

# Project 2

Zen and the art of sending signals

# Background – Context Switching

- One processor and multiple threads running concurrently – How?
- Give each thread a small time quantum to run.
- When this quantum expires, or the thread blocks, **context-switch** to a different thread.
  1. Where should I save the thread context during a context-switch?
  2. What should this context consist of?

# Background – Kernel Stack

- User process is a kernel thread with `USER_CONTEXT` structure.
- Store the current context (state) before context switching.
- Where is the kernel stack?

```
struct Kernel_Thread {  
    unsigned long esp; // Stack pointer (absolute)  
    void* stackPage; //The beginning of the stack  
    .....  
};
```

- **esp** points at the end of the stack (stack grows down from higher to lower address)

# Background – User Processes

- Two stacks: kernel stack and user stack.
- User Stack (stores state in user mode)
- Start\_User\_Thread:

set up the *kernel* stack to look as if the thread had previously been running and then context-switched to the ready queue.

# Background – Context Information

User Stack Location

Interrupt\_State

The items at the top are pushed first.

Program Counter → EIP

User stack pointer points to the end of the DS.

Stack grows down from higher address to lower address.

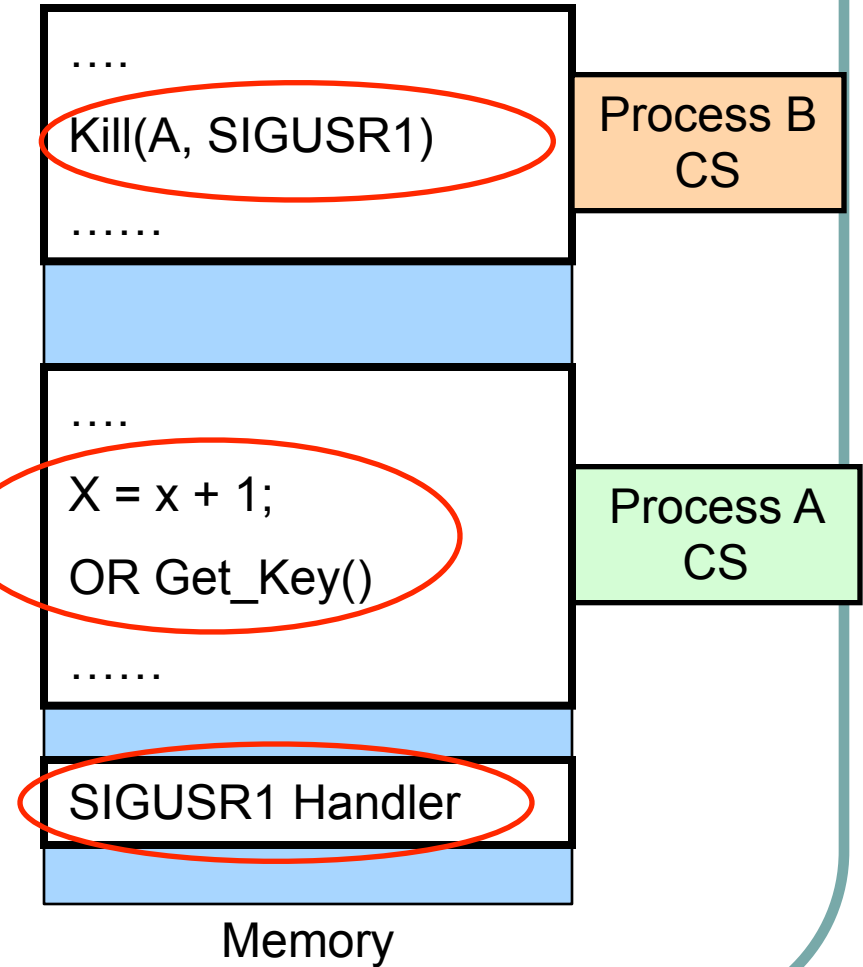
Stack Data Selector (data selector)
Stack Pointer (end of data memory)
Eflags
Text Selector (code selector)
Program Counter (entry addr)
Error Code (0)
Interrupt Number (0)
EAX (0)
EBX (0)
ECX (0)
EDX (0)
ESI (Argument Block address)
EDI (0)
EBP (0)
DS (data selector)
ES (data selector)
FS (data selector)
GS (data selector)

# Project 2: Signals

- Inter-process communication
- Signals are to user processes what interrupts are to the kernel .
- Process temporarily stops what it is doing, and is instead redirected to the **signal handler**.
- When the handler completes, the process goes back to what it was doing (unless another signal is pending!)

# Signals

1. Process A is executing then either finishes quantum OR blocked
2. Process B is now executing and sends a signal to A.
3. Process A is executing again. However, control is transferred to SIGUSR1 handler.
4. SIGUSR1 handler finishes. Then control transfers to Process A again (if no other signal pending).



# Project Requirements

1. Add the code to handle signals.
2. System calls.
3. Background processes are NOT detached. (Now they are children)

**Look for TODO macro**

# Supported Signals

- SIGKILL: treated as Sys\_Kill of project I.
- SIGUSR1 & SIGUSR2
- SIGCHLD
- Background processes are NOT detached any more (refCount will start as 2).
- Sent to a parent when the background child dies.
- Default handler - reap the child

# System Calls

- `Sys_Signal`: register a signal handler
- `Sys_RegDeliver`: initialize signal handling for a process
- `Sys_Kill`: send a signal
- `Sys_ReturnSignal`: indicate completion of signal handler
- `Sys_WaitNoPID`: wait for any child process to die

# Sys\_Signal

- Register a signal handler for a process
  - state->ebx - pointer to handler function
  - state->ecx - signal number
  - Returns: 0 on success or error code (< 0) on error
- Calling Sys\_Signal with a handler to SIGKILL should result in an error.

# Sys\_Signal

- Initial handler for SIGCHLD (reaps all zombie) is
- Def\_Child\_Handler
- Two predefined handlers:
- SIG\_IGN, SIG\_DFL (check include/libc/signal.h)
- Example: `Signal(SIGUSR1, SIG_IGN);`

# Sys\_RegDeliver

- Register three functions:
  1. Ignore
  2. Default
  3. Return\_Signal trampoline (calls Sys\_ReturnSignal)
- Signals cannot be delivered until this is registered.
  - state->ebx - pointer to Return\_Signal function
  - state->ecx - pointer to the default handler
  - state->edx - pointer to the ignore handler
  - Returns: 0 on success or error code (< 0) on error
- These routines are registered automatically.  
(check src/libc/entry.c)

# Sys\_Kill

- Send a signal to a process
  - state->ebx - pid of process
  - state->ecx - signal number
  - Returns: 0 on success or error code (< 0) on error

# Sys\_ReturnSignal

- Complete signal handling for this process.
  - No Parameters
  - Returns: 0 on success or error code ( $< 0$ ) on error
- Called by a process immediately after it has handled a signal.

# Sys\_WaitNoPID

- Reap a child process that has died
  - state->ebx - pointer to status of process reaped
  - Returns: pid of reaped process on success, -1 on error.

# Signals Golden Rules

- Any user process stores THREE pointers to handler functions corresponding to (SIGUSR1, SIGUSR2, SIGCHLD).
- These pointers could be NULL if there is no registered handler.
- Any process also stores THREE pointers to the Ign\_Handler, Def\_Handler, Signal\_Return
- If there no handler registered, the default handler will be executed.
- Signal handling is non-reentrant.

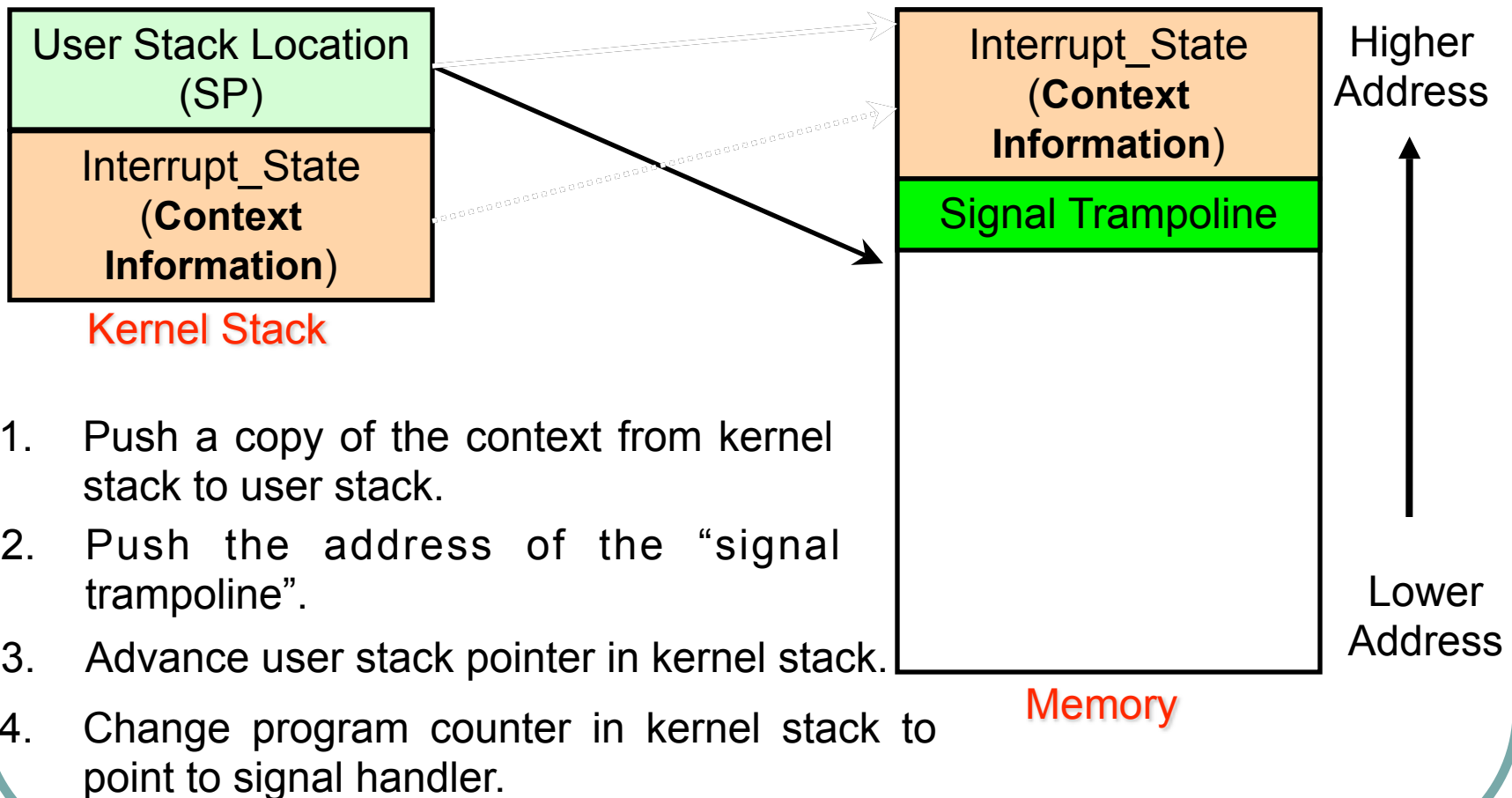
# Signals Delivery in Kernel

- `src/geekos/signal.c`
- `Send_Signal`
- `Check_Pending_Signal`
- `Set_Handler`
- `Setup_Frame`
- `Complete_Handler`

# Check\_Pending\_Signal

- A signal is pending for that user process.
- The process is about to start executing in user space.
- CS register != KERNEL\_CS
- (see include/geekos/defs.h)
- The process is not currently handling another signal.

# Setup\_Frame



# Complete\_Handler

