IMPLEMENTATION OF SIGNAL HANDLING

CS124 – Operating Systems

Winter 2015-2016, Lecture 15

Signal Handling

- UNIX operating systems allow user processes to register for and handle **signals**
  - Provides exceptional control flow mechanism for user processes
- User program registers a **signal handler** via a system call
- Example:

```
typedef void (*sig_t)(int);
```

  - `sig_t` is a function-pointer to a function that takes an `int` argument and returns `void`

```
sig_t signal(int sig, sig_t func);
```

  - This system call sets the signal handler for the specified signal type, and returns the previous signal handler
- Declarations are in C standard header `signal.h`

Signal Handling: Example

```
/* Print a message, then request another SIGALRM.  */
void handle_sigalrm(int sig) {
   printf("Hello!\n");
   alarm(1);  /* Request another SIGALRM in 1 second.  */
}

/* User typed Ctrl-C.  Taunt them. */
void handle_sigint(int sig) {
   printf("Ha ha, can't kill me!\n");
}

int main() {
   signal(SIGINT, handle_sigint);
   signal(SIGALRM, handle_sigalrm);
   alarm(1);  /* Request a SIGALRM in 1 second. */

   while (1) pause();  /* Wait for signals in a loop.  */

   return 0;
}
```

Advanced Signal Handling Support

- UNIX also provides more advanced signal handling:

```
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact)
```

- **`sigaction`** struct specifies various details, including the kind of handler function:
  - Either the simple **`void handler(int sig)`** as before…
  - Or, a more advanced handler function:
    `void sigact(int sig, siginfo *info, void *ctxt)`
- The **`siginfo`** struct includes many details about signals
  - e.g. sending process ID, memory address that caused fault, etc.
- **`ctxt`** points to a **`ucontext_t`** structure
  - A platform/architecture-dependent machine context, containing the CPU state of the user process, when it was interrupted by signal
  - Facilitates e.g. user-space threading libraries

Pending and Blocked Signals

- The kernel maintains two bit-vectors for every process
  - Every type of signal has a specific bit in this bit-vector
- **`pending`** bit-vector records what signals have yet to be delivered to the process
  - Note: if multiple instances of a given signal occur before a process receives the signal, it will see only one instance of the signal
  - Signals indicate one or more events of given type have occurred
- **`blocked`** bit-vector records what signals are currently not allowed to be delivered to the process
  - Can have a signal that is both blocked and pending
  - When the signal is unblocked, it will be delivered to the process
- When a signal is delivered to a process, that type of signal is automatically blocked for the process
  - Prevents a given signal handler from interrupting itself
  - One kind of signal can interrupt another kind of signal

The Kernel and Signal Handling

- How does the kernel provide signal handling to user processes?
- When is a signal delivered to the receiving process?
  - What if process is running on the CPU? What if ready or blocked?
  - How does signal handling affect process scheduling?
- If a process has multiple signals pending, how does the kernel dispatch all these signals to the process?

- Note: we are ignoring <u>many</u> issues caused by using signals in multithreaded programs
  - Individual threads can block signals so that only one thread handles signals, etc.

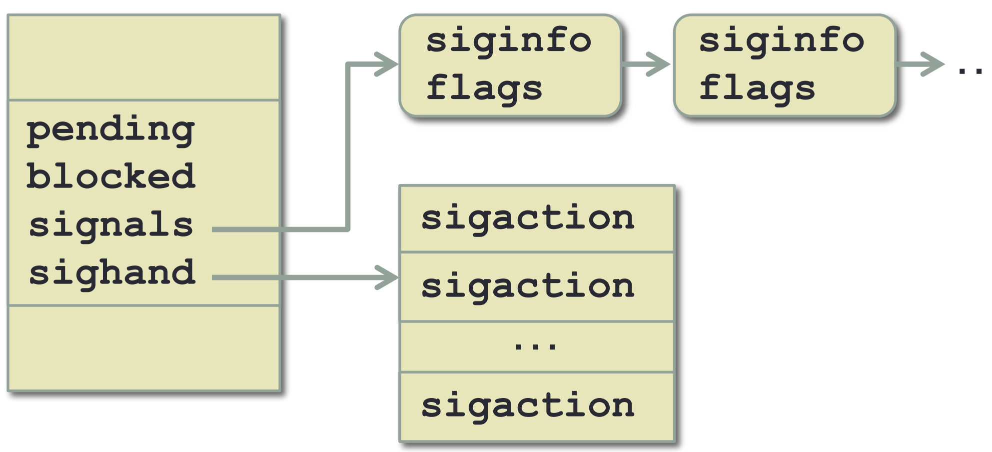
Generating and Delivering Signals

- A process isn't always running when a signal is sent to it
  - e.g. **`kill()`** syscall is invoked by another process
  - e.g. a child process dies, causing **`SIGCHLD`** to be sent to parent, but a higher priority process currently preempts the parent
- Kernels make a distinction between *generating* a signal and *delivering* the signal
- <u>Signal generation</u>: kernel updates the data structures of the receiving process to record that the signal was sent
- <u>Signal delivery</u>: kernel forces the receiving process to respond to the signal (e.g. by invoking a signal handler)
- Time may pass between generating and delivering signal

Process Signal Data Structures

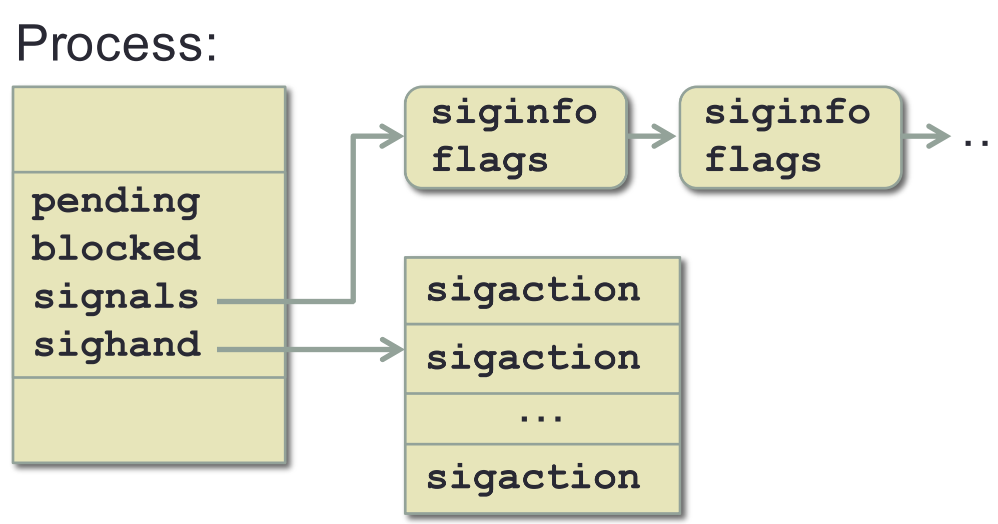
- Already mentioned **`pending`**/**`blocked`** signal bit-vectors
  - At a coarse-grain level of detail, records which signals need to be delivered, and which signals are currently blocked from delivery
- Each process also has a linked list of pending signals
  - **`siginfo_t`** struct records relevant details of the pending signal
- Each process also has an array of "signal action" structs
  - Specifies how to handle each kind of signal
  - e.g. "default action," "ignore," or a user-space handler
  - (Flags also record other options for handling signals)

Process:



Generating a Signal

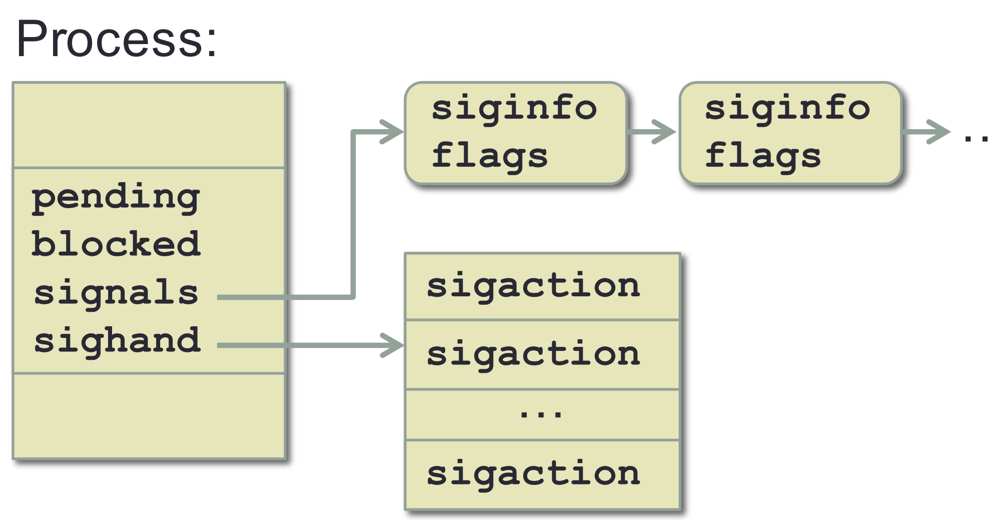
- When a signal is sent to a process:
  - The kernel invokes a specific function to update the process' signal structures, perform scheduling tasks, etc.
  - e.g. Linux 2.6 has **`specific_send_sig_info()`** kernel function
- If the process already has a pending signal of that type, the new signal is ignored
  - For real-time signals, this test is skipped
  - Every occurrence of a real-time signal is delivered
- If the process is ignoring the signal, nothing is done
  - No structures are updated
  - No scheduling tasks occur



Generating a Signal (2)

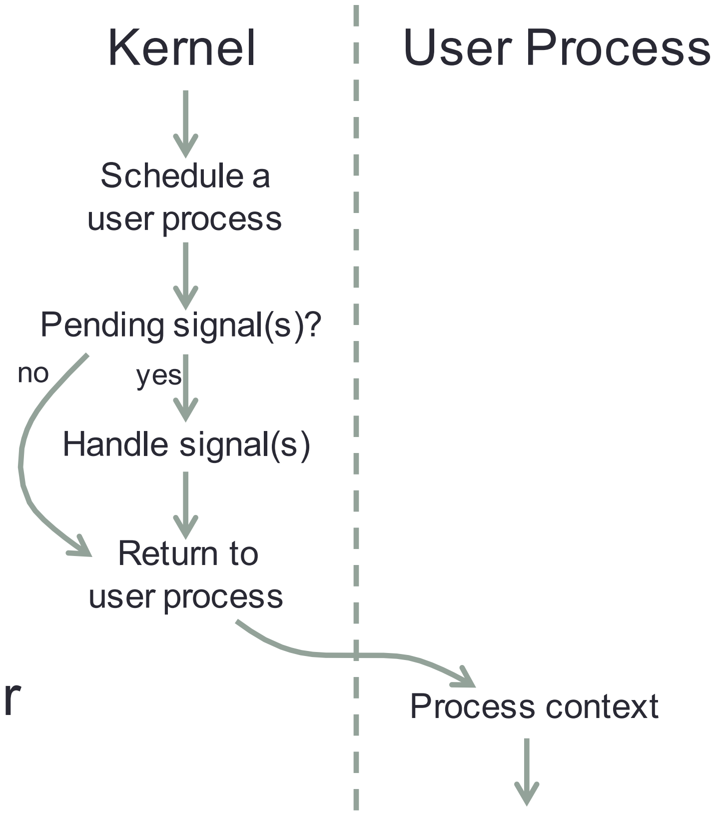
- Otherwise, new signal is appended to the signal queue
  - The **`pending`** bit-vector is also updated
- If the process is currently blocked or suspended, it is moved to the ready state

- Note: a few signal types are <u>not</u> added to signal queue
  - e.g. **`SIGKILL`**, **`SIGSTOP`**
  - These signals are enforced immediately by the kernel the next time the process runs
  - Affects the process' execution state in the scheduler



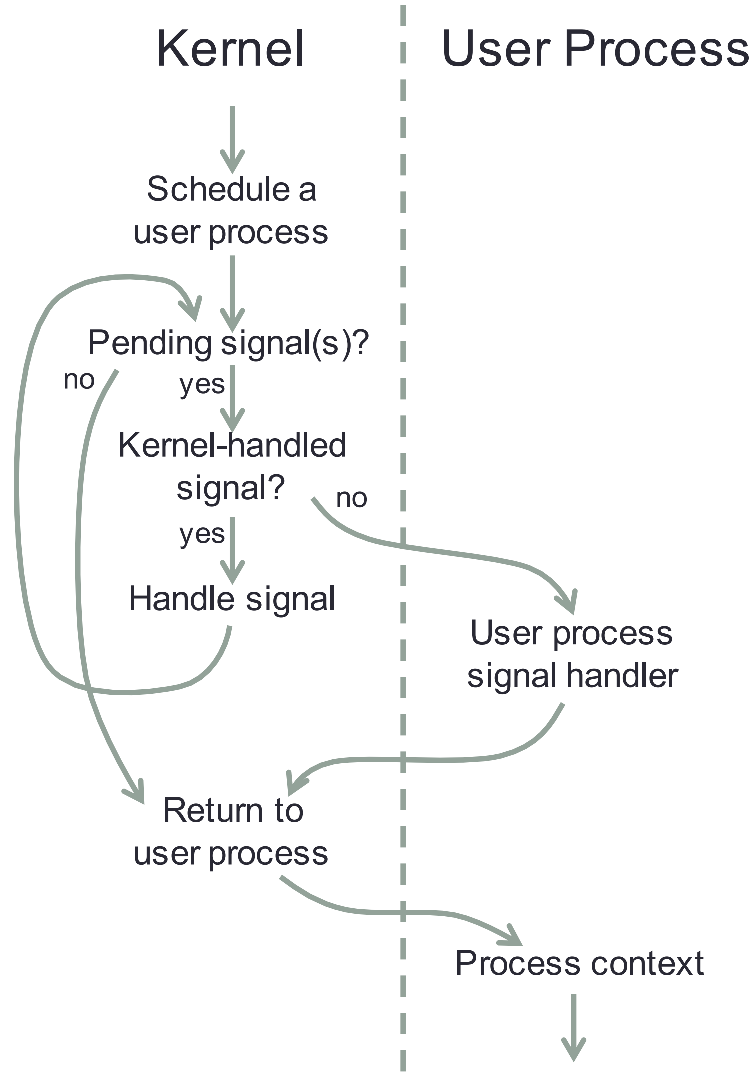
Delivering Signals

- Signals are only delivered to the currently running process
  - Obvious; the process must hold the CPU to run the signal handler
- The kernel checks for pending signals when it is about to return back to the user process
  - Kernel checks the process' signal state
  - If there are pending signals to deliver, they are delivered at this point
- Two ways a signal can be handled:
- Signal is ignored, or default action is to be performed
  - These signals are handled by the kernel
- Signal has a user-mode handler
  - Kernel must invoke the user-mode handler



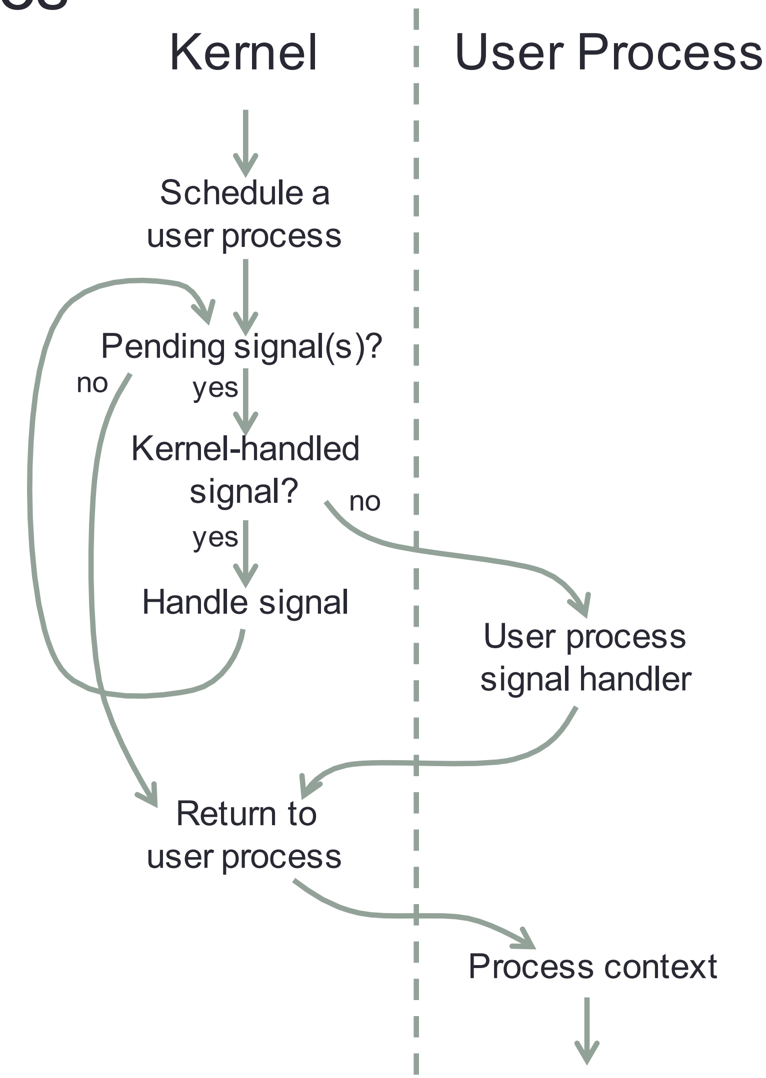
Delivering Signals (2)

- Two ways a signal can be handled:
  - Signal is ignored, or the default action is to be performed
  - Signal has a user-mode handler
- For kernel-handled signals, it can handle as many as are pending
- For user-process-handled signals, only one signal is handled
  - Other pending signals will be delivered the next time the scheduler is invoked
- In Linux 2.6 kernel, signal delivery handled by **`do_signal()`** function
  - This code has been greatly restructured in subsequent Linux kernel releases



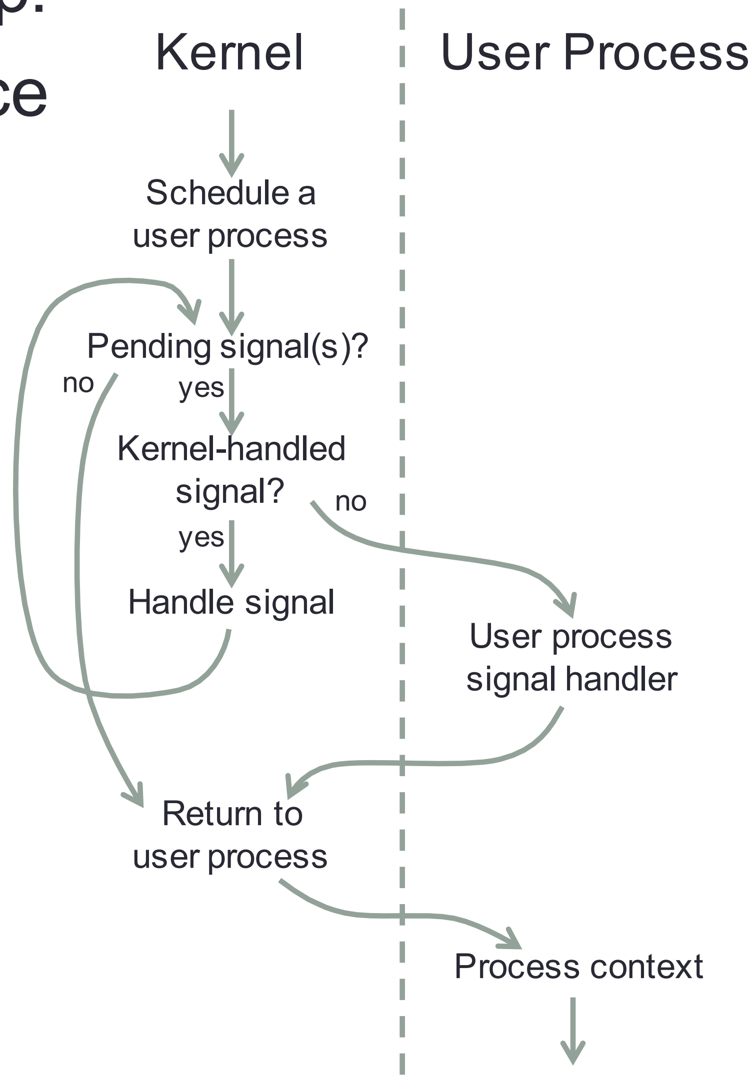
Delivering Signals (3)

- Several big signal delivery challenges
- Signal handler must return back to the kernel so that previous user process context can be restored
- Signals can interrupt system calls
  - Particularly on preemptible kernels
- Handlers can make system calls
  - When returning from syscall handler, must return to the signal handler, not the interrupted user process context
- Signals can `siglongjmp()` from the signal handler back to another part of the user process



Delivering Signals (4)

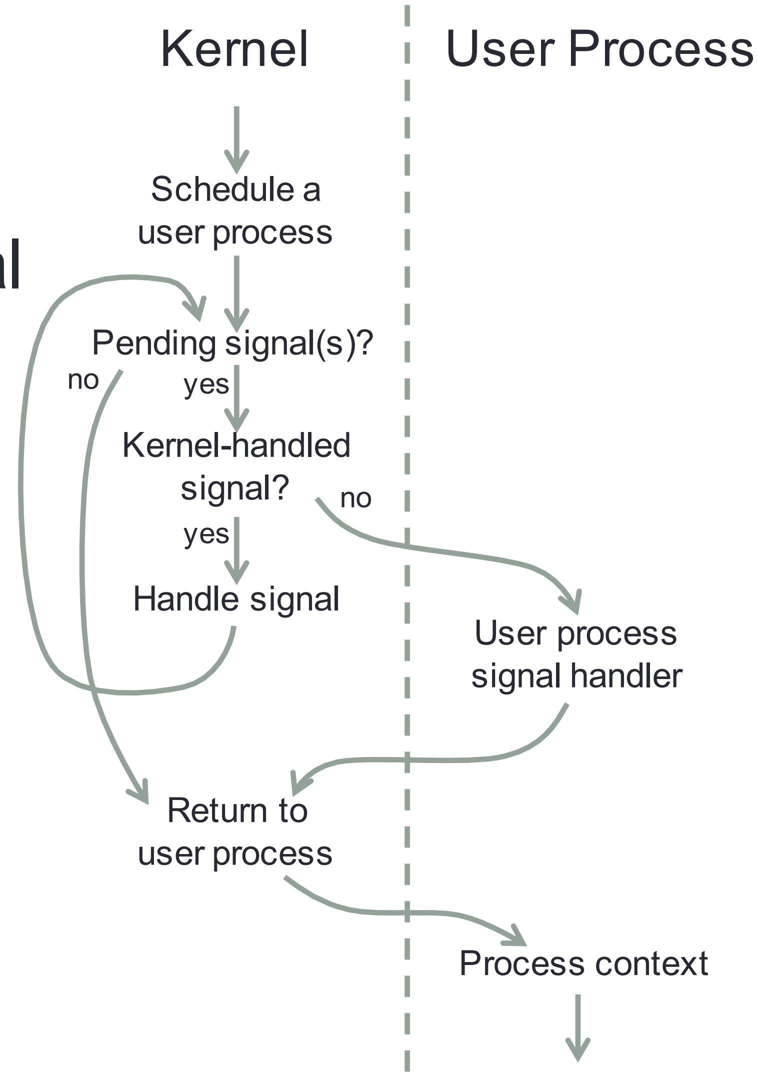
- Linux 2.6 **`do_signal()`** uses a loop:
- Try to find a pending signal to service
  - If there are multiple pending signals, kernel may choose to service them in a different order from queue order
  - (Some signals also cancel each other, like **`SIGSTOP`** and **`SIGCONT`**)
- If no more pending signals, the kernel returns to the user process
- If a pending signal was found, it is removed from pending queue
  - User process state is also updated to mark the signal as no longer pending



Delivering Signals (5)

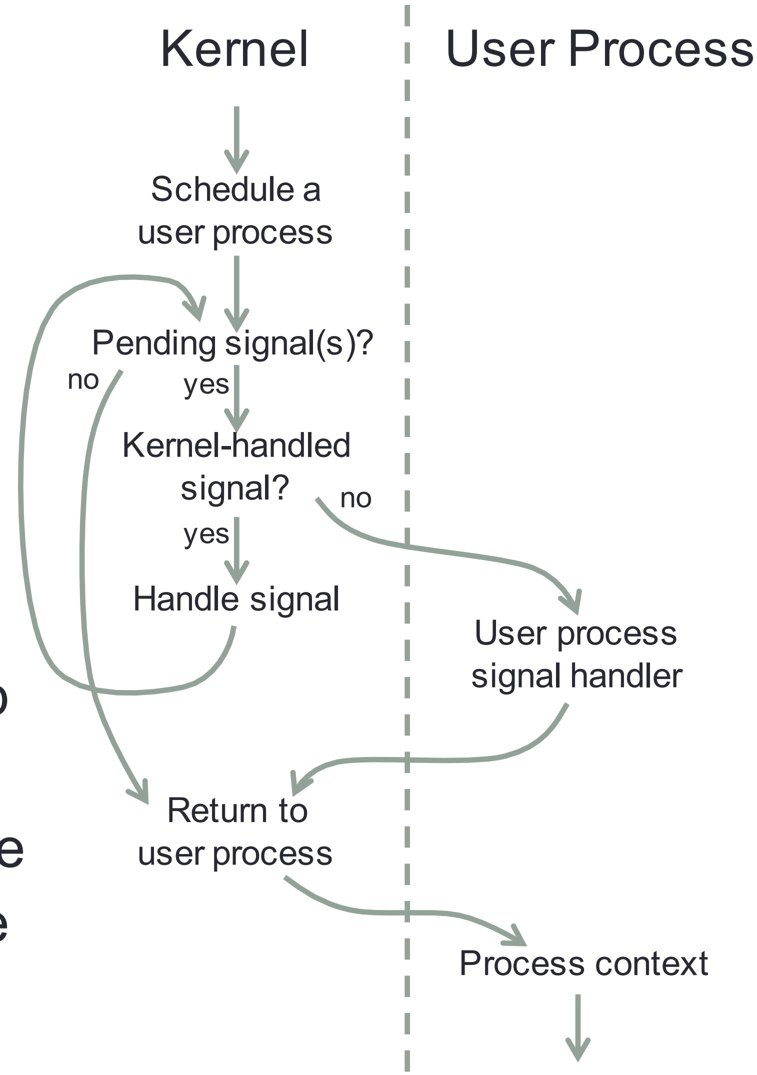
- Linux 2.6 `do_signal()`, cont.

- If signal is currently being ignored, `do_signal` continues to next signal
  - A signal may be ignored if the process has set its handler to `SIG_IGN`
  - Or, if signal's handler is `SIG_DFL` and default action is to ignore the signal, the signal is ignored
    - e.g. `SIGCHLD` is ignored by default



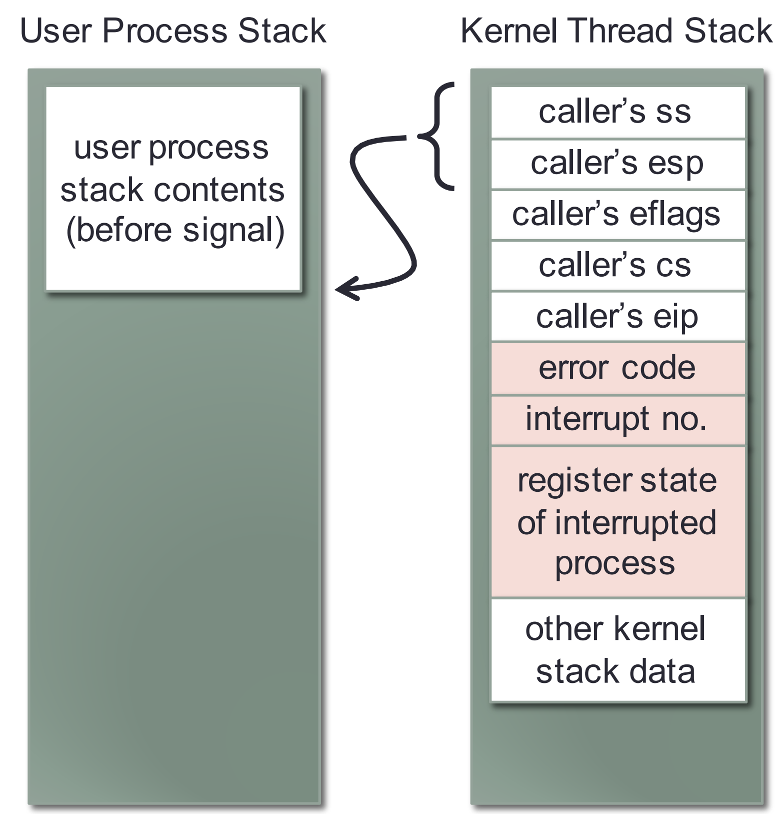
Delivering Signals (6)

- Linux 2.6 **`do_signal()`**, cont.
- Next, **`do_signal`** checks if the default action should be used
  - Signal's handler is set to **`SIG_DFL`** (and default action is not to ignore)
  - If so, **`do_signal`** carries out the action
- Default actions:
  - **Terminate** – kill the process
  - **Dump** – kill process, create a core dump
  - **Ignore** – ignore signal (handled earlier)
  - **Stop** – move process to suspended state
  - **Continue** – move process to ready state



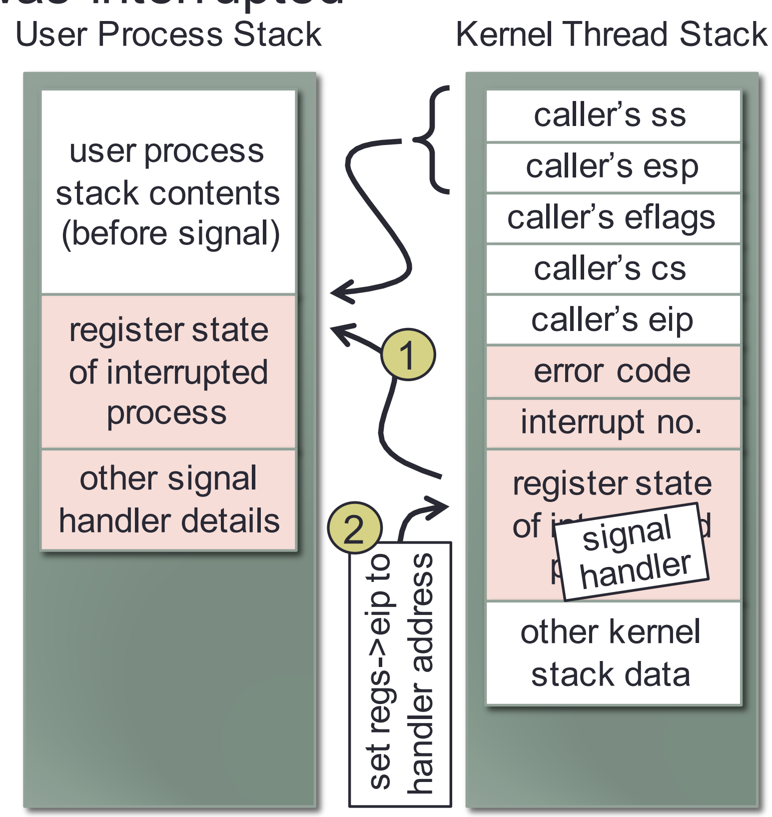
Delivering Signals (7)

- If none of the previous cases hold, **`do_signal`** must invoke the user process' signal handler
- Recall the state of our stacks within **`do_signal`** call:
- It's easy to cause the user process to run the signal handler…
  - Just set the process' EIP to the address of the signal handler
- Problem: can't just overwrite the previous CPU state of the process
  - When signal handler completes, must return to whatever was interrupted in the user process
- Must set up a new CPU context for the signal handler to use



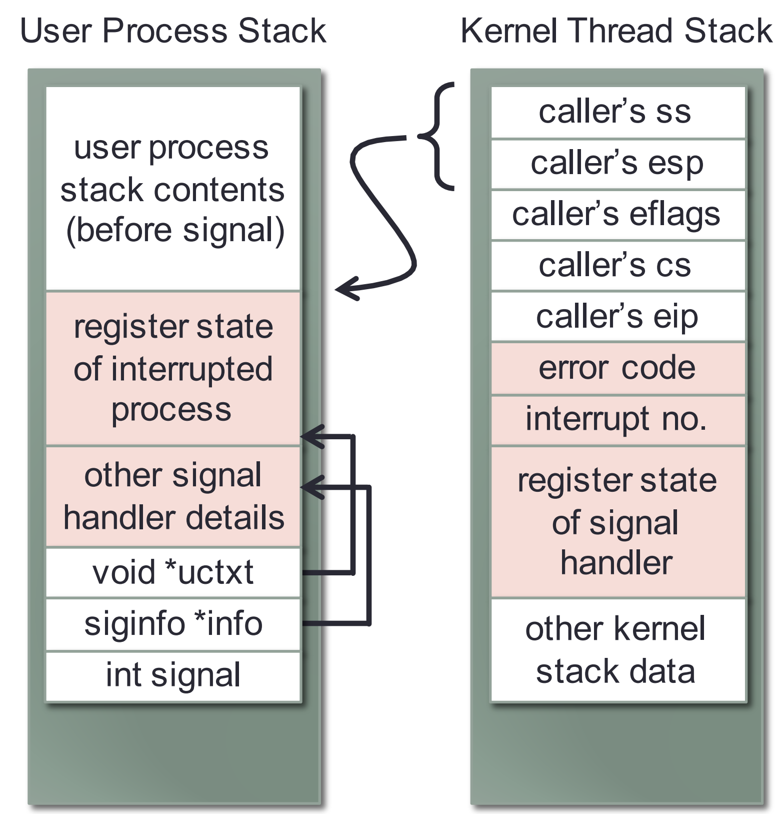
Delivering Signals (8)

- Another problem: the CPU context of the user process' interrupted execution is on the kernel stack…
  - …but kernel stack will be emptied when kernel returns to user mode!
- Solution: **`do_signal`** copies some critical details to user stack
  - CPU context of the user process before it was interrupted
  - Other details necessary for properly completing signal handler invocation
    - e.g. a bit-vector of blocked signals, other saved registers not in CPU context, etc.
- These details are used when the signal handler returns:
  - The kernel uses them to go back to the previous point in the interrupted process
- Then, the kernel can change the CPU context on the kernel stack to invoke the signal handler



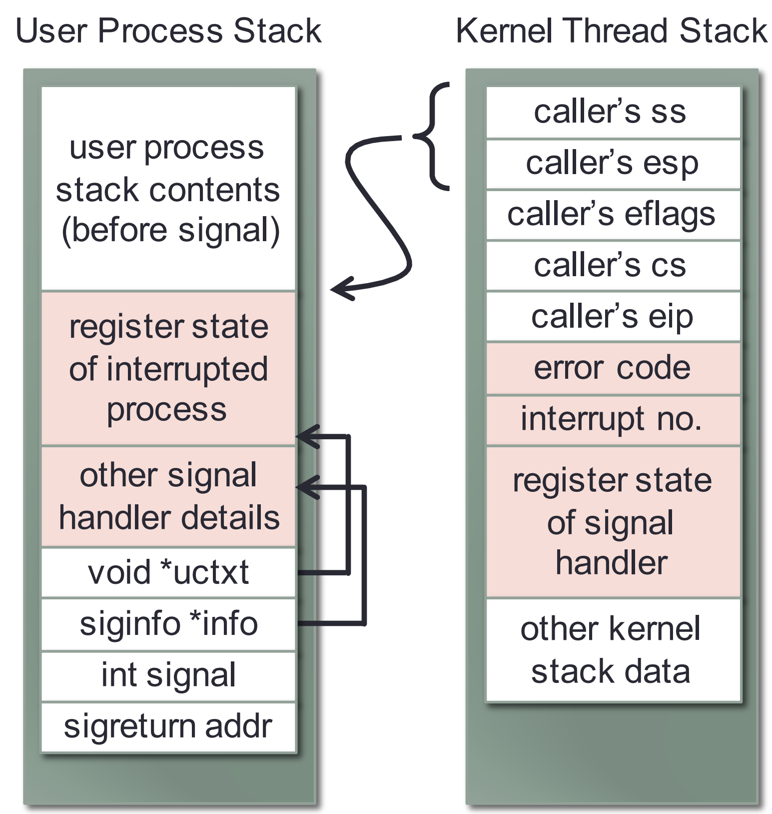
Delivering Signals (9)

- Finally, **`do_signal`** must set up the stack frame for signal handler function to use
  - If calling a 1-arg signal handler, just push signal # onto stack
  - If calling a 3-arg signal handler, also push signal details onto stack
- Since the process' interrupted CPU context is already on the stack, the "machine context" pointer is easy
  - And, it allows signal handlers to modify the process' CPU state directly, e.g. to implement user-mode threading libraries



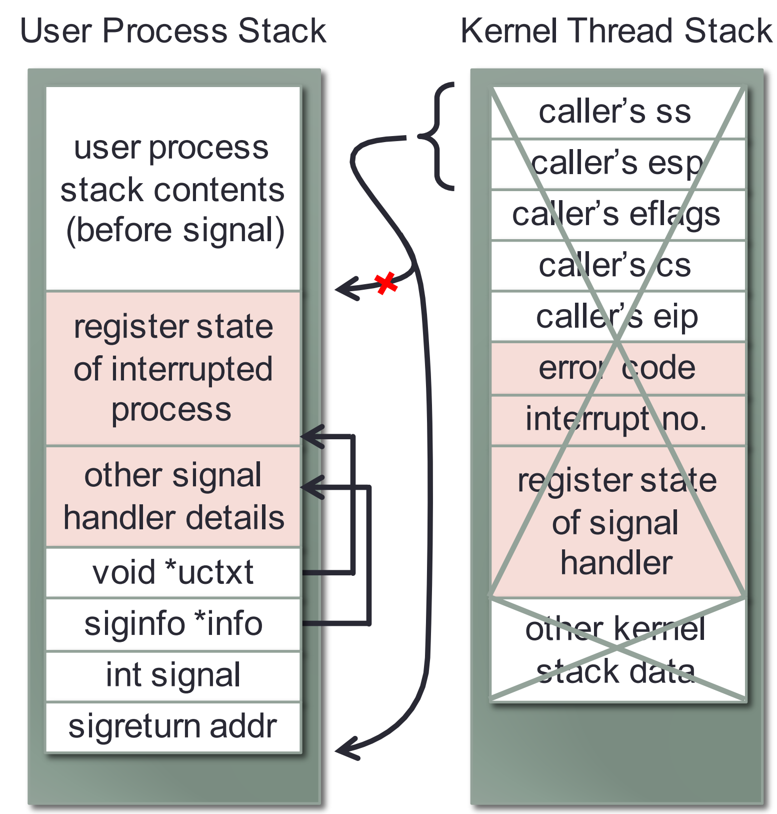
Delivering Signals (10)

- One last component needed for the stack frame:
  - The return address for when the signal handler returns
- Need the signal handler to return to the kernel:
  - Allow kernel to complete final signal-handling tasks, and restore the interrupted process' original context
- The kernel inserts address of code to invoke the **`sigreturn`** syscall
  - i.e. a wrapper to code that executes "mov NR_sysreturn, %eax; int $0x80"
- **`sigreturn`** has a single purpose:
  - Perform the final task of restoring the interrupted process' CPU context from its location on the stack



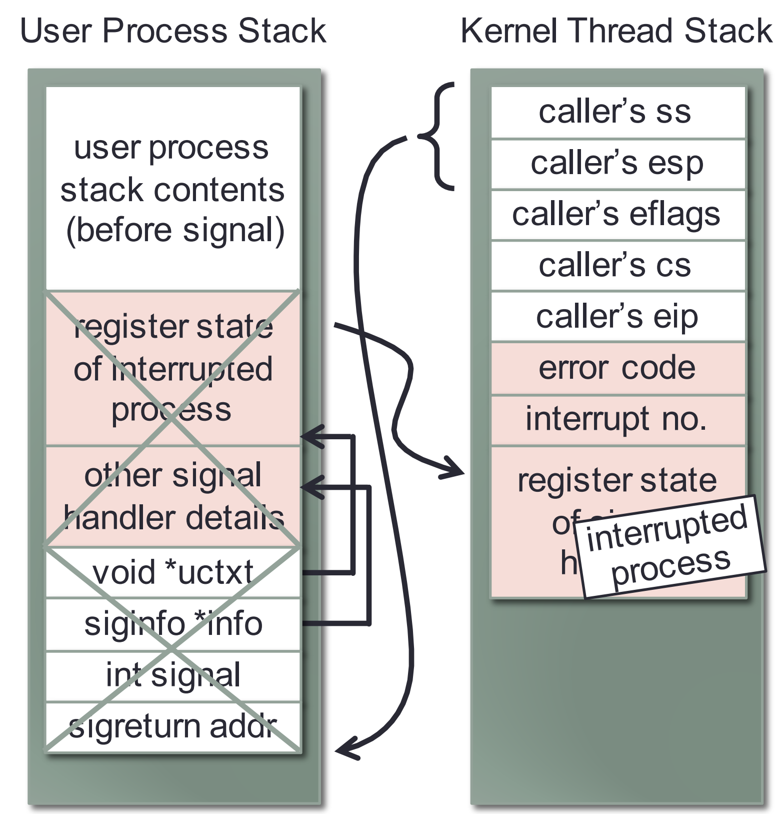
Delivering Signals (11)

- Since user stack was modified, must update caller's **`esp`**
- Finally, **`do_signal()`** is done
  - **`do_signal`** returns to its caller inside the kernel…
  - The kernel returns to user mode…
  - Register state of the user process is restored from the kernel stack…
  - User process begins executing the signal handler!
- Signal handler can make system calls with no problems
  - Original CPU context of the interrupted process is safely stored on user stack



Delivering Signals (12)

- When the signal handler returns, the **`sysreturn`** system call is invoked
  - Note: again the kernel thread stack contains user-process details
- The **`sysreturn`** syscall copies the original CPU context of interrupted process back into the kernel stack
  - (along with any changes to the CPU context made by the signal handler)
- Then, **`sysreturn`** returns back to the user process
  - Resumes process execution at the point where the signal interrupted the process
  - User stack pointer is restored as well, eliminating now-unneeded stack data

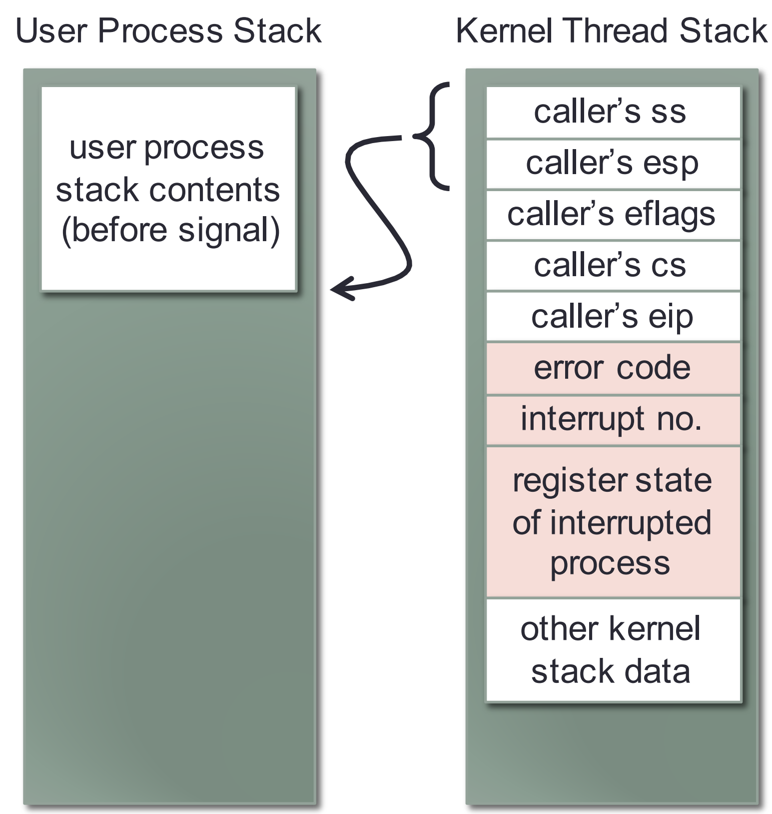


Signals and System Calls

- Process may be blocked on a syscall when signal occurs
  - The system call is interrupted by the signal
  - Generally, some action must be taken by the kernel in this case
  - e.g. the system call might return **`EINTR`** as its error code
- Some system calls can be automatically restarted when the signal occurs
  - e.g. **`read()`** or **`write()`** may be automatically restarted if they hadn't performed any work by the time the signal occurs
- Some system calls must report that they were interrupted
  - e.g. if **`nanosleep()`** is interrupted by a signal, it returns **`EINTR`**, and records the amount of time remaining in the sleep interval

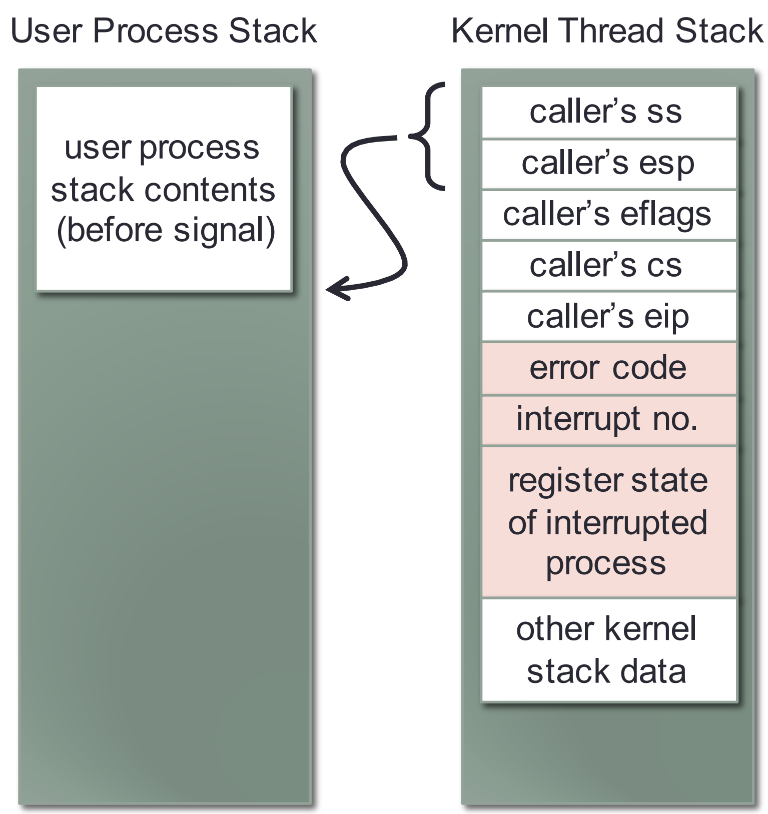
Signals and System Calls (2)

- If a process is in a system call when it receives a signal:
  - It was previously in kernel code when the signal is delivered…
  - It entered the kernel by performing an **`int $0x80`** trap operation (or a fast system-call, e.g. using **`sysenter`** instruction)
- The signal delivery mechanism will see the interrupt # in process' state
  - Original value of **`eax`** is also stored, indicates which syscall was interrupted



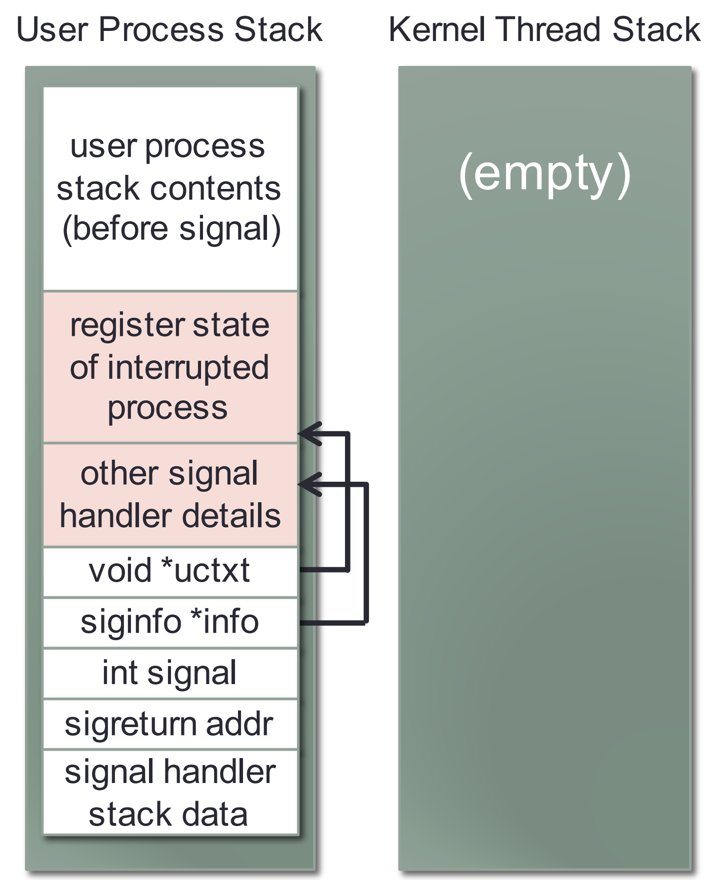
Signals and System Calls (3)

- Depending on the process' configuration, and on system call that was interrupted, kernel takes one of two actions:
- Option 1: the kernel modifies the interrupted process' CPU context to show the system call as interrupted
  - Sets the process' **`eax`** register to **`-EINTR`**
  - The process will see the system call return this error code
- Option 2: the kernel modifies process' CPU context to rerun the system call
  - Sets the process' **`eax`** register to the original system call number
  - Subtracts 2 from process' **`eip`** register to force **`int $0x80`** (or **`sysenter`**) to run again



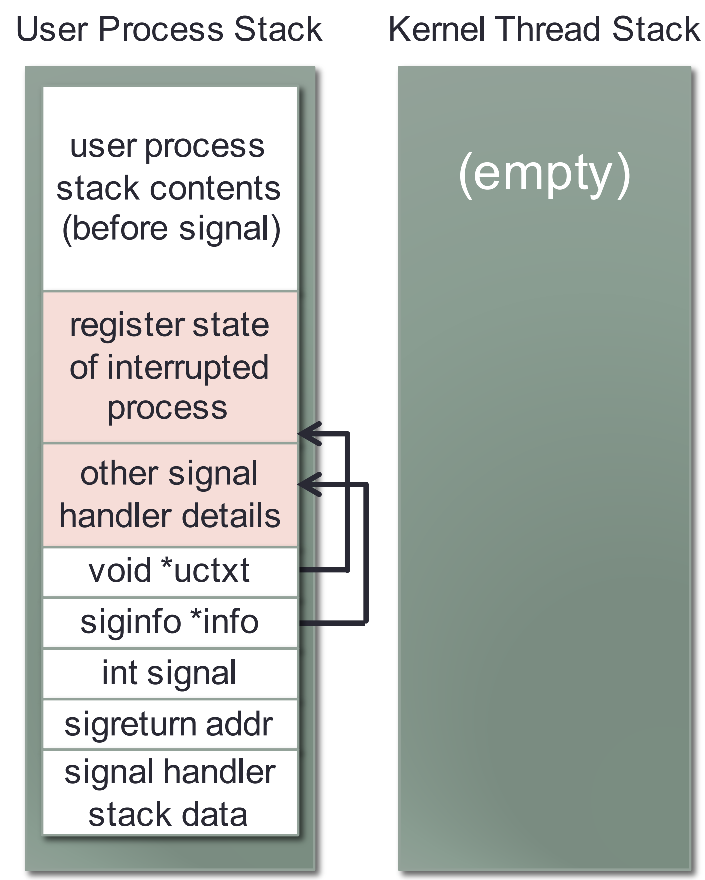
Signal Handlers and `siglongjmp`

- Signal handlers can use **`siglongjmp()`** to jump to another part of the user program
  - Effectively terminates the signal handler and resumes execution of the user program at a different point
  - Described as a "non-local goto"
- In fact, could use **`longjmp()`** or **`siglongjmp()`**
  - **`sigsetjmp`** / **`siglongjmp`** are strongly preferred because they can optionally save and restore blocked-signals mask
  - In practice, that is the only difference between **`setjmp`** and **`sigsetjmp`**



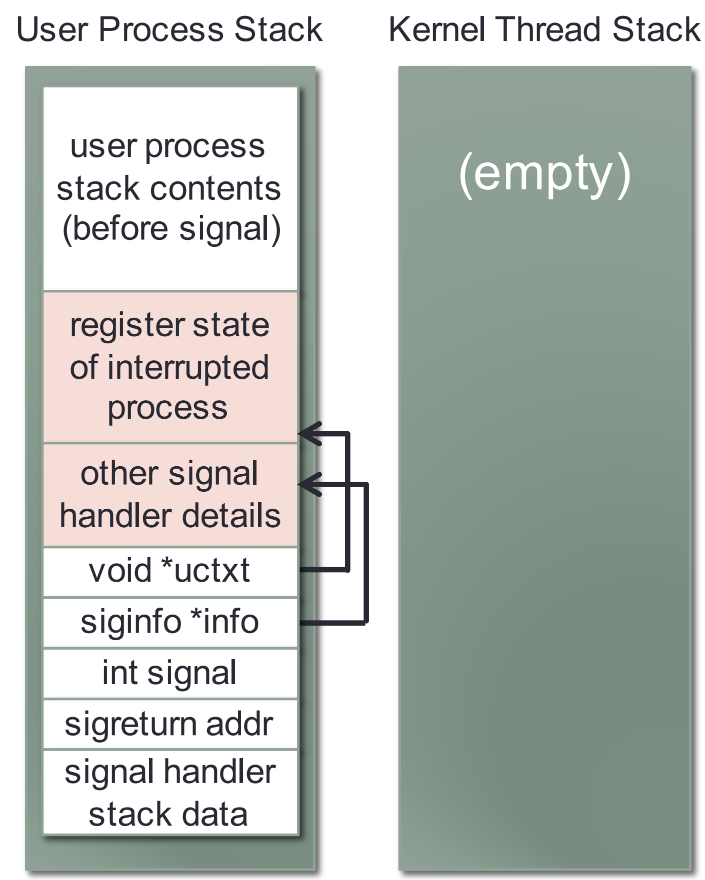
Signal Handlers and `siglongjmp` (2)

- Does performing a **`siglongjmp`** from a signal handler break the kernel's signal-delivery machinery at all?
- Answer should be obvious from the stack state, and from previous discussion
- Recall:
  - When a pending signal is dequeued by **`do_signal()`**, process' state is updated to show the signal as no longer pending
  - Kernel doesn't require the signal handler to return in order to "complete" processing
    - As far as kernel is concerned, signal is done!
    - Only task is to restore blocked signal mask (which **`siglongjmp`** can also take care of)



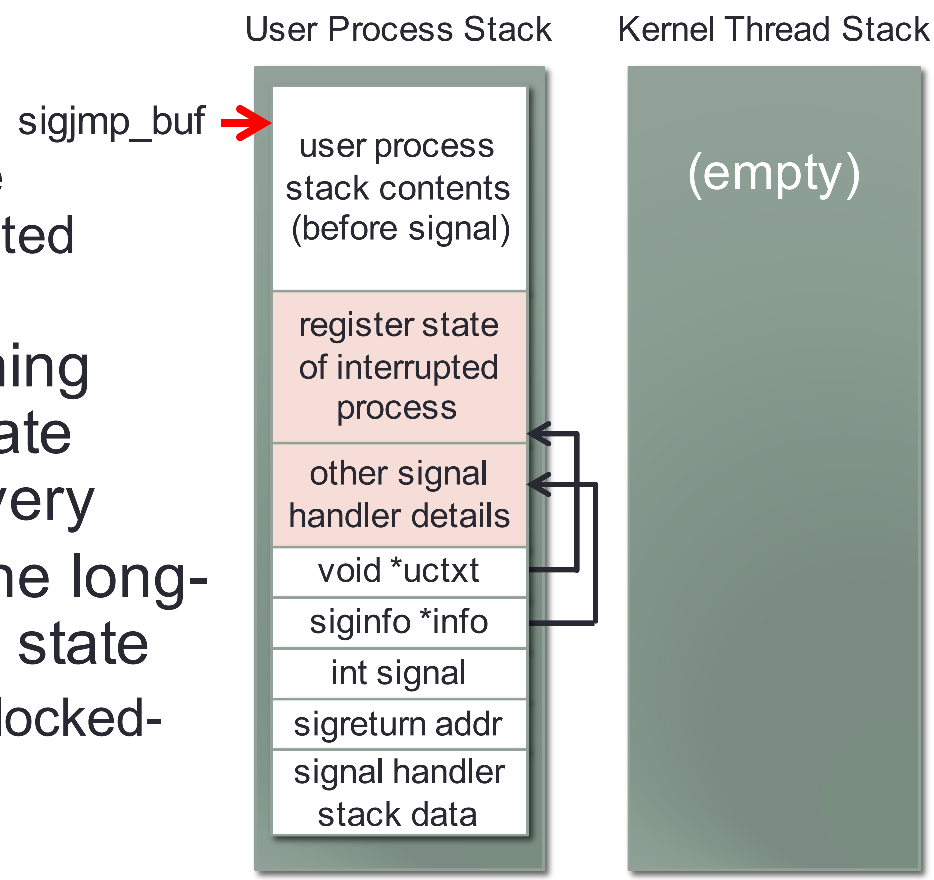
Signal Handlers and `siglongjmp` (3)

- Also: kernel will always check for pending signals to deliver, when returning to the user process
  - Any pending signals will likely be delivered the next time the scheduler starts running the process again
- Finally, the kernel's signal-delivery mechanism leaves no state on the kernel stack
  - All state for both the signal handler and the interrupted point in the process is contained within the user stack
- If a user process long-jumps out of signal handler, it will have <u>no effect</u> on delivery of any future signals



Signal Handlers and **`siglongjmp`** (4)

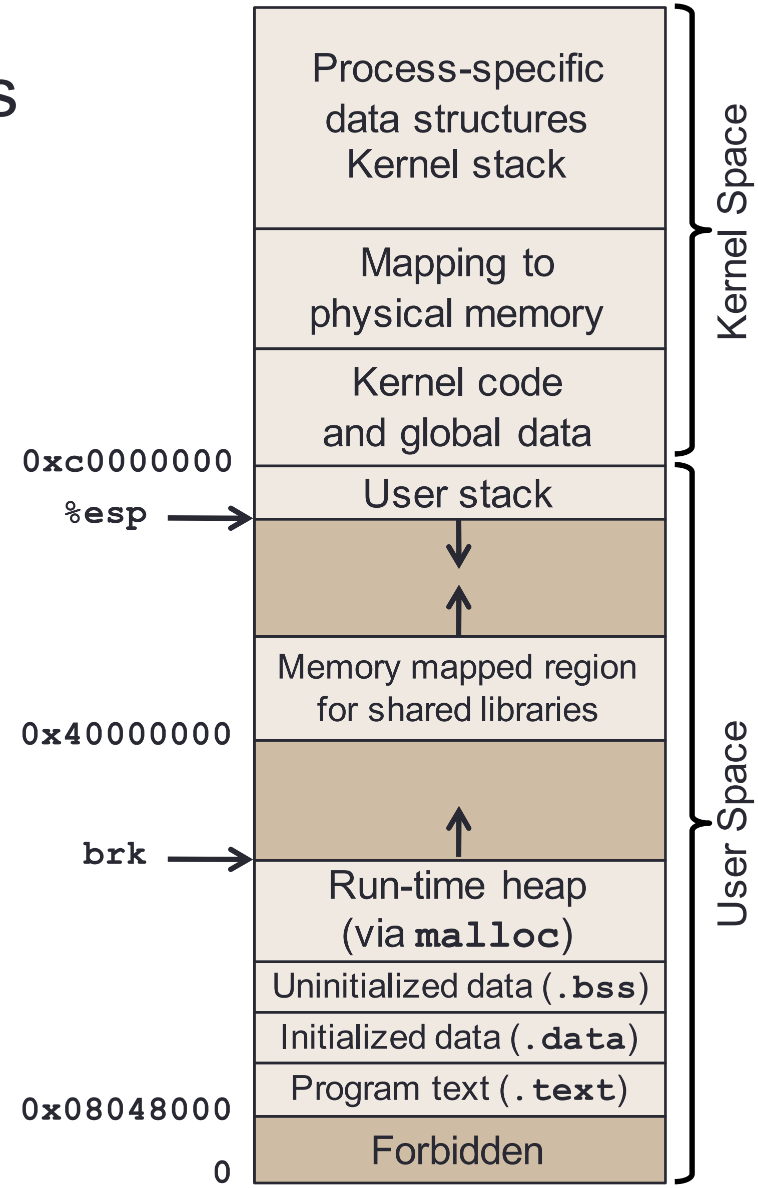
- Why would a process long-jump out of signal handler?
  - Needs to have a jump-buffer that records the CPU state at some earlier point in stack
- A process that long-jumps out of a signal handler *doesn't want to* restore the old execution state!
  - The process *doesn't want to* resume executing the code that was interrupted when the signal occurred
- Long-jump will discard all intervening execution state, including CPU state saved by the kernel at signal delivery
- No need to invoke **`sysreturn`**; the long-jump will restore the desired CPU state
  - And, **`siglongjmp`** will restore the blocked-signal mask to an appropriate state



Next Time

- Kernels maintain complex structures on behalf of user processes…
- Kernel has a very limited amount of memory for dynamic allocation
  - Kernel has a fixed memory area for data that pertains to all processes
  - And allocation needs to be fast

- Next time: kernel allocators



Next Time

- Begin discussing virtual memory management