time`: Time at which the process was forked/created.
   (b) `first_scheduling_time`: Time at which the process was scheduled for the first time.
   (c) `run_time`: Duration for which the process executed the CPU. Note that the a process can be RUNNABLE for a long time, but its actual time on the CPU can be lesser. This variable captures total time spent on the CPU.
   (d) `exit_time`: Time at which the process completes the execution and calls the `exit()` function. Note that process can languish in `ZOMBIE` state for a non-deterministic amount of time, update this variable somewhere in the *exit* function before the non-determinism kicks-in.
   (e) `priority` The priority associated with the process, to be used by the scheduler. Default value is 1.

2. Print the following information when a process exits (in the `exit` function).
   ```
   Exiting.  pid=5, prio=5, AT=363, IWT=6, FST=369, ET=1136, RT=84, TWT=690.
   ```
   Here, `AT` = arrival time, `IWT` = Initial wait time, `FST` = First schedule time, `ET` = Exiting time, `RT` = Run time, and `TWT` = Total wait time.
   `IWT` is the delay to schedule a process after it is created. `TWT` is the time for which the process was not running in the CPU, and was either blocked or not-scheduled.

Attaching priority to processes

1. Add a new system call to xv6 to set process priorities - `setprio()`.
   When a process calls `setprio(n)`, the priority of the process should be set to the specified value. The priority can be any positive integer value, with higher values denoting higher priority.

2. Add another system call `getprio()` to read back the priority of calling process, in order to verify that it has worked.

3. **fork2(int priority):**
   Add a new system call `fork2` which sets the priority of the child process. The implementation of `fork2` system call should be the same as existing `fork` function, and in addition should initialize priority of the child process. Copy/use existing `fork` implementation for this.

Test your implementations using the given test program and verify that the logging works correctly. Note we have not use changed the scheduler yet to use the priority.

A simple priority based scheduler

Modify the xv6 scheduler to use the priority of each process to choose the next process to schedule.

- The scheduling policy picks the highest priority process which can be scheduled and schedules it to run.
- If there two or more process with same priority, it selects any one of them.
- Once a process is scheduled, the scheduler should schedule the process again and again till the process completes, or blocks, or a higher priority process arrives.
- If all processes have same priority, this policy roughly approximates FCFS.
- Test your implementation using the testcases provided. Additionally, build your own testcases and outputs.

Note: For implementation of this task, update only the `scheduler` function.

Submission Guidelines

- All submissions via moodle. Name your submission as: **<rollno-lab10>.tar.gz**
- The tar should contain the following files in the following directory structure:
```
<rollno-lab10>/
|__<xv6>/
|____<all files in the xv6>/
|__outputs/
|____part1/
|____<programs and outputs of all testcases>
|____part2/
|____<programs and outputs of all testcases>
```
- We will evaluate your submission by reading through your code and executing it over several test cases.
- **Deadline: Monday, 22nd October 2018 - 05:00 PM**.

3. Extra cheese

Extend xv6 kernel to support for creating user threads. An user thread is like a process but shares the memory with parent process/thread and start execution from the thread function. For simplicity, assumes only parent process creates new threads.
Implement the following system calls,

- `int thread_create(void *start_function, void *arg)`, create an user thread. The user thread executes the function `star_function` and the argument `arg` should be passed to the starter function.
- `int thread_join(int pid)`, suspends execution of the calling process until the target thread terminates, unless the target thread has already terminated. Implementation of this system call is similar to `wait()` system call, but it waits for a specific process/thread to exit.

Hints:

- `exit` system call should be executed as the last statement in each starter function.
- Allocate separate userspace stack of one page size for each thread and initialize the stack with arguments to the starter function and the return address (set to zero).
- Dynamic memory allocation should be done carefully as both parent and other threads shares the memory. Need more attention on all accesses to `proc->sz`.