Project 2: Signals

Signals

- Interprocess communication
- Several signals exist, identified by number
- Target process “handles” the signal by invoking a handler function
- Handler runs when the target process is next chosen to run—unlike interrupts, signals don’t preempt other threads

Project 2: Signals

Two parts to this project:

- Make signals work (the big part)
- Send parents a signal when attached children terminate and they haven't waited, so they know to collect status

Project 2: System Calls

Two more visible:

- `Signal`: registers handler for a given signal
- `Kill`: sends signal from one process to another

Two less visible—we will discuss soon:

- `ReturnSignal`: terminates signal handling
- `RegDeliver`: registers user function that calls ReturnSignal

One more, `WaitNoPID`, for the second part

Project 2: Signal Machinery

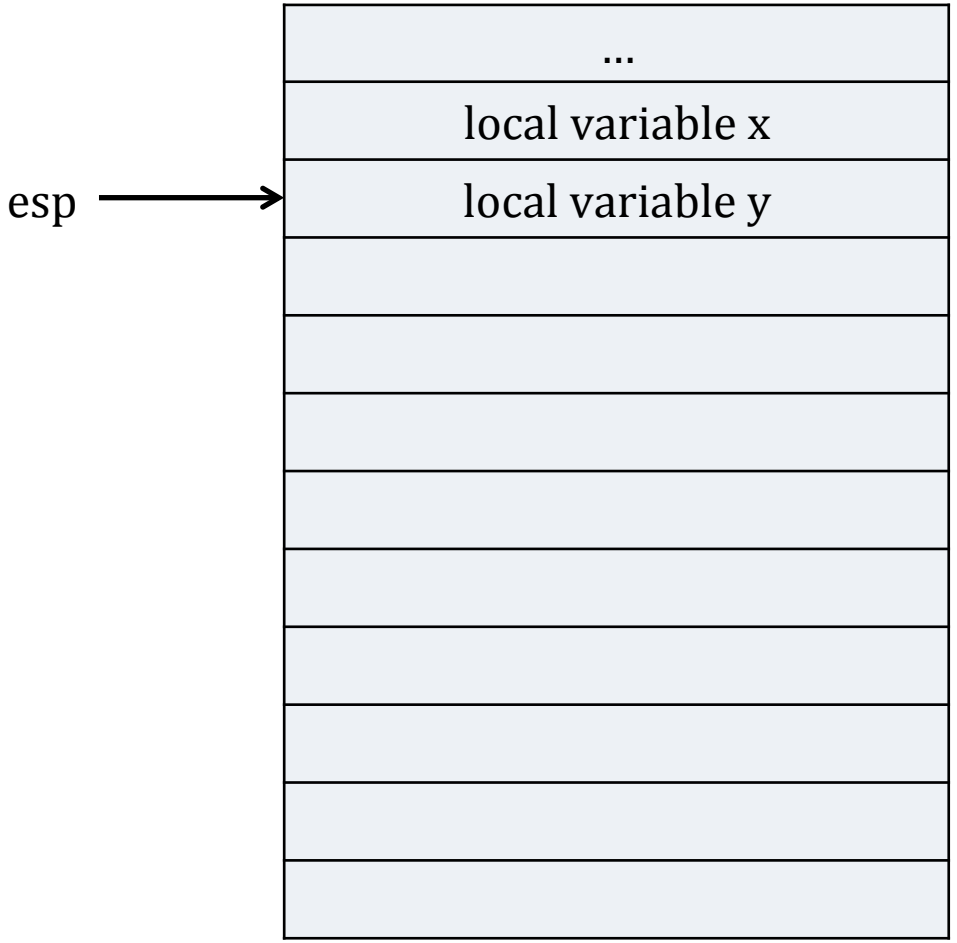
How does the kernel make a process drop what it was doing and execute its signal handler?

Need to understand:

- how context switching works
- how function calls and the stack work (C calling convention) – focus on parameters and return address

Stack and Function Calls

During function execution



Stack and Function Calls

Function call

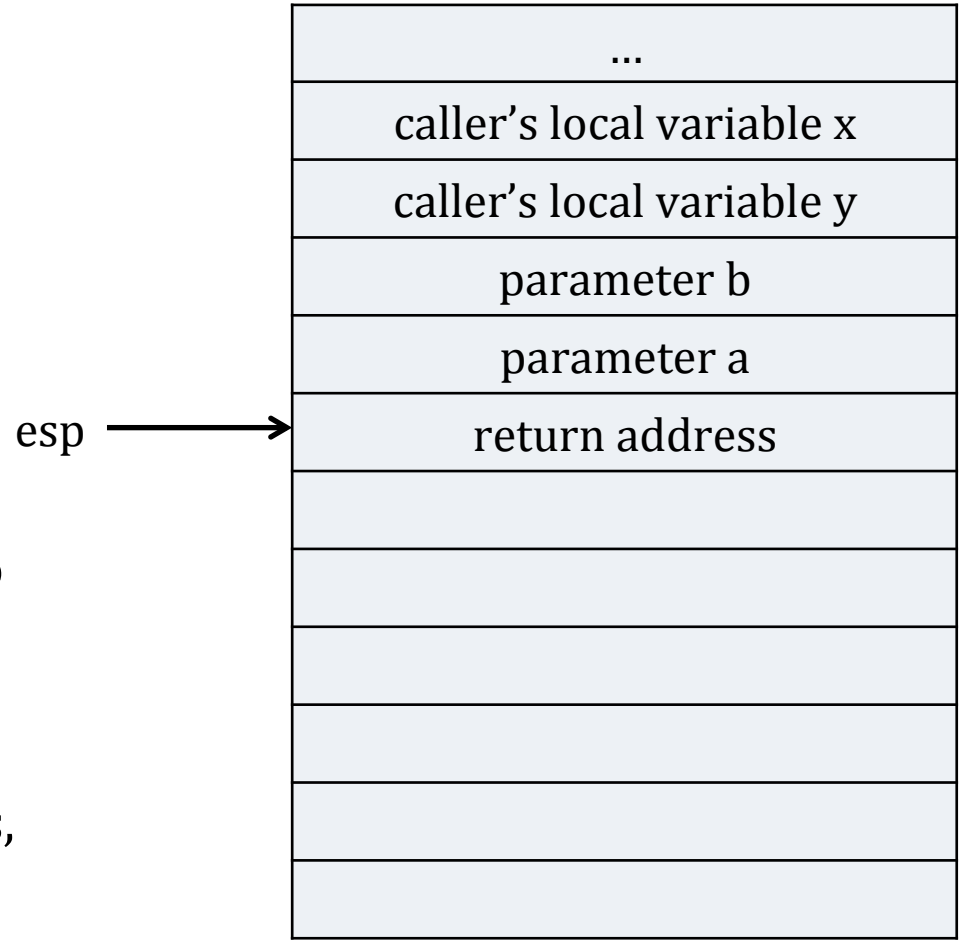
```
func(a, b);
```

or in assembly,

```
push b
push a
call func
```

- pushes return address
- sets instruction pointer to start of `func`

We will ignore some things (saving esp and ebp, saving register values, return values, etc.)



Stack and Function Calls

Return from function

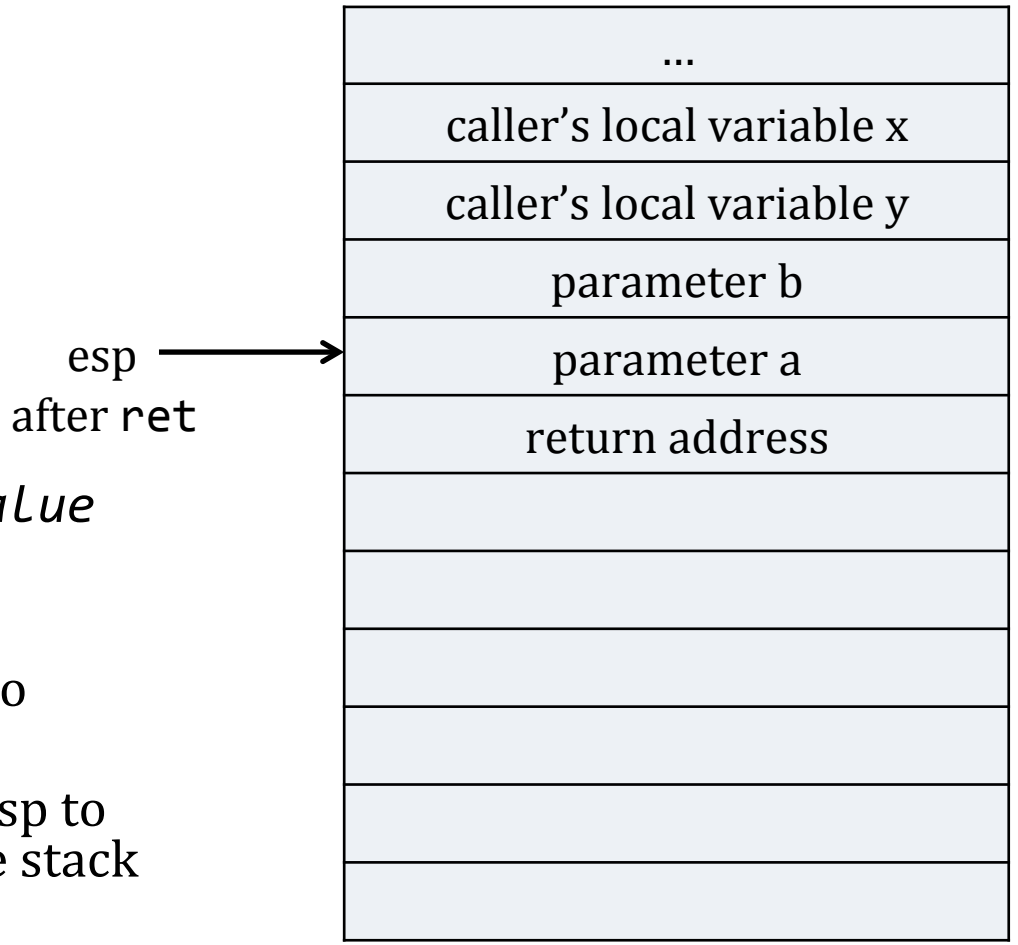
```
void func(a, b) {
…
return;
}
```

or in assembly,

```
restore esp to its value
when func was called
ret
```

- pops return address into instruction pointer

The caller then increases esp to take the parameters off the stack



Stack and Function Calls

Return from function

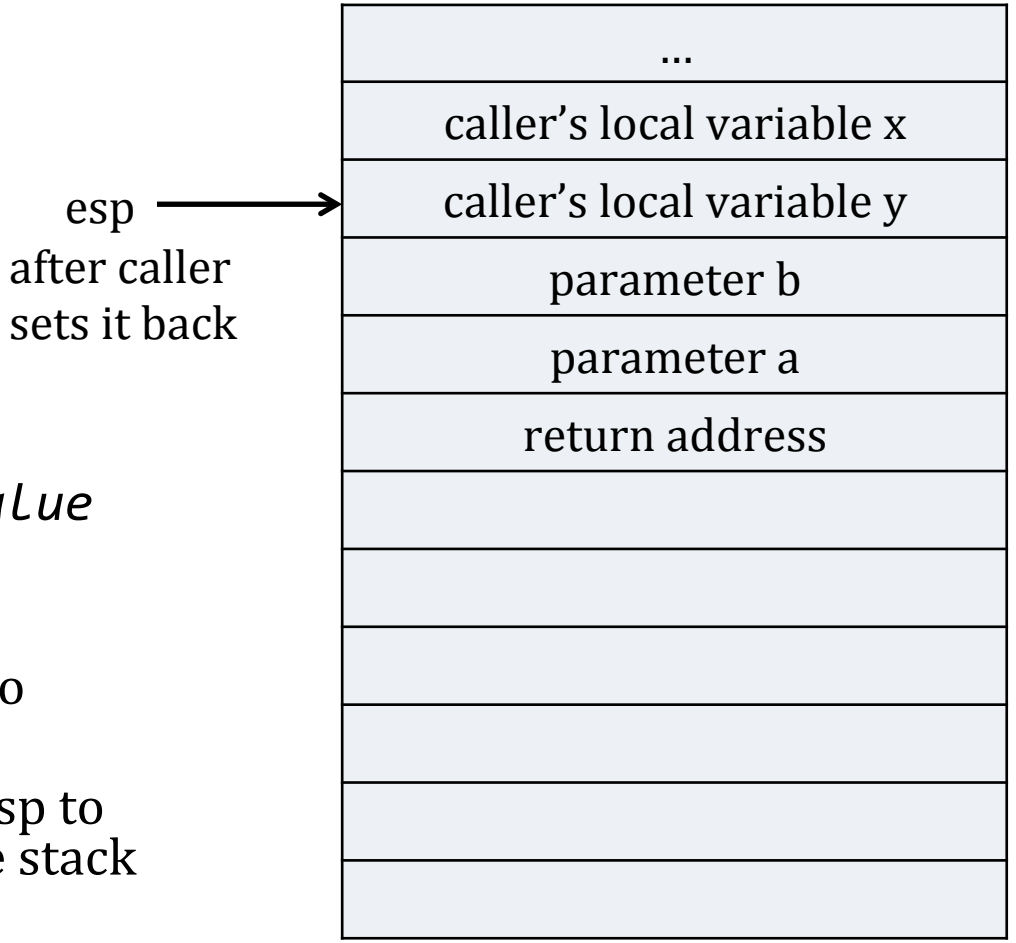
```
void func(a, b) {
…
return;
}
```

or in assembly,

```
restore esp to its value
when func was called
ret
```

- pops return address into instruction pointer

The caller then increases esp to take the parameters off the stack



Context Switch (1)

Context = register values, including memory segment selectors, instruction and stack pointers, and general purpose registers

Values are saved on *kernel stack* when a thread gets an interrupt or its time quantum expires

Kernel stack: call stack when in kernel mode; every kthread has a kernel stack, each user process has a separate user stack

Context Switch (2)

The saved context is the Interrupt_State that you've seen as the parameter to syscalls.

When the saved context is for a thread in user mode, there are two more parameters: the user stack pointer and stack segment selector.

(Note: **selectors** indicate what memory segment to use. In GeekOS, there are two: one for code, one for all data.)

Context and Signals (1)

How do we make a thread execute a system handler instead of whatever it was doing?

Make up a new context for it! (But save the old one first.)

How do we get it to go back to what it was doing?

Put back the old context!

Context and Signals (2)

How do we know when we should put back the old context?

Set the return address in our signal handler context to a special function, the *trampoline*. The trampoline issues a syscall to tell the kernel to restore the old context.

How do we know where the trampoline is?

It's registered at the beginning of every user program, using another syscall.

The Project

In terms of concepts, the hard part is getting the signal handler to execute and return. With this background, see the description of `Setup_Frame` in the spec.

You will add fields to the user context to hold signal handlers, flags for pending signals, and perhaps other things.