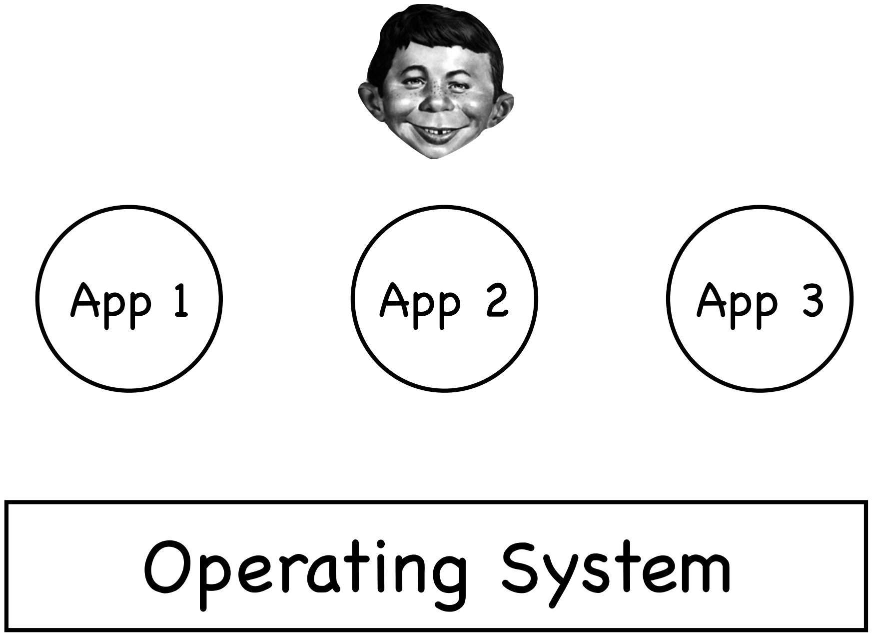
The Kernel

wants to be your friend

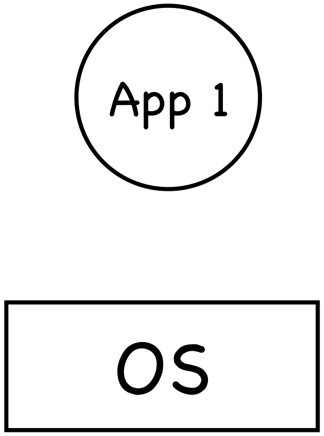
Boxing them in



Reading and writing memory, managing resources, accessing I/O...

- Buggy apps can crash other apps
- Buggy apps can crash OS
- Buggy apps can hog all resources
- Malicious apps can violate privacy of other apps
- Malicious apps can change the OS

The Process

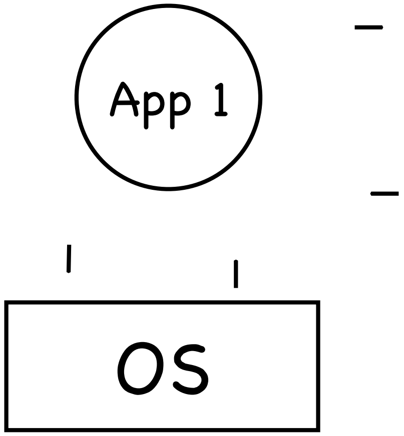


App 1

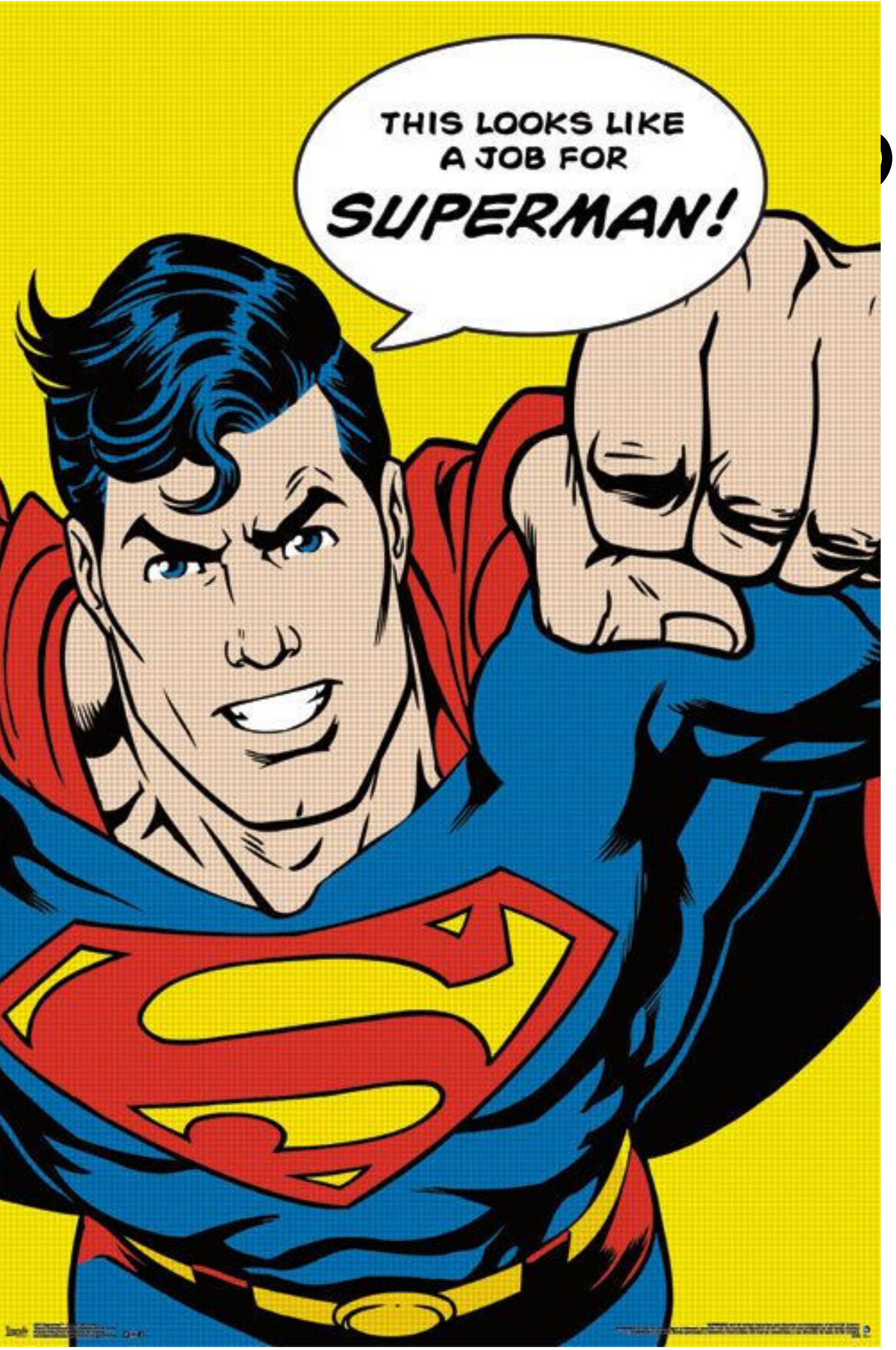
OS

- An abstraction for protection
  - the execution of an application program with restricted rights
- But there are tradeoffs (there always are tradeoffs!)
- Must not hinder functionality
  - still efficient use of hardware
  - enable safe communication

The Process



- An abstraction for protection
  - the execution of an application program with restricted rights
- But there are tradeoffs (there always are tradeoffs!)
- Must not hinder functionality
  - still efficient use of hardware
  - enable safe communication



ou are thinking...

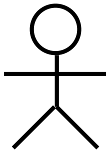
Actually...

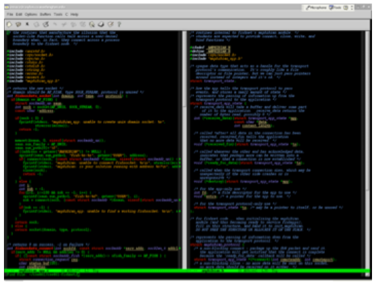
Special

- Part of the OS
    - all kernel is in the OS
    - not all the OS is in the kernel
        - (why not? robustness)
        - widgets libraries, window managers etc

Process:
Getting to know you

- A process is a program during execution
  - program is a static file
  - process = executing program = program + execution state





Source code

compiler

Header

Code

Initialized data

○

○

○

○

Executable Image

OS copy

	Code	Data	Heap	Stack		Code	Data	Heap	Stack	

Physical memory

Keeping track of a process

- A process has code
  - OS must track program counter
- A process has a stack
  - OS must track stack pointer
- OS stores state of process in Process Control Block (PCB)
  - Data (program instructions, stack & heap) resides in memory, metadata is in PCB

Process Control Block

PC
Stack Pointer
Registers
PID
UID
Priority
List of open files
Process status
...

How can the OS enforce restricted rights?

- Easy: kernel interprets each instruction!
  - slow
  - many instructions are safe: do we really need to involve the OS?

How can the OS enforce restricted rights?

- Easy: kernel interprets each instruction!
  - slow
  - many instructions are safe: do we really need to involve the OS?
- Dual Mode Operation
  - hardware to the rescue: use a mode bit
    - in user mode, processor checks every instruction
    - in kernel mode, unrestricted rights
  - hardware to the rescue (again) to make checks efficient

Efficient protection in dual mode operation

Privileged instructions

- in user mode, no way to execute potentially unsafe instructions

Memory protection

- in user mode, memory accesses outside a process' memory region are prohibited

Timer interrupts

- kernel must be able to periodically regain control from running process

⊕ Efficient mechanism for switching modes

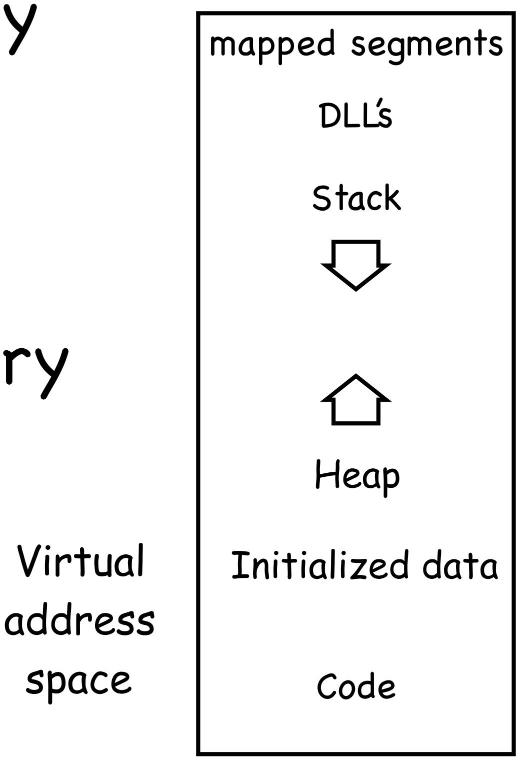
I. Privileged instructions

- Examples: Set mode bit; set accessible memory; disable interrupts; etc
- But how can an app do I/O then?
  - system calls achieve access to kernel mode only at specific locations specified by OS
- Executing a privileged instruction while in user mode causes a processor exception....
  - ...which passes control to the kernel

II. Memory Protection

Step 1: Virtualize Memory

- Virtual address space: set of memory addresses that process can “touch”
  - CPU works with virtual addresses
- Physical address space: set of memory addresses supported by hardware



Virtual address space

mapped segments

DLL’s

Stack

Heap

Initialized data

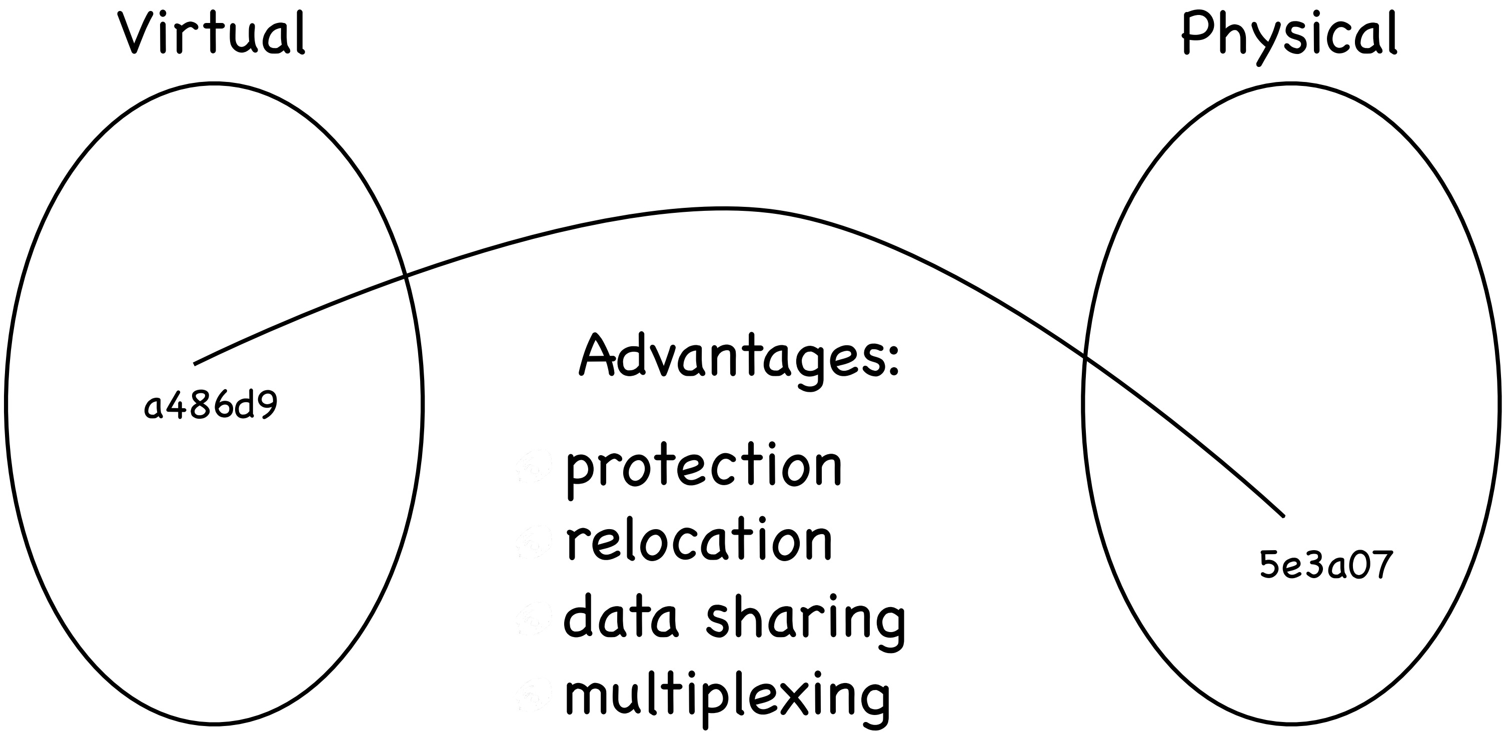
Code

II. Memory Protection

Step 2: Address Translation

- Implement a function mapping

into

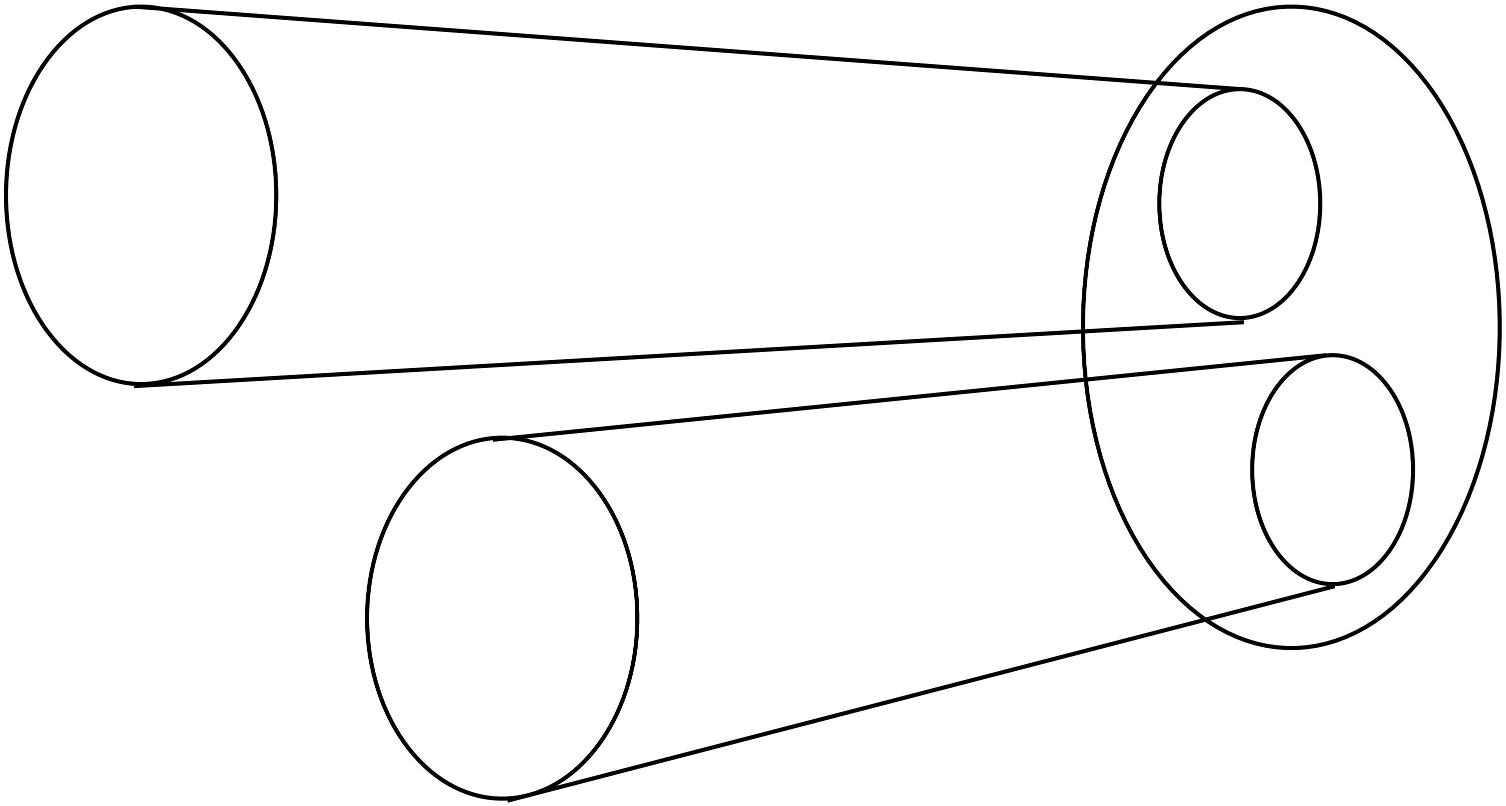


Advantages:

- protection
- relocation
- data sharing
- multiplexing

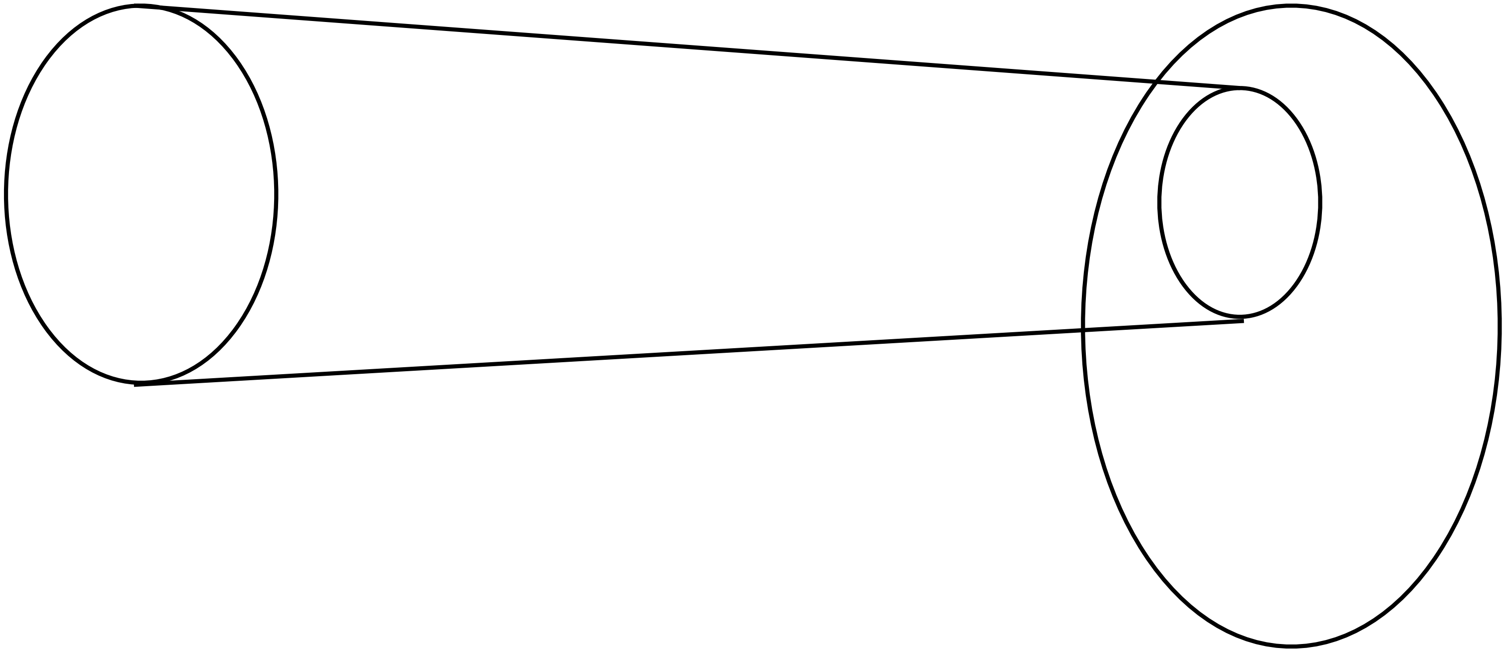
Protection

- At all times, the functions used by different processes map to disjoint ranges



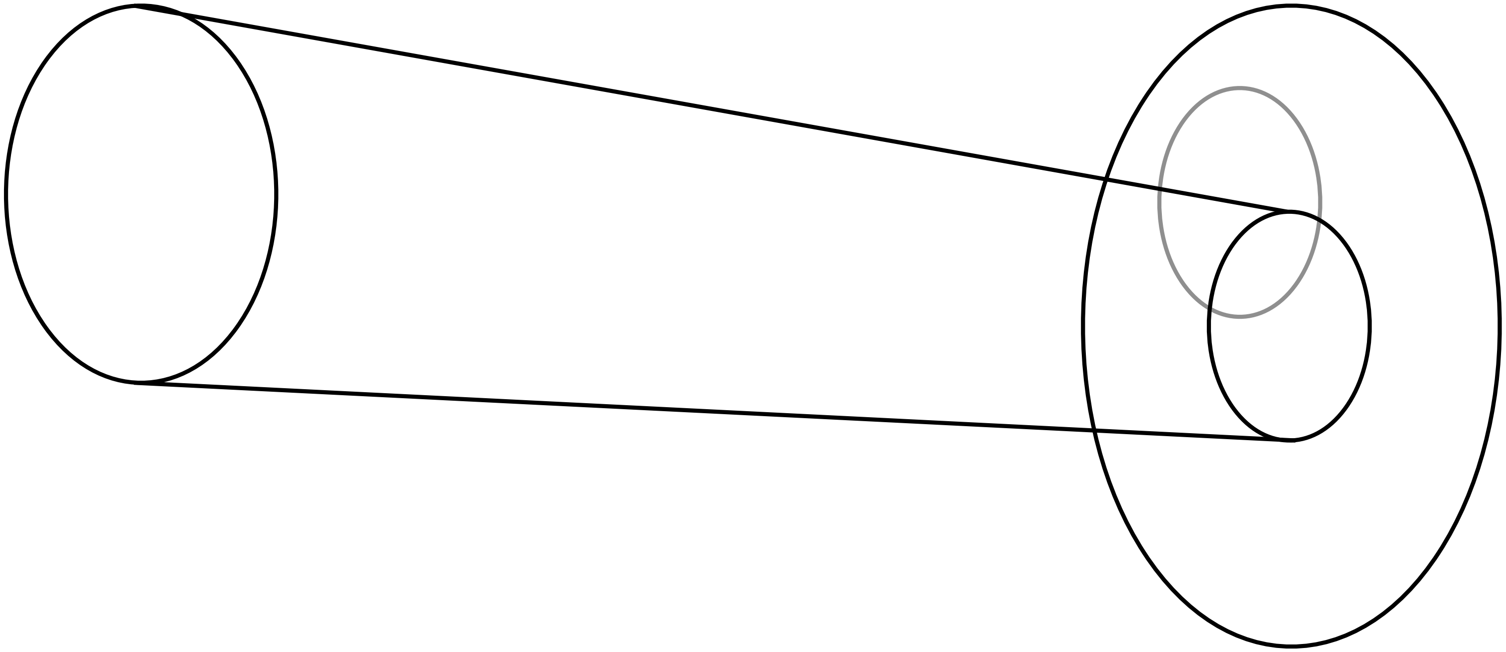
Relocation

- The range of the function used by a process can change over time



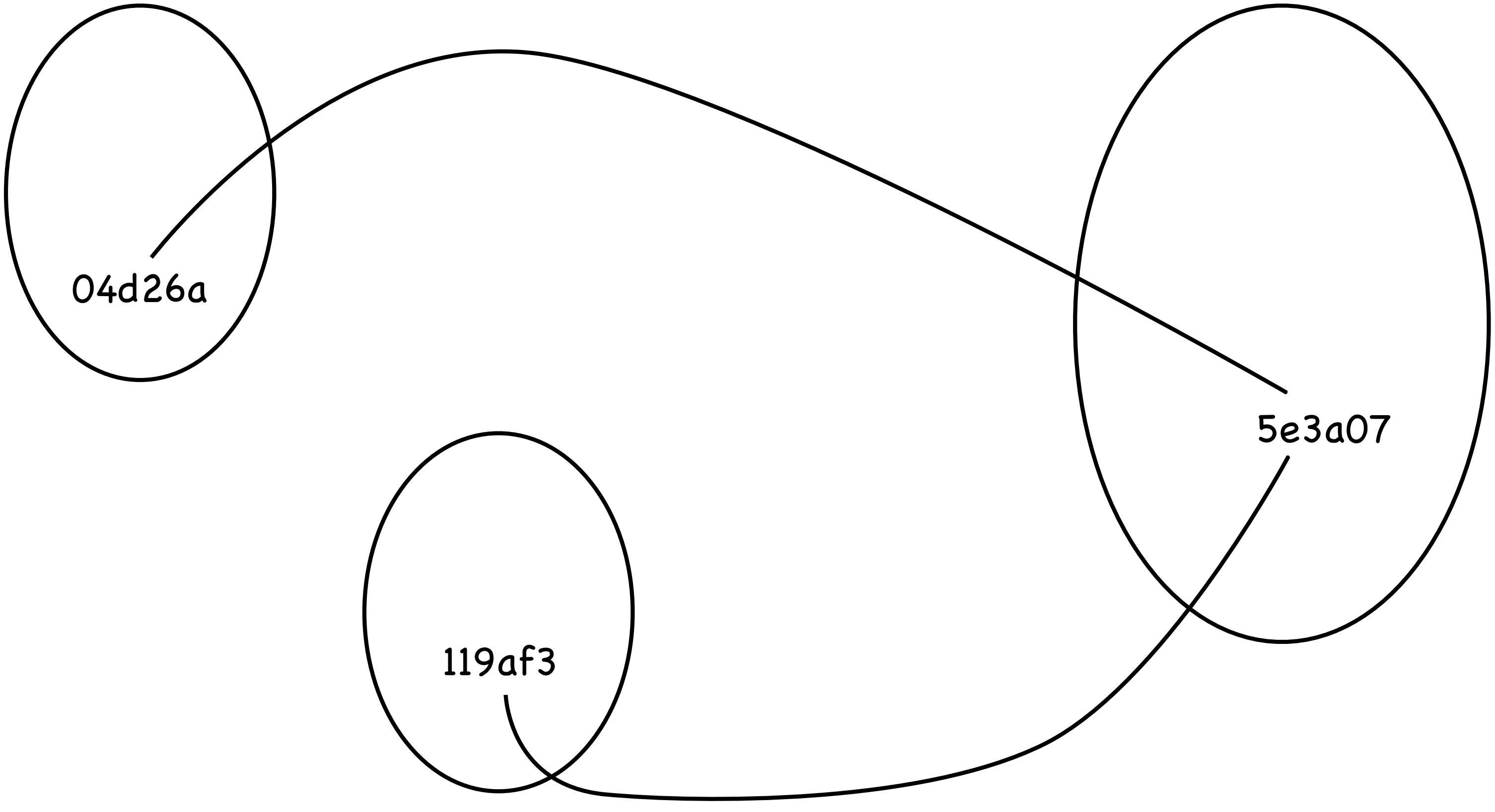
Relocation

- The range of the function used by a process can change over time



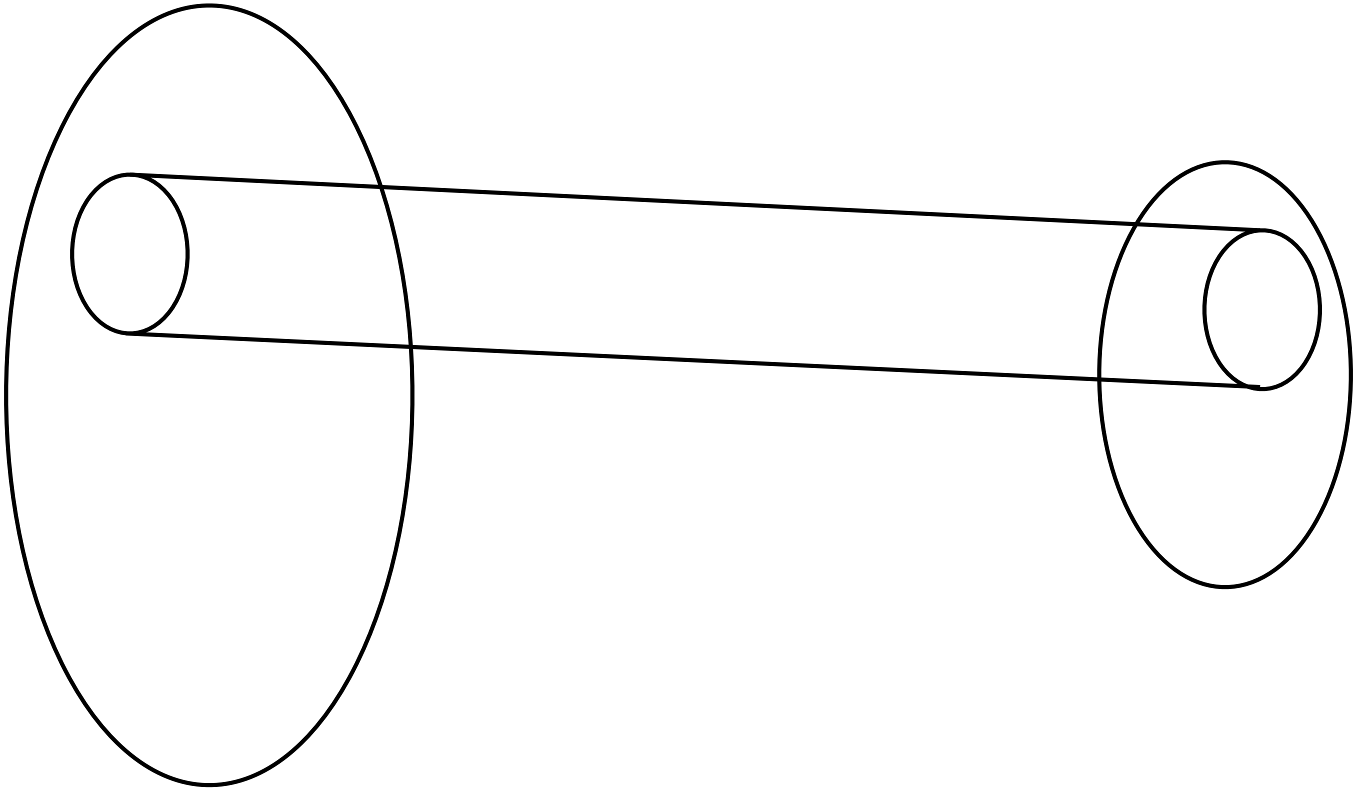
Data Sharing

- Map different virtual addresses of different processes to the same physical address



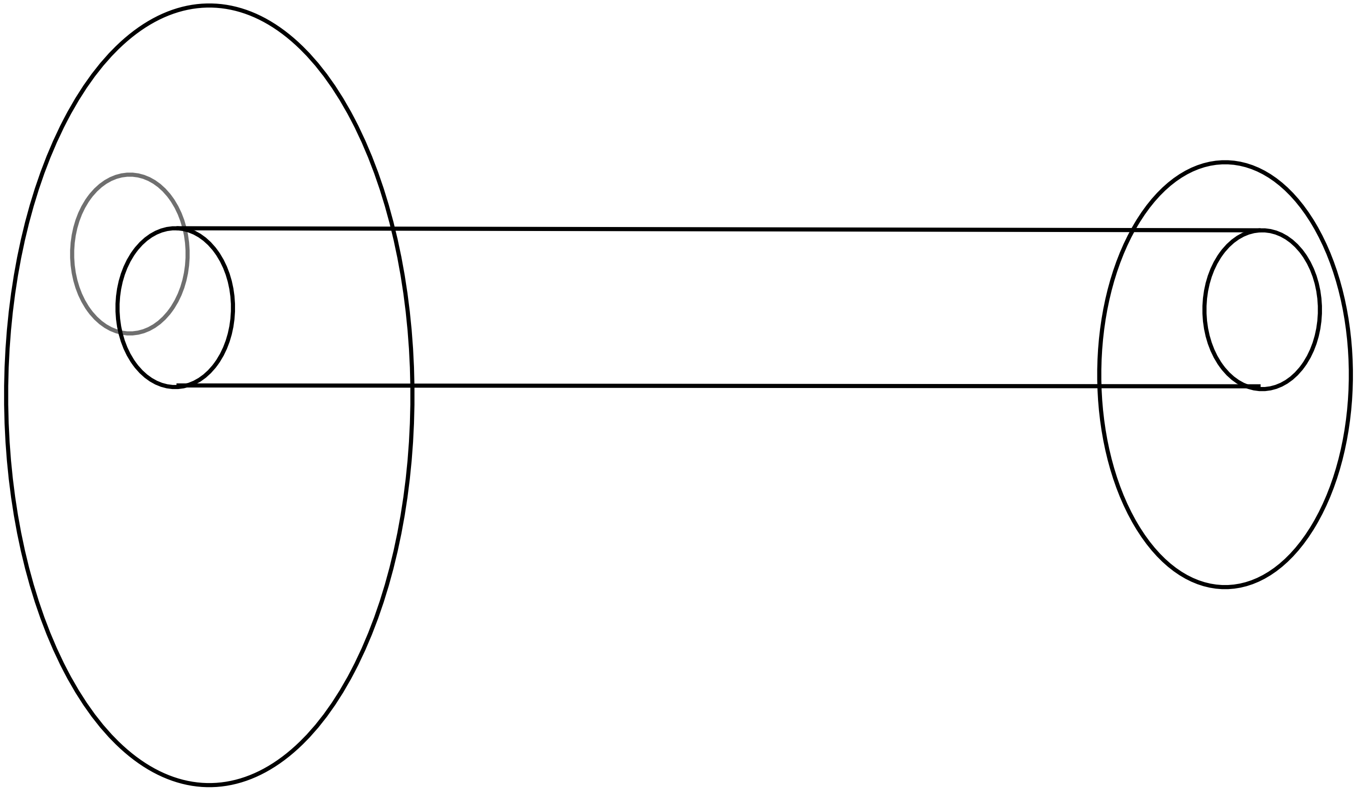
Multiplexing

- Create illusion of almost infinite memory by changing domain (set of virtual addresses) that maps to a given range of physical addresses



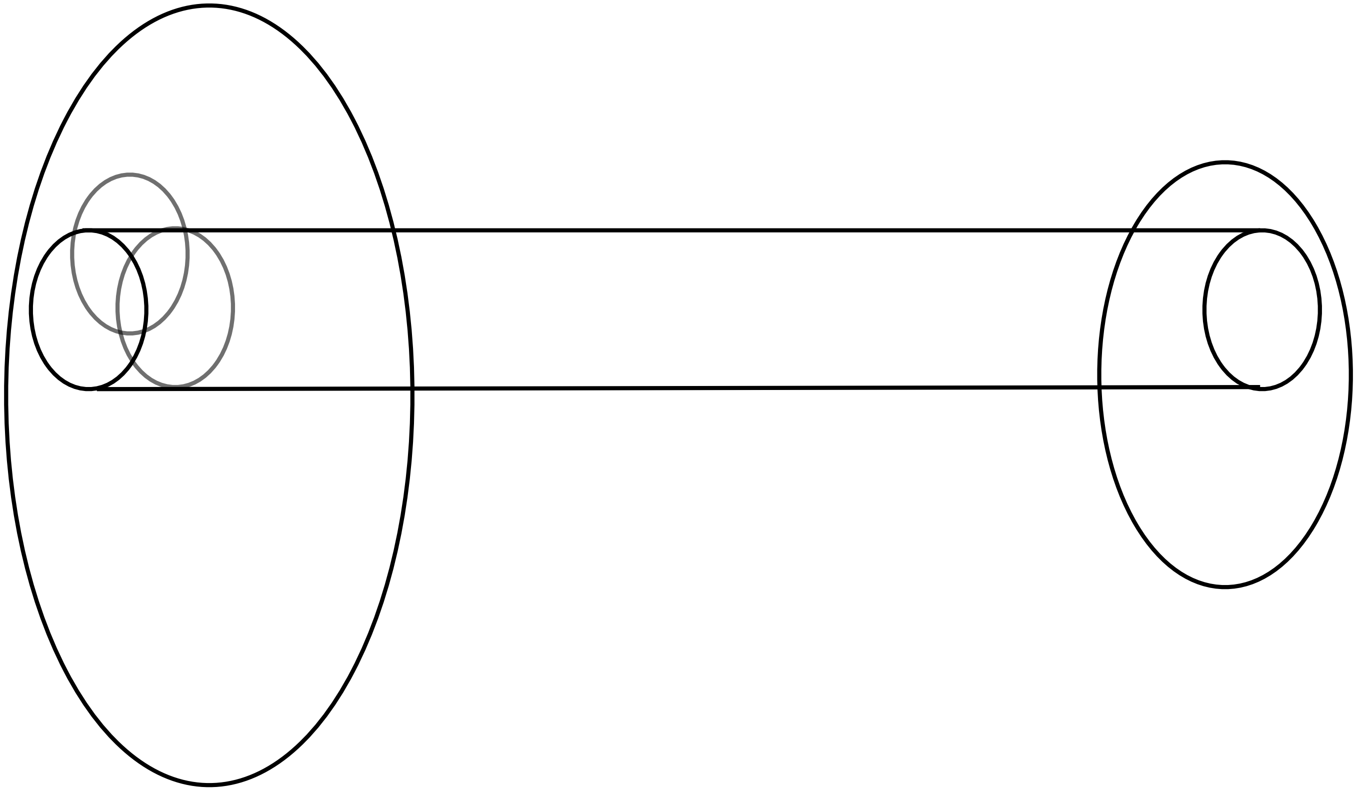
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



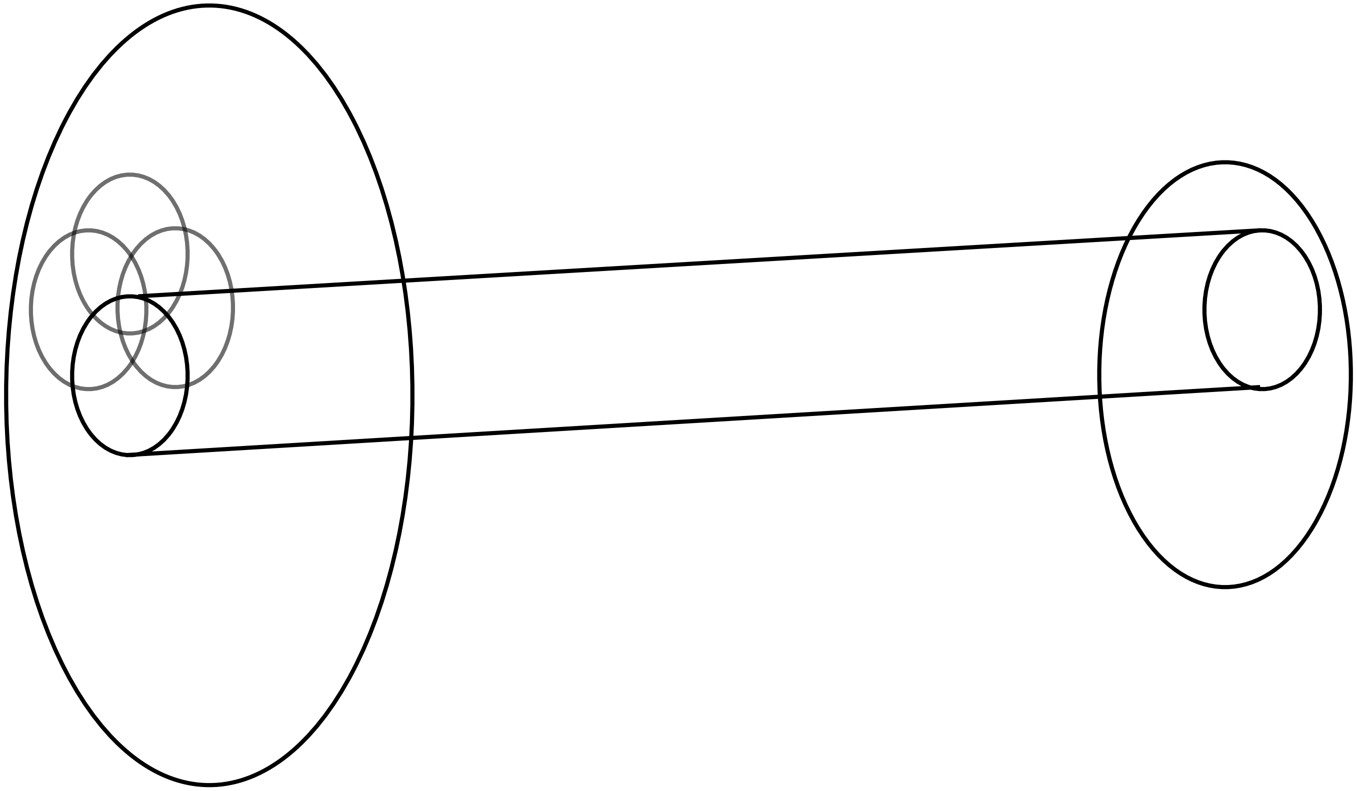
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



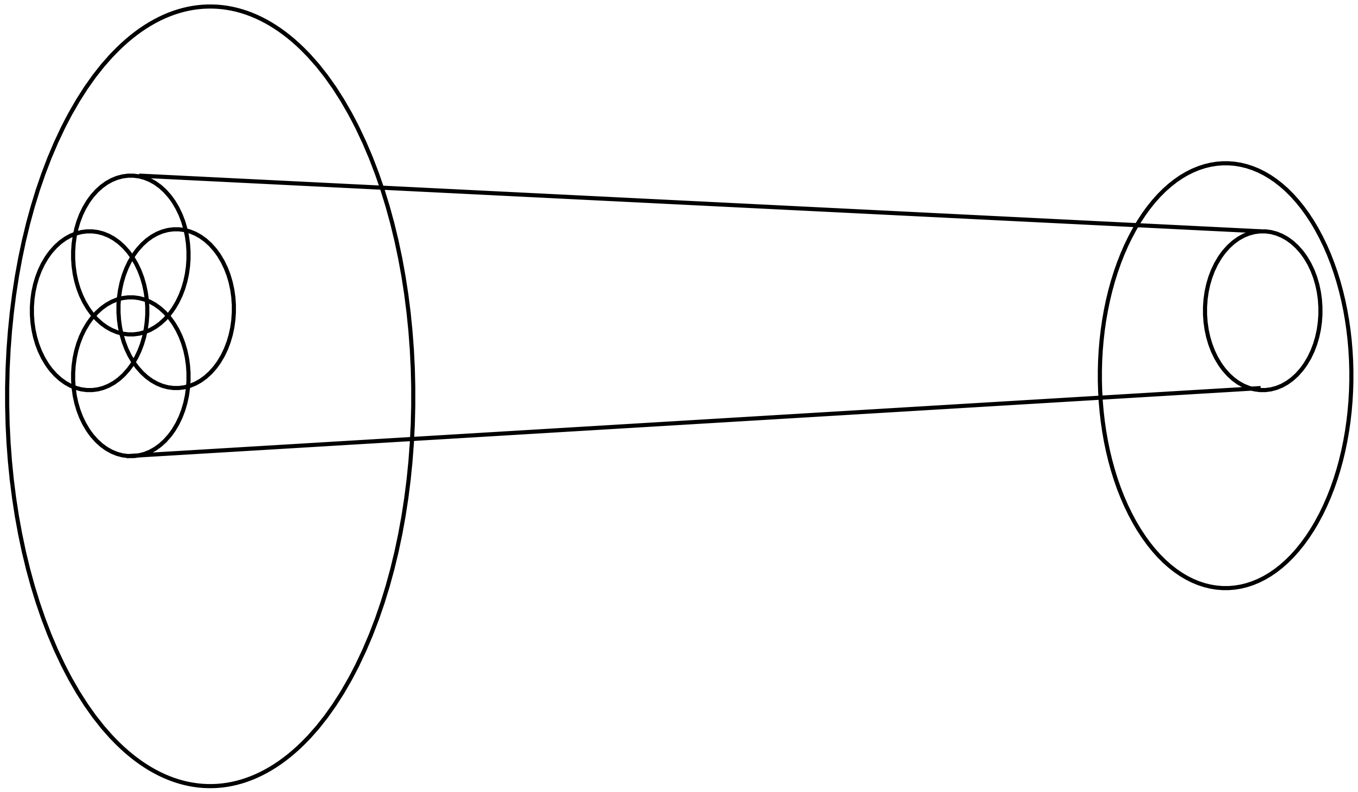
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

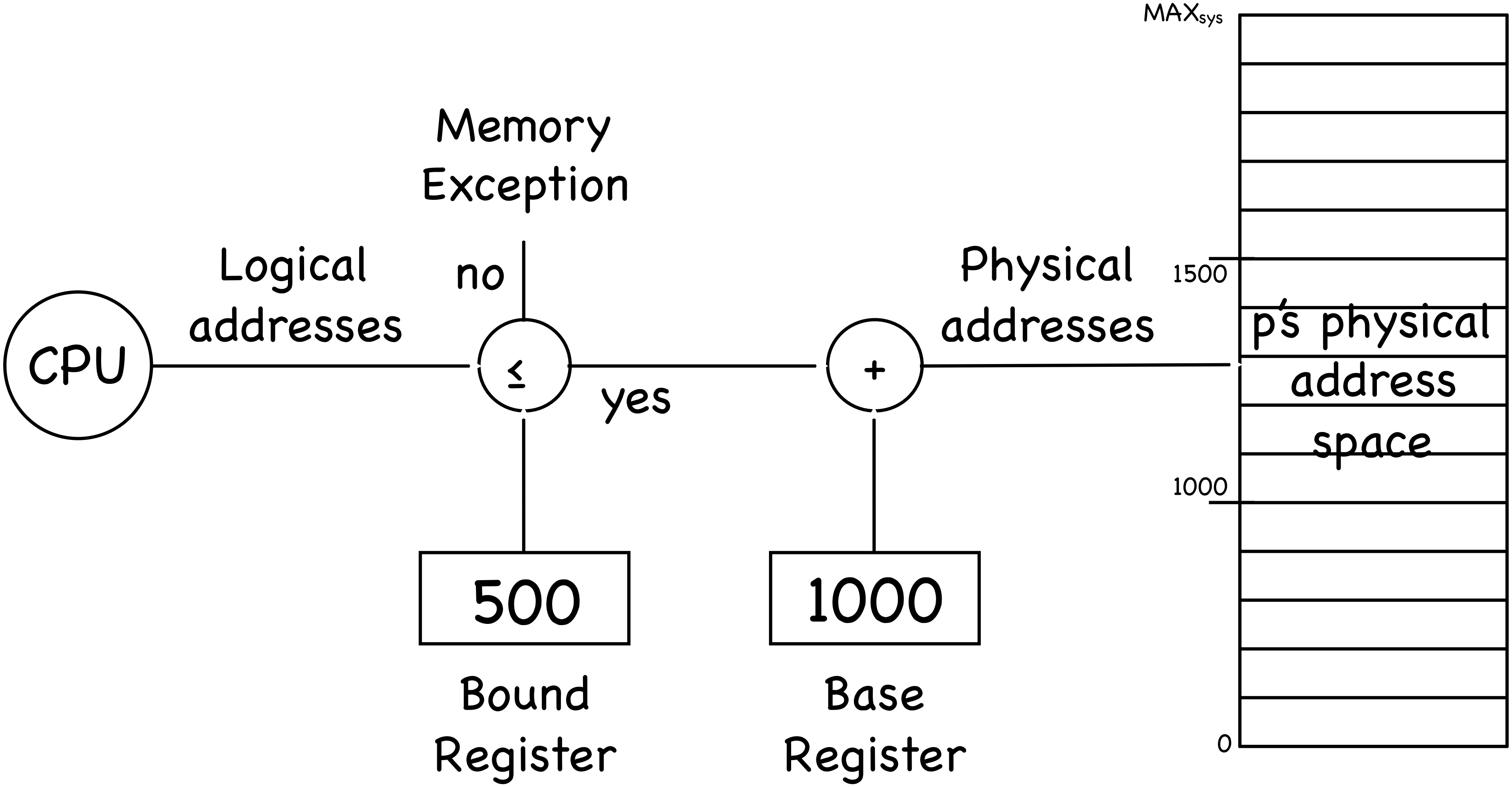


Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



A simple mapping mechanism: Base & Bound



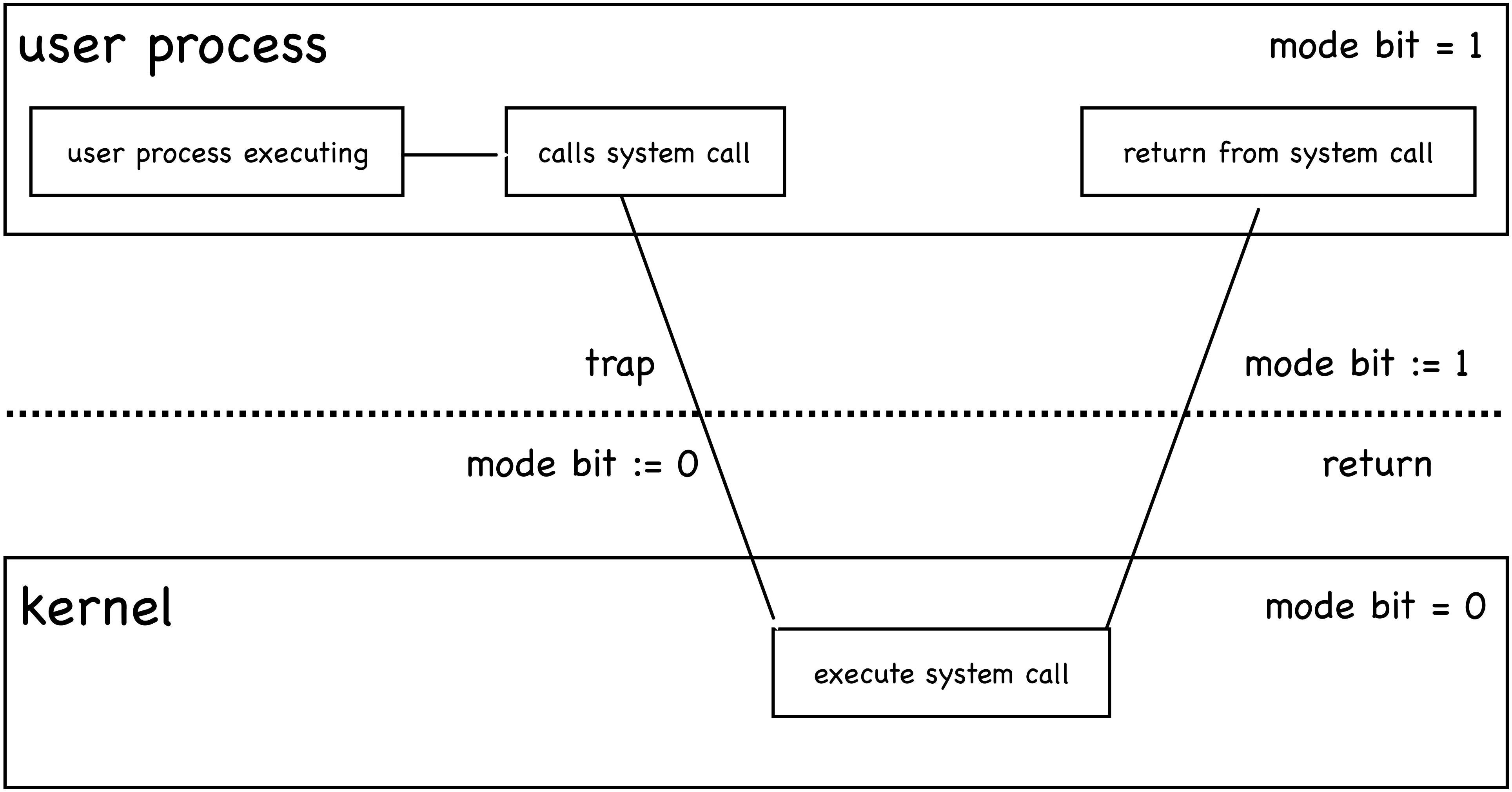
On Base & Limit

- Contiguous Allocation: contiguous virtual addresses are mapped to contiguous physical addresses
- Protection is easy, but sharing is hard
  - Two copies of emacs: want to share code, but have data and stack distinct...
- And there is more...
  - Hard to relocate
  - We want them as far as as possible in virtual address space, but...

III. Timer Interrupts

- Hardware timer
  - can be set to expire after specified delay (time or instructions)
  - when it does, control is passed back to the kernel
- Other interrupts (e.g. I/O completion) also give control to kernel

Crossing the line



user process

mode bit = 1

user process executing

calls system call

return from system call

trap

mode bit := 1

mode bit := 0

return

kernel

mode bit = 0

execute system call

From user mode to kernel mode...

Exceptions

- user program acts silly (e.g. division by zero)
- attempt to perform a privileged instruction
    - sometime on purpose! (breakpoints)
- synchronous

System calls/traps

- user program requests OS service
- synchronous

Interrupts

- HW device requires OS service
    - timer, I/O device, interprocessor
- asynchronous

…and viceversa

Resume after exception, interrupt or syscall

- restore PC, SP, registers;
- toggle mode

Switch to different process

- load PC, SP, registers from 's PCB
- toggles mode

If new process

- copy program in memory,
- set PC and SP
- toggle mode

User-level upcall

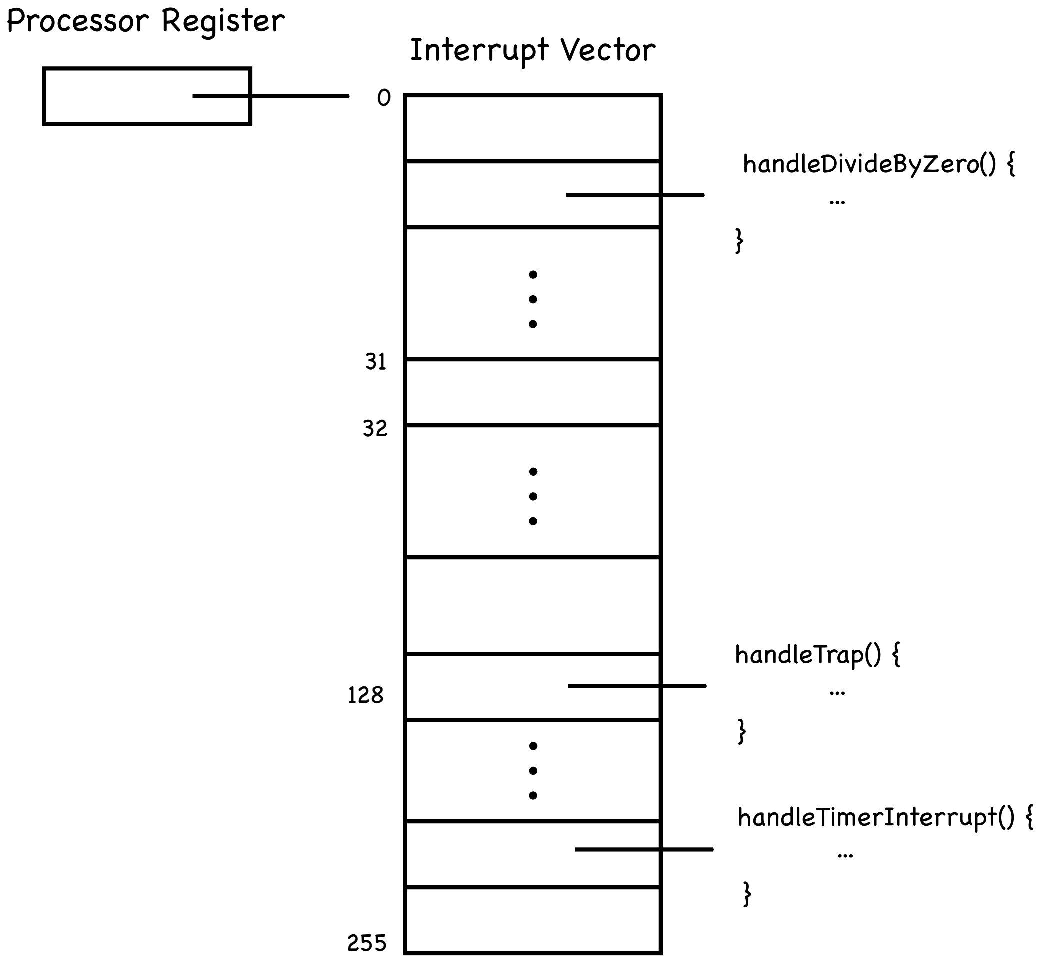
- a sort of user-level interrupt handling

Making the transition: Safe mode switch

- Common sequences of instructions to cross boundary, which provide:
    - Limited entry
        - entry point in the kernel set up by kernel
    - Atomic changes to process state
        - PC, SP, memory protection, mode
    - Transparent restartable execution
        - user program must be restarted exactly as it was before kernel got control

Interrupt vector

- Hardware identifies why boundary is crossed
    - trap?
    - interrupt (which device)?
    - exception?
- Hardware selects entry from interrupt vector
- Appropriate handler is invoked



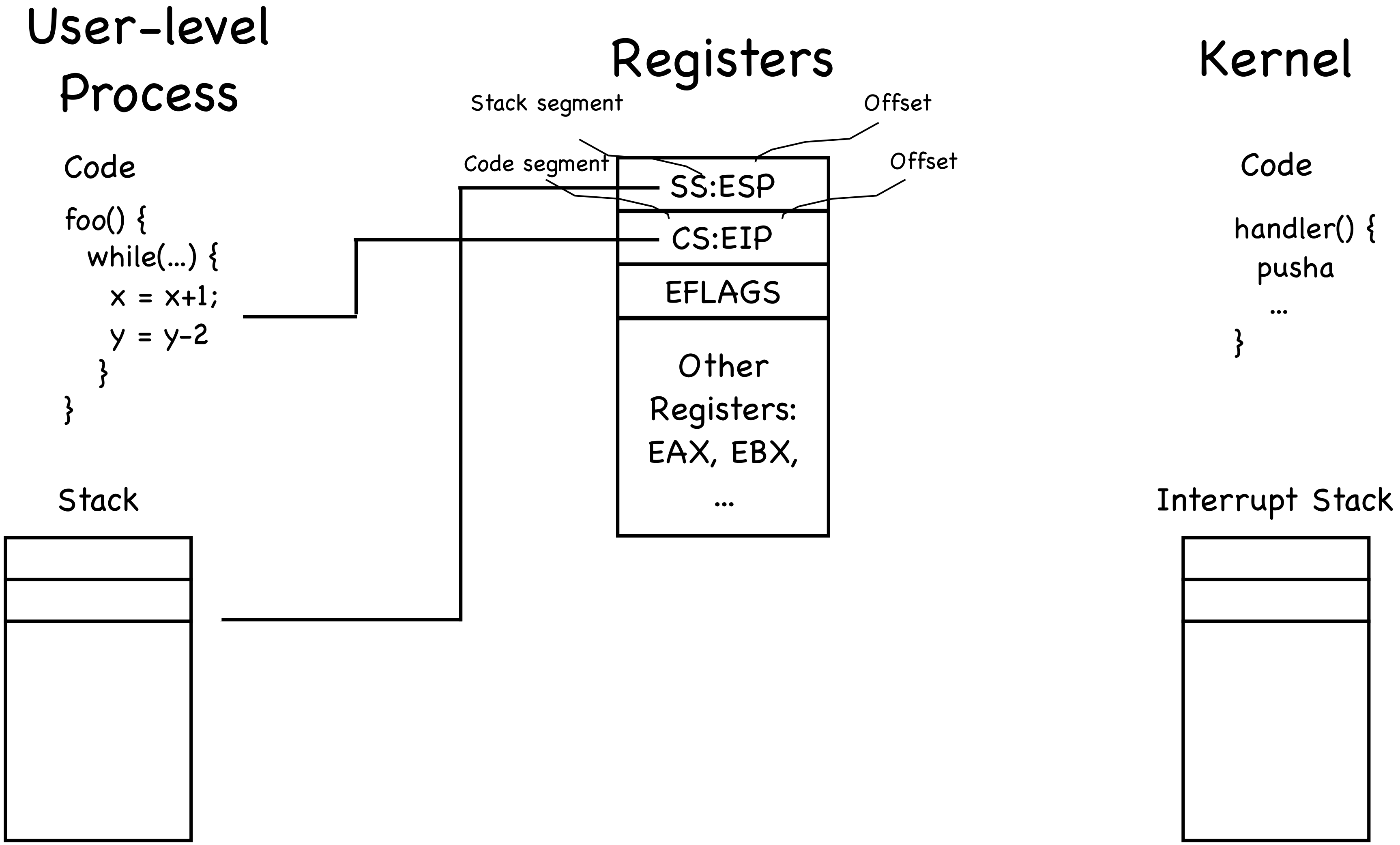
Interrupt stack

- Stores execution context of interrupted process
  - HW saves SP, PC
  - Handler saves remaining registers
- Stores handler's local variables
- Pointed by privileged register
- One per process (or per thread!)
  - Why not use the stack in user's space?

Interrupt masking

- What if an interrupt occurs while running an interrupt handler?
    - Disable interrupts via privileged instruction
        - Overdramatic… it actually defers them
    - Just use the current SP of Interrupt stack

Mode switch on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

Stack segment

Offset

Code segment

Offset

SS:ESP

CS:EIP

EFLAGS

Other Registers: EAX, EBX, ...

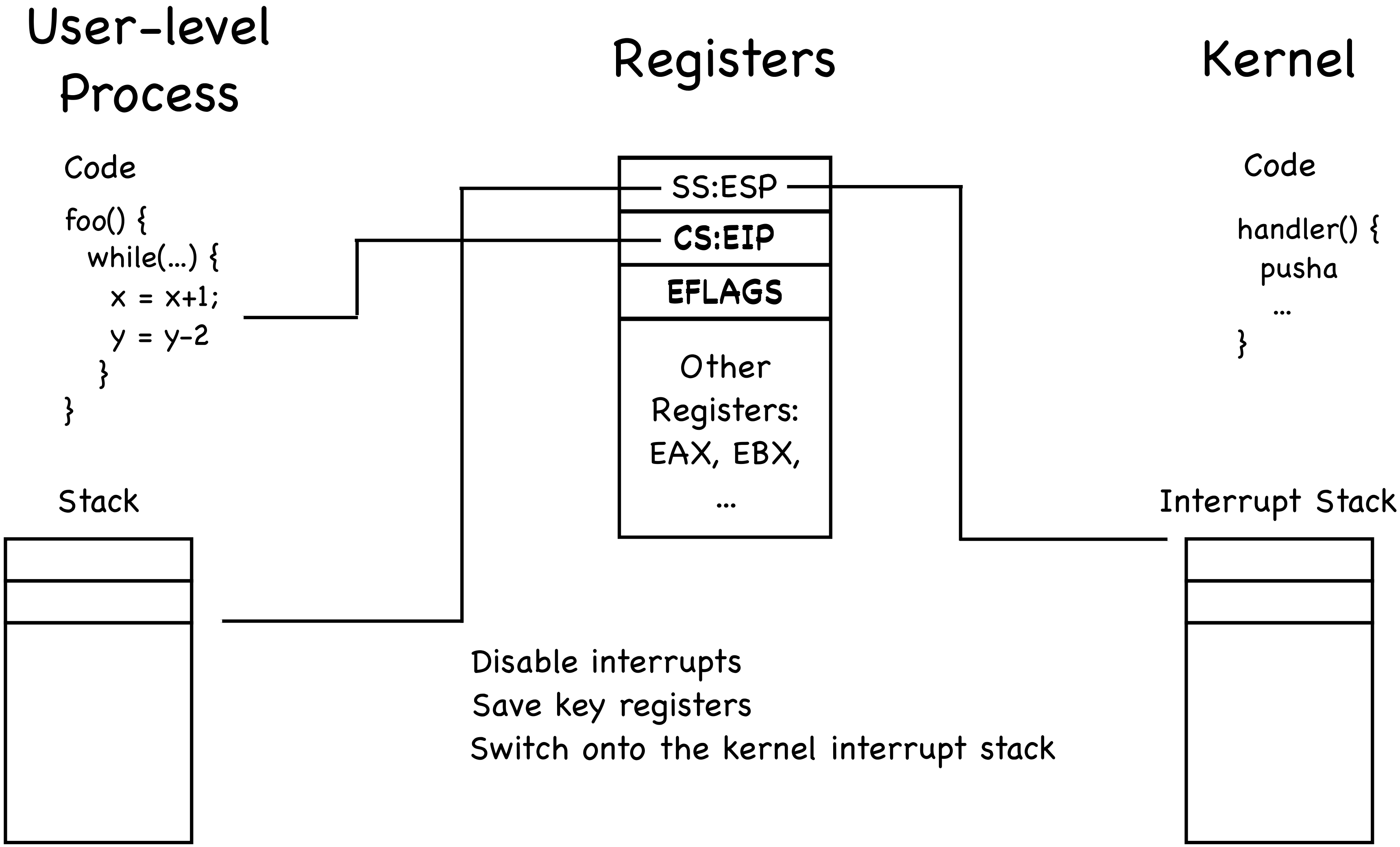
Kernel

Code

```
handler() {
  pusha
   ...
}
```

Interrupt Stack

Mode switch on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

SS:ESP

CS:EIP

EFLAGS

Other Registers: EAX, EBX, ...

Kernel

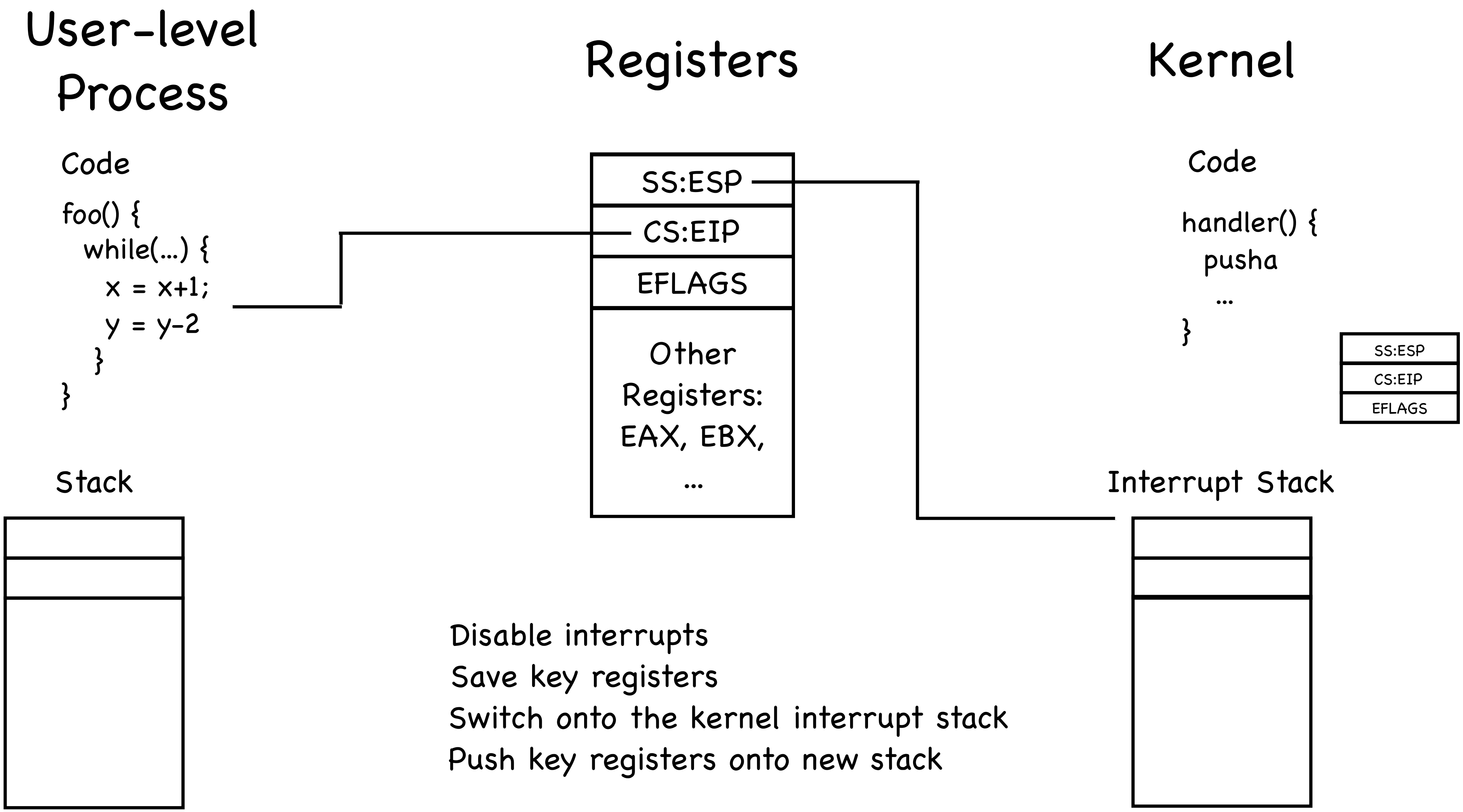
Code

```
handler() {
  pusha
    ...
}
```

Interrupt Stack

Disable interrupts
Save key registers
Switch onto the kernel interrupt stack

Mode switch on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

SS:ESP

CS:EIP

EFLAGS

Other Registers: EAX, EBX, ...

Kernel

Code

```
handler() {
  pusha
   ...
}
```

SS:ESP

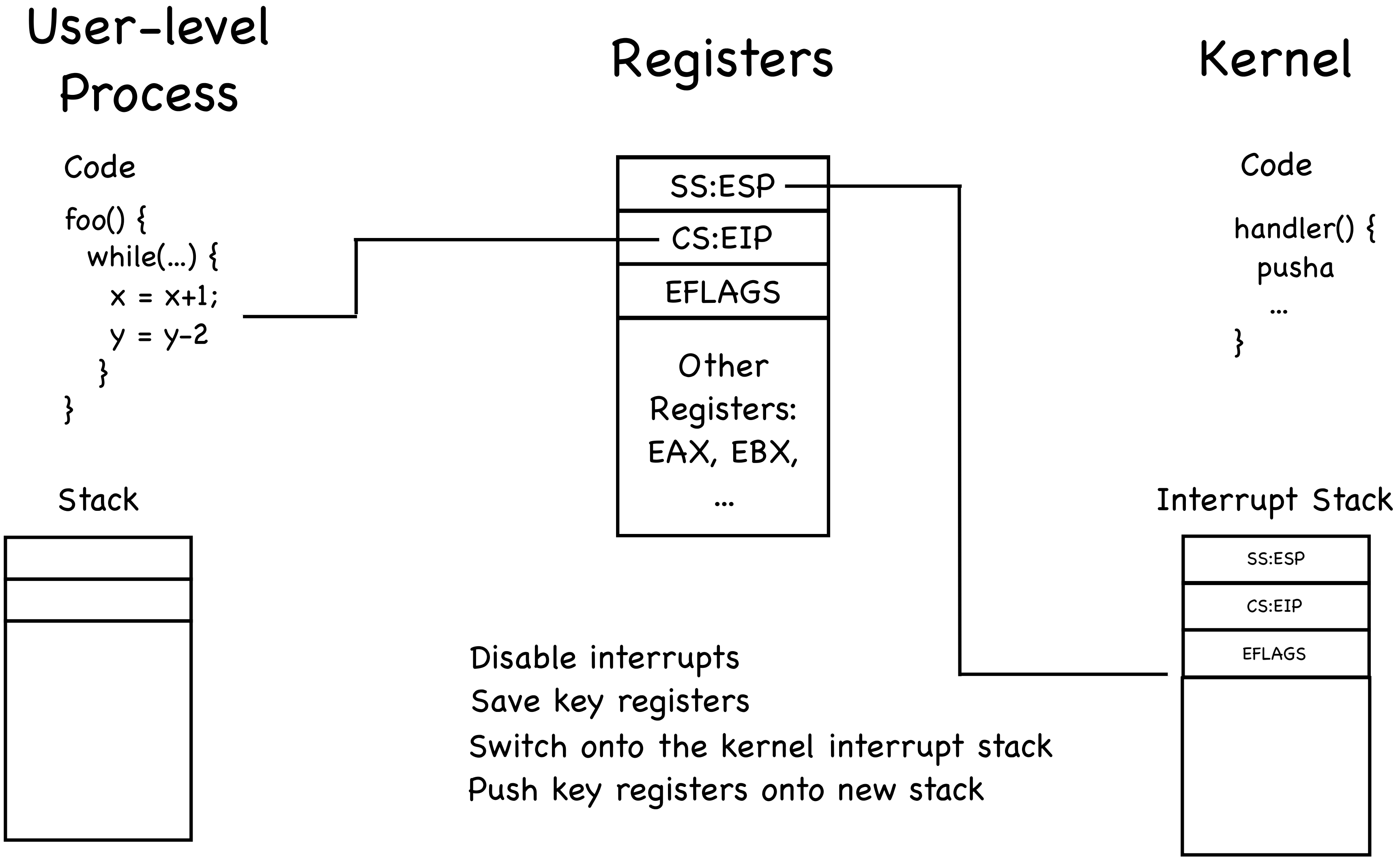
CS:EIP

EFLAGS

Interrupt Stack

Disable interrupts
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack

Mode switch on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

SS:ESP

CS:EIP

EFLAGS

Other Registers: EAX, EBX, ...

Kernel

Code

```
handler() {
  pusha
   ...
}
```

Interrupt Stack

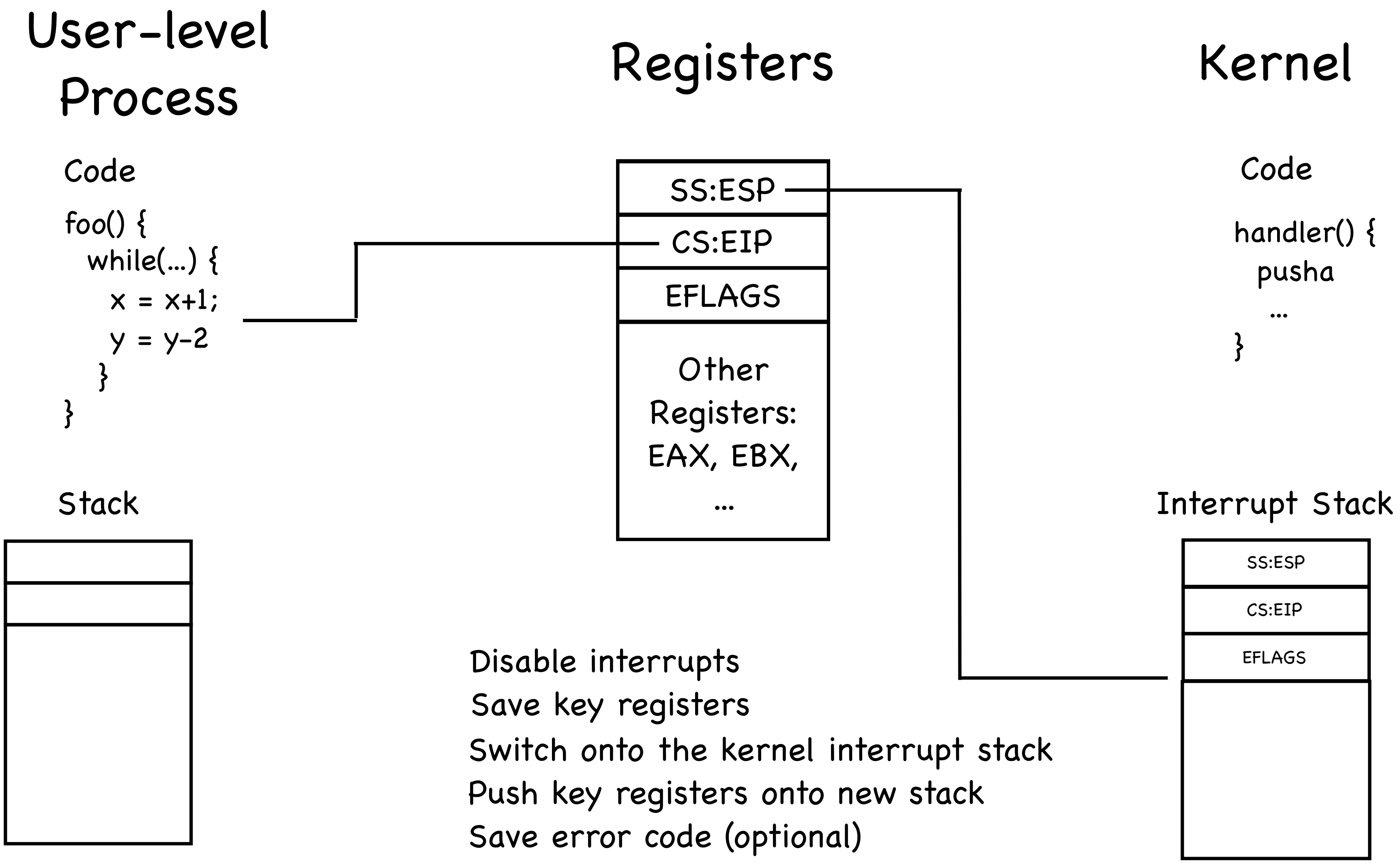
SS:ESP

CS:EIP

EFLAGS

Disable interrupts
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack

Mode switch on x86

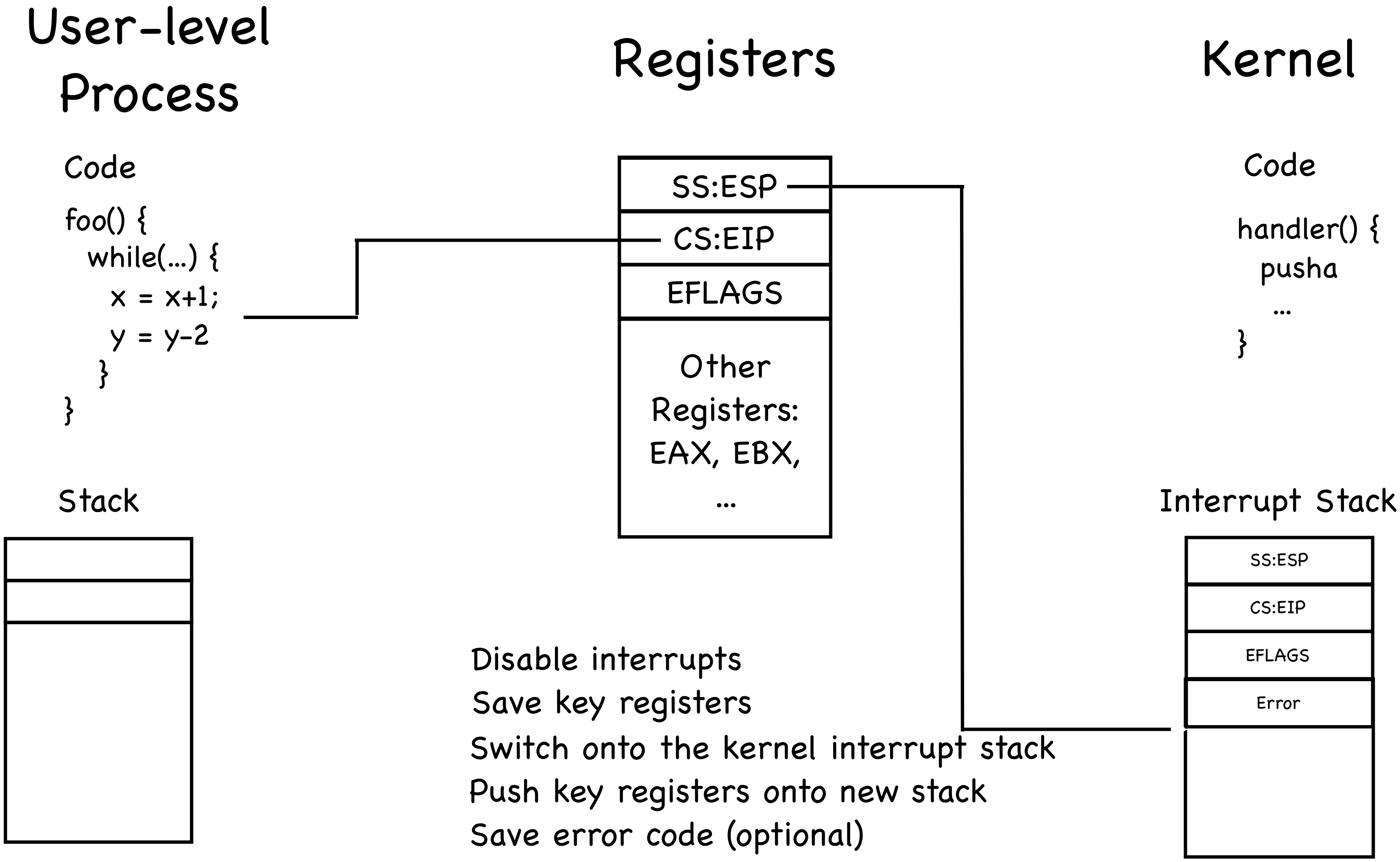


```
foo() {
  while(…) {
    x = x+1;
    y = y-2
  }
}
```

```
handler() {
  pusha
   …
}
```

Disable interrupts
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack
Save error code (optional)

Mode switch on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

Kernel

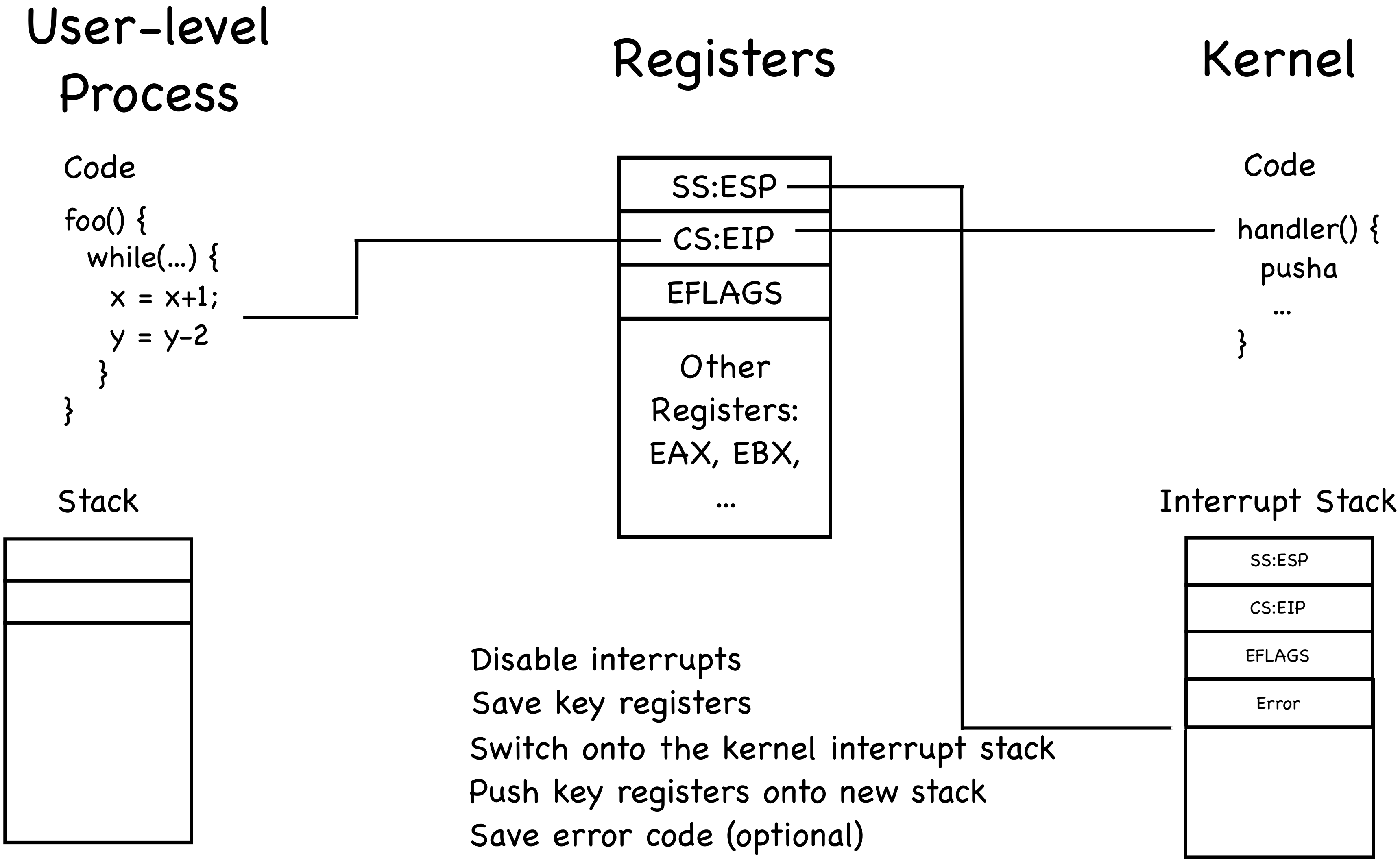
Code

```
handler() {
  pusha
    ...
}
```

Interrupt Stack

Disable interrupts
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack
Save error code (optional)

Mode switch on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

SS:ESP

CS:EIP

EFLAGS

Other Registers: EAX, EBX, ...

Kernel

Code

```
handler() {
  pusha
  ...
}
```

Interrupt Stack

SS:ESP

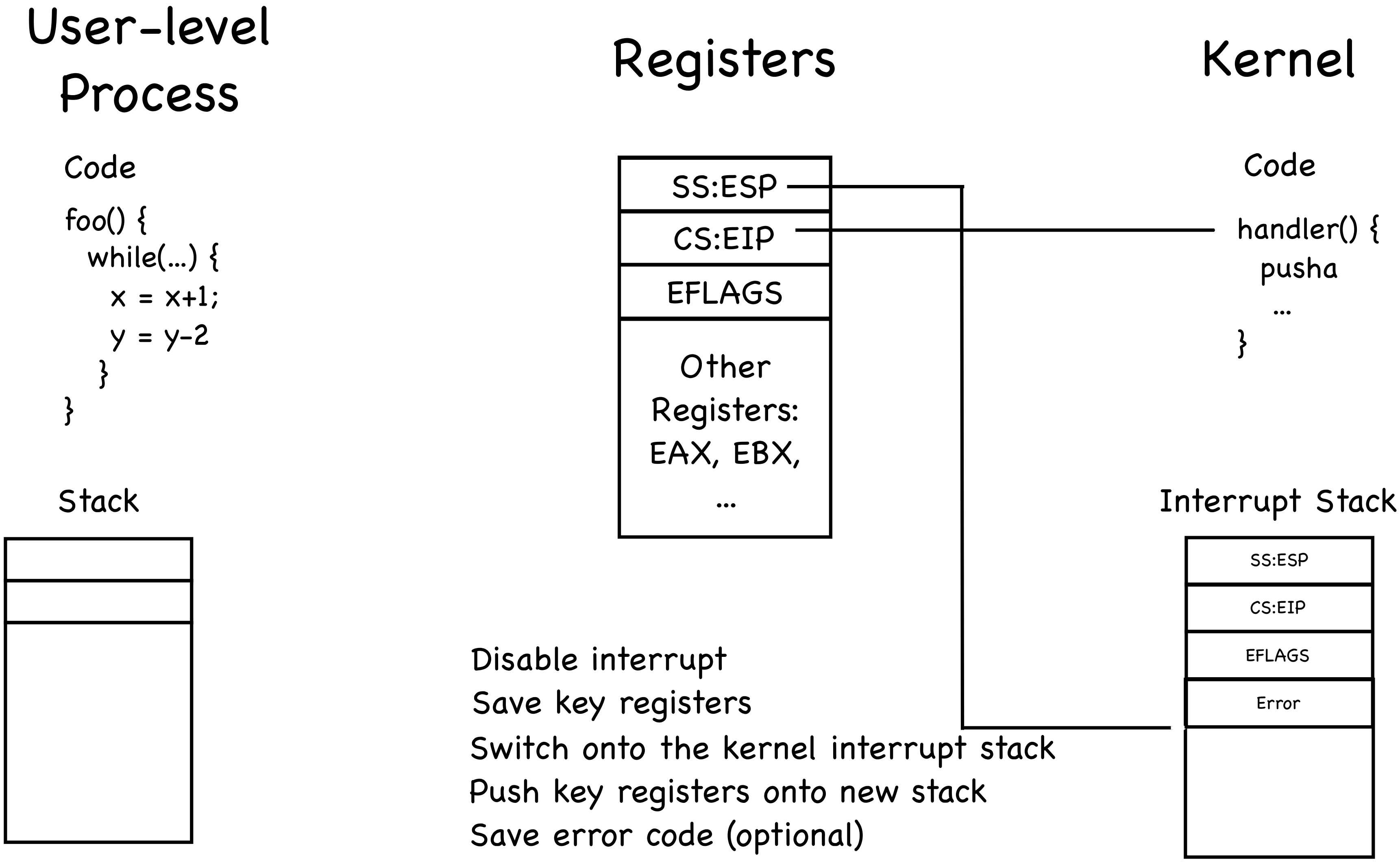
CS:EIP

EFLAGS

Error

Disable interrupts
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack
Save error code (optional)
Invoke interrupt handler

Mode switch on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

Kernel

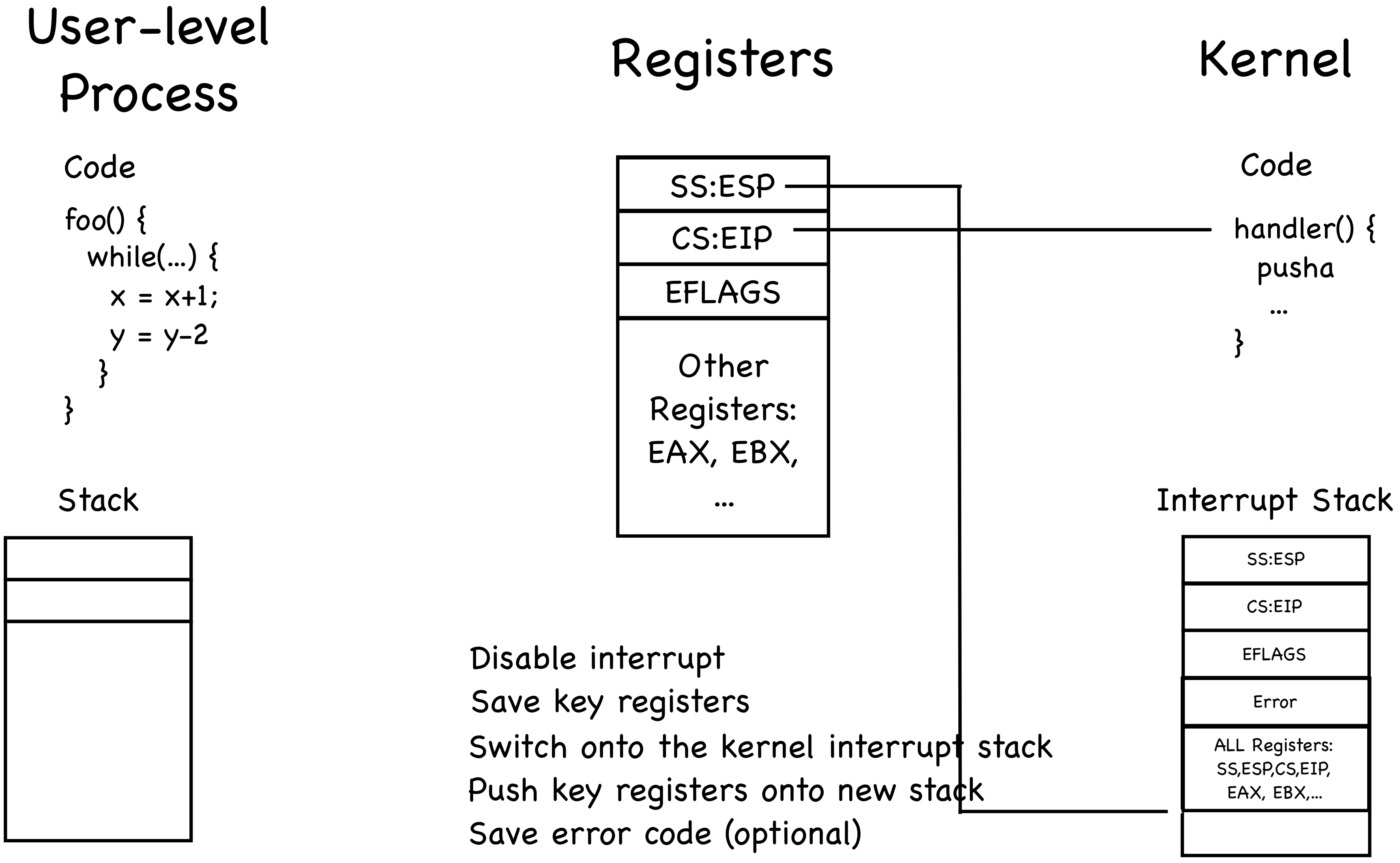
Code

```
handler() {
  pusha
   ...
}
```

Interrupt Stack

Disable interrupt
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack
Save error code (optional)
Invoke interrupt handler
Handler pushes all registers on stack

Mode switch on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

SS:ESP

CS:EIP

EFLAGS

Other Registers: EAX, EBX, ...

Kernel

Code

```
handler() {
  pusha
   ...
}
```

Interrupt Stack

SS:ESP

CS:EIP

EFLAGS

Error

ALL Registers: SS,ESP,CS,EIP, EAX, EBX,...

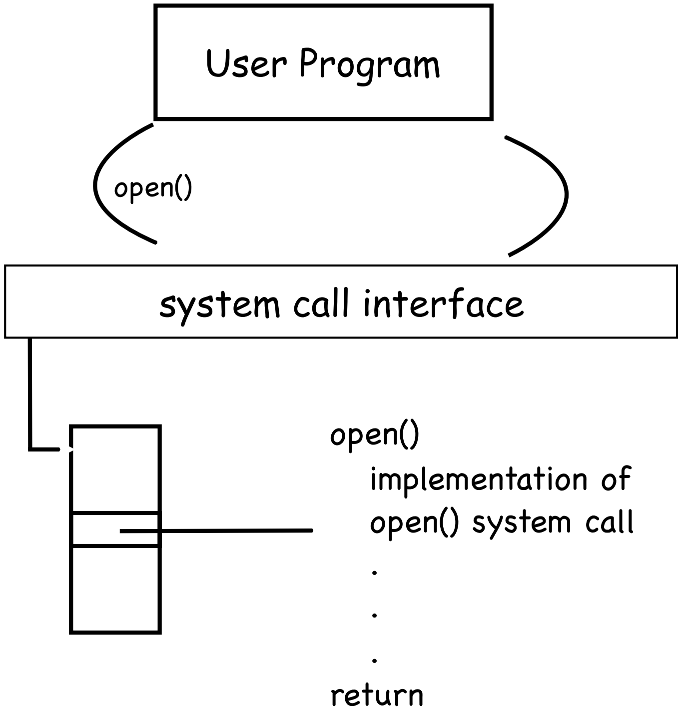
Disable interrupt
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack
Save error code (optional)
Invoke interrupt handler
Handler pushes all registers on stack

Switching back

- From an interrupt, just reverse all steps!
- From exception and system call, increment PC on return
    - on exception, handler changes PC at the base of the stack
    - on system call, increment is done by hw when saving user level state

System calls

- Programming interface to the services provided by the OS
- Mostly accessed through an API (Application Programming Interface)
  - Win32, POSIX, Java API
- Parameters passed according to calling convention
  - registers, stack, etc.



System call stubs

User

- Set up parameters
- call `int 080` to context switch

```
open:
  movl #SysCall_Open, %eax
  int 080
  ret
```

Kernel

- Validate parameters
  - defend against errors in content and format of args
- Copy before check
  - prevent TOCTOU
- Copy back any result

The Skinny

- Syscall interface allows separation of concern
  - Innovation
- Narrow
  - simple
  - powerful
  - highly portable
  - robust

Web Servers

Compilers

Word Processing

Databases

Web Browsers

Email

Portable OS Library

System call interface

Portable OS Kernel

x86 ARM PowerPC

10Mbps/100Mbps/1Gbps Ethernet

1802.11 a/b/g/n SCSI

Graphics accellerators LCD Screens

Upcalls: virtualizing interrupts

Interrupts/Exceptions	Upcalls/Signals
Hardware-defined Interrupts & exceptions	Kernel-defined signals
Interrupt vector for handlers (kernel)	Handlers (user)
Interrupt stack (kernel)	Signal stack (user)
Interrupt masking (kernel)	Signal masking (user)
Processor state (kernel)	Processor State (user)

Signaling

Why?

- To terminate an application
- To suspend it/resume it (e.g., for debugging)
- To alert of timer expiration

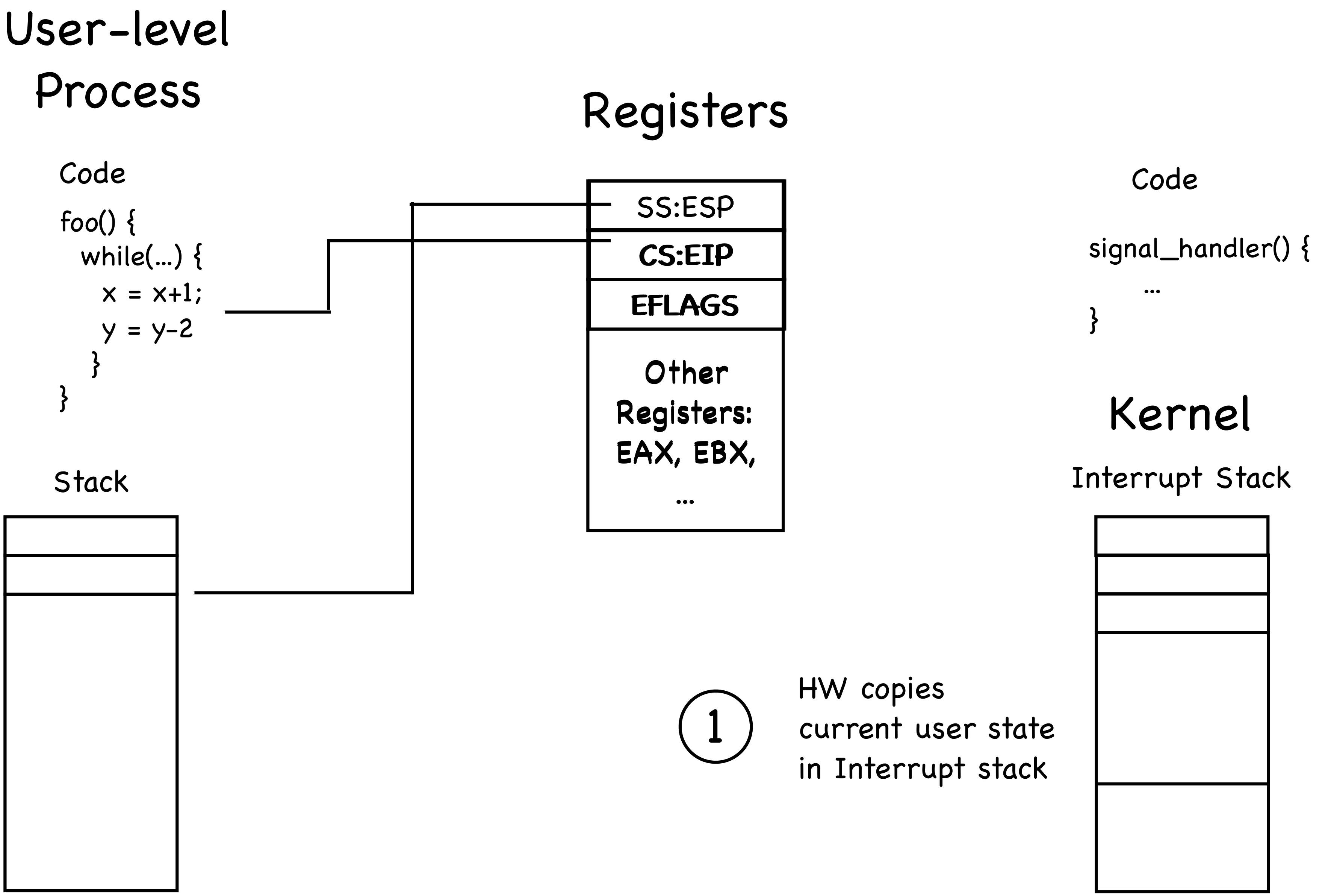
Upon receipt:

- Ignore
- Terminate process
- Catch through handler

More on signals

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., CTRL-C from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
14	SIGALRM	Terminate	Tmer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGSTP	Stop until SIGCONT	Stop signal from terminal (e.g., CTRL-Z from keyboard)

Unix signals



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

SS:ESP

CS:EIP

EFLAGS

Other Registers: EAX, EBX, ...

Code

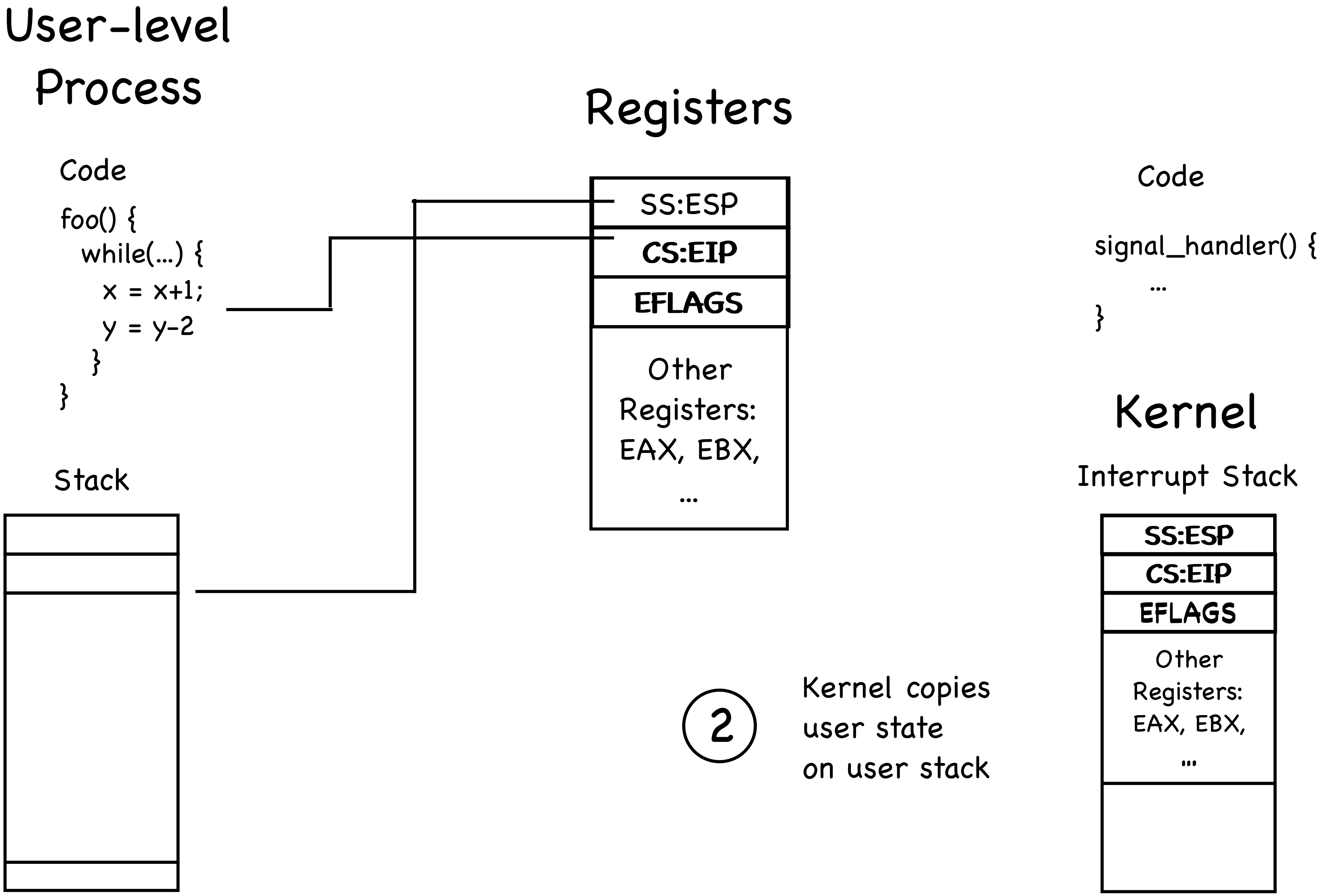
```
signal_handler() {
      ...
}
```

Kernel

Interrupt Stack

① HW copies current user state in Interrupt stack

Unix signals



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

- SS:ESP
- CS:EIP
- EFLAGS
- Other Registers: EAX, EBX, ...

Code

```
signal_handler() {
      ...
}
```

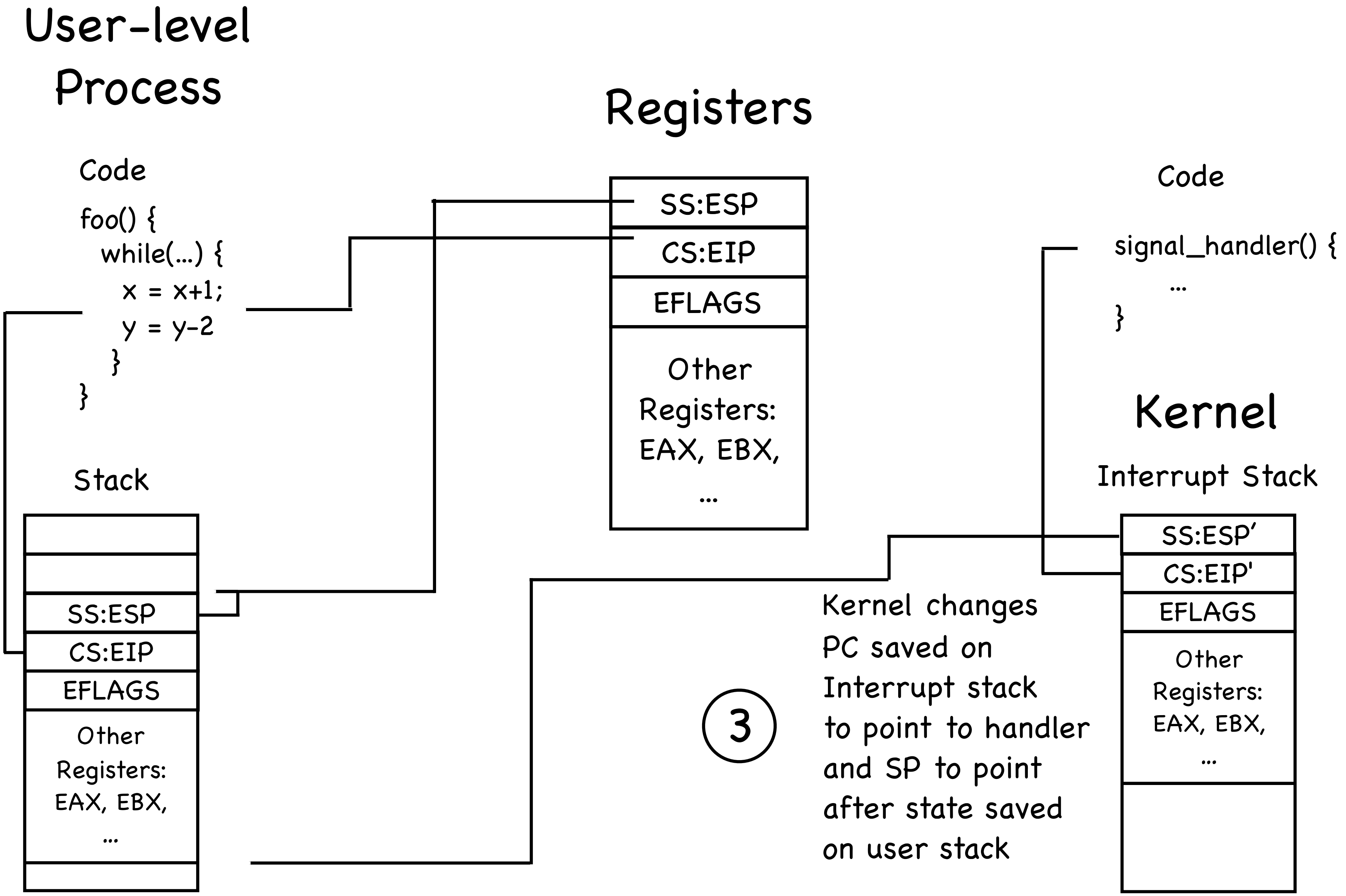
Kernel

Interrupt Stack

- SS:ESP
- CS:EIP
- EFLAGS
- Other Registers: EAX, EBX, ...

② Kernel copies user state on user stack

Unix signals



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, ...

Registers

SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, ...

Code

```
signal_handler() {
    ...
}
```

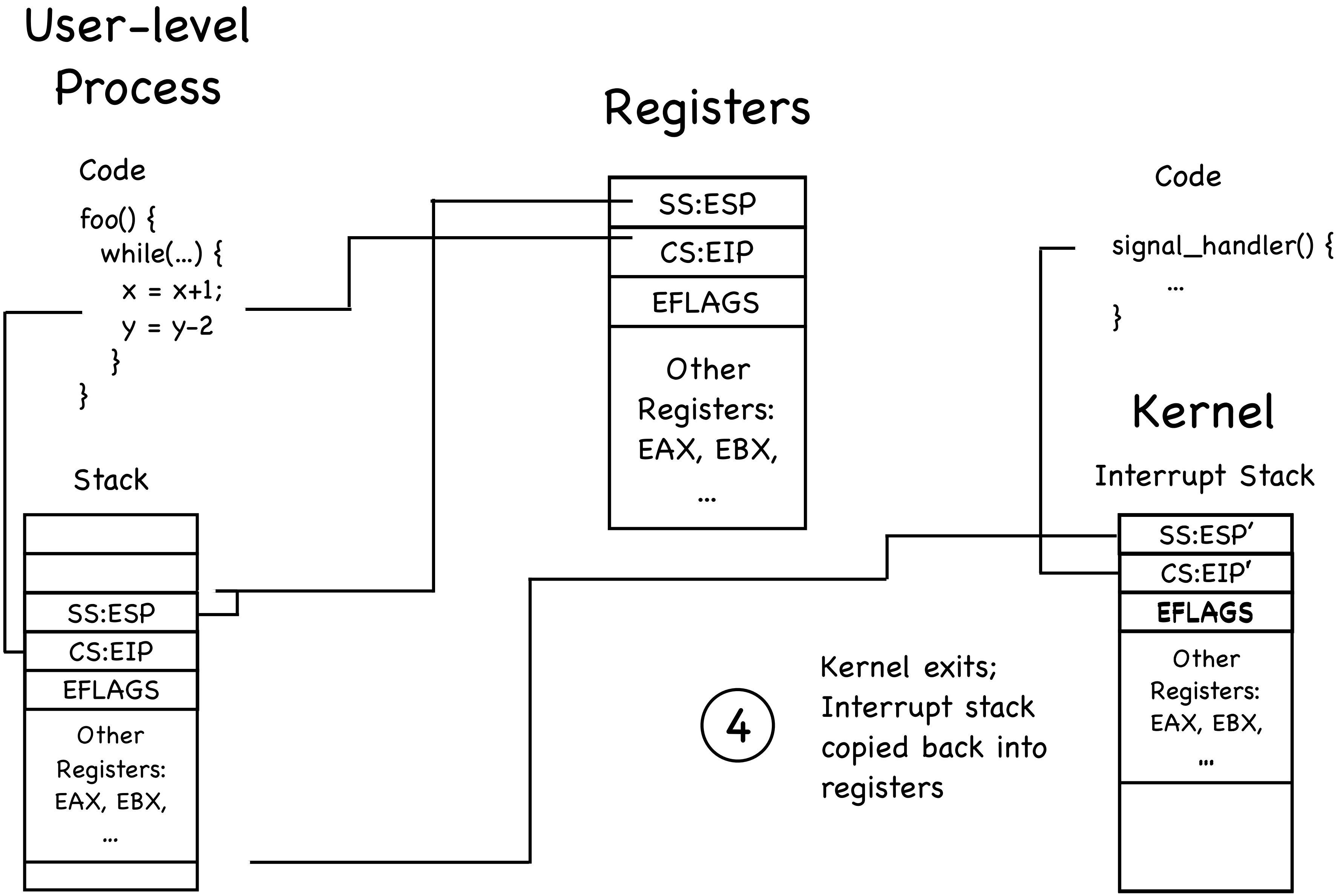
Kernel

Interrupt Stack

SS:ESP’
CS:EIP'
EFLAGS
Other Registers: EAX, EBX, ...

③ Kernel changes PC saved on Interrupt stack to point to handler and SP to point after state saved on user stack

Unix signals

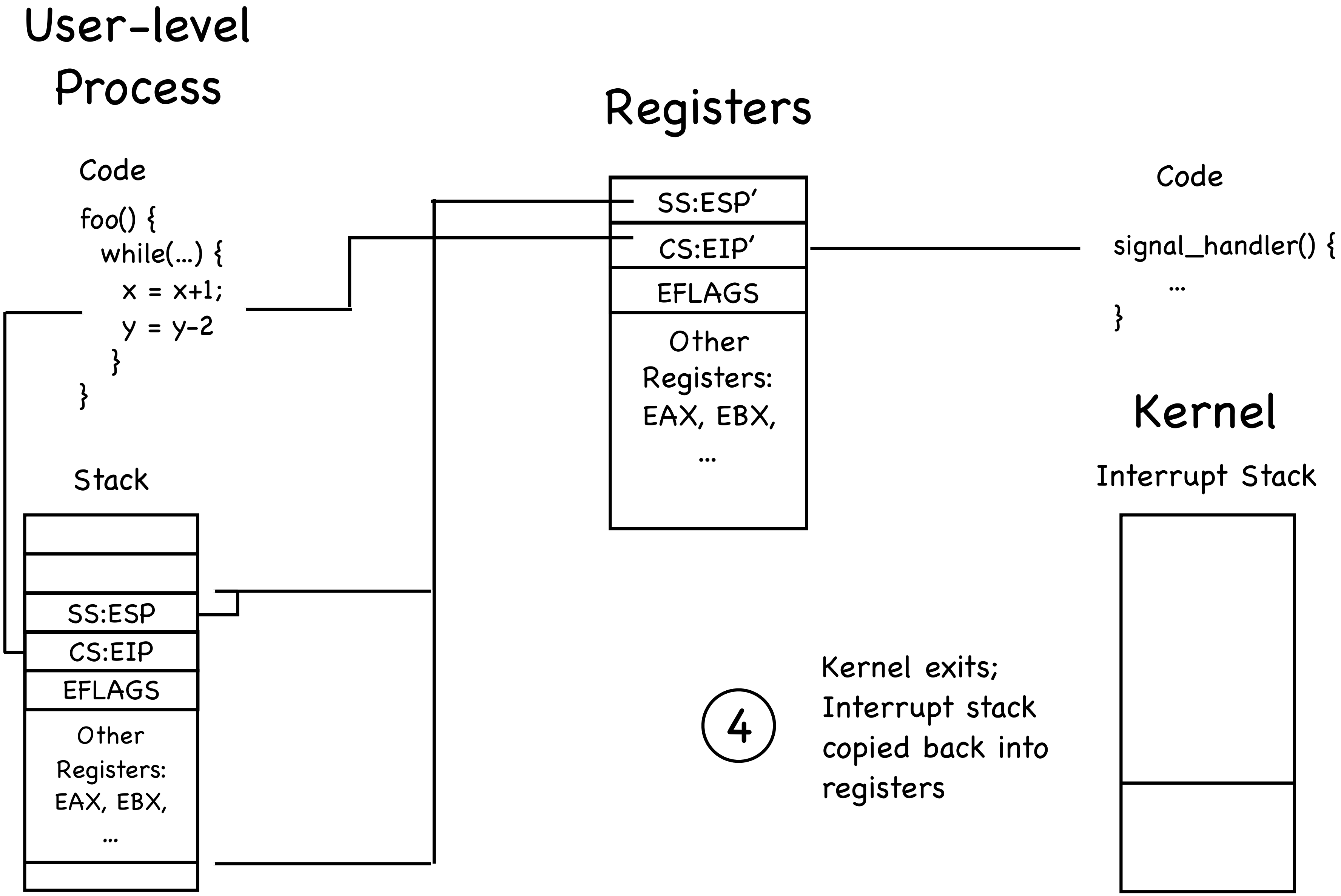


```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

```
signal_handler() {
    ...
}
```

④ Kernel exits; Interrupt stack copied back into registers

Unix signals



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, ...

Registers

SS:ESP’
CS:EIP’
EFLAGS
Other Registers: EAX, EBX, ...

Code

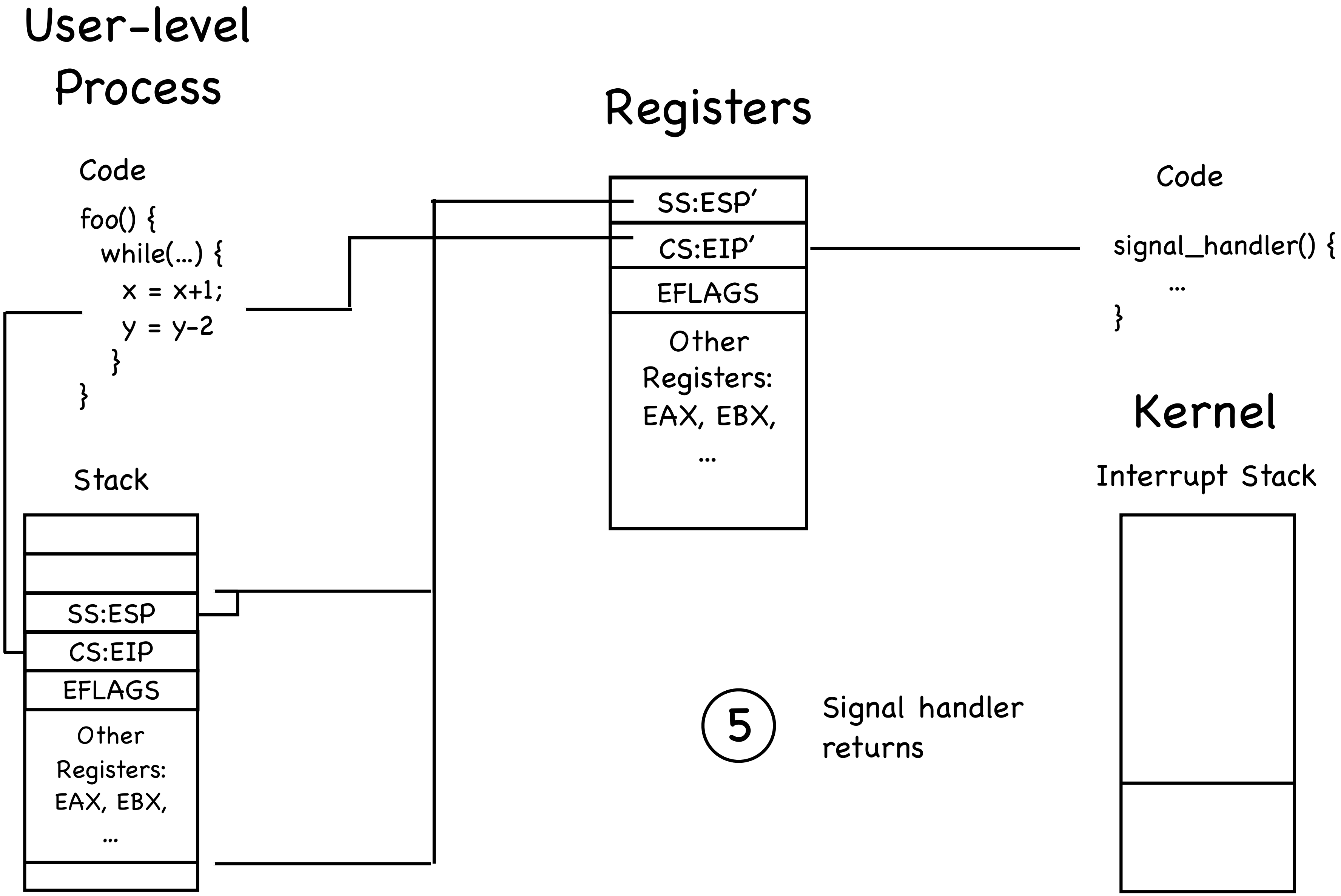
```
signal_handler() {
    ...
}
```

Kernel

Interrupt Stack

④ Kernel exits; Interrupt stack copied back into registers

Unix signals



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

- SS:ESP
- CS:EIP
- EFLAGS
- Other Registers: EAX, EBX, ...

Registers

- SS:ESP’
- CS:EIP’
- EFLAGS
- Other Registers: EAX, EBX, ...

Code

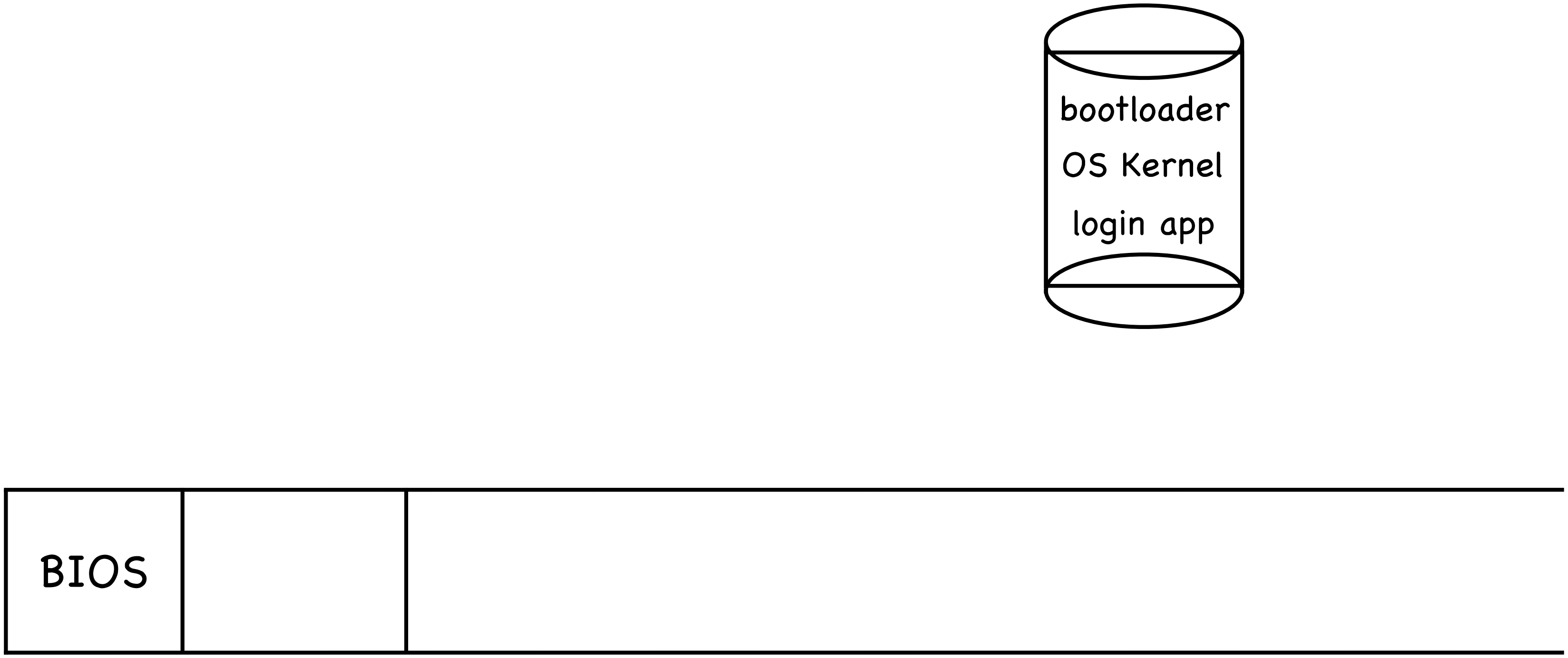
```
signal_handler() {
    ...
}
```

Kernel

Interrupt Stack

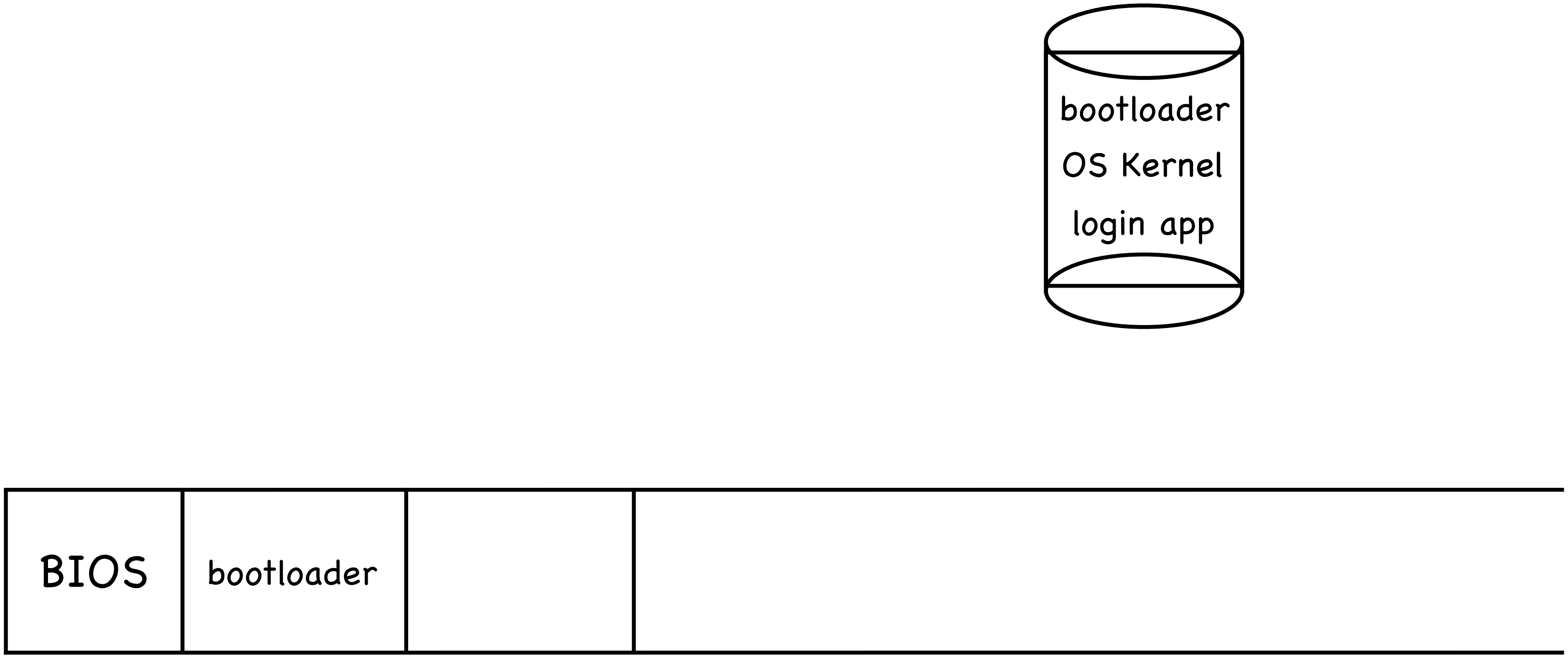
⑤ Signal handler returns

Booting an OS Kernel



- Basic Input/Output System
- In ROM, includes the first instructions fetched and executed
- BIOS copies bootloader, using a cryptographic signature to make sure it has not been tampered with

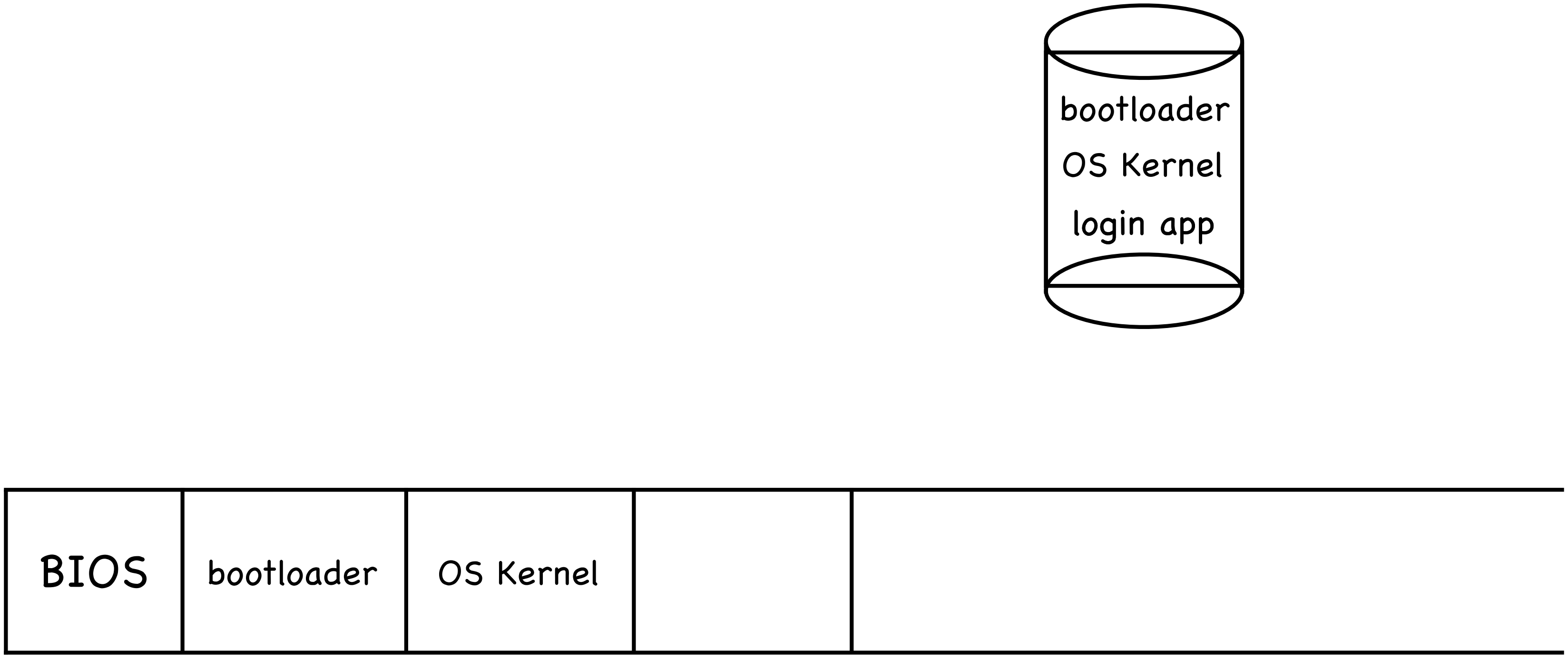
Booting an OS Kernel


bootloader
OS Kernel
login app

BIOS
bootloader

- Bootloader copies OS kernel, checking its cryptographic signature

Booting an OS Kernel



bootloader
OS Kernel
login app

BIOS
bootloader
OS Kernel

- Kernel initializes its data structures
- Starts first process by copying it from disk
- Let the dance BEGIN!

Shall we dance?

- All processes are progeny of that first process
- Created with a little help from its friend...

- ...via system calls!

Starting a new process

- A simple recipe:
  - Allocate & initialize PCB
  - Create and initialize a new address space
  - Load program into address space
  - Allocate user-level and kernel-level stacks
  - Initialize hw context to start execution at “start”
  - Copy arguments (if any) to the base of the user-level stack
  - Inform scheduler process new process is ready
  - Transfer control to user-mode

Which API?

- Windows: CreateProcess System Call

```
if (!CreateProcess(
    NULL,           // No module name (use command line)
    argv[1],       // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,             // No creation flags
    NULL,           // Use parent's environment block
    NULL,           // Use parent's starting directory
    &si,           // Pointer to STARTUPINFO structure
    &pi )          // Ptr to PROCESS_INFORMATION structure
)
```

- Everything you might want to control… but wait!
  - CreateProcessAsUser
  - CreateProcessWithLogonW

Which API?

Unix: fork() and exec()

fork()

Creates a complete copy (child) of the invoking process (parent) — but for return value:

child := 0;
parent := child's pid

exec()

Loads executable in memory & starts executing it

- code, stack, heap are overwritten
- the process is now running a different program!

The genius of fork() and exec()

To redirect stdin/stdout:

fork, close/open files, exec

To switch users:

fork, setuid, exec

To start a process with a different current directory:

fork, chdir, exec

You get the idea!

But what about overhead?

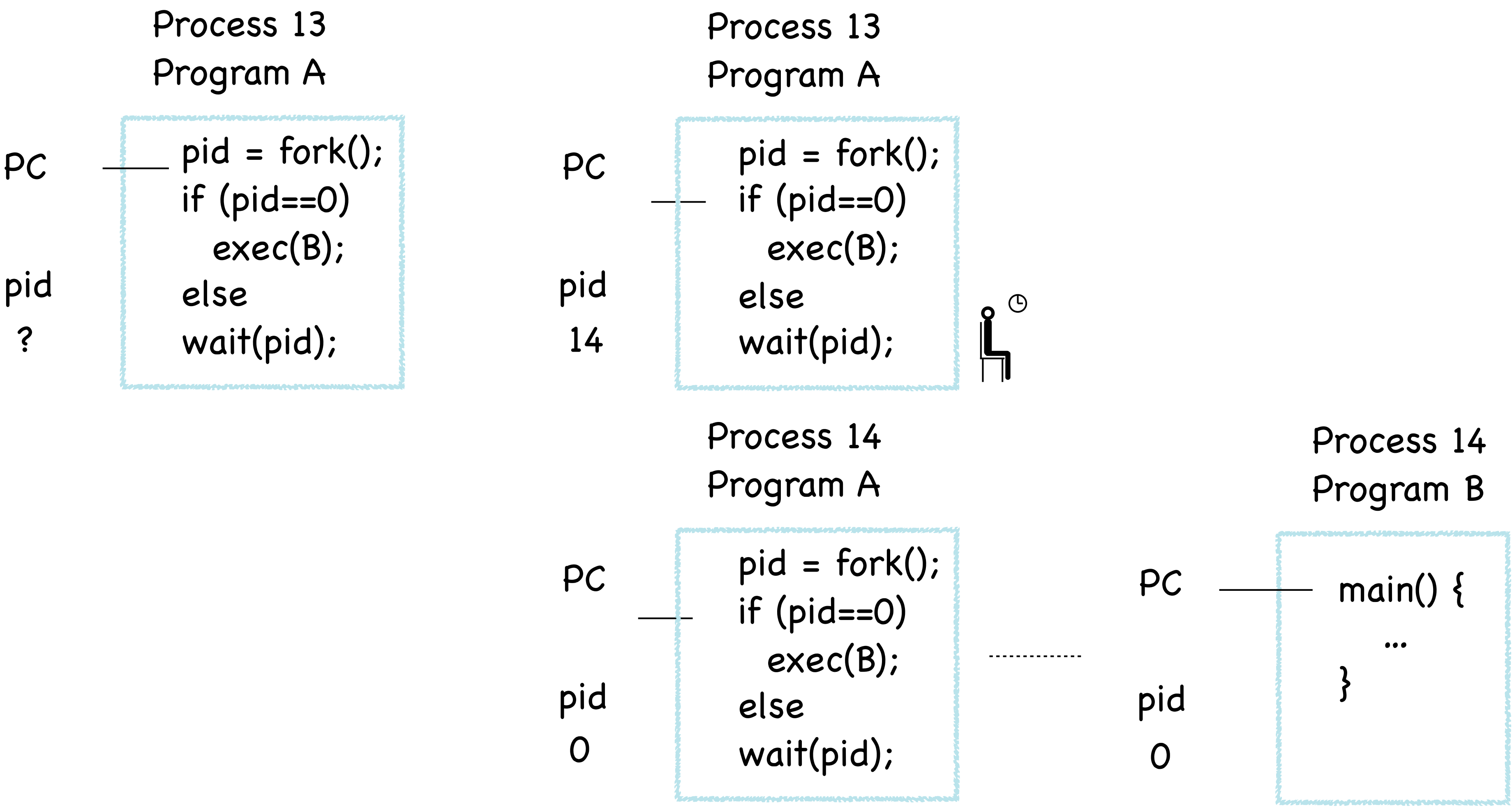
wait() and exit()

- wait() causes parent to wait until child terminates
  - parent gets return value from child
  - if no children alive, wait() return immediately
- exit() is called after program terminates
  - closes open files
  - deallocates memory
  - dellaocates most OS structures
  - checks if parent is alive. If so...

In action

```
/* See Figure 3.5 in textbook*/

#include <stdio.h>
#include <unistd.h>

int main() {

   int child_pid = fork();

   if (child_pid == 0) {          // child process
        printf("I am process #%d\n", getpid());
        return 0;
   } else {                       // parent process
        printf("I am the parent of process #%d\n", child_pid);
        return 0;
   }
}
```

In action



Process 13
Program A

PC

pid
?

```
pid = fork();
if (pid==0)
  exec(B);
else
wait(pid);
```

Process 13
Program A

PC

pid
14

```
pid = fork();
if (pid==0)
  exec(B);
else
wait(pid);
```

Process 14
Program A

PC

pid
0

```
pid = fork();
if (pid==0)
  exec(B);
else
wait(pid);
```

Process 14
Program B

PC

pid
0

```
main() {
    ...
}
```

Shell

- Runs programs on behalf of the user
- Allows programmer to create/manage set of programs

sh	Original Unix shell (Bourne, 1977)
csh	BSD Unix C shell (tcsh enhances it)
bash	“Bourne again” shell

- Every command typed in the shell starts a child process of the shell