CMSC 412, Spring 2017
Project 2
Due: Thursday, March 2nd, 2017 5:00 pm

Overview

Project 2 requires you to implement signals and signal handlers. A signal delivered to a process will interrupt what it is doing to process the event.

Quick Links

- Download Project Source:
  `git clone https://gitlab.cs.umd.edu/cmsc412_sp2017/spring2017.git`
- Submission: run make submit.

Signals

A signal is an inter-process communication mechanism that involves causing one of a small array of functions to be invoked in another process. Each process can use the "signal" system call to manipulate a table that lists signal handlers (function pointers) to be invoked at the request of another process. To send a signal, a process calls the "kill" system call with the pid of the target and the signal number to send. The signal number represents the index into the table of signal handlers in the target process. The kernel will then arrange for the signal handler function to be called within the process. The slightly tricky part is to ensure that when the signal handler returns, control is passed back to the kernel and the process can resume from wherever it was. To accomplish this task, we define a "return signal" call that will be invoked as the signal handler returns. In implementing signals, you will need to arrange for the process to have both a context when executing the signal handler and a saved context that the signal handler will return to after execution. In preparation, we describe context switching next.

1. Background

1.2. The Kernel Stack (Thread Stack)

The *kernel stack* is the stack used by a Geekos kernel thread while it is executing in the kernel. As usual, the kernel stack stores the local variables used by the kernel thread while running GeekOS kernel routines. This could be for kernel threads performing system processes, like the reaper thread, or it could be for kernel threads implementing user processes, executing system calls on their behalf.

When performing a context switch, the current state (or "context") of a thread is saved on its kernel stack. This state consists of the current values of (most of) the registers. The fields *stackPage* and *esp* defined in the Kernel_Thread structure, specify where the thread's kernel stack is (*esp* is the kernel stack pointer). This way, when a thread is to be context-switched to, the current thread switches to the new thread's stack, and then restores the context.

Stacks grow downward, from numerically higher addresses to numerically lower addresses.

1.3. User Processes

User processes have a kernel stack, for calls within the kernel when the kernel is running on the process's behalf, and a user stack, for local variables while running user-level code.

To prepare a user process to be run for the first time, GeekOS pushes the same state on the kernel stack that it would have had, if it has been previously running and interrupted in a systme call or by being preempted. The state pushed onto the kernel stack includes the following:

1. Context Information: this includes (almost) all the registers used by the user (GS, FS, ES, DS, EBP, EDI, ESI, EDX, ECX, EBX, EAX)
2. Error code and Interrupt number.
3. Program counter: this contains the value that should be loaded into the instruction pointer register (EIP). Initially, when the user process is about to run, GeekOS pushes the entry point for the process and this value will be subsequently loaded into EIP.
4. Text selector: this is the selector corresponding to the code segment (CS) of the process.
5. The EFlags register.
6. User stack data selector and user stack pointer: these point to the location of the *user stack*.

When the thread is scheduled for the first time, these initial values will be loaded into the corresponding processor registers and the thread can run. The initial stack state for a user thread is described in the following figure (check `Setup_User_Thread()` in `src/geekos/kthread.c`):

User Stack Data Selector (data selector)	**User Stack Location**
User Stack Pointer (to end of user's data segment)	
Eflags	**Interrupt_State**
Text Selector (code selector)	
Program Counter (entry addr)	
Error Code (0)	
Interrupt Number (0)	
EAX (0)	
EBX (0)	
ECX (0)	
EDX (0)	
ESI (Argument Block address)	
EDI (0)	
EBP (0)	
DS (data selector)	
ES (data selector)	
FS (data selector)	
GS (data selector)	

The items at the top of this diagram (in high memory) are pushed first, the items at the bottom (in lower memory) are pushed last (i.e., the stack grows downward). In this figure, the contents of the stack, not including the user stack location, are defined in `struct Interrupt_State` in `geekos/int.h`. This is the structure you're familiar with from modifying system calls in `syscall.c`.

1.4. The User Stack

The user stack selector is the same as the data selector: both the stack and the data segment occupy the same memory segment. The user stack starts at the high end of the data segment and grows downward. Initially, the user stack pointer should indicate an empty stack. So it points to the end of the data segment.

When switching from kernel mode to user mode, the kernel calls `Switch_To_User_Context()` in `src/geekos/user.c`. Switching the context includes the following steps:

- Save the context of the currently executing thread
- Switch to a new address space by loading the LDT of the new thread (`ldtSelector` of User_Context) using the `lldt` assembly instruction (see `Switch_To_Address_Space()` in `src/geekos/userseg.c`).
- Move the stack pointer up one page via `Set_Kernel_Stack_Pointer()`.

2. Project Requirements

This project will require you to make changes to several files. In general, look for the calls to the `TODO()` macro. These are places where you will need to add code, and they will generally contain a comment giving you some hints on how to proceed. There are two primary goals of this project:

- Add the code necessary to handle signals
- Implement a collection of system calls to setup and call the signals.

2.1. Signals

In this project, you must implement signal handling and delivery for the following four signals (defined in include/geekos/signal.h):

SIGKILL:
: This is the signal sent to a process to kill it. You should write the handler for this signal such that it results in the same behavior as in project 1's Sys_Kill. The process is not permitted to install a signal handler for SIGKILL.

SIGUSR1, SIGUSR2:
: "User-defined" signals with no pre-determined purpose. These will be sent only by other processes.

SIGALARM:
: This signal is sent to the process when an alarm expires.

Further Reading: More information about signal handling can be found in Chapter 4 of the text.

2.2. System Calls

In this project, you will implement five system calls; the user-space portion of these calls is defined for you in `src/libc/signal.c`:

Sys_Signal:
: This system call registers a signal handler for a given signal number. The signal handler is a function that takes the signal number as an argument (it may not be useful to it), processes the signal in some way, then returns nothing (void). If called with SIGKILL, return an error (EINVALID). The handler may be set as the pre-defined "SIG_DFL" or "SIG_IGN" handlers. SIG_IGN tells the kernel that the process wants to ignore the signal (it need not be delivered). SIG_DFL tells the kernel to revert to its default behavior,

which is to terminate the process on KILL, USR1, and USR2, and to discard (ignore) SIGALARM. A process may need to set SIG_DFL after setting the handler to something else.

Sys_RegDeliver:

The signal handling infrastructure requires a special "trampoline" function to be implemented. This "trampoline" invokes the system call `Sys_ReturnSignal` (see below) at the conclusion of signal handler. This system call is invoked by `Sig_Init` when called by the _Entry function in `src/libc/entry.c`; this function is invoked prior to running the user program's main(). It returns 0 on successes and EINVALID if the passed function is invalid.

Sys_Kill:

In project 1, this system call took as its argument the PID of a process to kill. In this project, it will be used to send a signal to a certain process. So in addition to the PID, `Sys_Kill` will take a signal number: one of the four defined above. It should be implemented as setting a flag in the process to which the signal is to be delivered, so that when the given process is about to start executing in user space again, rather than returning to where it left off, it will execute the appropriate signal handler instead.

Sys_ReturnSignal:

This system call is not invoked by user-space programs directly, but rather is executed by some stub code at the completion of a signal handler. That is, Sys_Kill/Send_Signal sends process P a signal, which causes it to run its signal handler. When this handler completes, we will have set up its stack so that it will "return" to the trampoline registered by `Sys_RegDeliver`. This trampoline wil invoke `Sys_ReturnSignal` to indicate that signal handling is complete.

Sys_Alarm:

Sets an alarm to happen mili-seconds in the future. Calling Sys_Alarm a second time before the first alam goes off replaces the first alarm by the second one. If mili-seconds is 0, it cancels any pending alarm. Alarms happen only once (i.e. they are not periodic). When an alarm happens, a SIGALARM is delivered to the process. An invalid (negative time) should not change the alarm and return an EINVALID. Sys_Alarm return 0 on success.

Termination

If the default handler is invoked for SIGKILL, SIGUSR1, or SIGUSR2, `Print("Terminated %d.\n", g_currentThread->pid);` and invoke Exit. The default handler for SIGALARM is to do nothing.

Reentrancy and Preemption

Sending a signal should appear as if setting a flag in the PCB about the pending signal; the signal handler should not necessarily be executed immediately. In particular, if the process is executing a signal handler, it cannot start executing another signal handler. Further, multiple invocations of kill() to send the same signal to the same process before it has a chance to handle even one will have the same effect as just one invocation of kill(). The delivery order of pending signals is not specified (they need not be queued).

2.3. Signal Delivery

To implement signal delivery, you will need to implement (at least) five routines in `src/geekos/signal.c`:

Send_Signal:

this takes as its arguments a pointer to the kernel thread to which to deliver the signal, and the signal number to deliver. This should set a flag in the given thread to indicate that a signal is pending. This flag is used by `Check_Pending_Signal`, described next.

Check_Pending_Signal:

this routine is called by code in `lowlevel.asm` when a kernel thread is about to be context-switched to. It returns true if the following THREE conditions hold:

1. A signal is pending for that user process.
2. The process is about to start executing in user space. This can be determined by checking the `Interrupt_State`'s CS register: if it is not the kernel's CS register (see `include/geekos/defs.h`), then the process is about to return to user space.
3. The process is not currently handling another signal (recall that signal handling is non-reentrant).

Set_Handler:

use this routine to register a signal handler provided by the `Sys_Signal` system call.

Setup_Frame:

this routine is called when `Check_Pending_Signal` returns true, to set up a user process's user stack and kernel stack so that when it starts executing, it will execute the correct signal handler, and when that handler completes, the process will invoke the `Sys_ReturnSignal` system call to go back to what it was doing. IF instead the process is relying on SIG_IGN or SIG_DFL, handle the signal within the kernel. IF the process has defined a signal handler for this signal, this function will have to do the following:

1. Choose the correct handler to invoke.
2. Acquire the pointer to the top of the *user stack*. This is below the saved interrupt state stored on the *kernel stack* as shown in the figure above.
3. Push onto the *user stack* a snapshot of the interrupt state that is currently stored at the top of the *kernel stack*. The interrupt state is the topmost portion of the kernel stack, defined in `include/geekos/int.h` in struct `Interrupt_State`, shown above.
4. Push onto the *user stack* the number of the signal being delivered.
5. Push onto the *user stack* the address of the "signal trampoline" that invokes the `Sys_ReturnSignal` system call, and was registered by the `Sys_RegDeliver` system call, mentioned above.
6. Change the current *kernel stack* such that (notice that you already saved a copy in the *user stack*)

   (1) The *user stack* pointer is updated to reflect the changes made in steps 3-5.
   (2) The saved program counter (eip) points to the signal handler.

Complete_Handler:

this routine should be called (by your code) when the `Sys_ReturnSignal` call is invoked, to indicate a signal handler has completed. It needs to restore back on the top of the kernel stack the snapshot of the interrupt state currently on the top of the user stack.

You will also need to write a routine to implement the Alarm system call. There is already an alarm system in the kernel (implemented in the file src/geekos/alarm.c.). You should be able to make calls to this alarm code to setup an alarm at the requested number of mili-seconds in the future. The callback from the alarm should cause the SIGALARM signal to be delivered to the process that requested it.