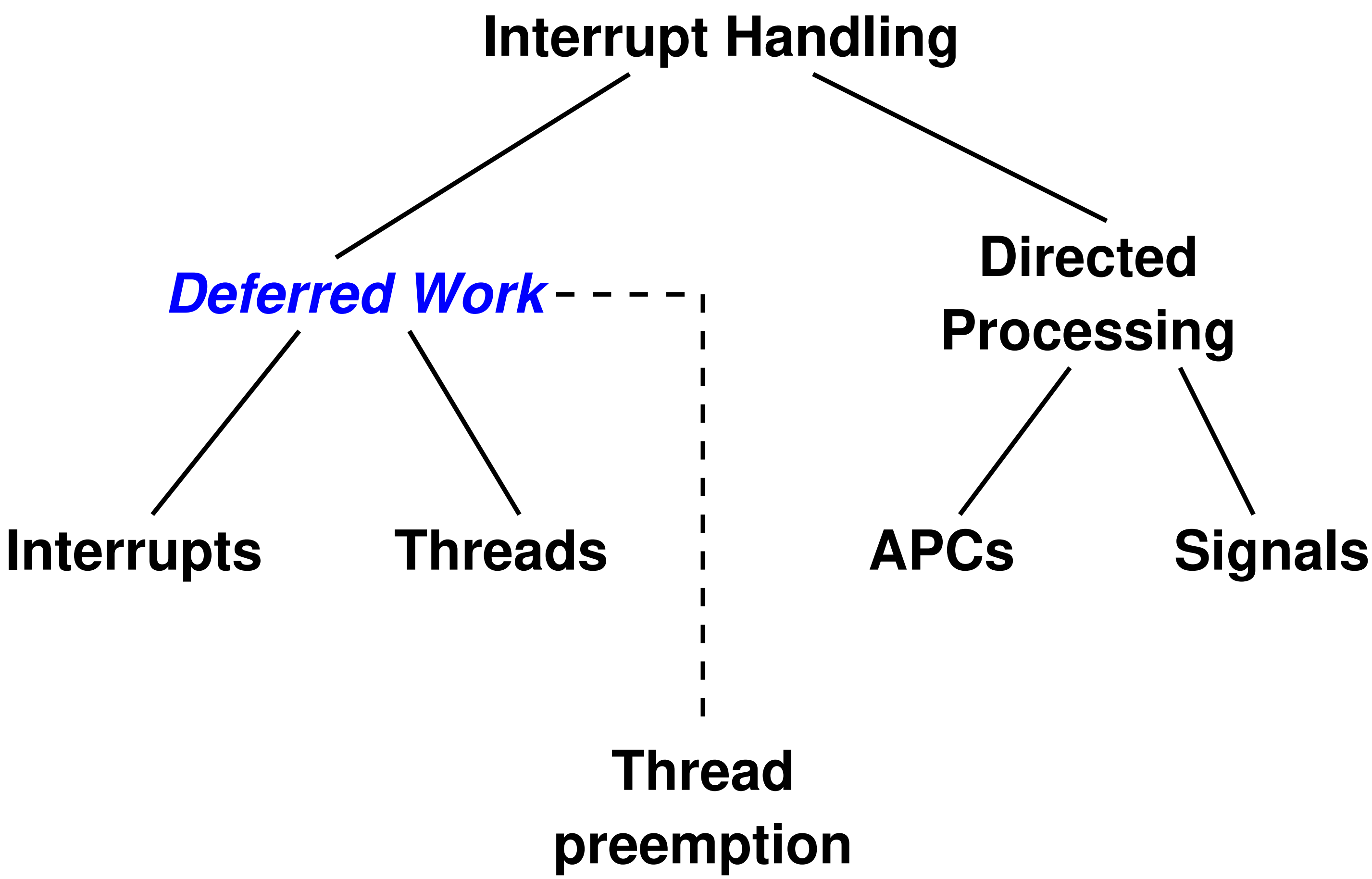
Interrupt Handling - Overview



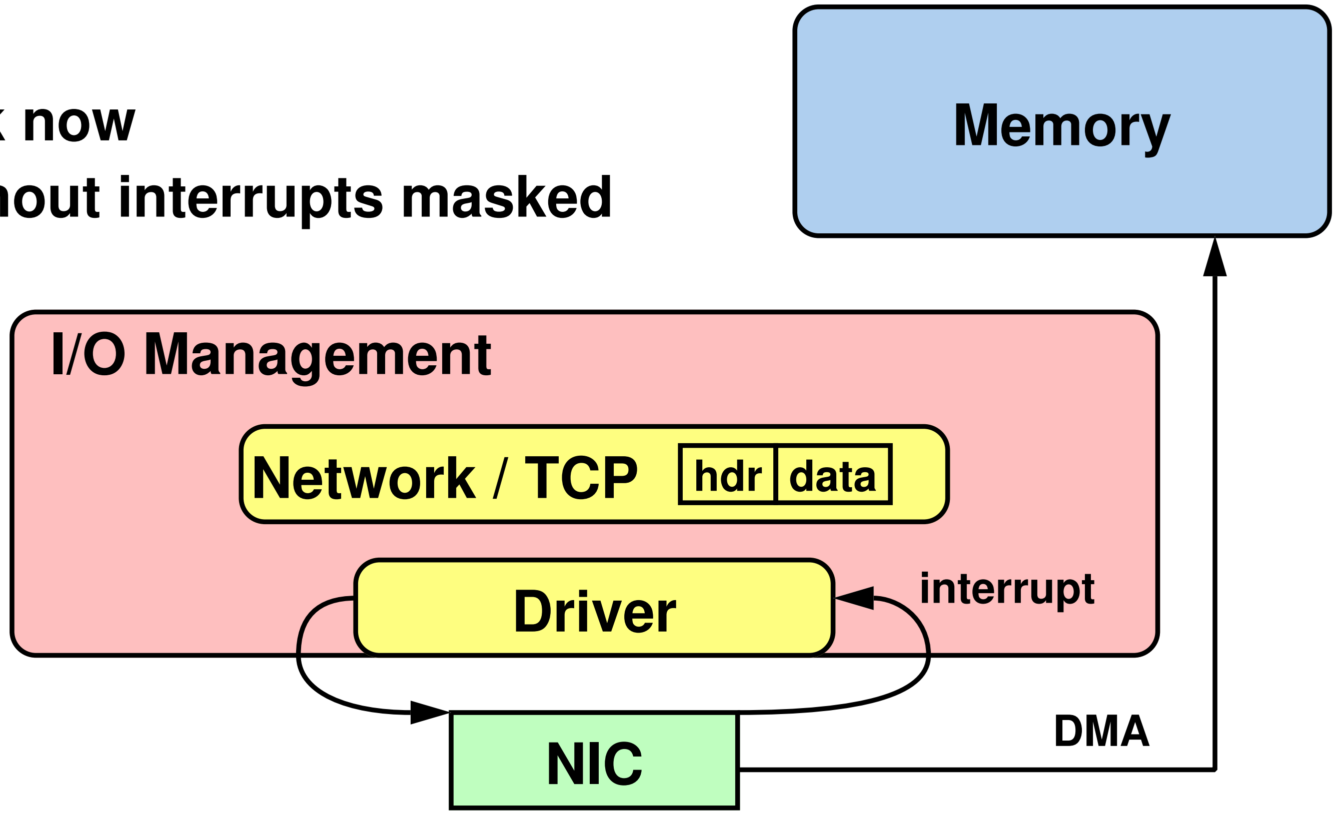
Deferred Work

- Interrupt handlers run with interrupts masked (up to its interrupt priority level)
  - both when executed in interrupt context or thread context
  - may interfere with handling of other interrupts
  - they must run to completion (but may be interrupted by a higher priority interrupt)
    - it must *complete quickly*

- What to do if an interrupt handler has a lot of work to be done?
  - only do what you must do inside the interrupt handler
  - *defer* most of the work to be done after the interrupt handler returns

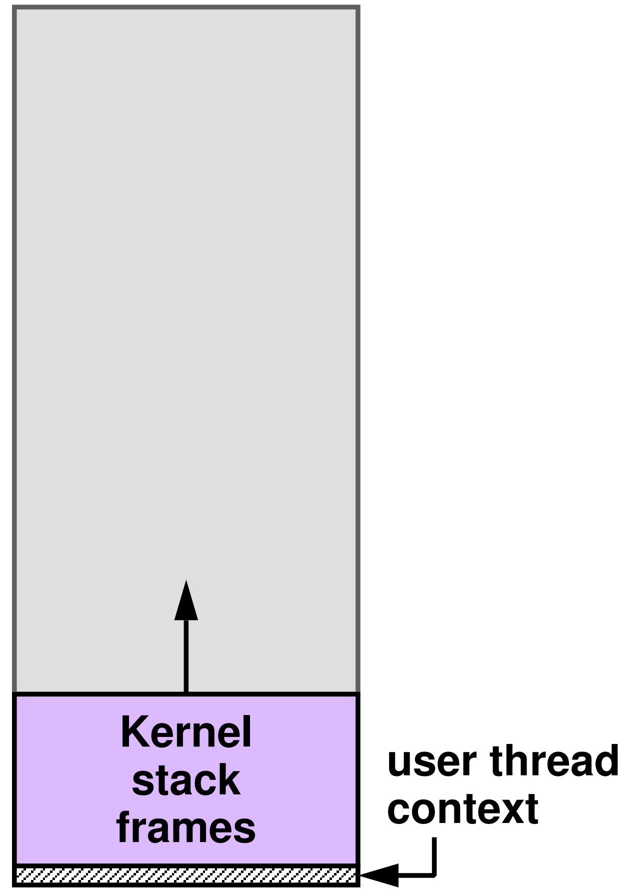
Deferred Work

- Ex: network packet processing
  - TCP header processing can take a long time
    - not suitable to do them in a interrupt handler
- Solution
  - do minimal work now
  - do rest later without interrupts masked
- How?



Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}
```



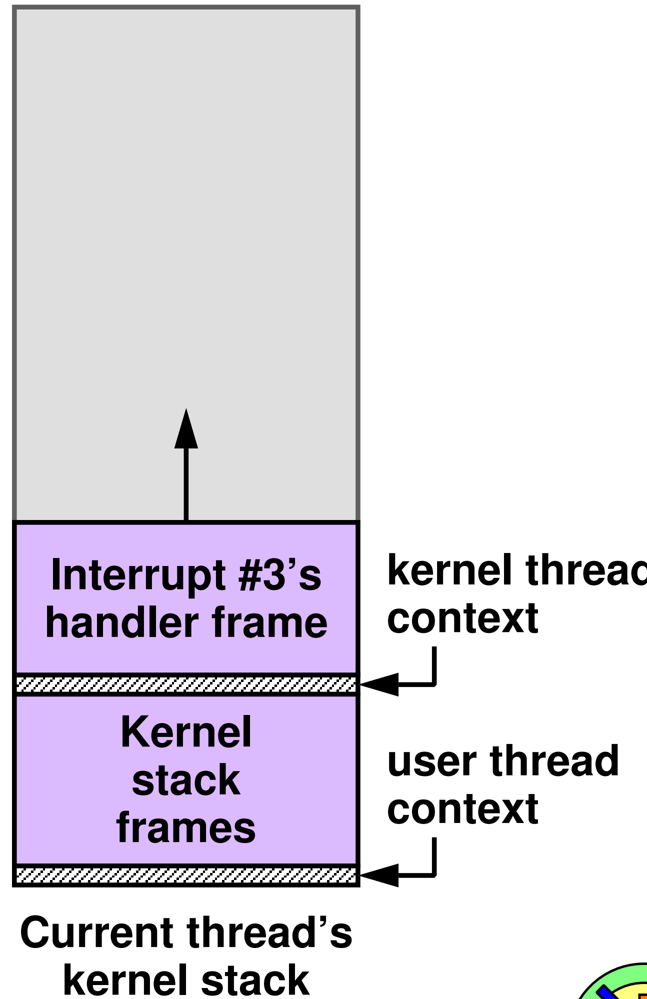
Current thread's kernel stack

- Ex: interrupting a kernel thread
  - stack allows us to return to previous context

Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void KeyboardInterruptHandler() {
  // deal with interrupt
  // do minimal work
  ...
  EnQueue(WorkQueue, MoreWork);
}
```

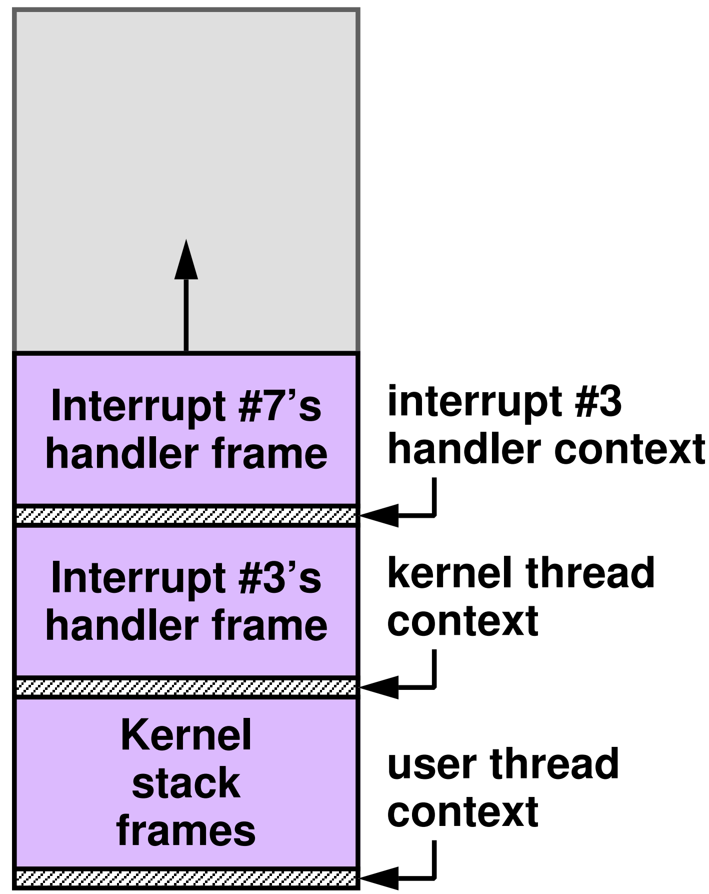


- Ex: interrupting a kernel thread
  - stack allows us to return to previous context

Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void DiskInterruptHandler() {
  // deal with interrupt
  // do minimal work
  ...
  EnQueue(WorkQueue, MoreWork);
}
```



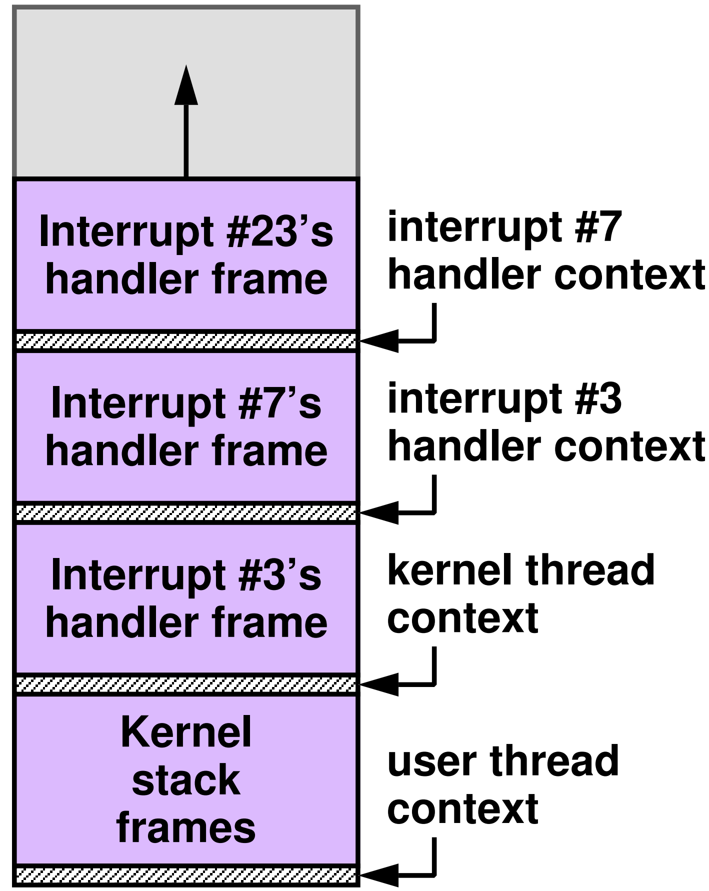
Current thread's kernel stack

- Ex: interrupting a kernel thread
  - stack allows us to return to previous context

Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void NetworkInterruptHandler() {
  // deal with interrupt
  // do minimal work
  ...
  EnQueue(WorkQueue, MoreWork);
}
```



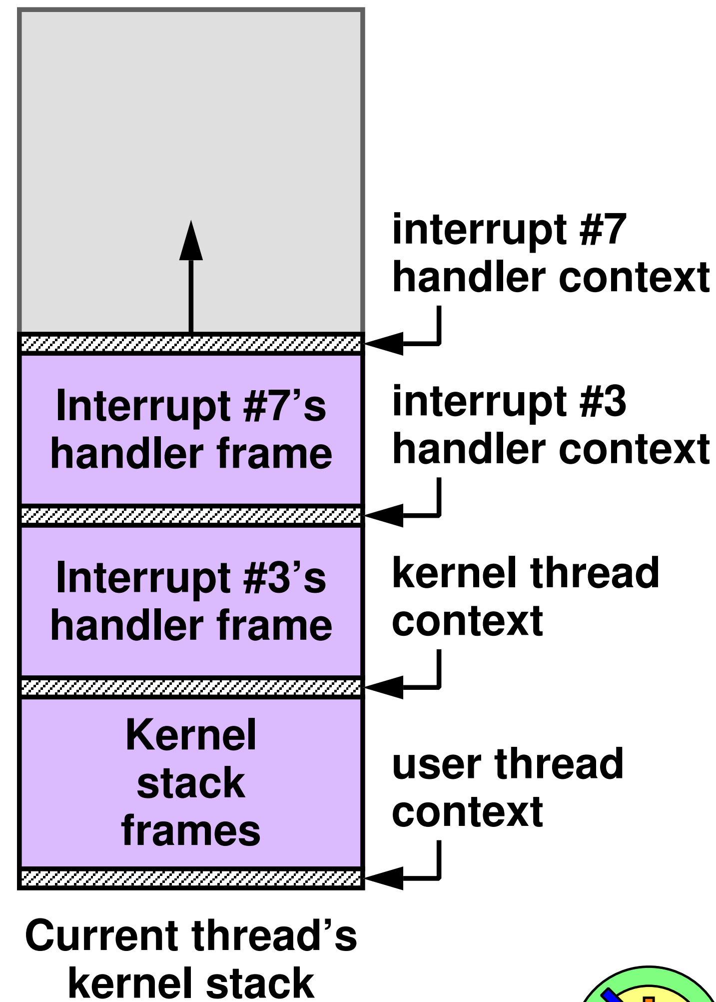
Current thread's kernel stack

- Ex: interrupting a kernel thread
  - stack allows us to return to previous context

Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void NetworkInterruptHandler() {
  // deal with interrupt
  // do minimal work
  ...
  EnQueue(WorkQueue, MoreWork);
}
```

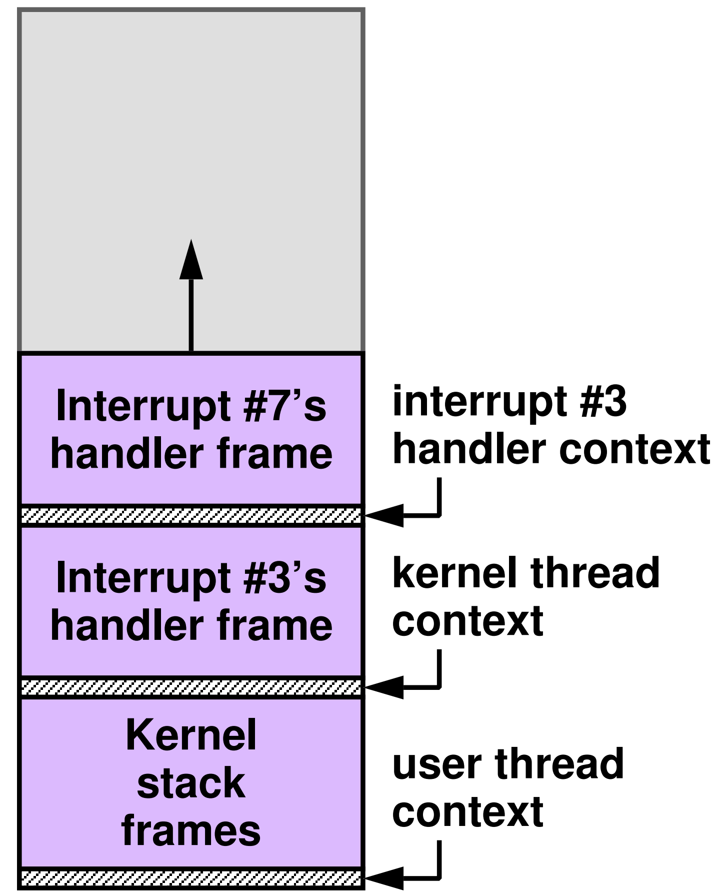


- Ex: interrupting a kernel thread
  - stack allows us to return to previous context

Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void DiskInterruptHandler() {
  // deal with interrupt
  // do minimal work
  ...
  EnQueue(WorkQueue, MoreWork);
}
```



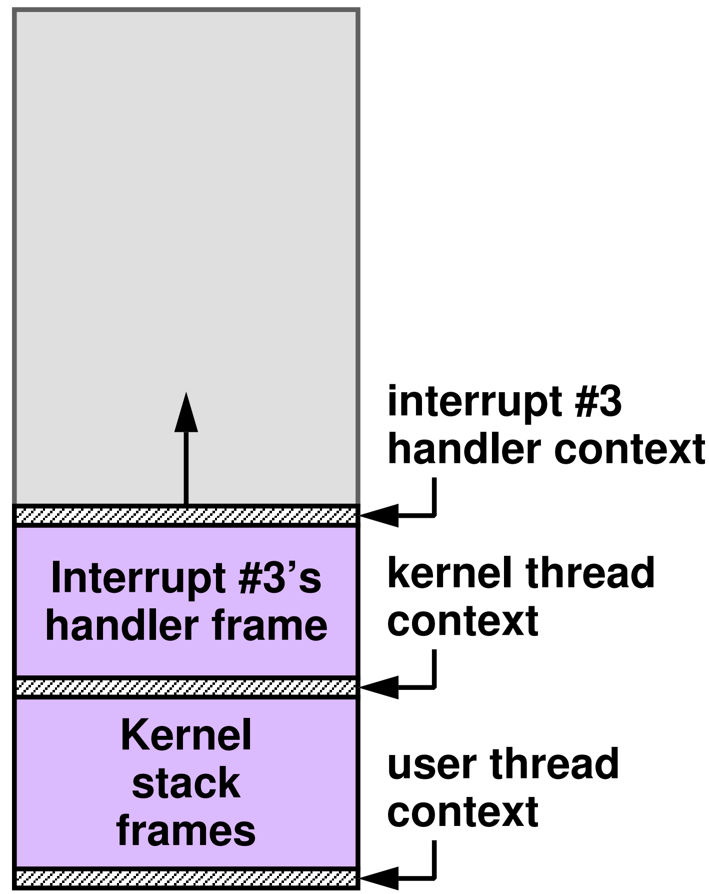
Current thread's kernel stack

- Ex: interrupting a kernel thread
  - stack allows us to return to previous context

Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void DiskInterruptHandler() {
  // deal with interrupt
  // do minimal work
  ...
  EnQueue(WorkQueue, MoreWork);
}
```

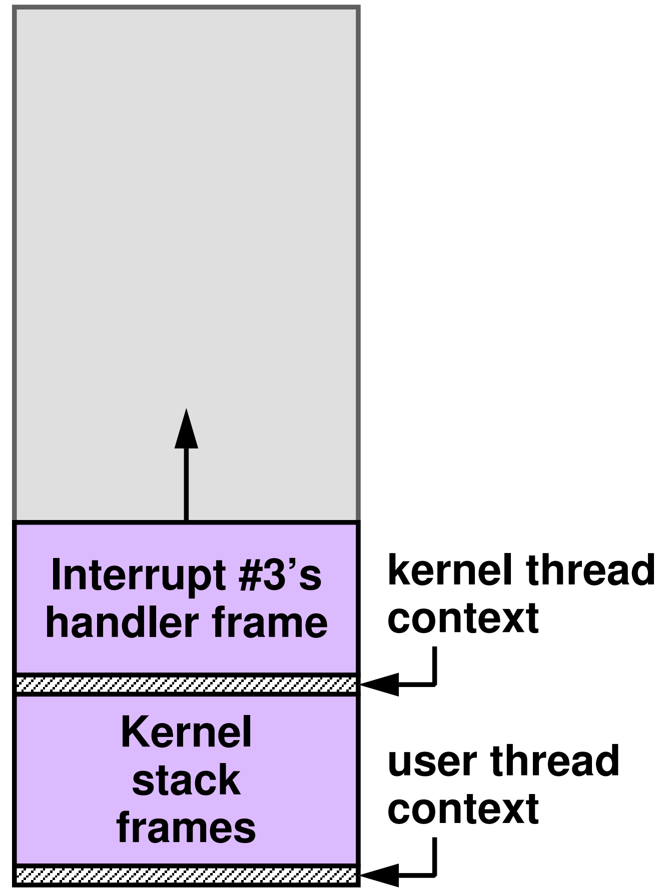


- Ex: interrupting a kernel thread
  - stack allows us to return to previous context

Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void KeyboardInterruptHandler() {
  // deal with interrupt
  // do minimal work
  ...
  EnQueue(WorkQueue, MoreWork);
}
```



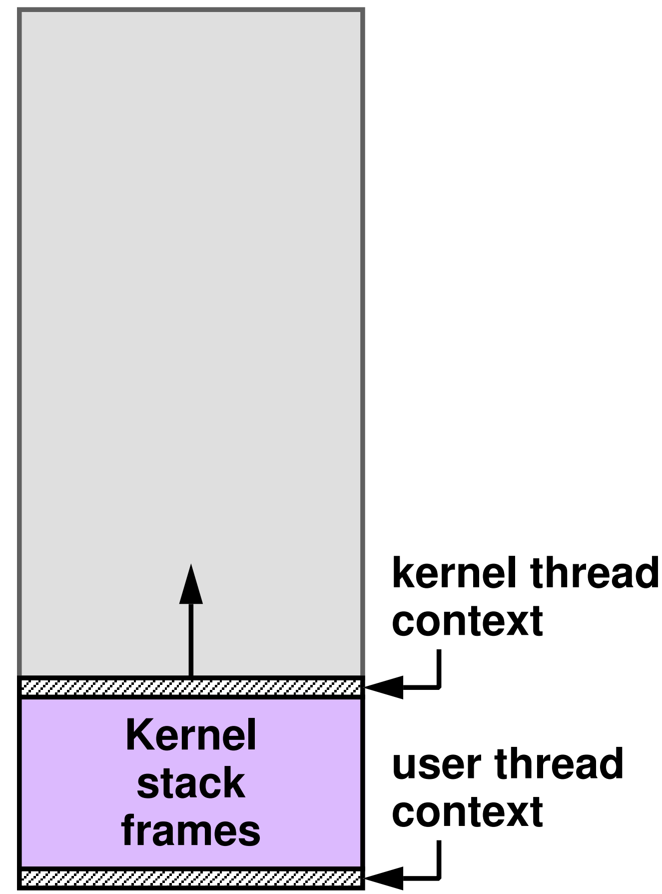
Current thread's kernel stack

- Ex: interrupting a kernel thread
  - stack allows us to return to previous context

Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void KeyboardInterruptHandler() {
  // deal with interrupt
  // do minimal work
  ...
  EnQueue(WorkQueue, MoreWork);
}
```



Current thread's kernel stack

- Ex: interrupting a kernel thread
  - stack allows us to return to previous context

Windows Interrupt Priority Levels

hardware	31	High
	30	Power fail
	29	Inter-processor
	28	Clock
	.	.
	.	.
	.	.
	4	Device 2
	3	Device 1
software	2	*DPC*
	1	APC
	0	Thread

Windows handles deferred work in a special interrupt context

- DPC (deferred procedure call) is a *software interrupt*

Deferred Procedure Calls

```
void InterruptHandler( ) {
  // deal with interrupt, IPL already ≥ 3
  ...
  QueueDPC(MoreWork);
  /* requests an asynchronous DPC interrupt */
}

void DPCHandler( ... ) {
  while(!Empty(DPCQueue)) {
    Work = DeQueue(DPCQueue);
    Work();
  }
}
```

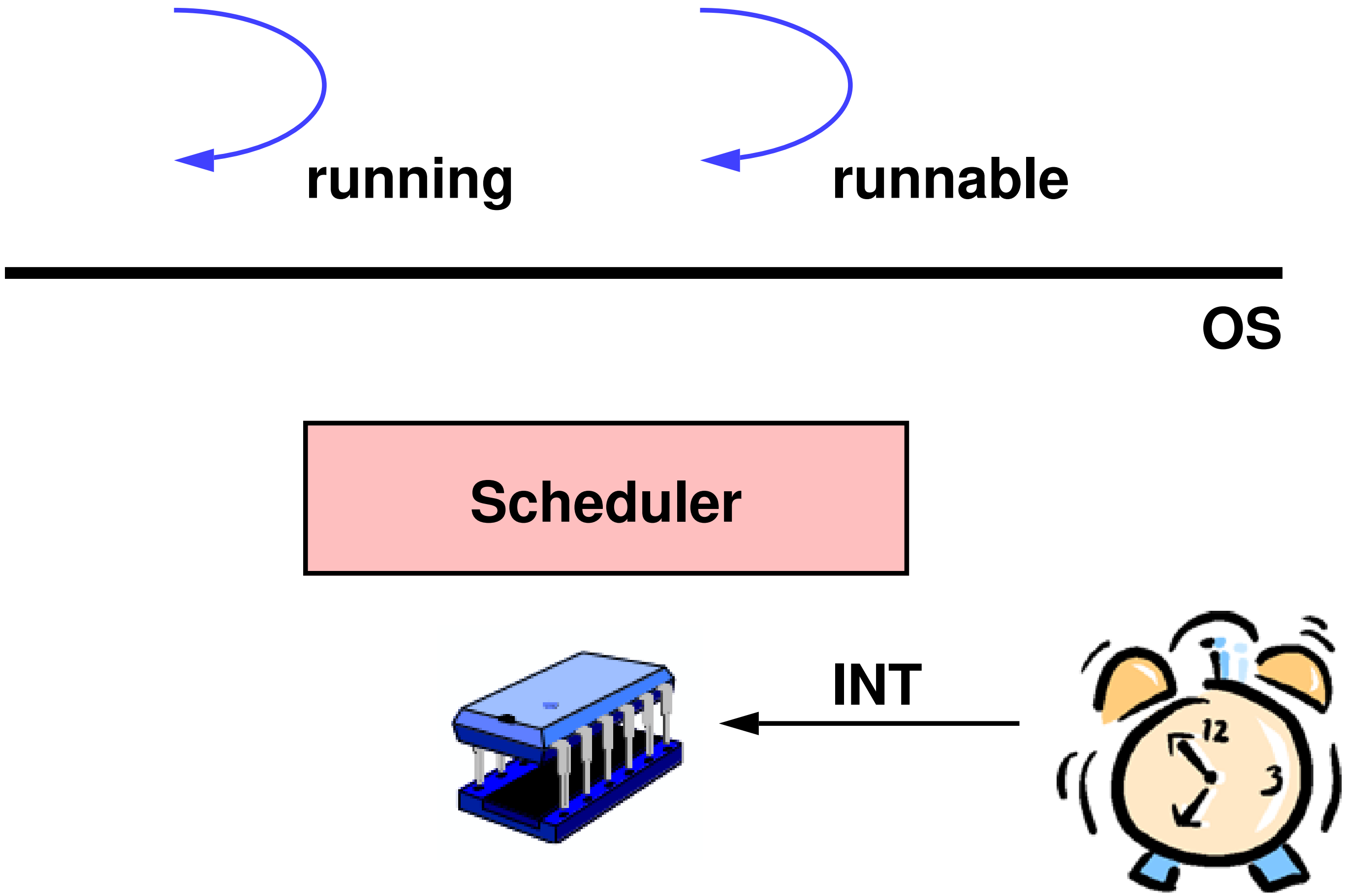
Software Interrupt Threads

- Linux handles deferred work in a special kernel thread
  - this kernel thread is scheduled like any other kernel thread

```
void InterruptHandler( ) {
  // deal with interrupt
  ...
  EnQueue(WorkQueue, MoreWork);
  SetEvent(Work);
}

void SoftwareInterruptThread() {
  while(TRUE) {
    WaitEvent(Work)
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}
```

Thread Preemption



Preemption: User-Level Only

- Non-preemptive kernel
  - preempt only threads running in user mode
  - if clock-interrupt happens, just set a global flag

```
void ClockHandler( ) {
  // deal with clock
  //     interrupt
  ...
  if (TimeSliceOver())
    ShouldReschedule = 1;
}
```

Preemption: User-Level Only

- If interrupted a user thread

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev]();
  if (PreviousMode == UserMode) {
    // the clock interrupted user-mode code
      if (ShouldReschedule)
        Reschedule();
  }
}
```

- If interrupted a kernel thread

```
void TopLevelTrapHandler(...) {
  SpecificTrapHandler();
  if (ShouldReschedule) {
    /* the time slice expired
       while the thread was
       in kernel mode */
    Reschedule();
  }
}
```

`Reschedule()` puts the calling thread on the run queue
- then call `thread_switch()` to give up the processor

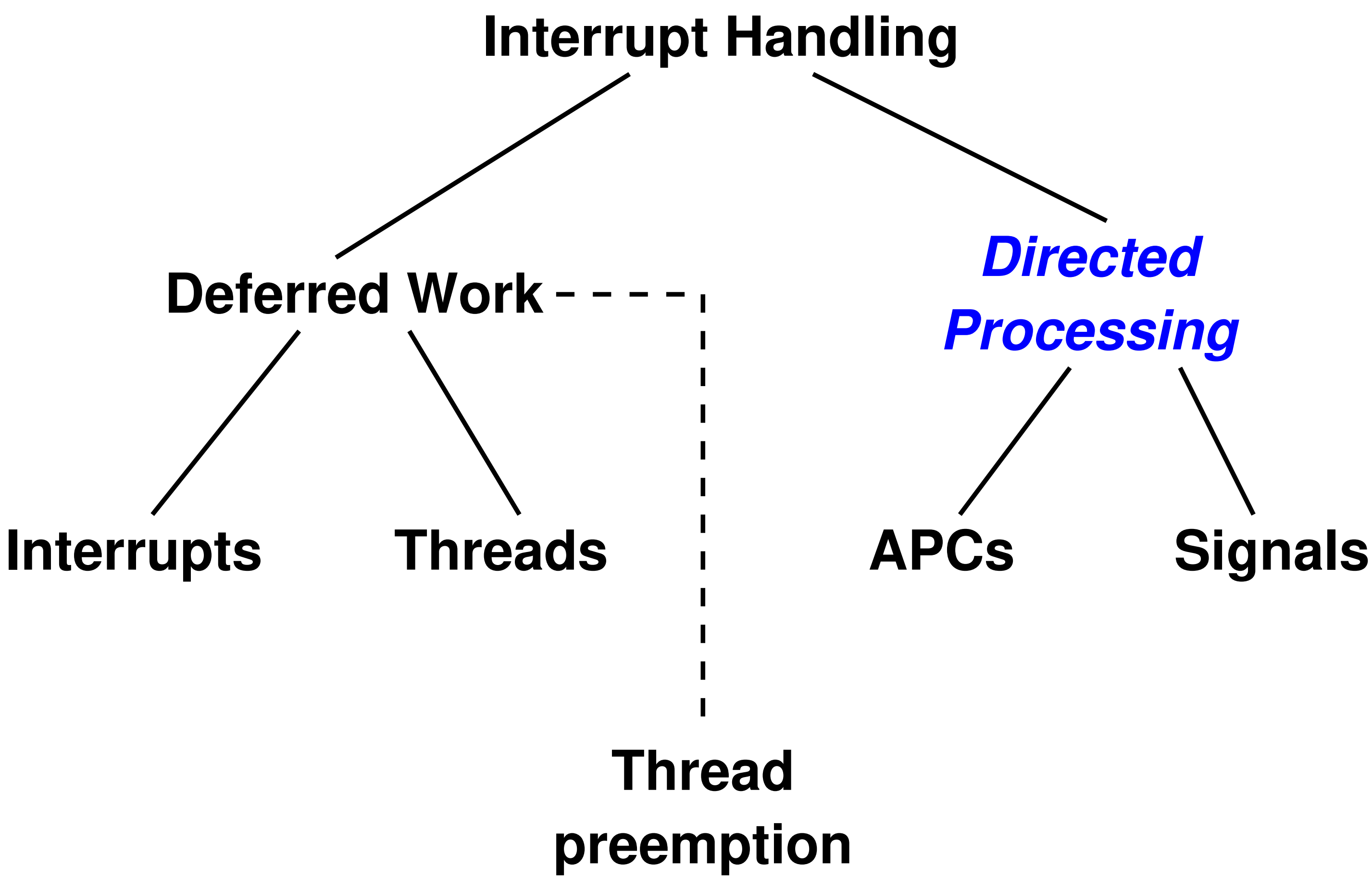
The work of *rescheduling* is *deferred*

Preemption: Full (i.e., Preemptive Kernel)

- Preemptive kernel
  - preemption can happen for a kernel thread
  - if clock-interrupt happens, setup the kernel thread to give up the processor when the processor is about to return to the thread's context
  - how?
    - e.g., in Windows, add the `Reschedule()` function to the DPC queue and invoke a DPC interrupt

```
void ClockInterruptHandler( ) {
  // deal with clock interrupt
  if (TimeSliceOver()) {
    QueueDPC(Reschedule);
    /* requests an asynchronous DPC interrupt */
  }
}
```

Interrupt Handling - Overview



Directed Processing

- Signals: Unix
  - perform given action in context of a particular thread in user mode
  - e.g., <Ctrl+C>
    - generated by hardware and needs to be delievered to the user process to invoke a singal handler

- APC (Asynchronous Procedure Calls): Windows
  - roughly same thing, but also may be done in *privileged mode*
    - provides a *general callback mechanism* for the *kernel* and for *user processes*
      - thus, APC is *more general* than Unix signals

software

2	DPC
1	*APC*
0	Thread

Windows Interrupt Priority Levels

Invoking the Signal Handler

- Basic idea is to set up the user stack so that the handler is called as a subroutine and so that when it returns, normal execution of the thread may continue

- Complications:
  - saving and restoring registers
    - must first save *all* registers and later on restore all of them
  - signal mask
    - must block the signal and later on unblock the signal
  - therefore, when the signal handler returns, it needs to return to some code that restores all the registers, unblocks the signal, then return to the interrupted code

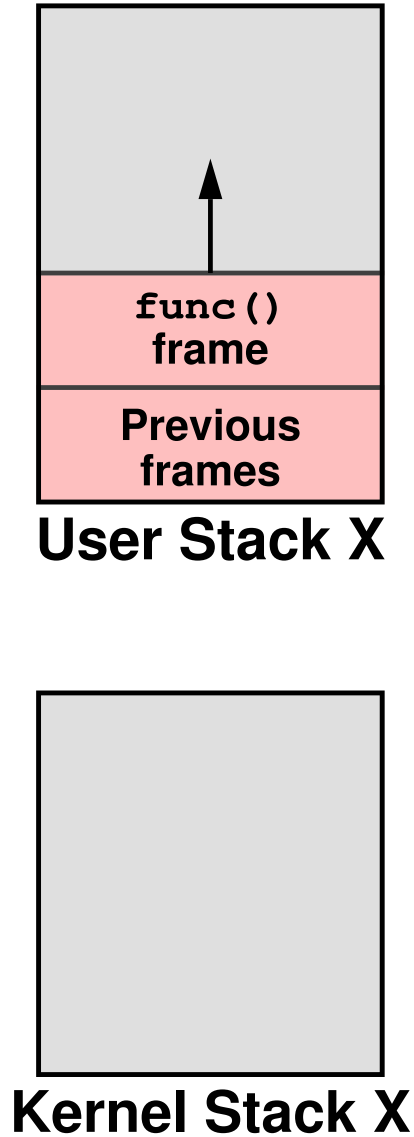
Invoking the Signal Handler (1)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;     <-- IP
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```



- "borrow a thread" means borrow ***both*** of its stacks!

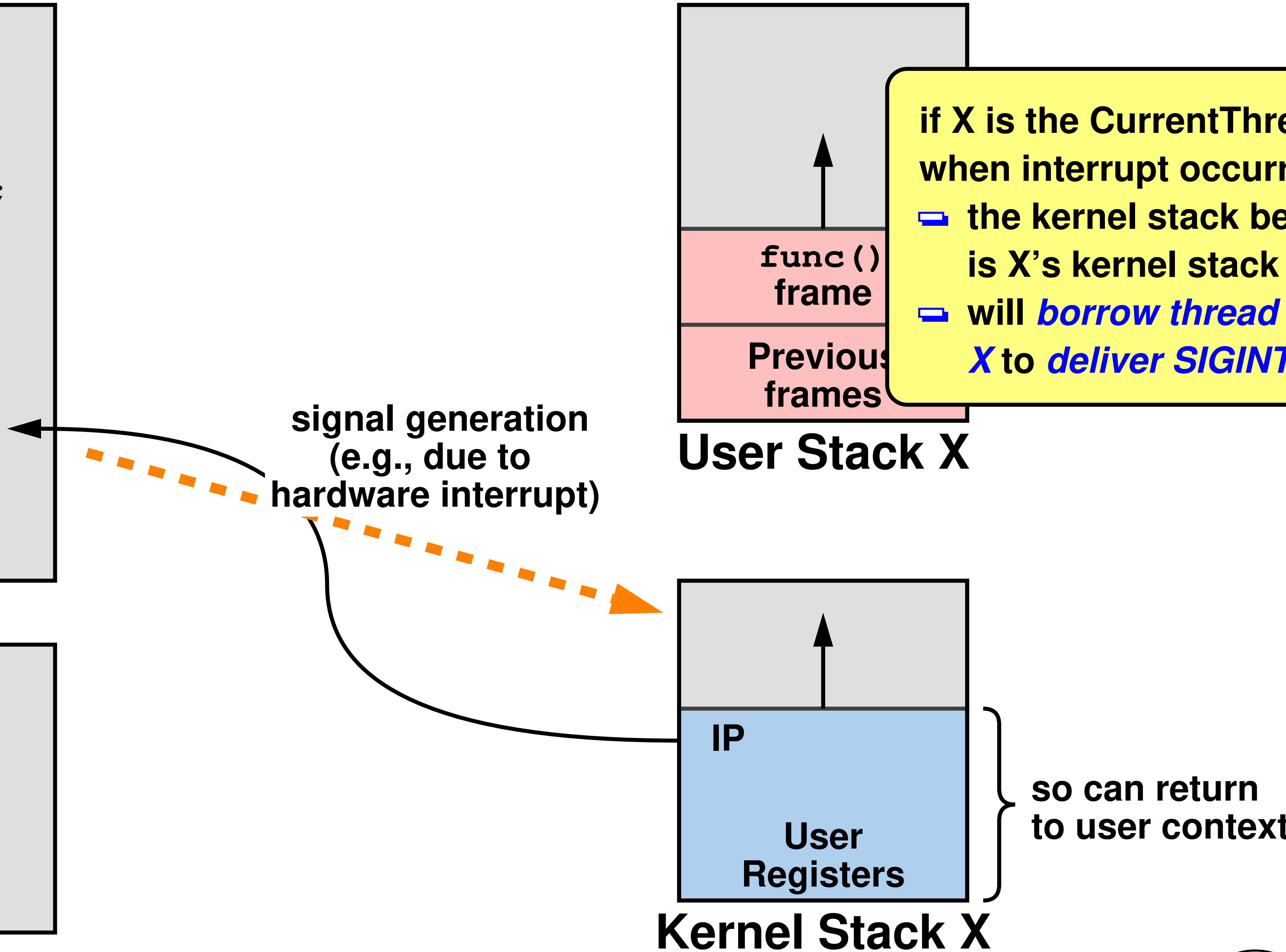
Invoking the Signal Handler (2)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```



if X is the CurrentThread when interrupt occurred
- the kernel stack below is X's kernel stack
- will *borrow thread X* to *deliver SIGINT*

- find thread X's kernel stack

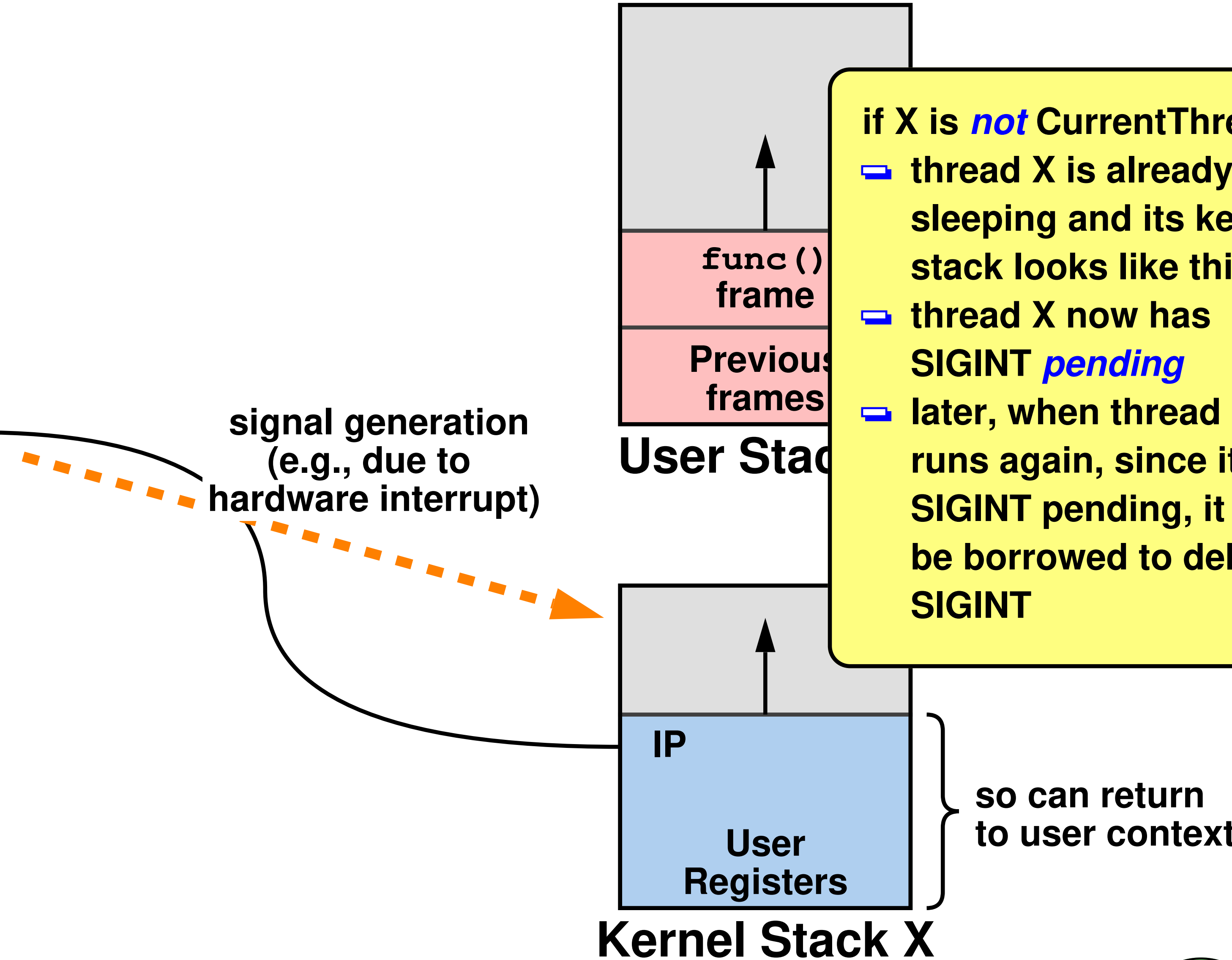
Invoking the Signal Handler (2)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```

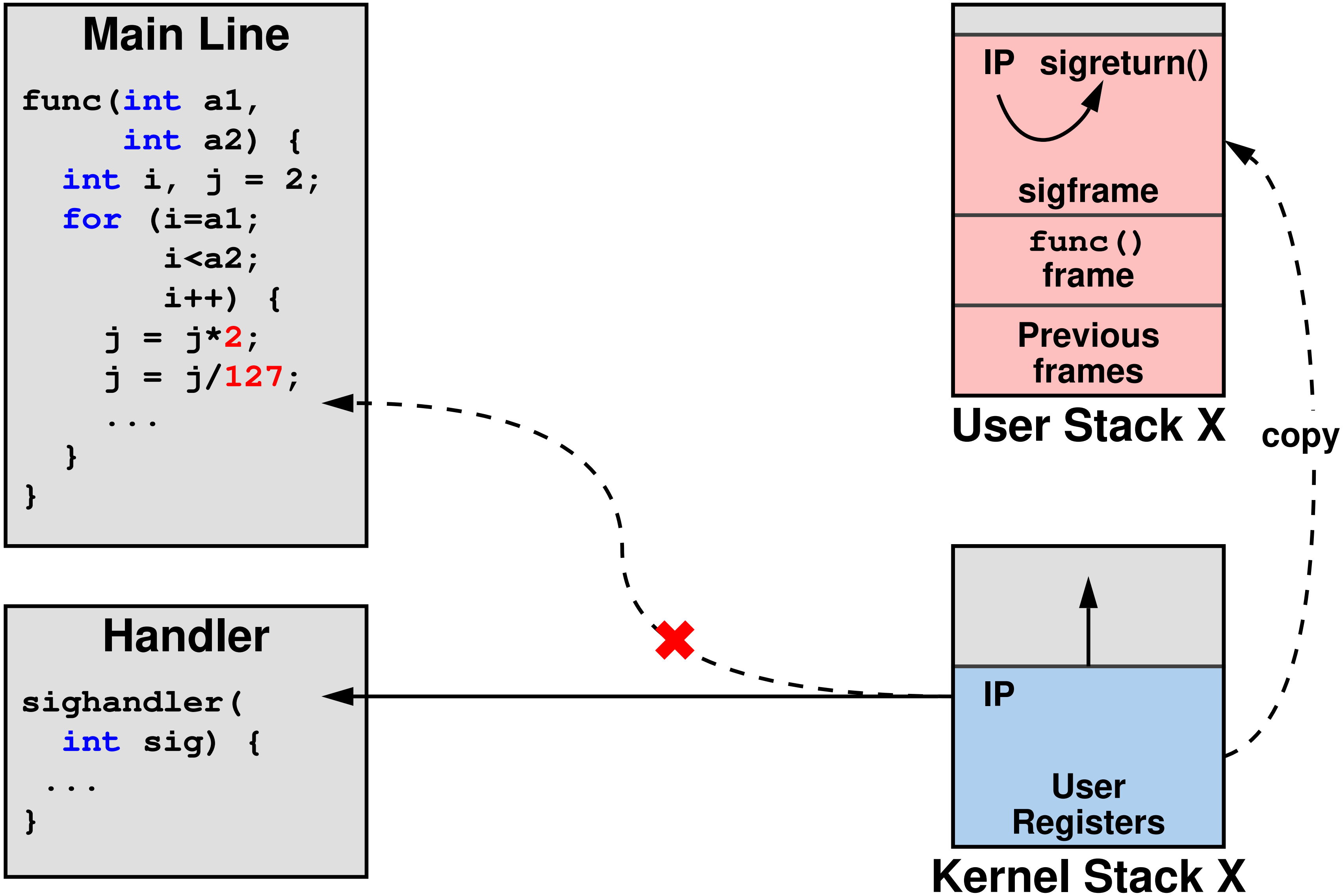


if X is *not* CurrentThread

- thread X is already sleeping and its kernel stack looks like this
- thread X now has SIGINT *pending*
- later, when thread X runs again, since it has SIGINT pending, it will be borrowed to deliver SIGINT

- find thread X's kernel stack

Invoking the Signal Handler (3)



- save X's kernel stack contents in "sigframe" in X's user stack

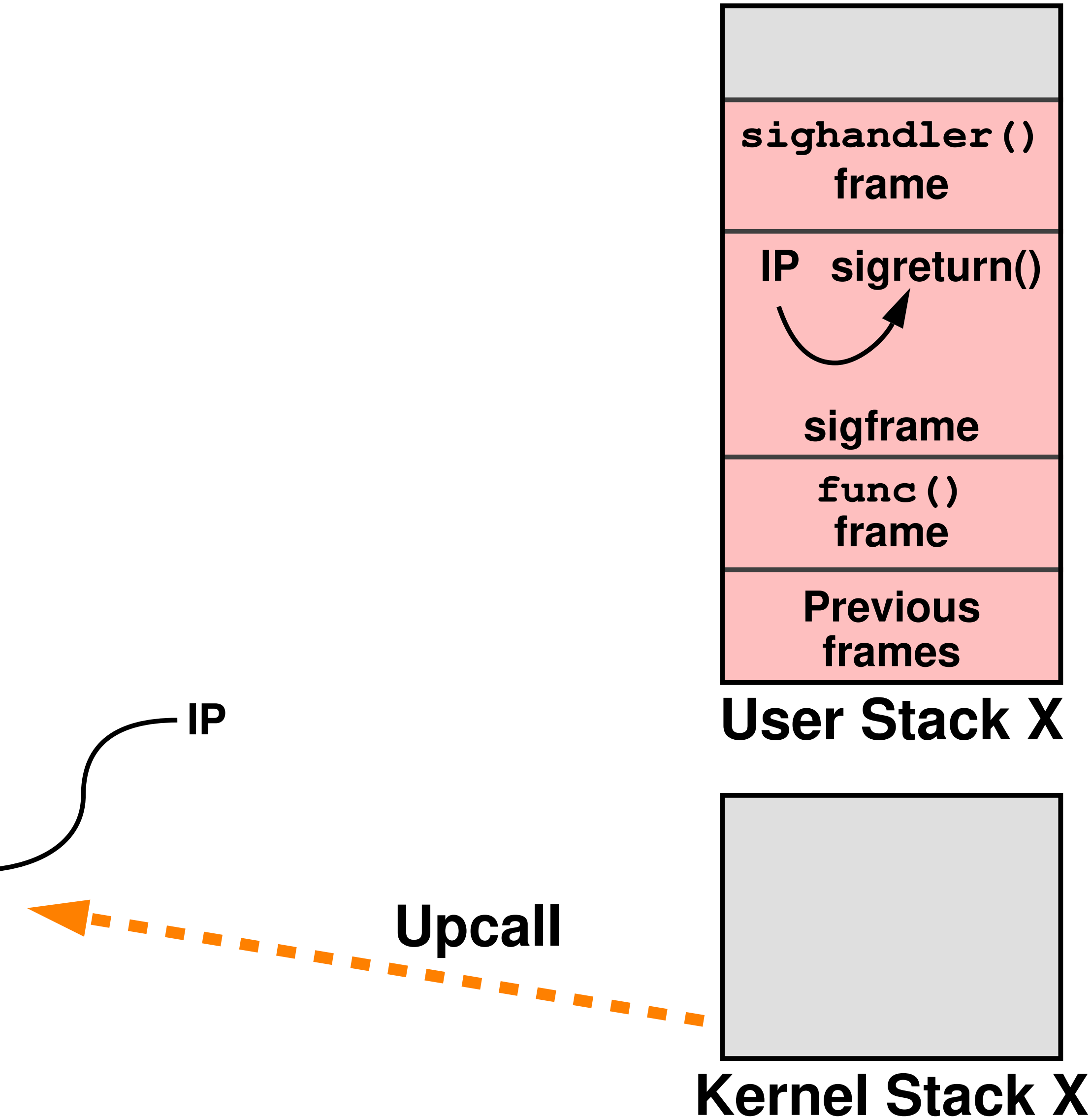
Invoking the Signal Handler (4)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```



- signal handler executed on X's user stack

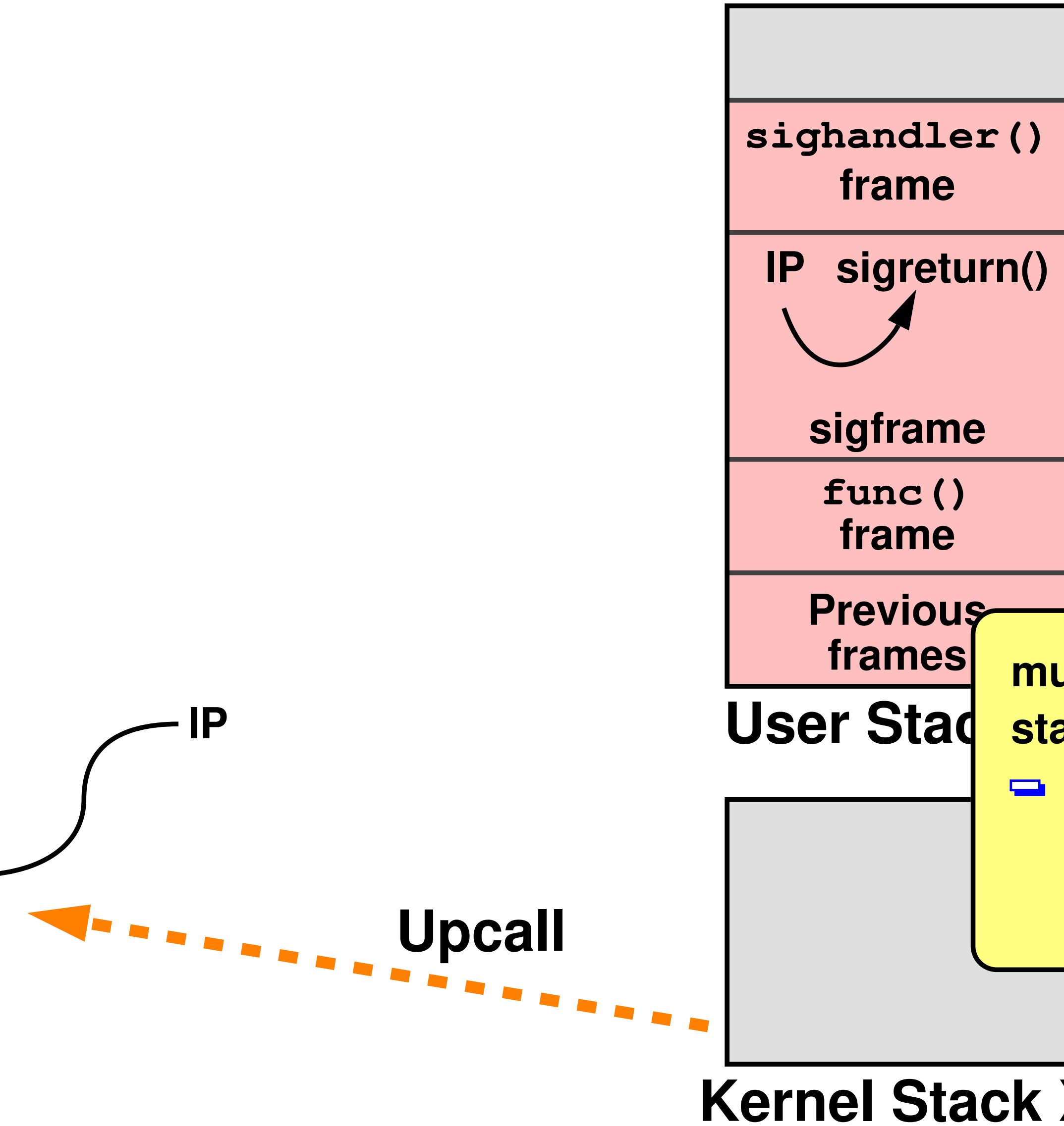
Invoking the Signal Handler (4)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
  ...
}
```



must keep the kernel stack X empty

- if `sighandler()` makes a system call, we are fine!

- signal handler executed on X's user stack when X runs again

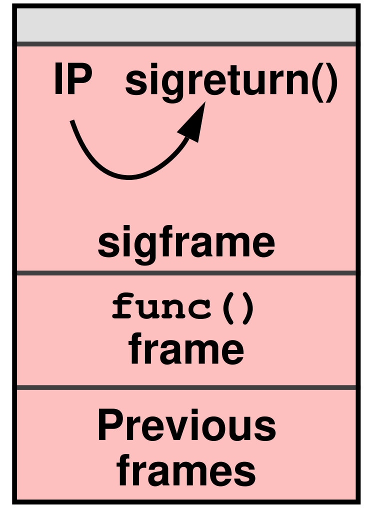
Invoking the Signal Handler (5)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
  ...
}
```



User Stack X

invoke **`sigreturn()`** system call on return from signal handler

Kernel Stack X

- kernel Stack X will be used since this is a system call by X

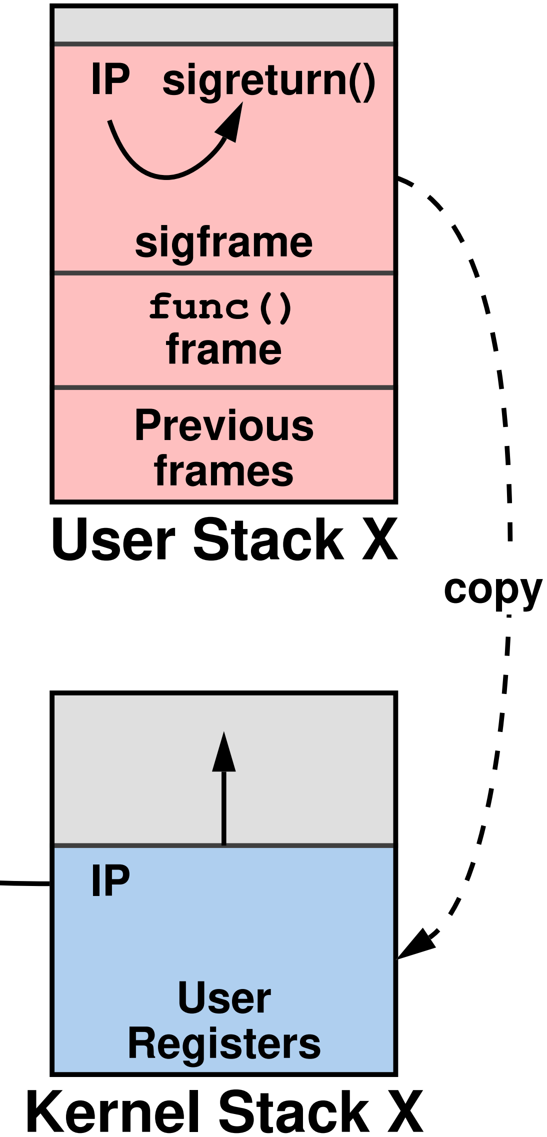
Invoking the Signal Handler (6)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```



- copy context back into kernel stack and continue

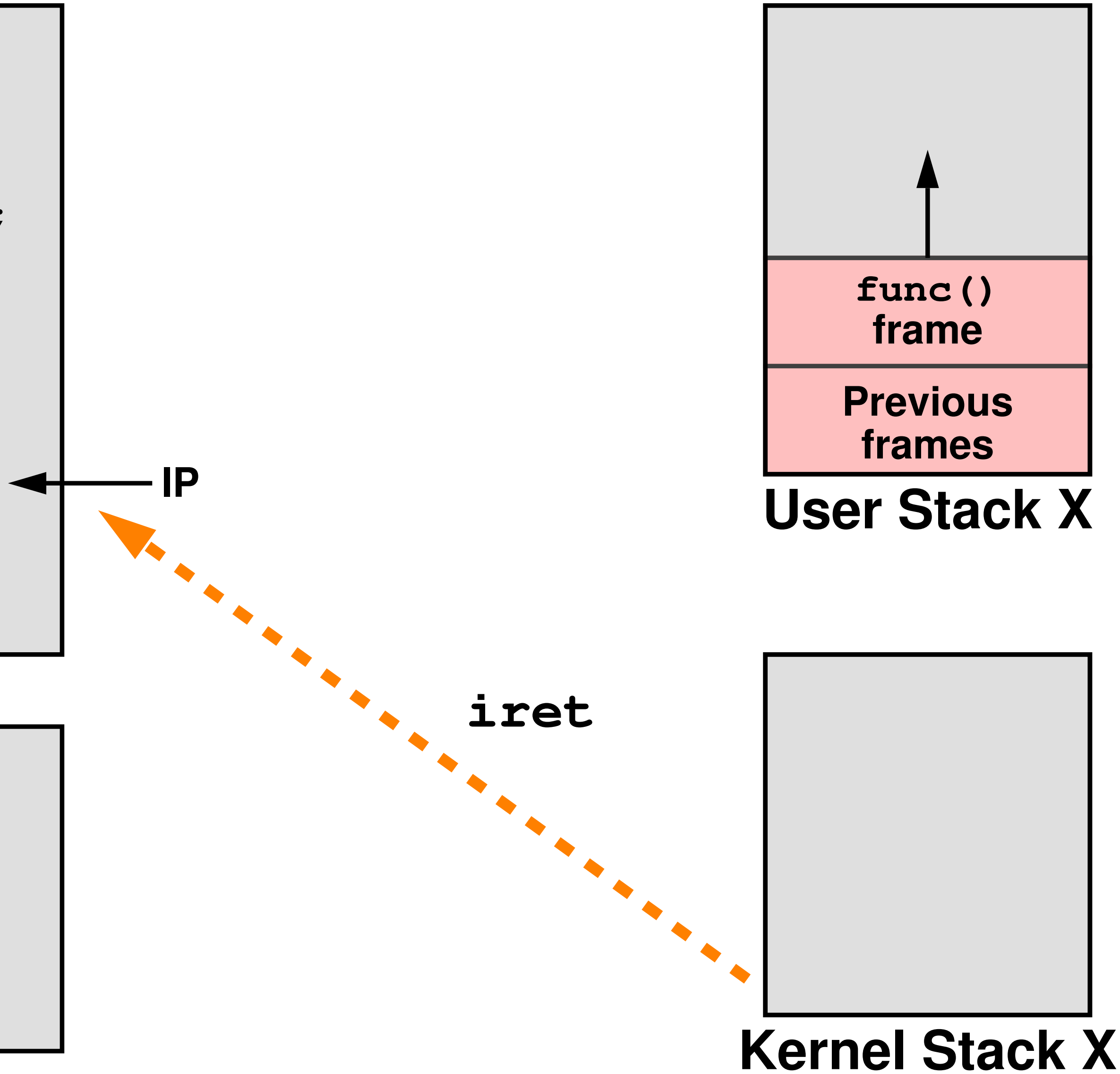
Invoking the Signal Handler (7)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```



- exactly where we were before signal delivery starts

Extra Slides

Asynchronous Procedure Calls

- Two uses
  - kernel APC: release of kernel resources
  - user APC: notifying a thread of an external event

Kernel APC

- Release of kernel resources
  - interrupt handler cannot free storage for buffer and control blocks until info passed to process
  - can’t be done unless in context of process
  - otherwise address space not mapped in
  - interrupt handler requests kernel APC to have thread, running in kernel mode, absorb info in buffer and control blocks and then free them

User APC

- Notifying thread of external event
  - example: asynchronous I/O
  - thread supplies *completion routine* when starting asynchronous I/O request
  - called in thread's context when I/O completes
    - similar to a Unix signal
    - called only when thread is in *alertable wait state*
      - an option in certain blocking system calls

APC Implementation

- Per-thread list of pending APCs
  - on notification, thread executes them
- User APC
  - thread in alertable state is woken up and executes pending APCs when it returns to user mode
- Kernel APC
  - running thread interrupted by APC interrupt (lowest priority interrupt)
  - waiting thread is "unwaited"
  - execute pending kernel APCs