Project 2 Roadmap

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 (store local variables)
- 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	**Stack Data Selector (data selector)**
	Stack Pointer (end of data memory)
Interrupt_State	**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)

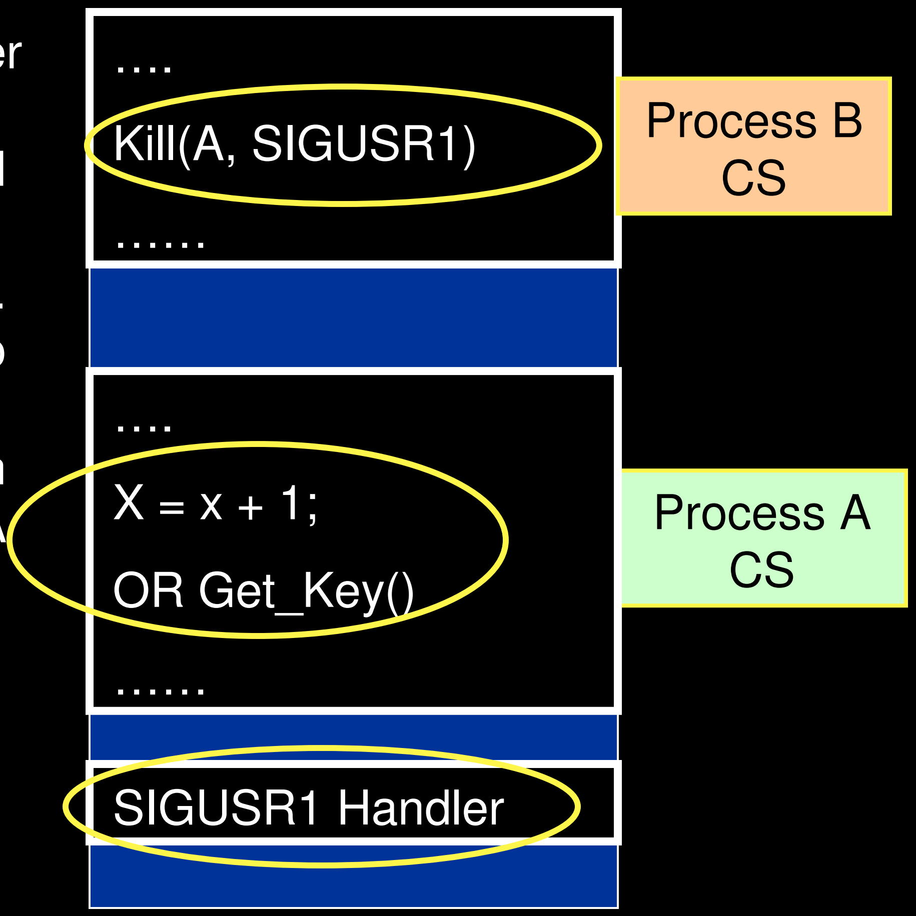
- The items at the top are pushed first.
- Program Counter →EIP
- User stack pointer points to the end of the DS.
- Stack grows down from higher address to lower address.

Project 2: Signals

- Signals are to user processes what interrupts are to the kernel .
- Process temporarily stop 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
1. Process B is now executing and sends a signal to A.
1. Process A is executing again. However, control is transferred to SIGUSR1 handler.
1. SIGUSR1 handler finishes. Then control transfers to Process A again (if no other signal pending).



….

Kill(A, SIGUSR1)

……

Process B CS

….

X = x + 1;

OR Get_Key()

……

Process A CS

SIGUSR1 Handler

Memory

Project Requirements

1. Add the code to handle signals.
2. System calls.
3. Background processes are NOT detached.

Look for TODO macro

Supported Signals

1. SIGKILL: treated as Sys_Kill of project1.
2. SIGUSR1 & SIGUSR2
3. SIGCHLD

- Background processes are NOT detached any more (refCount = 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.
- Initial handler for SIGCHLD (reaps all zombie) is

  Def_Child_Handler
- Two predefined handlers:
  - SIG_IGN, SIG_DFL (check inlcude/libc/signal.h)
  - Used #define to set a fake address
  - Should be handled directly from kernel
- Example: Signal(SIGUSR1,SIG_IGN);

Sys_ RegDeliver

- Register trampoline function:
  - calls Sys_ReturnSignal
- Signals cannot be delivered until this is registered.
  - state->ebx - pointer to Return_Signal function
  - Returns: 0 on success or error code (< 0) on error

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, -1on 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

src/**geekos**/signal.c

1. Send_Signal
2. Check_Pending_Signal
3. Set_Handler
4. Setup_Frame
5. Complete_Handler

Check_Pending_Signal

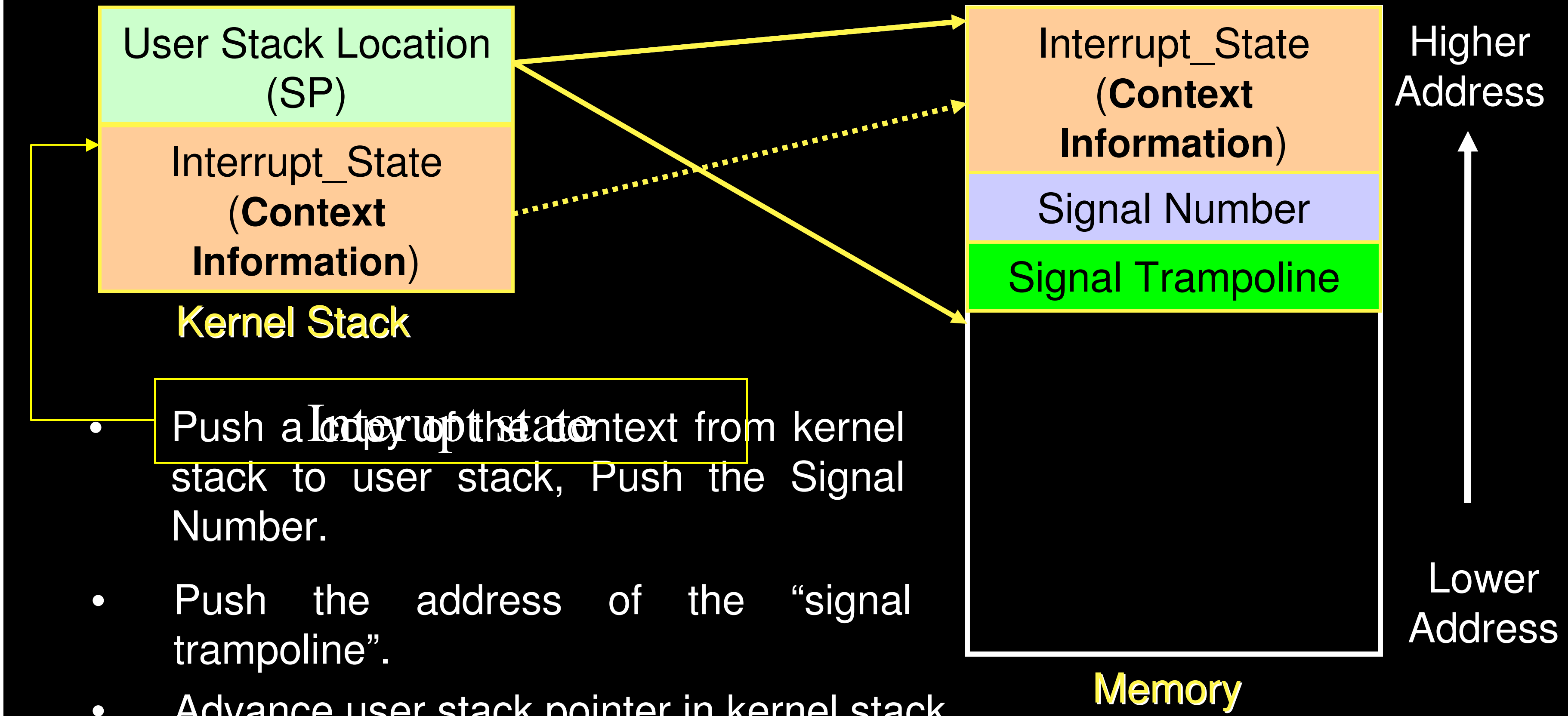
1. A signal is pending for that user process.
2. The process is about to start executing in user space.

CS register != KERNEL_CS

(see include/geekos/defs.h)

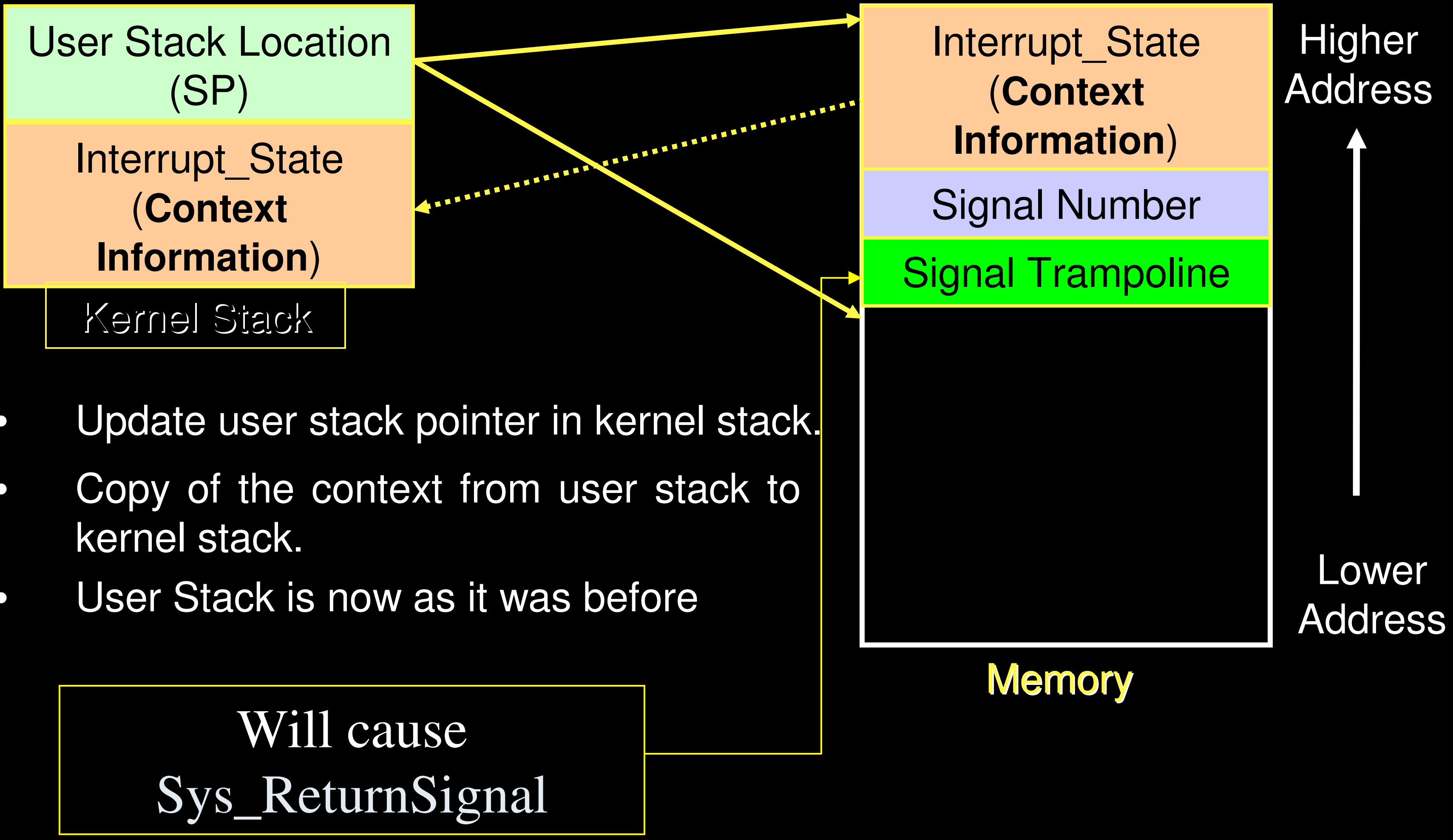
1. The process is not currently handling another signal.

Setup_Frame



- Push a copy of the context from kernel stack to user stack, Push the Signal Number.
- Push the address of the “signal trampoline”.
- Advance user stack pointer in kernel stack.
- Change program counter in kernel stack to point to signal handler.

Complete_Handler



- Update user stack pointer in kernel stack.
- Copy of the context from user stack to kernel stack.
- User Stack is now as it was before

Will cause Sys_ReturnSignal

Project 2 Roadmap++

Review

Process A

```
main() {
    for(1000)
        Print "A"
    Kill(B, SIGUSR1)
}
```

Process B

```
function handler() {
    Print "HANDLING"
}

main() {
    Signal(&handler, SIGUSR1)
    for(;;)
        Print "B"
}
```

Output

$$\underbrace{\text{AABBAABBAABB………}}_{\text{1000 A'S}}\text{HANDLING BBBBBBBBBBBBBB…..}$$

System Calls

Sys_Signal: register a signal handler

Sys_Kill: send a signal

Sys_RegDeliver: initialize signal handling for a process

Sys_WaitNoPID: wait for any child process to die

Sys_ReturnSignal: indicate completion of signal handler

System Calls

Sys_Signal: register a signal handler

Sys_Kill: send a signal

Referenced in user code

Sys_RegDeliver: initialize signal handling for a process

Sys_WaitNoPID: wait for any child process to die

Sys_ReturnSignal: indicate completion of signal handler

Review

Process A

```
main() {
    for(1000)
        Print “A”
    Kill(B, SIGUSR1)
}
```

Process B

```
function handler() {
    Print “HANDLING”
}

main() {
    Signal(&handler, SIGUSR1)
    for(;;)
        Print “B”
}
```

Output

$\underbrace{\text{AABBAABBAABB.........}}_{\text{1000 A'S}}$ HANDLING BBBBBBBBBBBBBB.....

System Calls

Sys_Signal: register a signal handler

Sys_Kill: send a signal

Sys_RegDeliver: initialize signal handling for a process

Sys_WaitNoPID: wait for any child process to die

Sys_ReturnSignal: indicate completion of signal handler

System Calls

Sys_Signal: register a signal handler

Sys_Kill: send a signal

Sys_RegDeliver: initialize signal handling for a process

Sys_WaitNoPID: wait for any child process to die

Sys_ReturnSignal: indicate completion of signal handler

Executed by stub code once a signal has been handled

(from trampoline)

Helper Functions

- Send_Signal
- Set_Handler
- Check_Pending_Signal
- Setup_Frame
- Complete_Handler

Review

Process A

```
main() {
    for(1000)
        Print “A”
    Kill(B, SIGUSR1)
}
```

Process B

```
function handler() {
    Print “HANDLING”
}

main() {
    Signal(&handler, SIGUSR1)
    for(;;)
        Print “B”
}
```

Overview

A

B

Overview

A

B

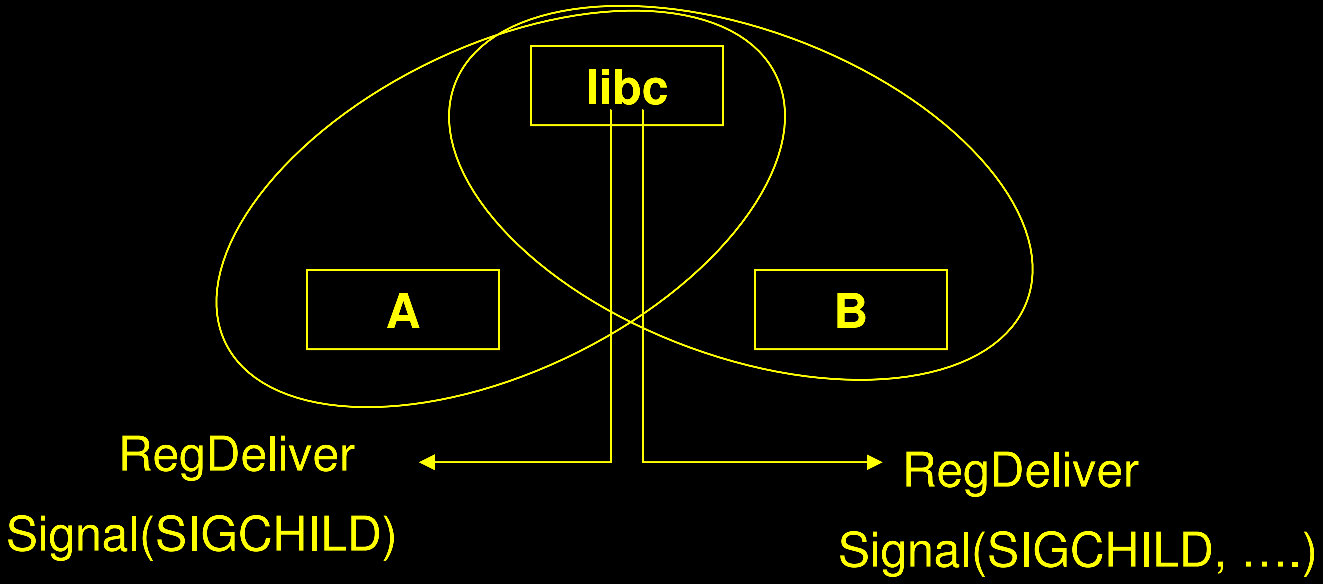
RegDeliver

Signal(SIGCHILD)

RegDeliver

Signal(SIGCHILD, ….)

Overview



libc

A

B

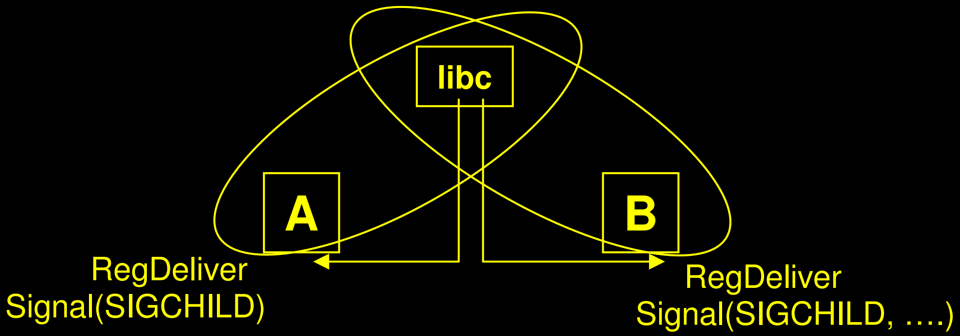
RegDeliver

Signal(SIGCHILD)

RegDeliver

Signal(SIGCHILD, ….)

Overview



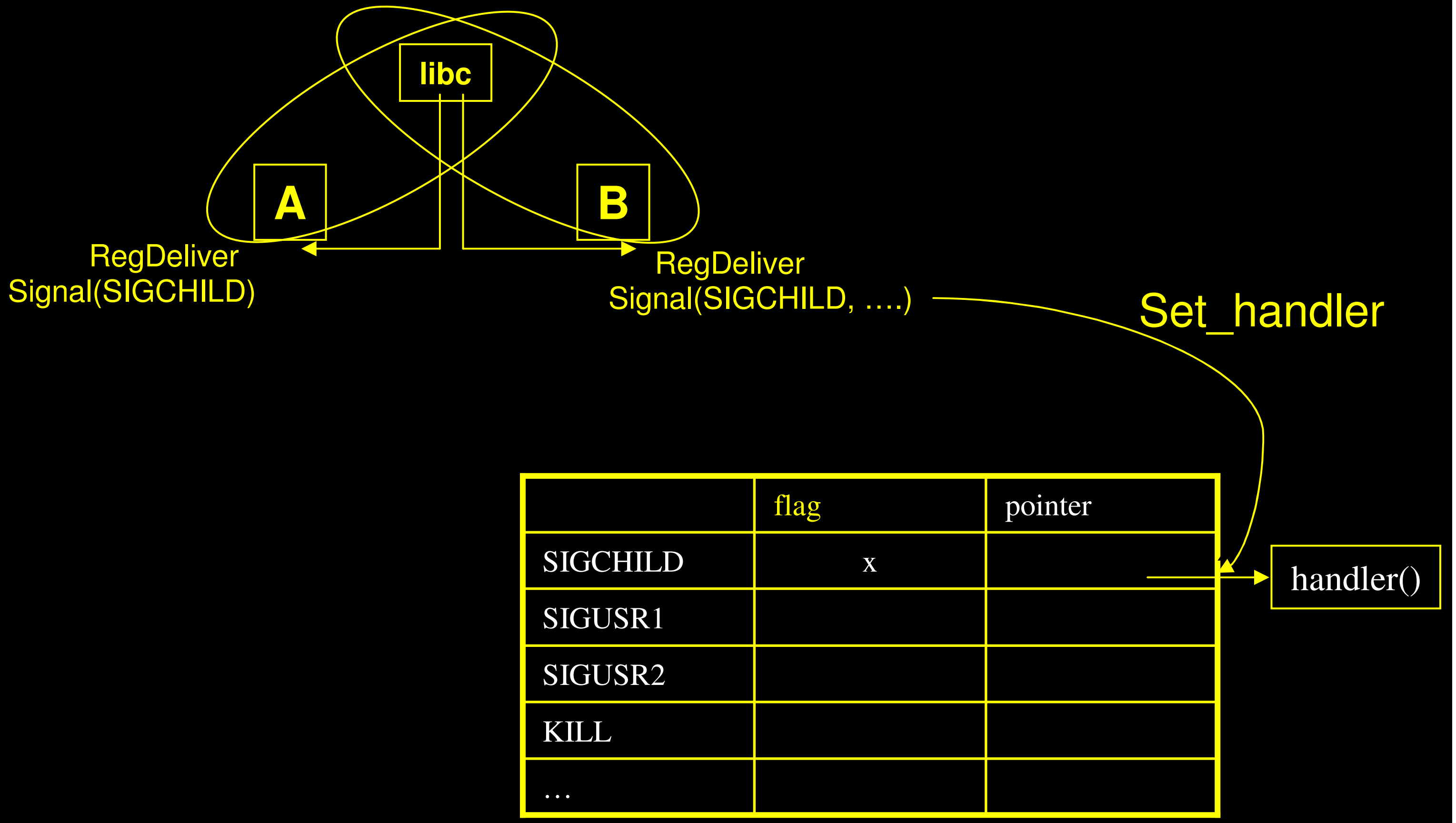
RegDeliver
Signal(SIGCHILD)

RegDeliver
Signal(SIGCHILD, ….)

Set_handler

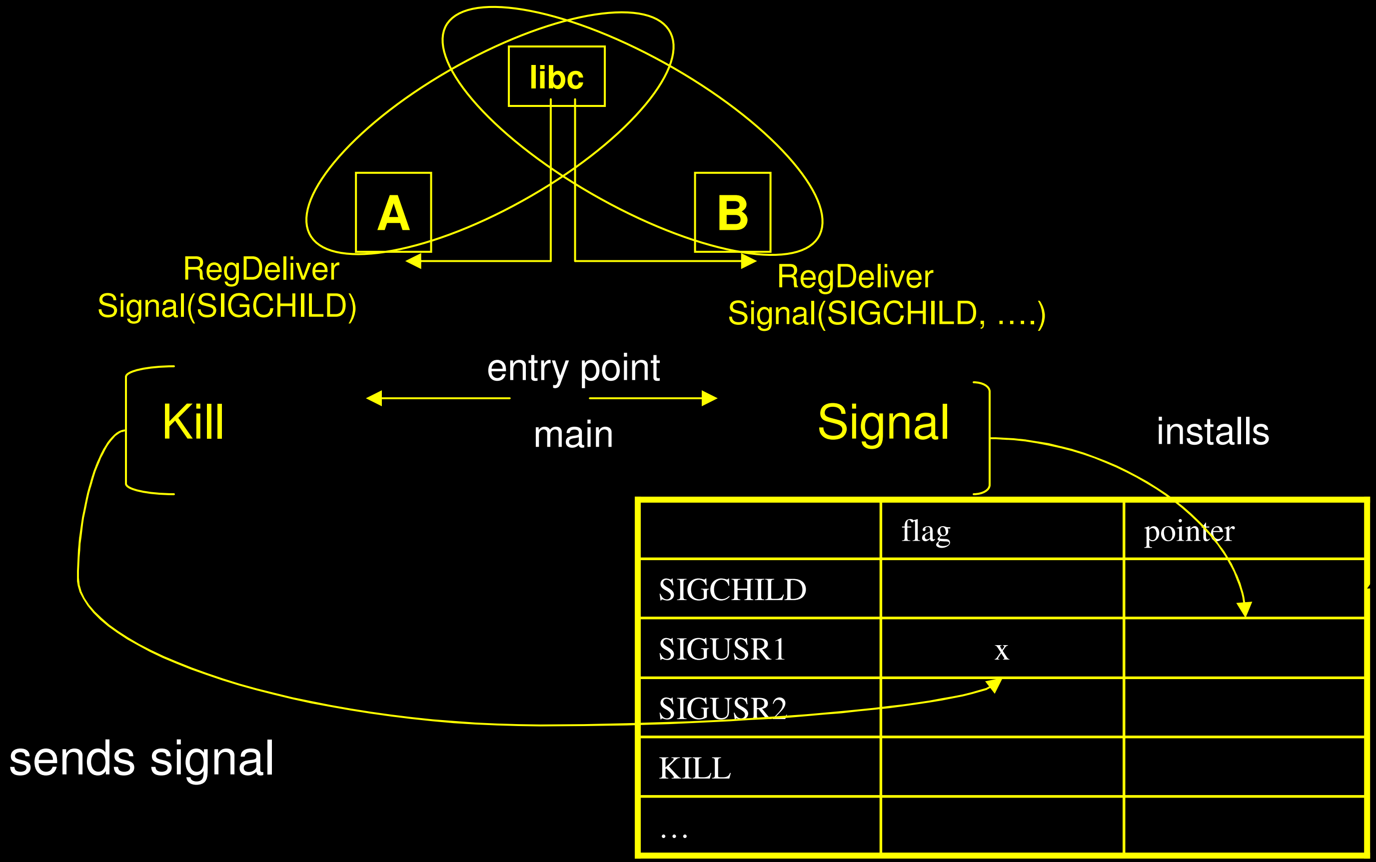
	flag	pointer
SIGCHILD		
SIGUSR1		
SIGUSR2		
KILL		
…		

Overview



	flag	pointer
SIGCHILD	x	
SIGUSR1		
SIGUSR2		
KILL		
…		

Overview



libc

A

B

RegDeliver
Signal(SIGCHILD)

RegDeliver
Signal(SIGCHILD, ….)

entry point

main

Kill

Signal

installs

sends signal

	flag	pointer
SIGCHILD		
SIGUSR1	x	
SIGUSR2		
KILL		
…		

Helper Functions

Send_Signal

Set_Handler

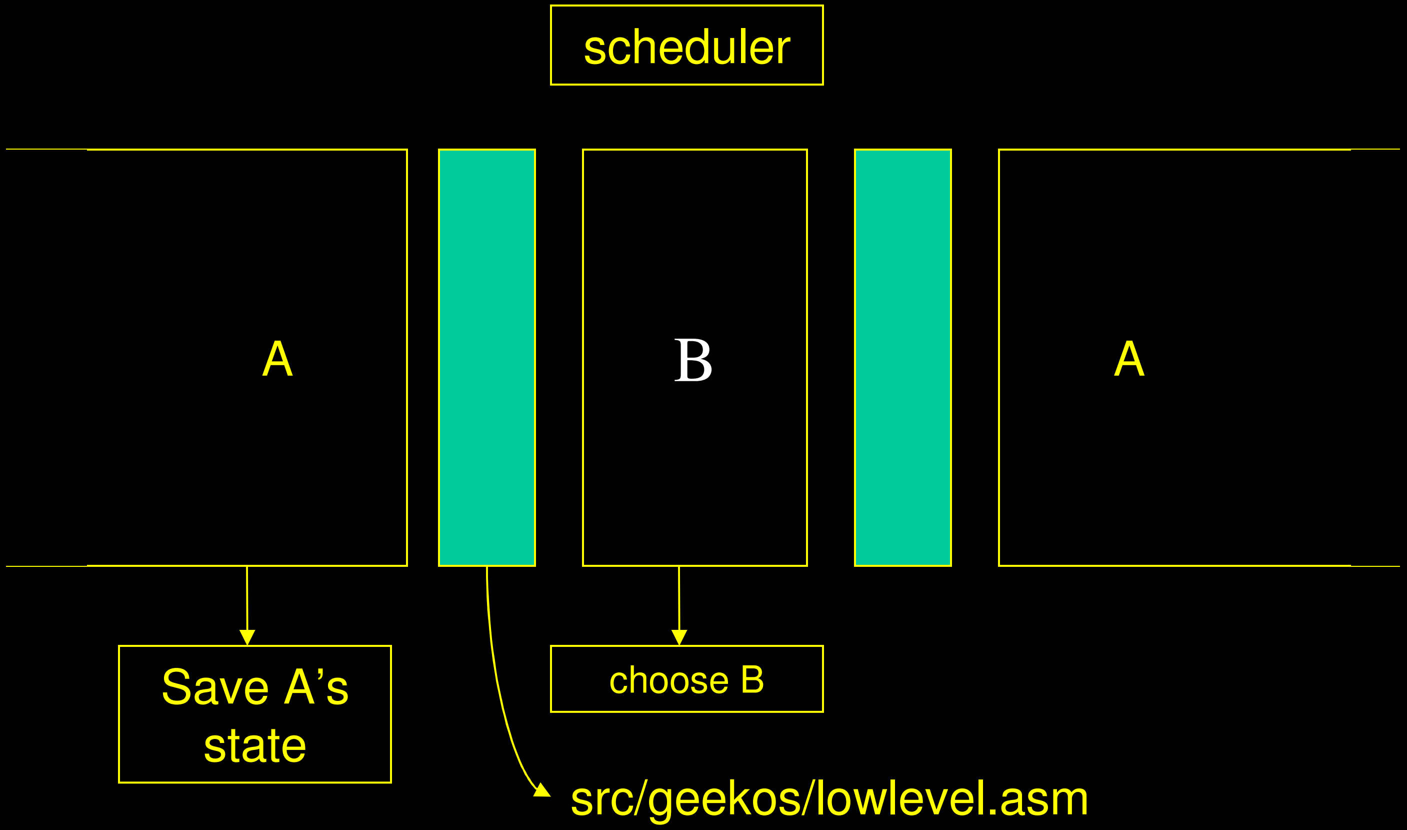
Check_Pending_Signal

Setup_Frame

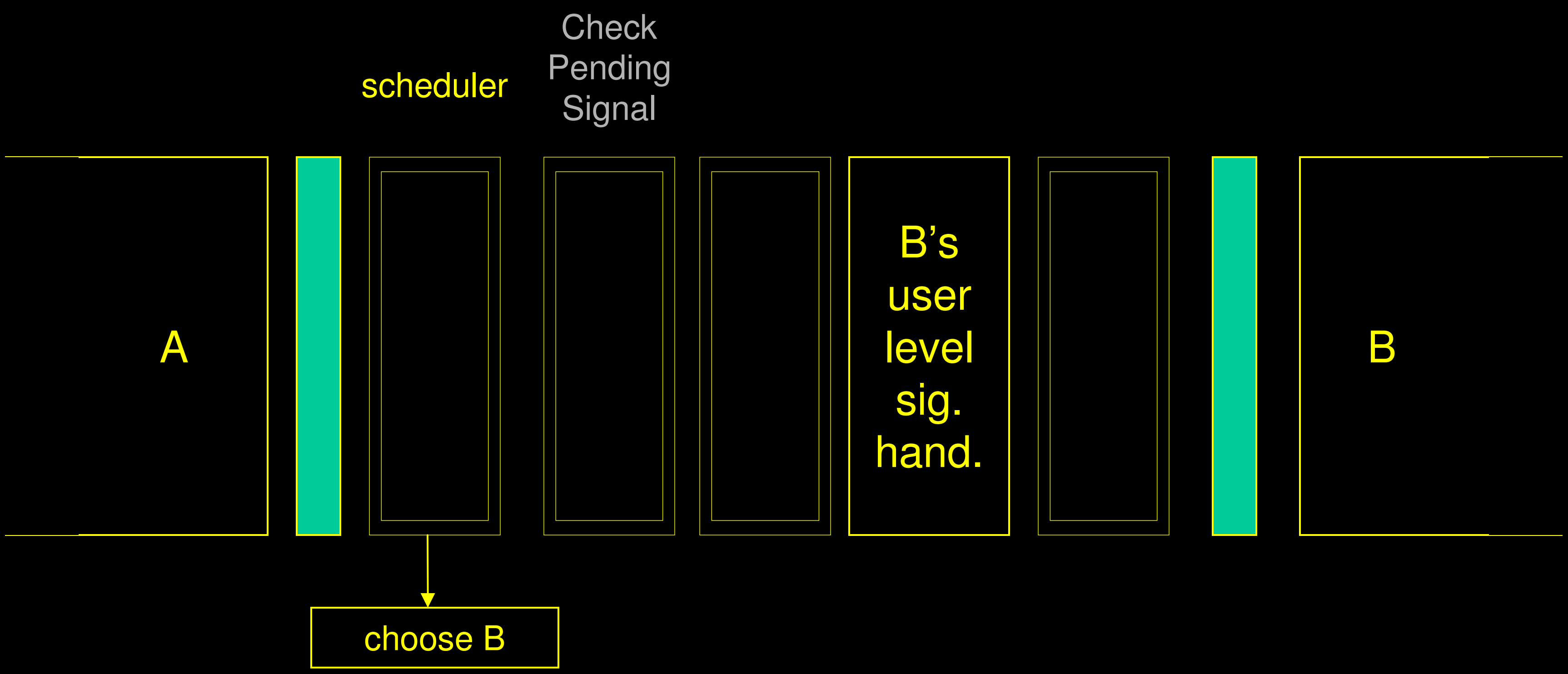
Complete_Handler

Look at Scheduler

Scheduler: w/o signals



Scheduler: w/ signals

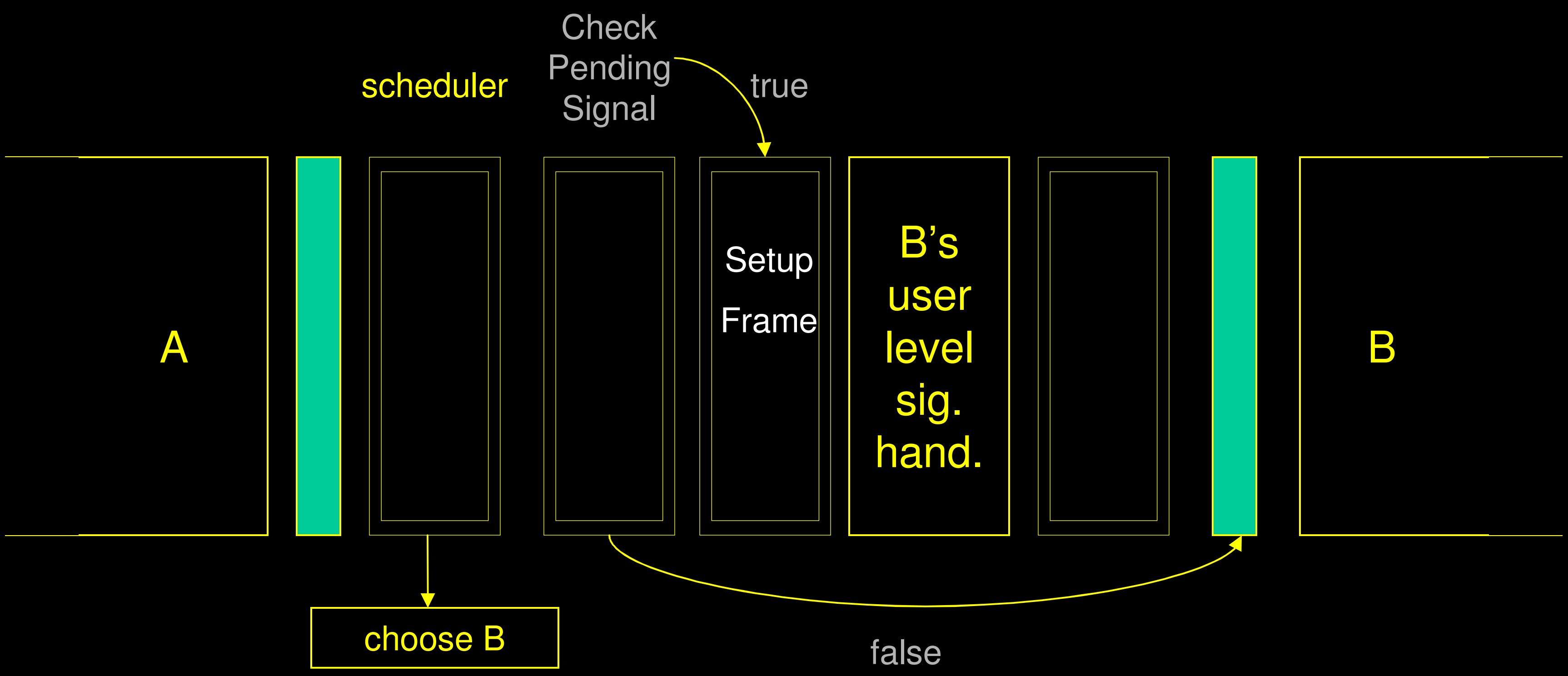


Check Pending Signal

Boolean output

Determines whether to proceed with signal handling

Scheduler: w/ signals



Setup Frame

Sets up state to enable user-level handling code execution

Setup Frame

Sets up state to enable user-level handling code execution

How are functions called?

Function Calls

Parameter of return address is stored on the stack so when finished

- Pop off stack
- Continue execution

Setup Frame

- Enables user stack to keep:
  - Interrupt_State Vector
  - Return address

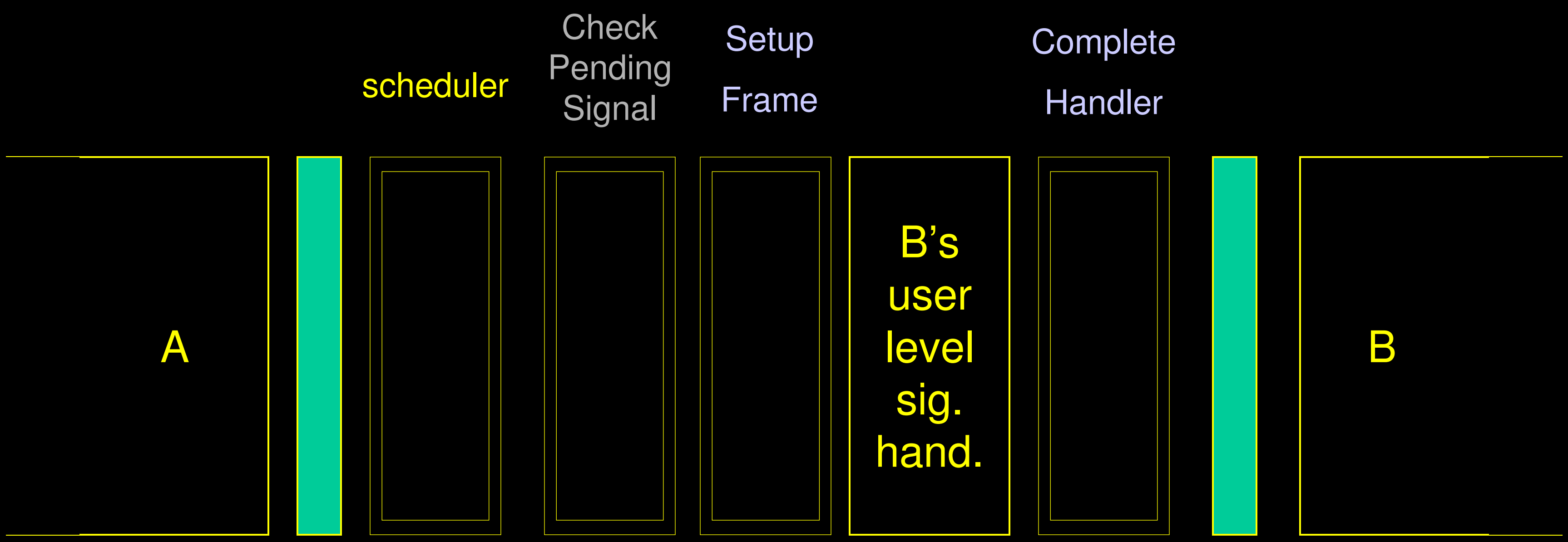
Storing Return Address

Want complete_handler to execute once user level handling done.

Hack

- Place address of return_signal as return address on stack
- Now return_signal stred as function

Scheduler: w/ signals



scheduler

Check Pending Signal

Setup Frame

Complete Handler

A

B's user level sig. hand.

B