Project 2

Zen and the art of sending signals

Background – Context Switching

Background – Context Switching

- One processor and multiple threads running concurrently – How?

Background – Context Switching

- One processor and multiple threads running concurrently – How?
- Give each thread a small time quantum to run.

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.

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?

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

Background – Kernel Stack

- User process is a kernel thread with USER_CONTEXT structure.

Background – Kernel Stack

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

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?

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
  ........................
  };
```

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

Background – User Processes

- Two stacks: kernel stack and user stack.

Background – User Processes

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

Background – User Processes

- Two stacks: kernel stack and user stack.
- User Stack (stores state in user mode)
- Start_User_Thread:

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	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

Project 2: Signals

- Inter-process communication

Project 2: Signals

- Inter-process communication
- Signals are to user processes what interrupts are to the kernel .

Project 2: Signals

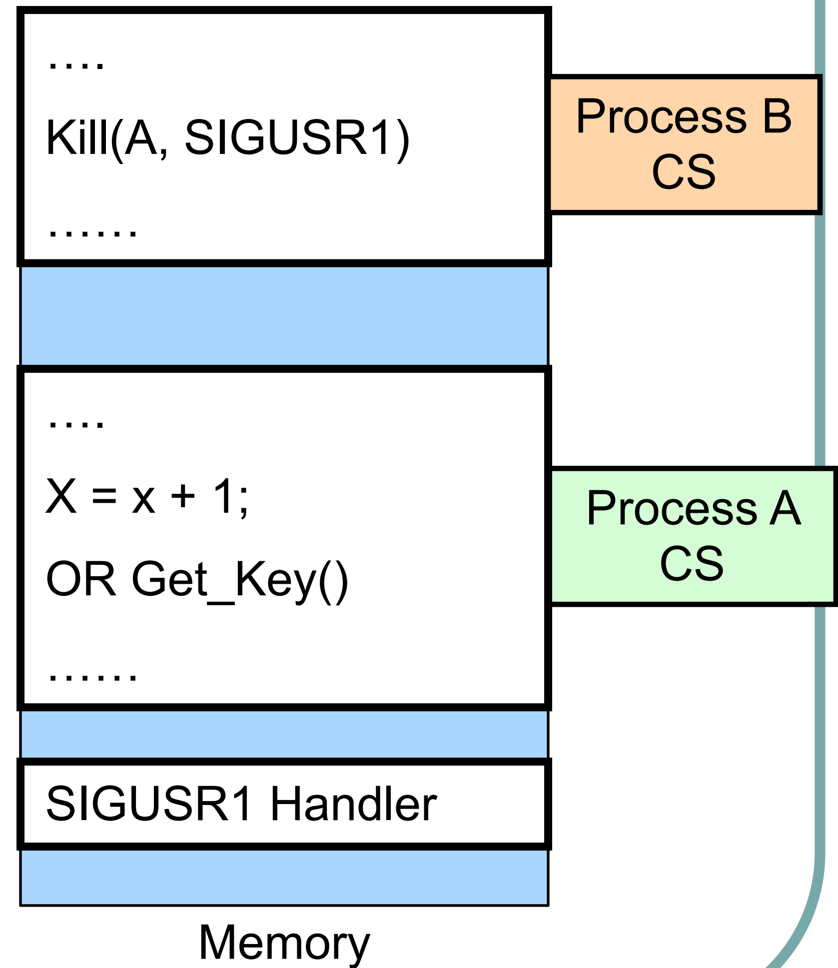
- Inter-process communication
- Signals are to user processes what interrupts are to the kernel .
- Process temporarily stops what it is doing, and is instead redirected to the signal handler.

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!)

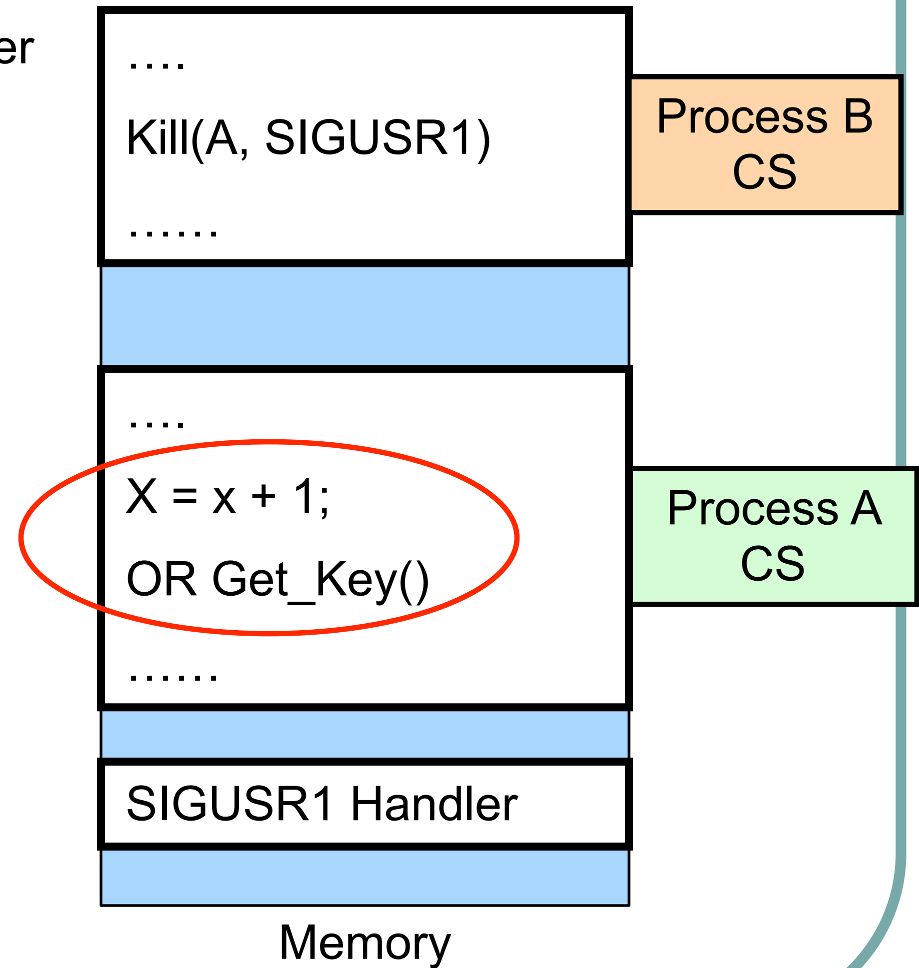
Signals

1. Process A is executing then either finishes quantum OR blocked



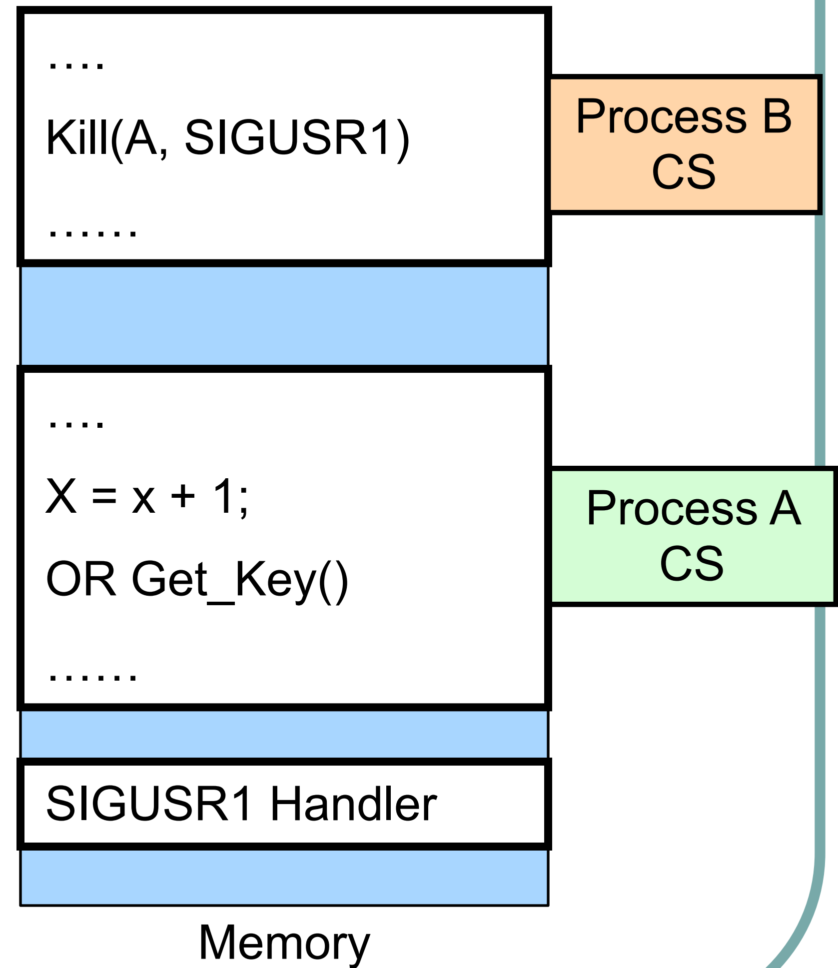
Signals

1. Process A is executing then either finishes quantum OR blocked



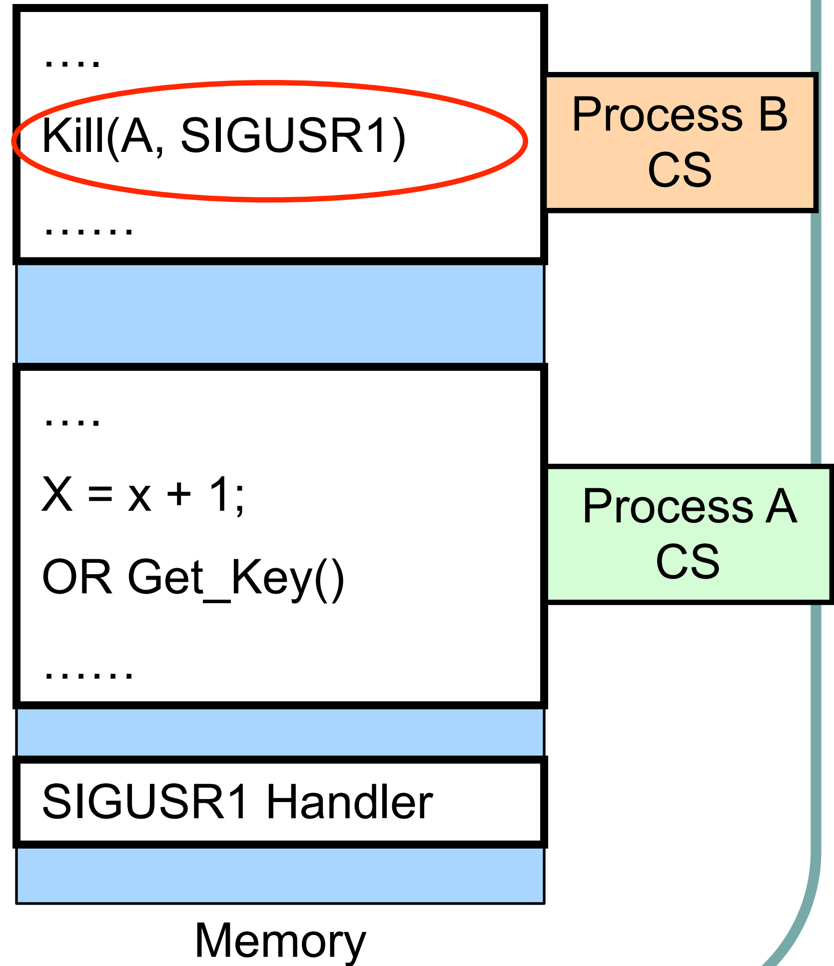
Signals

1. Process A is executing then either finishes quantum OR blocked



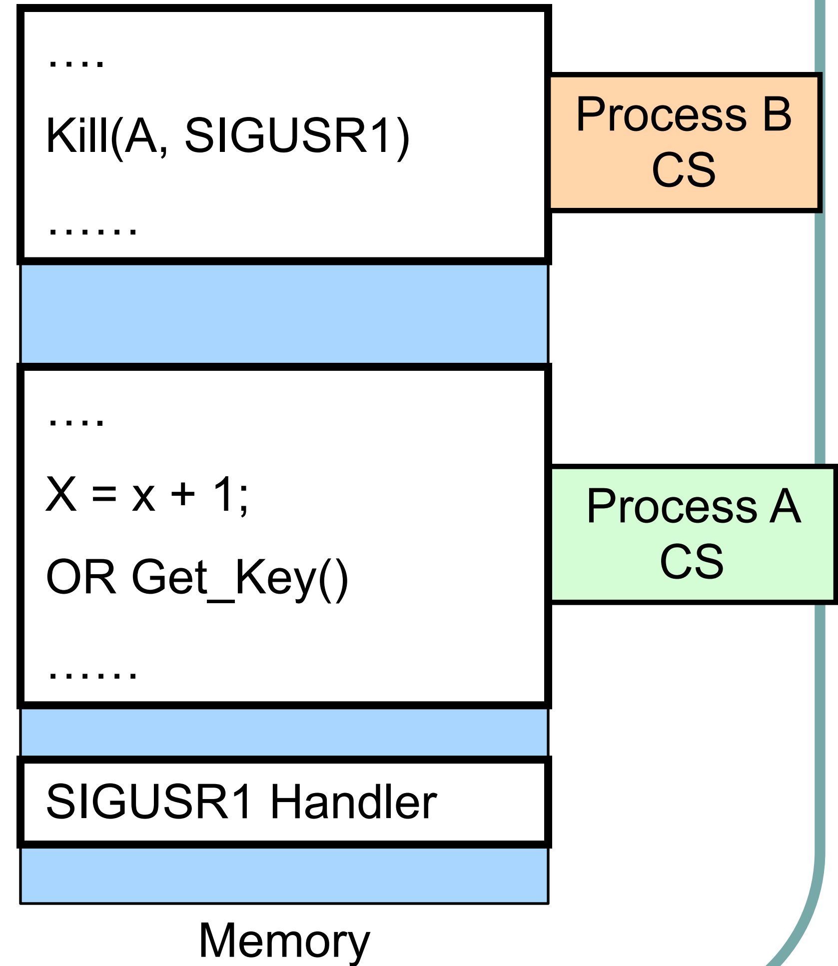
Signals

1. Process A is executing then either finishes quantum OR blocked
2. Process B is now executing and sends a signal to A.



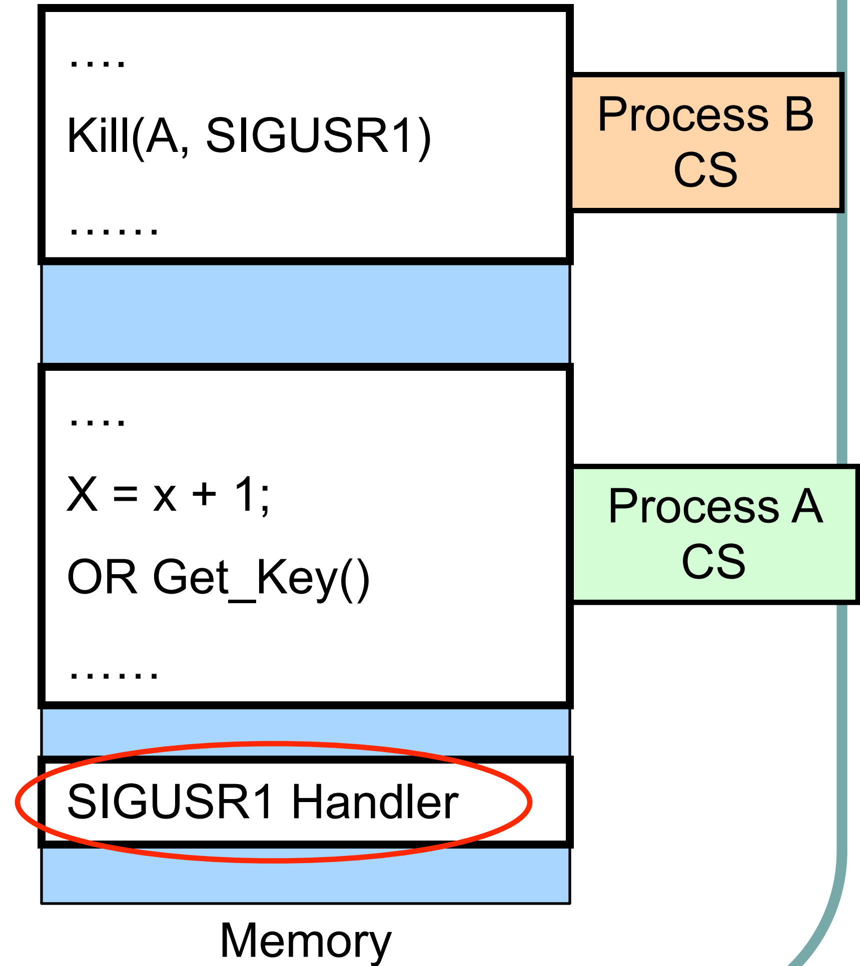
Signals

1. Process A is executing then either finishes quantum OR blocked
2. Process B is now executing and sends a signal to A.



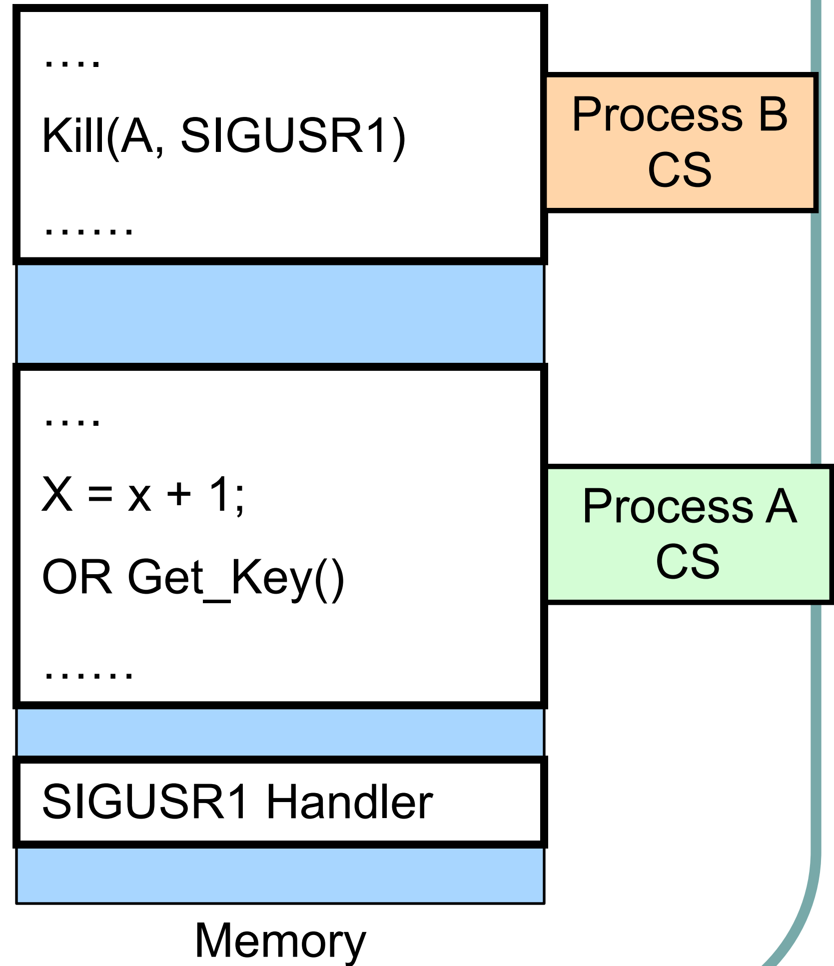
Signals

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.



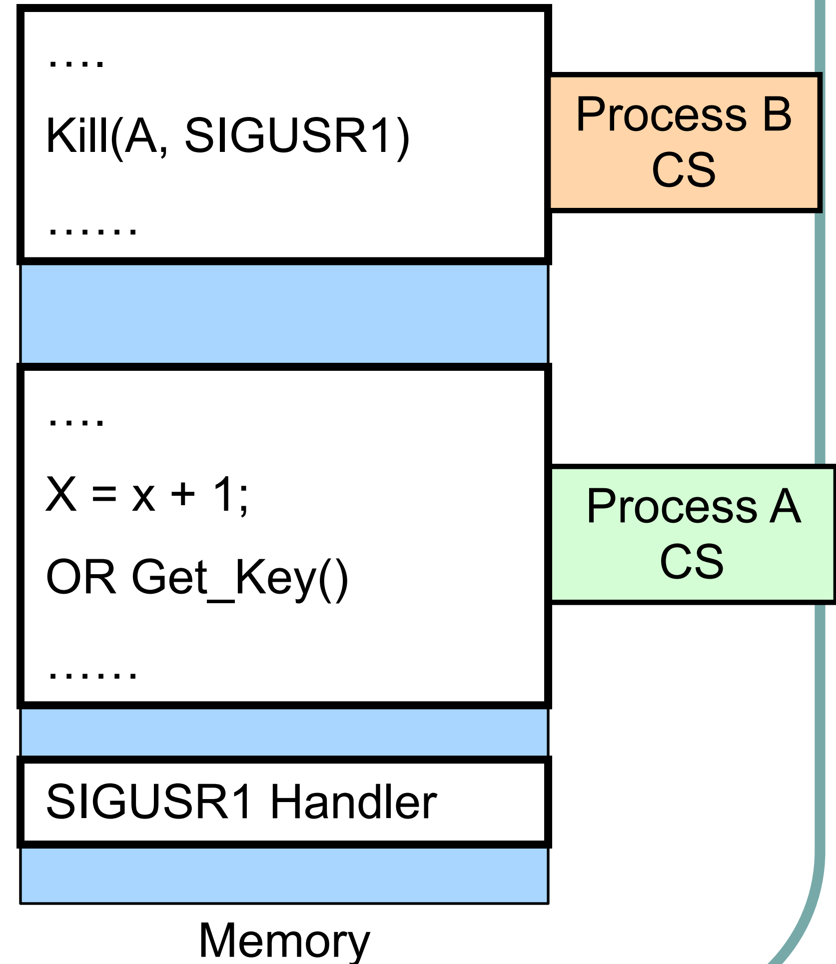
Signals

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.



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

Project Requirements

1. Add the code to handle signals.

Project Requirements

1. Add the code to handle signals.
2. System calls.

Project Requirements

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

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

Supported Signals

- SIGKILL: treated as Sys_Kill of project1.

Supported Signals

- SIGKILL: treated as Sys_Kill of project1.
- SIGUSR1 & SIGUSR2

Supported Signals

- SIGKILL: treated as Sys_Kill of project1.
- SIGUSR1 & SIGUSR2
- SIGCHLD

Supported Signals

- SIGKILL: treated as Sys_Kill of project1.
- SIGUSR1 & SIGUSR2
- SIGCHLD
- Background processes are NOT detached any more (refCount will start as 2).

Supported Signals

- SIGKILL: treated as Sys_Kill of project1.
- SIGUSR1 & SIGUSR2
- SIGCHLD
- Background processes are NOT detached any more (refCount will start as 2).
- Sent to a parent when the background child dies.

Supported Signals

- SIGKILL: treated as Sys_Kill of project1.
- 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

System Calls

- Sys_Signal: register a signal handler

System Calls

- Sys_Signal: register a signal handler
- Sys_RegDeliver: initialize signal handling for a process

System Calls

- Sys_Signal: register a signal handler
- Sys_RegDeliver: initialize signal handling for a process
- Sys_Kill: send a signal

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

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

Sys_ Signal

- Register a signal handler for a process

Sys_ Signal

- Register a signal handler for a process
  - state->ebx - pointer to handler function

Sys_ Signal

- Register a signal handler for a process
  - state->ebx - pointer to handler function
  - state->ecx - signal number

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

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

Sys_Signal

- Initial handler for SIGCHLD (reaps all zombie) is

Sys_Signal

- Initial handler for SIGCHLD (reaps all zombie) is
- Def_Child_Handler

Sys_Signal

- Initial handler for SIGCHLD (reaps all zombie) is
- Def_Child_Handler
- Two predefined handlers:

Sys_Signal

- Initial handler for SIGCHLD (reaps all zombie) is
- Def_Child_Handler
- Two predefined handlers:
- SIG_IGN, SIG_DFL (check inlcude/libc/signal.h)

Sys_Signal

- Initial handler for SIGCHLD (reaps all zombie) is
- Def_Child_Handler
- Two predefined handlers:
- SIG_IGN, SIG_DFL (check inlcude/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

Sys_Kill

- Send a signal to a process

Sys_Kill

- Send a signal to a process
  - state->ebx - pid of process

Sys_Kill

- Send a signal to a process
  - state->ebx - pid of process
  - state->ecx - signal number

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

Sys_ReturnSignal

- Complete signal handling for this process.

Sys_ReturnSignal

- Complete signal handling for this process.
    - No Parameters

Sys_ReturnSignal

- Complete signal handling for this process.
  - No Parameters
  - 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

Sys_WaitNoPID

- Reap a child process that has died

Sys_WaitNoPID

- Reap a child process that has died
  - state->ebx - pointer to status of process reaped

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

Signals Golden Rules

- Any user process stores THREE pointers to handler functions corresponding to (SIGUSR1, SIGUSR2, SIGCHLD).

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.

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

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.

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

Signals Delivery in Kernel

- src/geekos/signal.c

Signals Delivery in Kernel

- src/geekos/signal.c
- Send_Signal

Signals Delivery in Kernel

- src/geekos/signal.c
- Send_Signal
- Check_Pending_Signal

Signals Delivery in Kernel

- src/geekos/signal.c
- Send_Signal
- Check_Pending_Signal
- Set_Handler

Signals Delivery in Kernel

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

Signals Delivery in Kernel

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

Check_Pending_Signal

Check_Pending_Signal

- A signal is pending for that user process.

Check_Pending_Signal

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

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

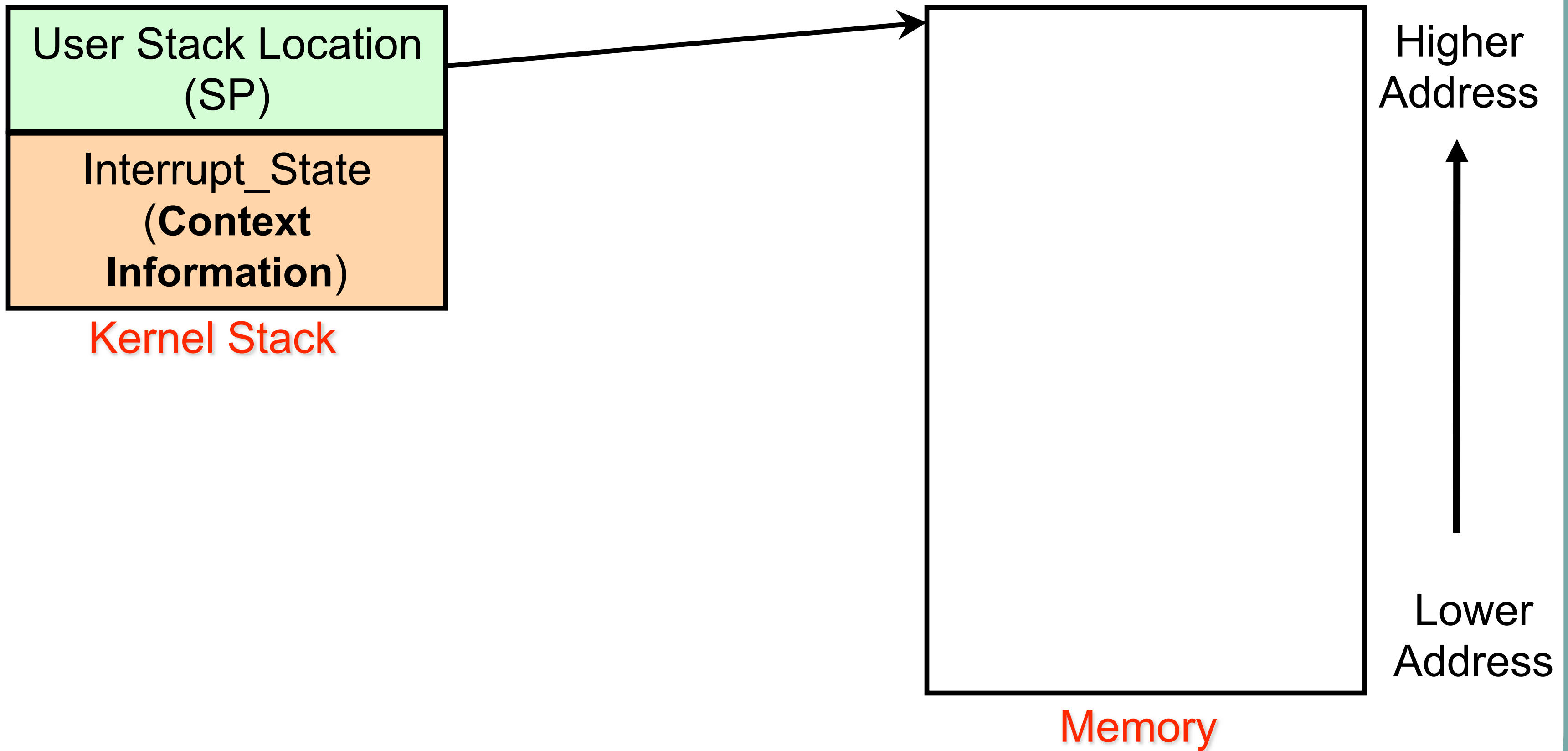
Check_Pending_Signal

- 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)

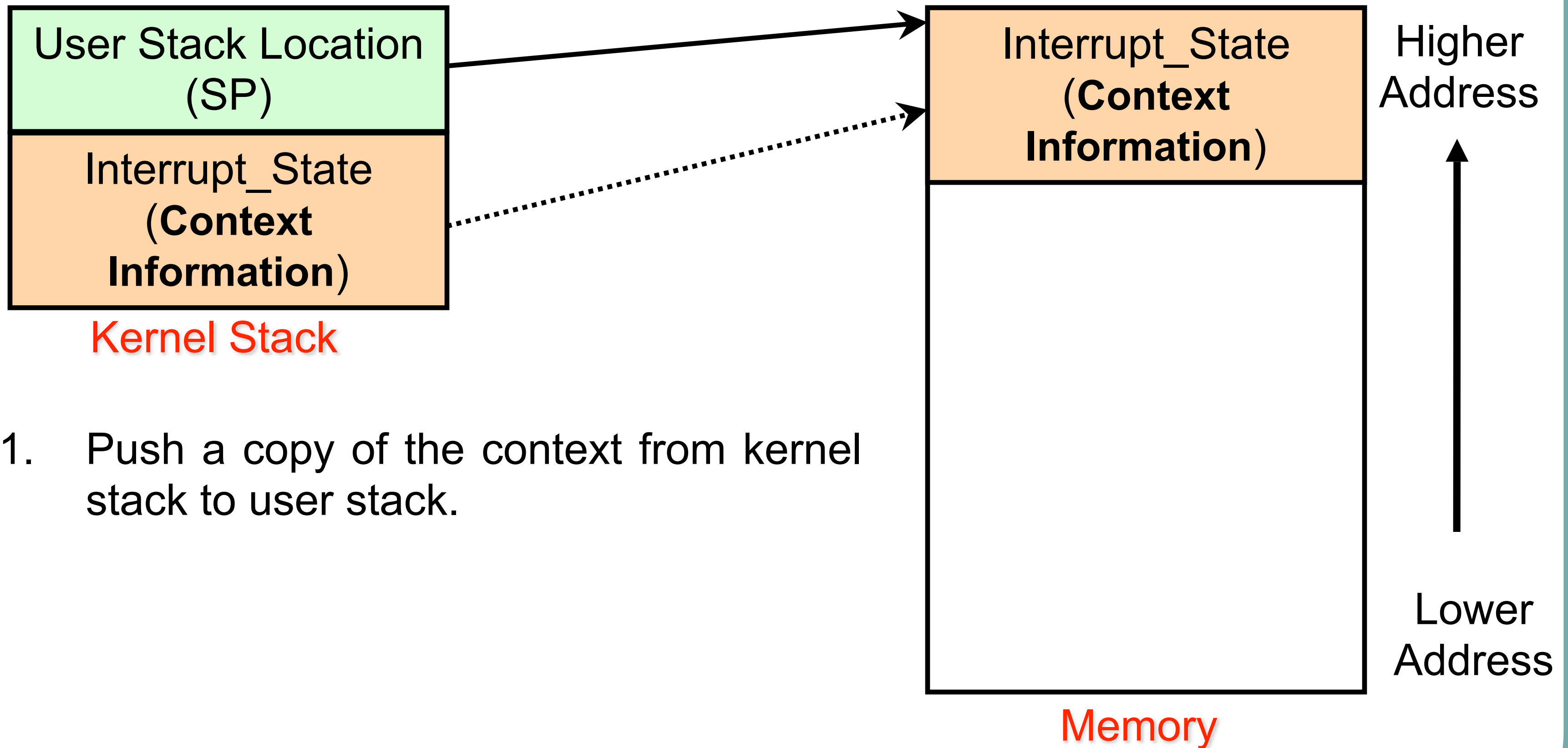
Check_Pending_Signal

- 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

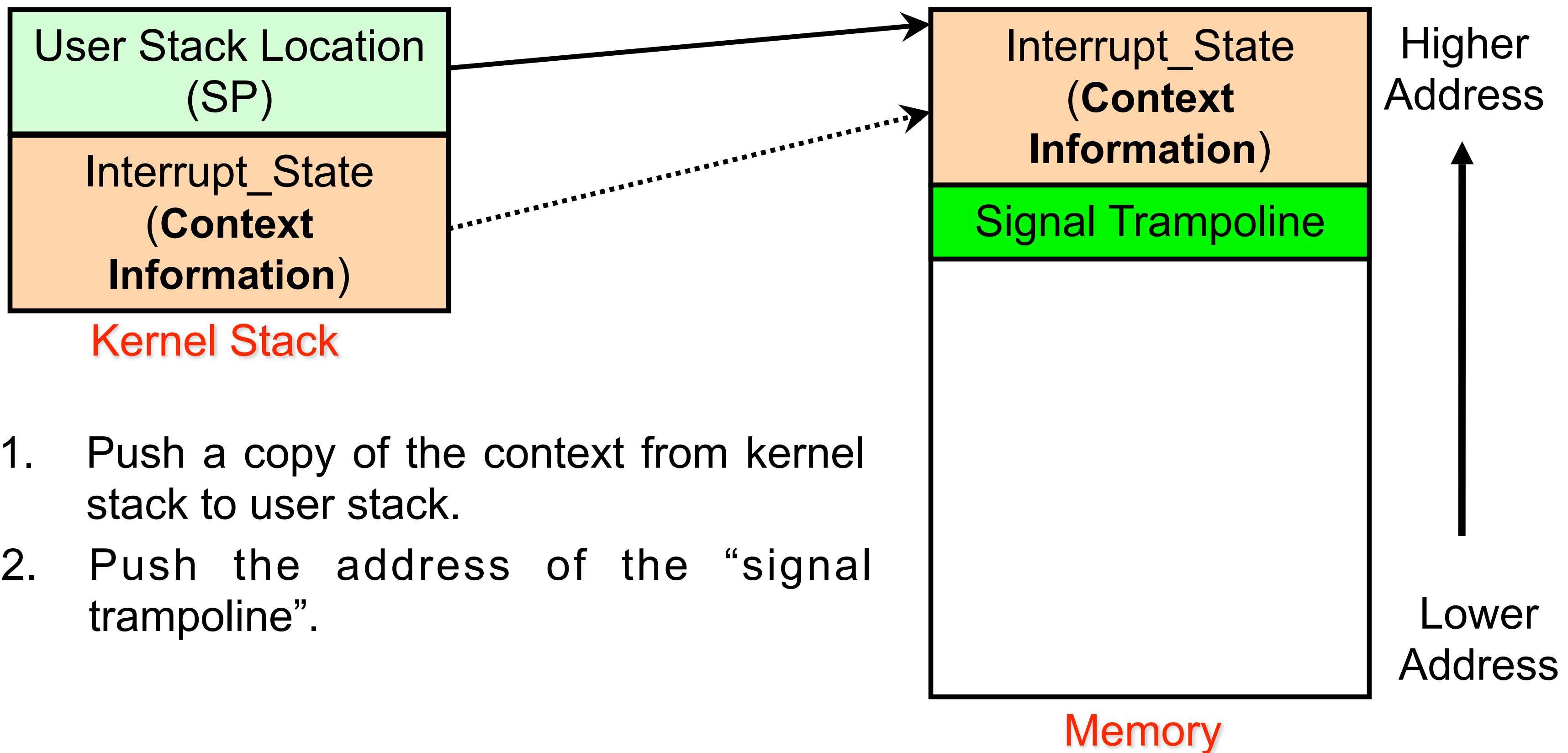


Setup_Frame



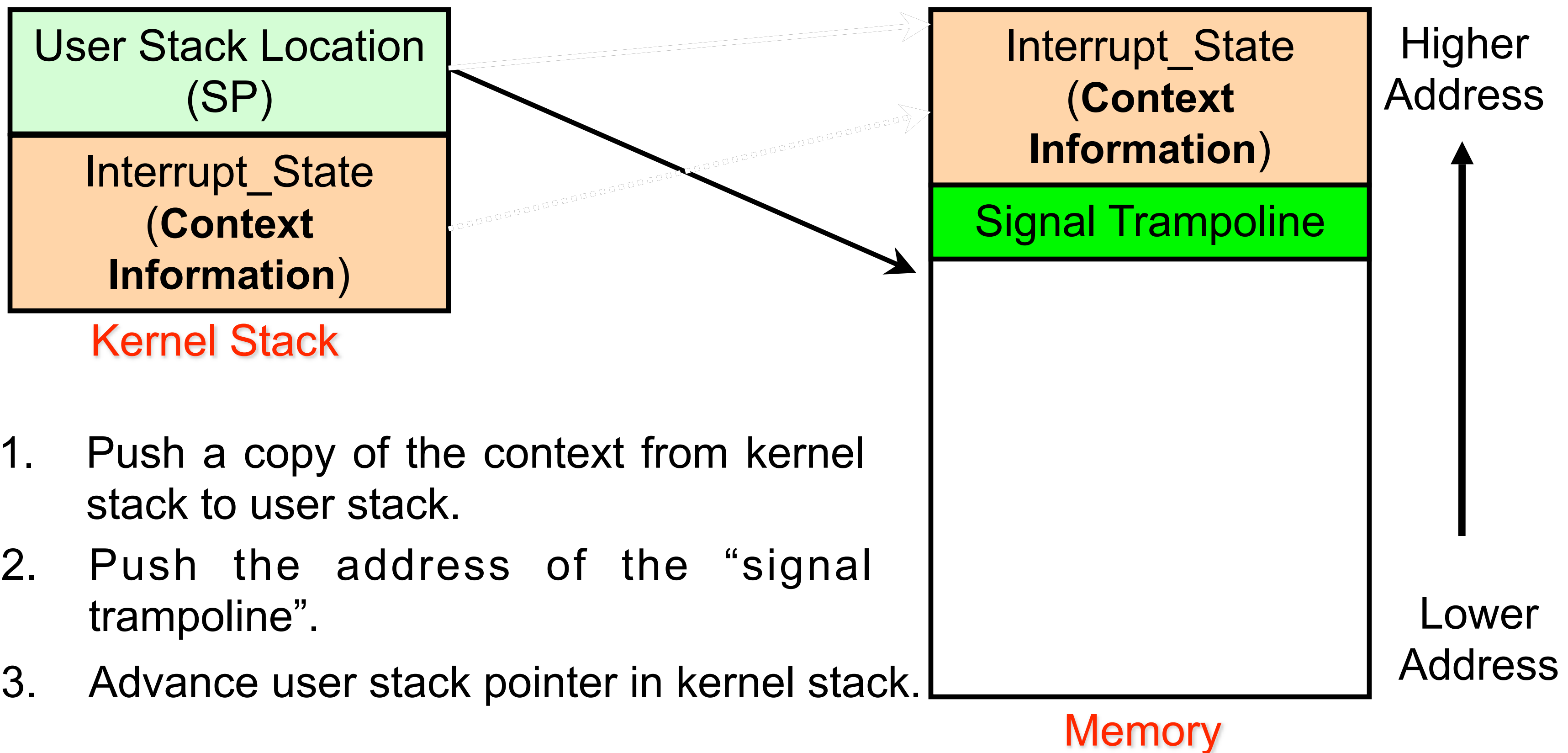
1. Push a copy of the context from kernel stack to user stack.

Setup_Frame



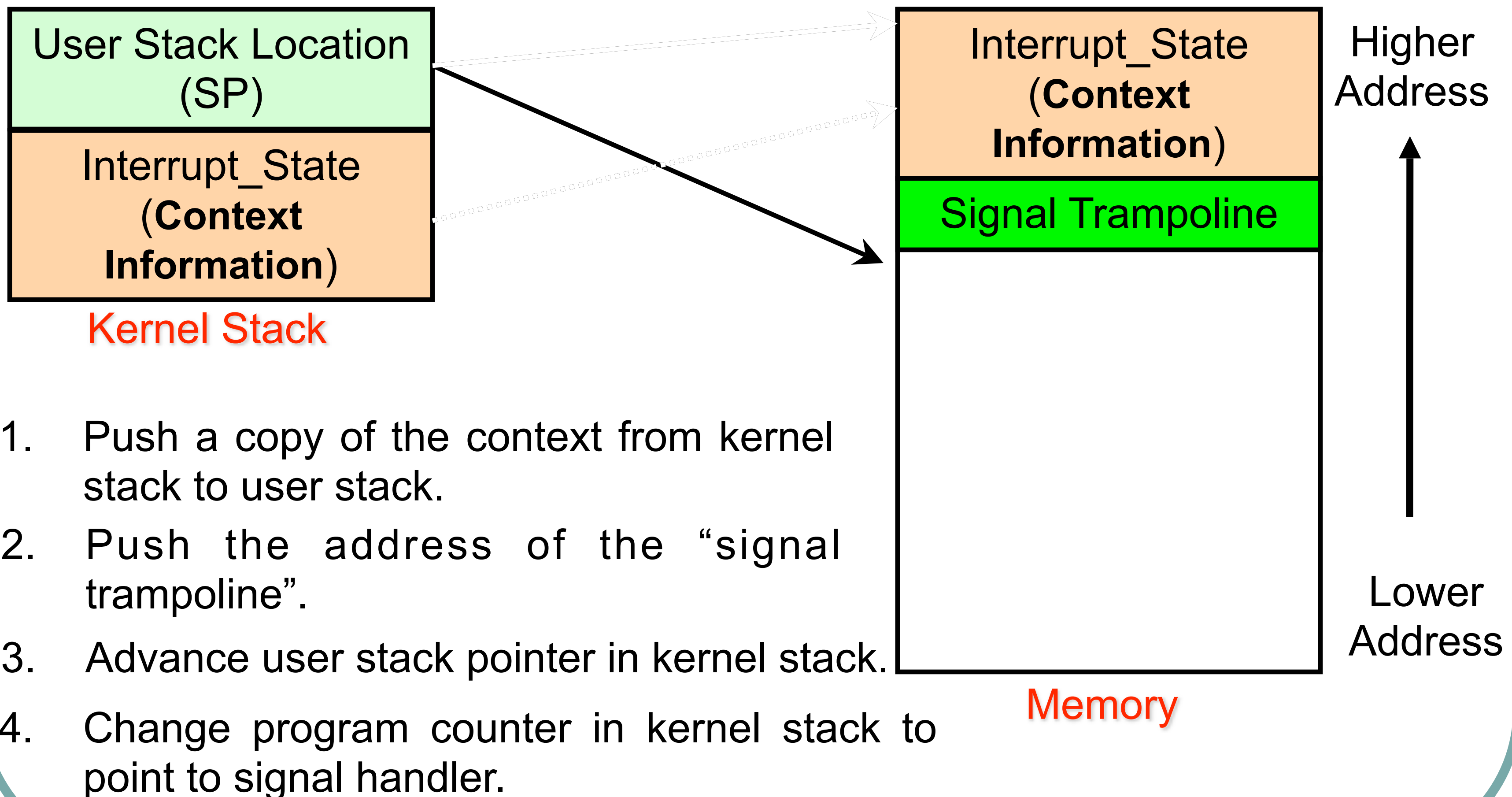
1. Push a copy of the context from kernel stack to user stack.
2. Push the address of the “signal trampoline”.

Setup_Frame

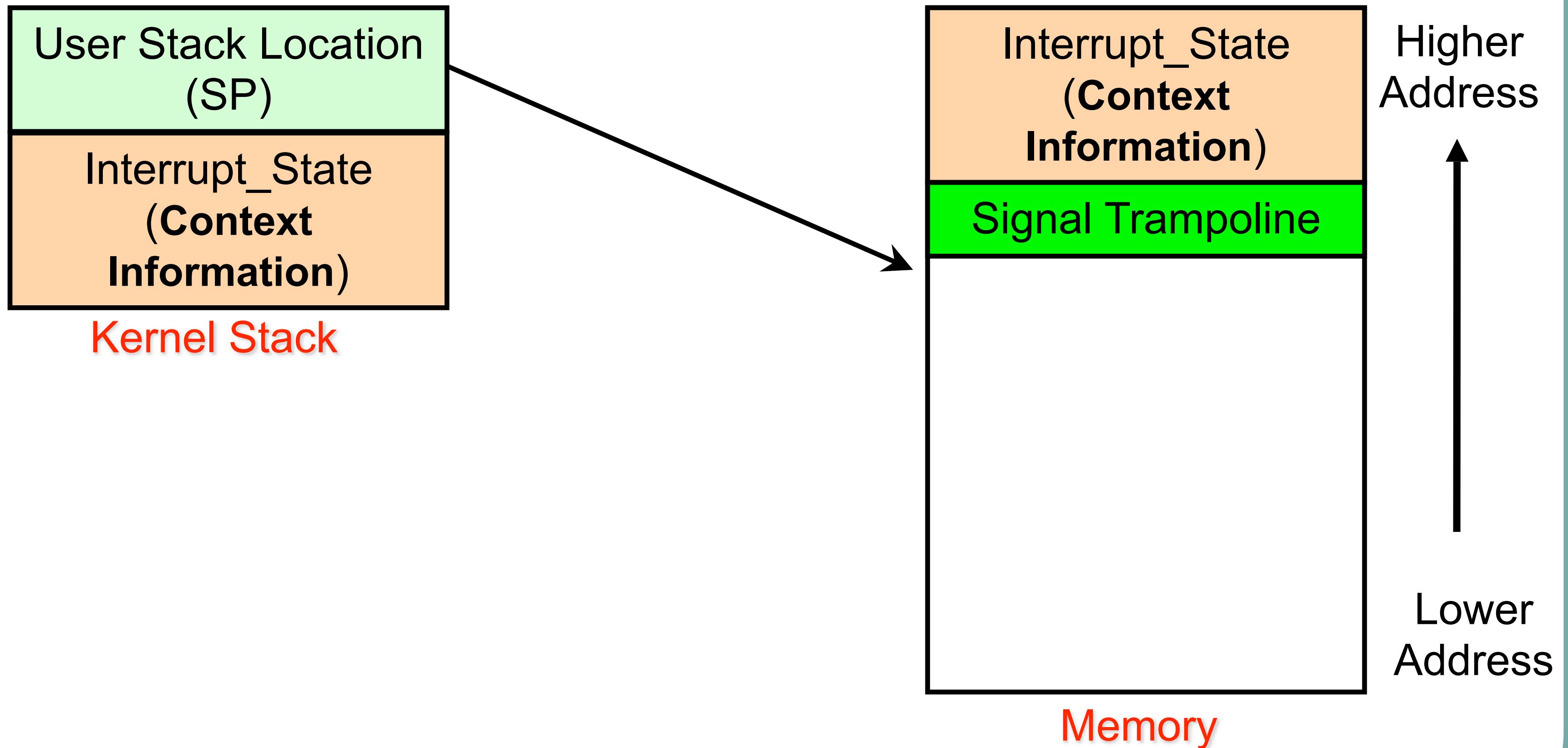


1. Push a copy of the context from kernel stack to user stack.
2. Push the address of the "signal trampoline".
3. Advance user stack pointer in kernel stack.

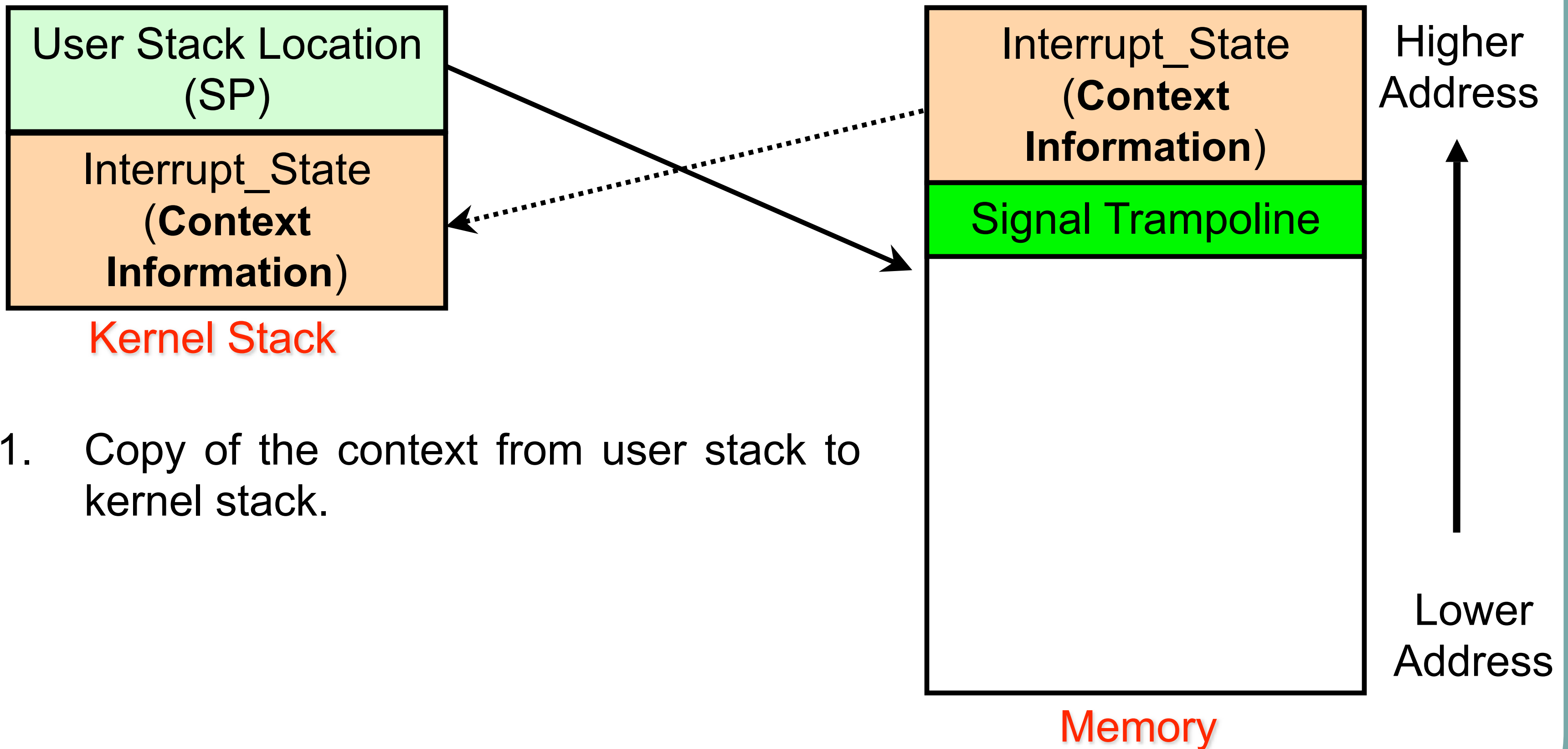
Setup_Frame



Complete_Handler

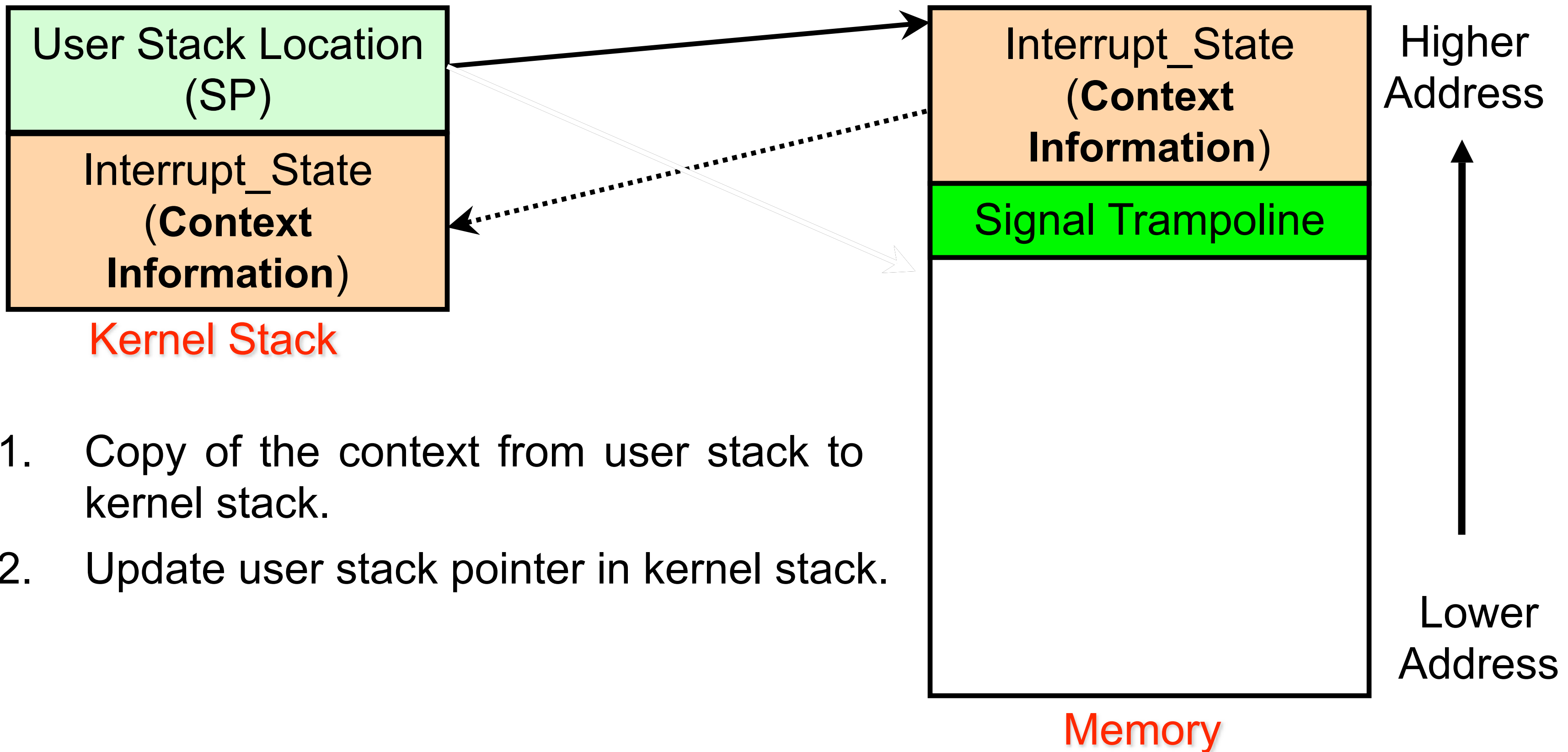


Complete_Handler



1. Copy of the context from user stack to kernel stack.

Complete_Handler



1. Copy of the context from user stack to kernel stack.
2. Update user stack pointer in kernel stack.