# Project 2
# Signals

**Deadline: October** 10, Monday **at 11**.00 **PM**
**Minimum Requirement: 9 public tests**

# Test Breakdown

test breakdown:
12 public tests
1 release test
7 secret tests
5 points each except test_built

# Overview

Project 2 requires you to implement signals and signal handlers.

# Introduction to signals

- A signal is an inter-process communication mechanism that allows one process to invoke a signal-handler function in another process.
  - A process can send a signal to another process.
  - The process receiving the signal will, at some point, stop what it is doing, execute a signal-handler function, and then resume what it was doing.
- There are several signals that one process can send to another, each identified by a number.
  - Each process maintains a table of signal handlers (as function pointers), one for each signal the process can handle.
  - The signal number is used as an index into the table of signal handlers in the target process.
- When process A sends a signal to B, the kernel must create a new context that causes B to execute its signal handler, then return B to its original context.

# Things to be implemented

**In syscall.c**

Sys_Signal

Sys_RegDeliver

Sys_Kill

Sys_ReturnSignal

Sys_WaitNoPid

**In signal.c**

Check_Pending_Signal

Setup_Frame

Complete_Handler

# Sys_Signal(struct Interrupt_State *state)

- A process calls "signal" to indicate what handler function should run when it receives a signal.
- This system call handler registers a signal handler for a given signal number.
- Recommend to check signal.h to know the structure
- Arguments: signal handler, signal number
  - state->ebx: pointer to handler function
  - state->ecx: signal number
- EINVALID ( can use IS_SIGNUM to check if its a valid signal

- **Define & initialize a signal handlers table**
  - **SIG_IGN**: tells the kernel the process wants to ignore the signal.
  - **SIG_DFL**: perform the default behavior of the process.
  - **Default behaviors**:
    - SIGKILL, SIGPIPE, SIGUSER1, SIGUSER2: terminate
    - SIGCHLD: ignore

**Include/geekos/signal.h**

```
/* Signal numbers */
#define SIGKILL  1              /* can't be handled by users */
#define SIGUSR1  2
#define SIGUSR2  3
#define SIGCHLD  4
#define SIGALARM  5
#define SIGPIPE  6

/* The largest signal number supported */
#define MAXSIG   6

/* Macro to determine whether a number is a valid signal number */
#define IS_SIGNUM(n) (((n) > 0) && ((n) <= MAXSIG))

/* Definition of a signal handler */
typedef void (*signal_handler) (int);

/* Default handlers */
#define SIG_DFL  (signal_handler)1
#define SIG_IGN  (signal_handler)2
```

# Return Signal and Trampoline

- When a signal is sent to a process, the kernel arranges for the signal handler function to be called within the process.
- When the signal handler returns, control must pass back to the kernel, which arranges for the process to resume from wherever it was.
- To accomplish this, we define a system call **ReturnSignal** that is to be invoked by the process when its signal handler returns.
- However, we cannot count on every signal handler to call ReturnSignal explicitly.
    - Instead, the compiler will include a user-side function, called the **"Trampoline,"** in every executable and set the return address of each signal handler to go to Trampoline.
- In userspace: calls Sys_ReturnSignal syscall

# Sys_ReturnSignal(struct Interrupt_State *state)

- Complete signal handling for this process
- return state->eax;

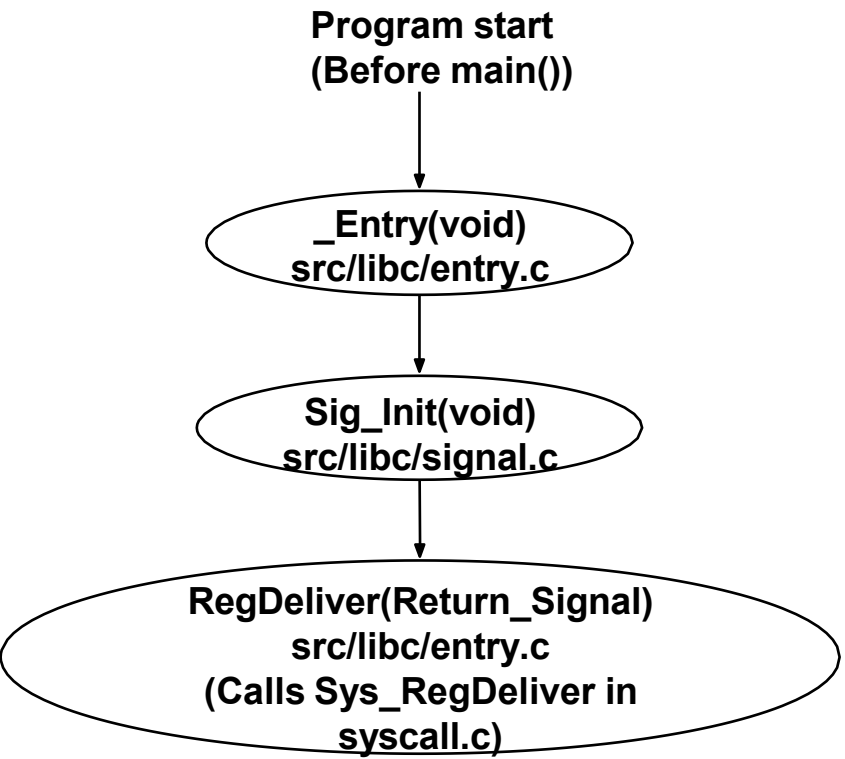
# How does the kernel know where trampoline is?

- Sys_Regdeliever is used to register the address at the beginning of each user program.

# Sys_RegDeliver(struct Interrupt_State *state)

- Must be invoked before a user process begins to execute.
- Provides the kernel with the address of the trampoline function, as described for Sys_ReturnSignal.
- The user code that calls Sys_RegDeliver and Sys_ReturnSignal is already written for you in libc.
- Your task is :
  - To Store the values of the Return_Signal(trampoline) address in User_Context

# Purpose:
# Register trampoline




Sys_RegDeliver registers the Return_Signal trampoline function for current process.

```
/*
 * Entry point.  Calls user program's main() routine, then exits.
 */
void _Entry(void) {
    struct Argument_Block *argBlock;

    /* The argument block pointer is in the ESI register. */
    __asm__ __volatile__("movl %%esi, %0":"=r"(argBlock));

    /* Initialize the signal handling trampoline */
    {
        int ret = Sig_Init();
        if(ret != 0)
            Exit(ret);
    }

    /* Call main(), and then exit with whatever value it returns. */
    Exit(main(argBlock->argc, argBlock->argv));
```

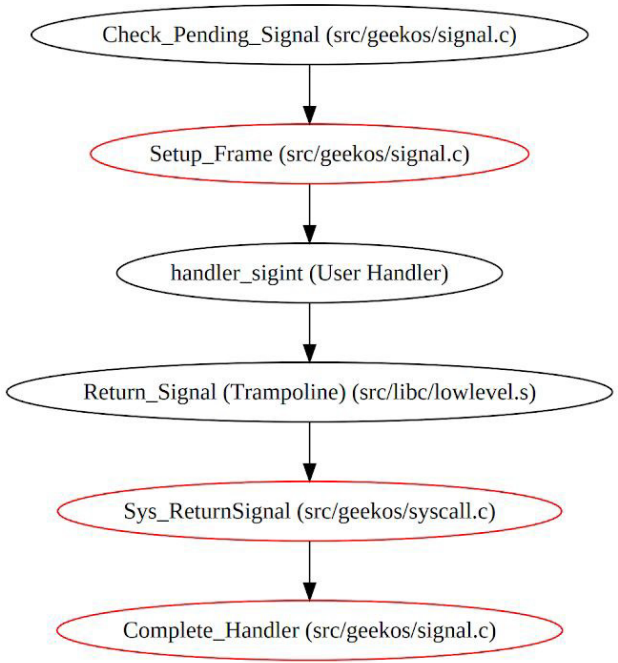
```
/* Should be called when the program starts up */
int Sig_Init(void) {
  return RegDeliver(Return_Signal);
}
```

# Sys_Kill(struct Interrupt_State *state)

- It will be used to send a signal to a certain process.
- If called with a different signal number, it should return EINVALID.
- Lookup_Thread (src/geekos/kthread.c): get the target thread.
- 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.
- This is a kernel process; userspace isn't allowed to send signals to it. Set the appropriate error conditions.

# Sys_WaitNoPID(struct Interrupt_State *state)

- This call allows a parent to collect the exit status for a child that has terminated and become a zombie, without knowing its PID or going into a blocking wait.
- Argument: a pointer to an integer
- Reaps one zombie child process per call
- Places the exit status of the child process in the memory location the argument points to
- Returns the pid of the zombie.
- If no dead child process, return ENOZOMBIES


Check_Pending_Signal (src/geekos/signal.c)
Setup_Frame (src/geekos/signal.c)
handler_sigint (User Handler)
Return_Signal (Trampoline) (src/libc/lowlevel.s)
Sys_ReturnSignal (src/geekos/syscall.c)
Complete_Handler (src/geekos/signal.c)

# Check_Pending_Signal(struct Kernel_Thread *kthread, struct Interrupt_State *state)

Check_Pending_Signal:

This routine is called by code in lowlevel.asm when a kernel thread is about to be context-switched in. 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 equal to the kernel's CS register (see include/geekos/defs.h), then the process will return to user space.
3. The process is not currently handling another signal (recall that signal handling is non-reentrant).

# …Continued

In Kernel Space

Run every time when a process returns from kernel model to user mode

Basically checks if the process

- Has pending signal AND
- It's the right time to handle the signal
- Start executing in user space

•not handling another signal If

we passes these tests, then

- Iterate through all signal flags and set busy flag.
- Check for error conditions

# Setup_Frame()

- Called when switching to a user process, if Check_Pending_Signal returns true.
- Sets up a user processes user stack and kernel stack so that

  (1) when the process returns to user mode, it will execute the correct signal handler, and

  (2) when that handler completes, the process will return to the trampoline function so that it can go back to what it was doing.

# ...Continued

- If the process has defined a signal handler for this signal, function Setup_Frame 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 available when you cast the saved Interrupt_State (on the *kernel stack*) to a User_Interrupt_State.
  3. Push onto the *user stack* a snapshot of the interrupt state that is currently stored at the top of the *kernel stack*.
  4. Push onto the user stack the number of the signal being delivered.

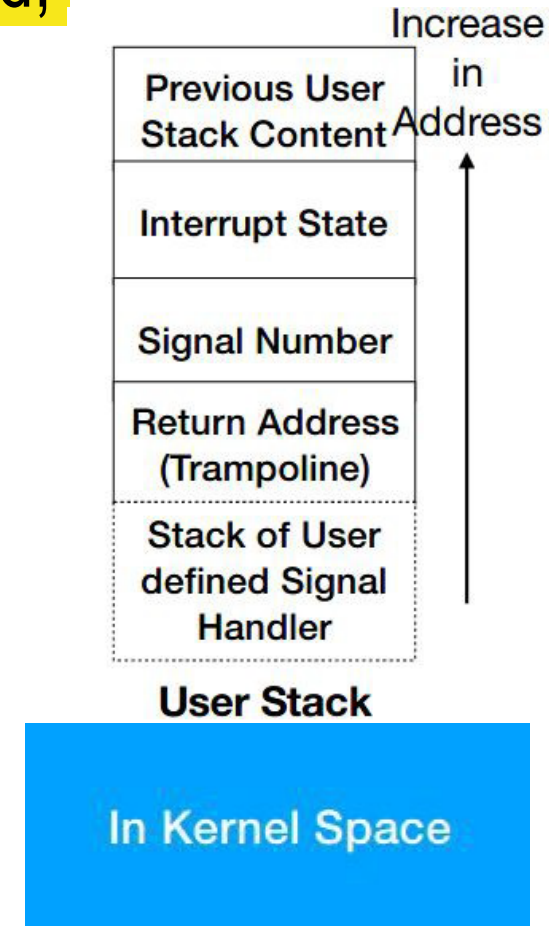
# ...Continued

5. Push onto the *user stack* the address of the trampoline (which invokes the Sys_ReturnSignal system call handler). The trampoline was registered by the Sys_RegDeliver system call handler, mentioned above.
6. Now that you have saved a copy of the *kernel stack*, change the original User_Interrupt_State such that
    i. The *user stack* pointer is updated to reflect the changes made in steps 3--5.
    ii. The saved program counter (eip) points to the signal handler.

# Setup_Frame(struct Kernel_Thread *kthread, struct Interrupt_State *state)

Find the registered handler pointer

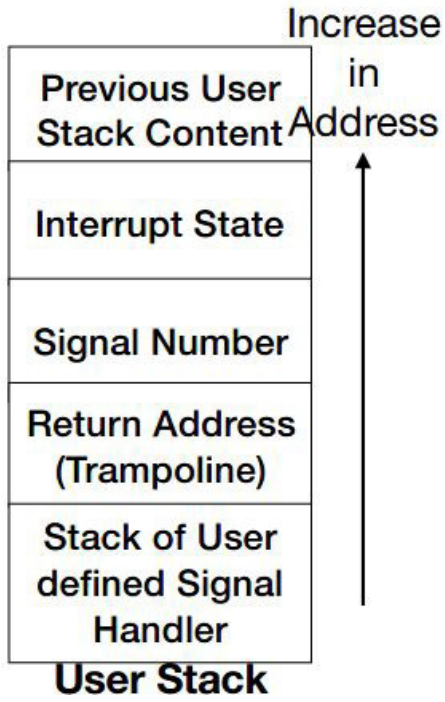
- (cast to User_Interrupt_State to find user esp)
- Push user interrupt state to USER stack
- Push signal number to USER stack (argument to user defined signal handler pointer)
- Push trampoline to USER stack
- Jump to user defined signal handler pointer (state→eip)



# handle_sigint()

User supplied function

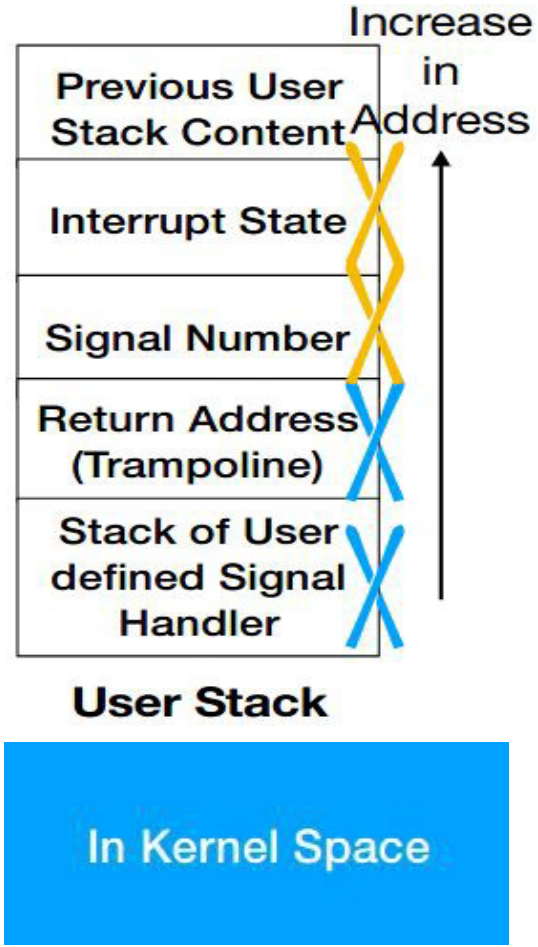
- Has function signature void signal_handler(int signum);




In User Space

# Complete_Handler(struct Kernel_Thread *kthread, struct Interrupt_State *state)

- This routine should be called when the Sys_ReturnSignal call handler is invoked (after 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.
  - Reverse the operations in setup_frame
  - Restore interrupt_state on user stack to interrupt_state on current thread
  - Remove signal number
- No need to pop the pushed return_signal_function here; it's been popped by the ret instruction at the end  of the signal handler (to call the libc function Return_Signal).


Increase in Address
Previous User Stack Content
Interrupt State
Signal Number
Return Address (Trampoline)
Stack of User defined Signal Handler
User Stack
In Kernel Space

# Other Functions to Modify

1. Sys_Fork :  copy signal handlers, trampoline
2. Pipe_Write: send SIGPIPE
3. Exit: send SIGCHLD

- You have a lot of freedom for this project
- Feel free to add additional fields to existing structs
- kernel_thread for kernel space related data
- user_context for user space related data

# Demo: sigpipe.c

Remember the flow:

- Before main(): Sys_RegDeliver -> Register trampoline function
- In main():
  - Check_Pending_Signal is always running
  - If user used Sys_Signal to change a signal handler, when a signal is received (Check_Pending_Signal returns true):
    i. Setup_Frame (Kernel mode)
    ii. User defined signal handler (User mode)
    iii. Trampoline (User mode)
    iv. Sys_returnSignal (Kernel mode)
    v. Complete_Handler (Kernel mode)
    vi. Back to user program (User mode)
  - If user didn't change the signal handler
    i. Handle the signal in kernel (Terminate or ignore)
    ii. Back to user program (User mode)

# Demo: sigpipe.c

What this test is doing:

- The parent:
  - **Signal(sigpipe_handler, SIGPIPE)**: It will register a signal handler sigpipe_handler for SIGPIPE. So when the parent received SIGPIPE, it will execute the code in sigpipe_handler.
  - Will try to read from the buffer. Then a read end will be closed.
  - Wait for the child to finish
- The child:
  - Close a read end.
  - Tries to write a lot of stuff to the buffer.
- Until this step, the two read ends should be closed. If any of the child or the parent tries to write to the buffer, a SIGPIPE will be raised before returning EPIPE.
- For the parent, SIGPIPE should invoke sigpipe_handler. For the child, SIGPIPE should perform its default behavior.

```
$ sigpipe
In Sys_RegDeliver

Parent should handle sigpipe, child should terminate on sigpipe.
original pid=9
parent n=1, global=1, child_pid=10, my_pid=9

In Sys_Signal, Trying to register signal number: 6
child n=1, global=1, child_pid=0, my_pid=10
In Setup_Frame. Find pending signal: 6
Terminated 10.
expected to reap child_pid 10, Wait returned 262
In Setup_Frame. Find pending signal: 6
GOOD: sigpiped the parent!

In Sys_returnSignal

In Complete_Handler
```

parent

child

parent