Minimum Requirements: 9 public tests

Public tests – 12 tests | 57 points

Release tests – 1 tests | 5 points

Secret tests – 7 tests | 35 points

# 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
- Insignal.c
  - Check_Pending_Signal
  - Setup_Frame
  - Complete_Handler

# SYS_SIGNAL

- A process calls "Signal" to indicate what handler function should run when it receives the 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 (in user)
    - state->ecx: signal number
- Create your own data structure to keep track of the handler function.
- Error: EINVALID if the signal number is not a valid signal (1-6)
    - use IS_SIGNUM to check if it's a valid signal.

# SIGNAL.H

- Default handlers:
  - 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

```
/* 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 register this address in kernel. (Sys_RegDeliver)
- Kernel would explicitly set the return address of each signal handler to go to Trampoline.

- Complete signal handling for this process.
- Return state->eax.

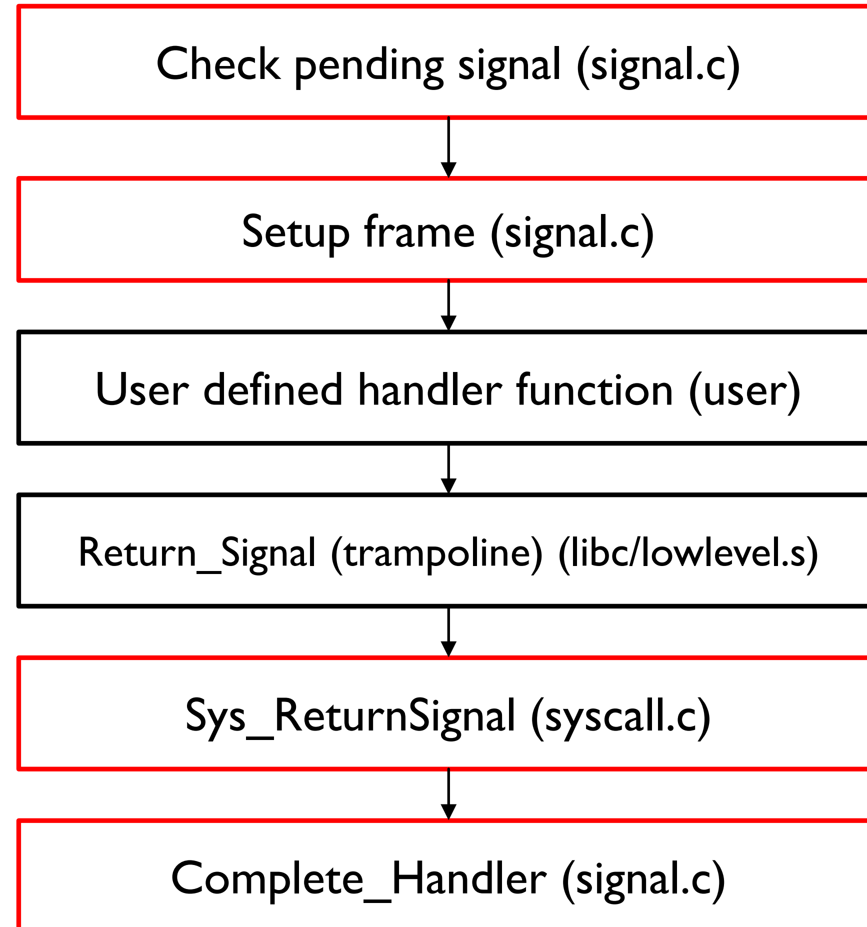
# SYS_REGDELIVER

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

# SYS_KILL

- It will be used to send a signal to a certain process.
  - Not just SIG_KILL, any signal that is valid.
- If called with an invalid 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.
- User process is not allowed to send signal to a kernel process. Set the appropriate error conditions.

# CALL SEQUENCE OF SYGNAL



# 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:
  - A signal is pending for that user process.
  - 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.
  - The process is not currently handling another signal(recall that signal handling is non-reentrant).

# SETUP_FRAME

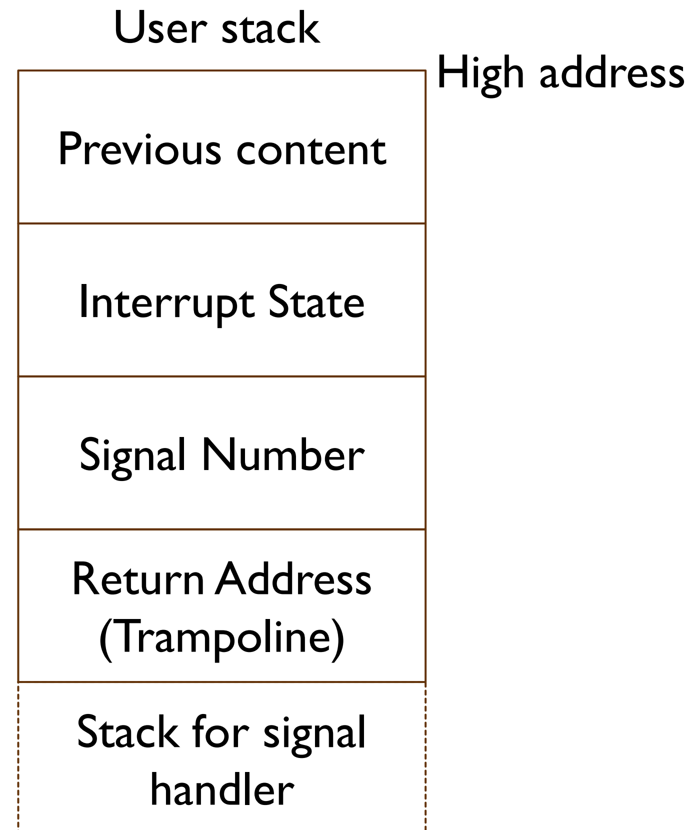
- Invoked when Check_Pending_Signal returns true.
- Takes care of the signal if it does not have user defined handler function.
- Sets up a user processes user stack and kernel stack so that
    - When the process returns to user mode, it will execute the correct signal handler.
    - When that handler completes, the process will return to the trampoline function so that it can go back to what it was doing.

# SETUP_FRAME – HOW TO SET USER & KERNEL STACK

- Choose the correct handler to invoke by using the signal number and the handler table.
- 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.
- Push onto the user stack a snapshot of the interrupt state that is currently stored at the top of the kernel stack.
- Push onto the user stack the signal number being delivered.
- Push onto the user stack the address of the trampoline that was saved in the Sys_RegDeliver system call.
- Now that you have saved a copy of the interrupt state, change the original User_Interrupt_State such that
  - The user stack pointer is updated.
  - The saved program counter(eip) points to the signal handler.

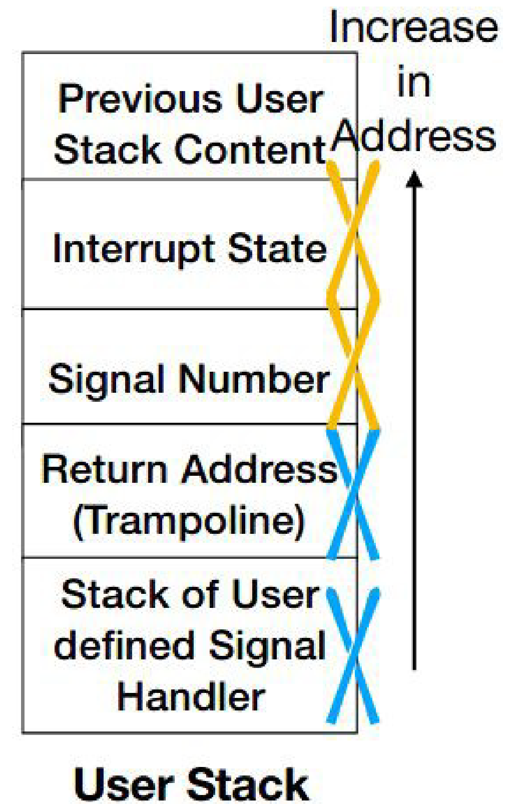
# SETUP_FRAME – SUMMARIZE

- Determine the handler.
- Push the user interrupt state on the USER stack.
- Push the signal number on the USER stack.
- Push the trampoline address on the USER stack.
- Change the saved eip.



# COMPLETE_HANDLER

- This routine should be called when the Sys_ReturnSignal is invoked(after a signal handler has completed).
- Reverse the operations in setup_frame.
  - 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.
  - Remove signal number.
- No need to pop the pushed return address here. It’s been popped by the ret instruction at the end of the signal handler(to execute the libc function Return_Signal).
- Make sure user esp is adjusted correctly.
- To debug: check user esp after pop off everything should be equal to the esp in setup frame before push anything.



# SYS_WAITNOPID

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

- Sys_Fork: copy signal handlers, trampoline
- Pipe_Write: send SIGPIPE
- Exit: send SIGCHLD
- Refer to spec for more details.

# TESTING: SIGPIPE.C

- 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.
- At this point, 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, which terminate the process.

# TESTING: SIGPIPE.C

```
$ 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
```