

# Kernel and processes

# Process management

- What
  - Can a program create an instance of another?
  - Wait for its completion?
  - Stop/resume another program?
  - Send asynchronous events?
- Where
  - Everything on the kernel?
    - Batch systems
  - Allow user code to manage processes

# User level process management

- Processes can create processes
  - Without kernel recompilation
- Shell command line interpreters
  - User level processes
  - Job control system
    - Creation / killing / suspending resuming
  - Job coordination system
    - Pipe-lining /sequencing

# Compiling a program

- `cc -c file1.c`
  - Creation of a new process
    - Arguments: `-c file1.c`
  - Cc program reads file and produces output
  - Shell waits for it completion
- `cc -c file1.c`
  - Creation of a new process
  - Cc program reads file and produces output
  - Shell waits for it completion
- ...
  - Can be defined in a file
    - Becoming a program

# Process creation

- Kernel operations
  - Allocate process data-structures
  - Allocate memory for process
  - Copy program from disk to allocated memory
  - Allocate Stacks
    - User-level for functions
    - Kernel-level for system-calls / interrupts
  - Process startup

# Process startup

- Kernel
  - Copy arguments for user memory
    - Argc/argv
  - Copy environment
  - Transfer control to user mode
    - POP + IRET
      - After manipulation of kernel stack

# Process termination

- Call of **exit** system call
  - Inserted by compiler
- Executed by kernel
  - Free stacks
  - Free process memory
  - Free kernel datastructures
  - Notify “parent”

# Process creation

- In a OS process creation can follow different policies
  - Execution mode
    - Father and son execute in parallel
    - Father blocks till son terminates
  - Memory management
    - Son gets new memory (data and code)
    - Son gets new data
    - Son gets a shared copy of father memory
    - Son gets a NON SHARED copy of father memory
  - Resource sharing (files, ...)
    - Father and son share all resources
    - Father and son share some resources
      - Open files IPC objects, ...)
    - Father and son do not share any resource

# Windows process creation

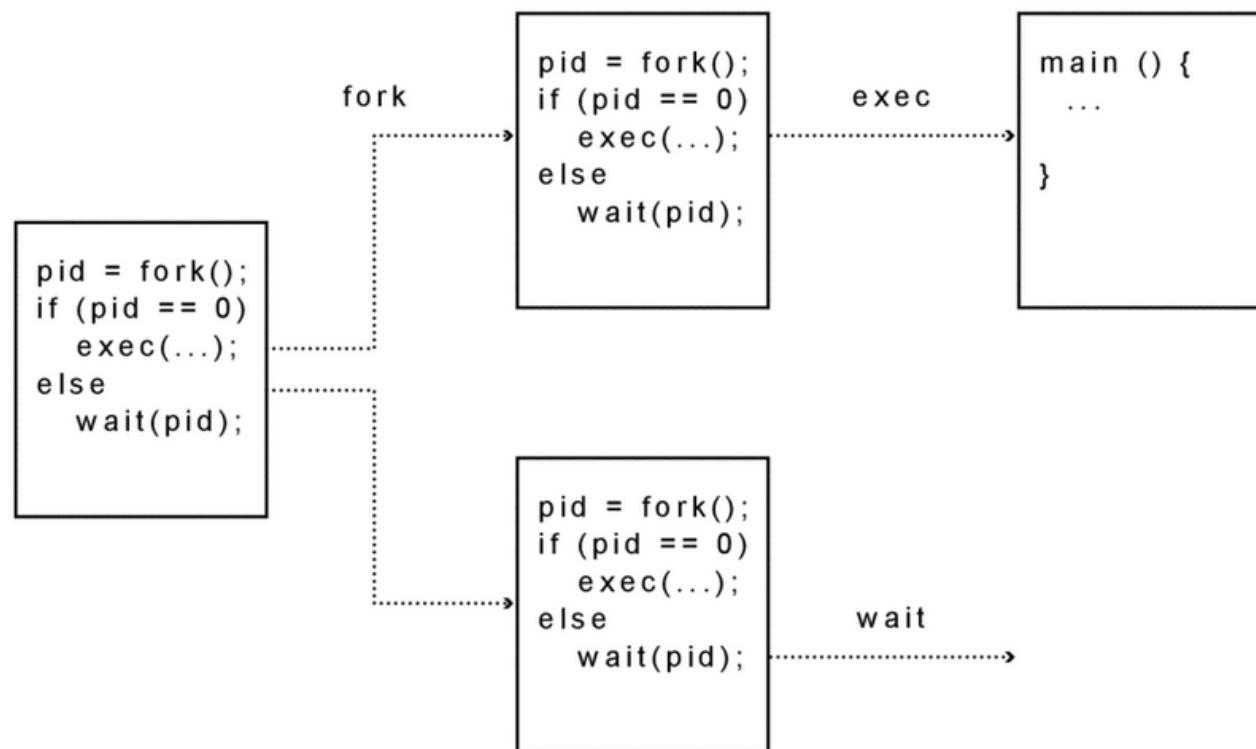
- Boolean Createprocess(char \* prog, char \* arg)
  - Create and initialize Process Control Block
  - Create and initialize address space
  - Load program into address space
  - Copy arguments into process memory
  - Initialize hardware context
  - Inform the scheduler of the new process
- Further security configuration is necessary
  - Limit privileges
  - Change priority

# Windows process creation

- BOOL WINAPI CreateProcess(
  - \_In\_opt\_ LPCTSTR lpApplicationName,
  - \_Inout\_opt\_ LPTSTR lpCommandLine,
  - \_In\_opt\_ LPSECURITY\_ATTRIBUTES lpProcessAttributes,
  - \_In\_opt\_ LPSECURITY\_ATTRIBUTES lpThreadAttributes,
  - \_In\_ BOOL bInheritHandles,
  - \_In\_ DWORD dwCreationFlags,
  - \_In\_opt\_ LPVOID lpEnvironment,
  - \_In\_opt\_ LPCTSTR lpCurrentDirectory,
  - \_In\_ LPSTARTUPINFO lpStartupInfo,
  - \_Out\_ LPPROCESS\_INFORMATION lpProcessInformation
- );

# Unix process creation

- Two steps
  - Copy of the current process
  - Execution of a different program



# UNIX Process Management

- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec – system call to change the program being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send a notification to another process

# How fork returns two values?

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

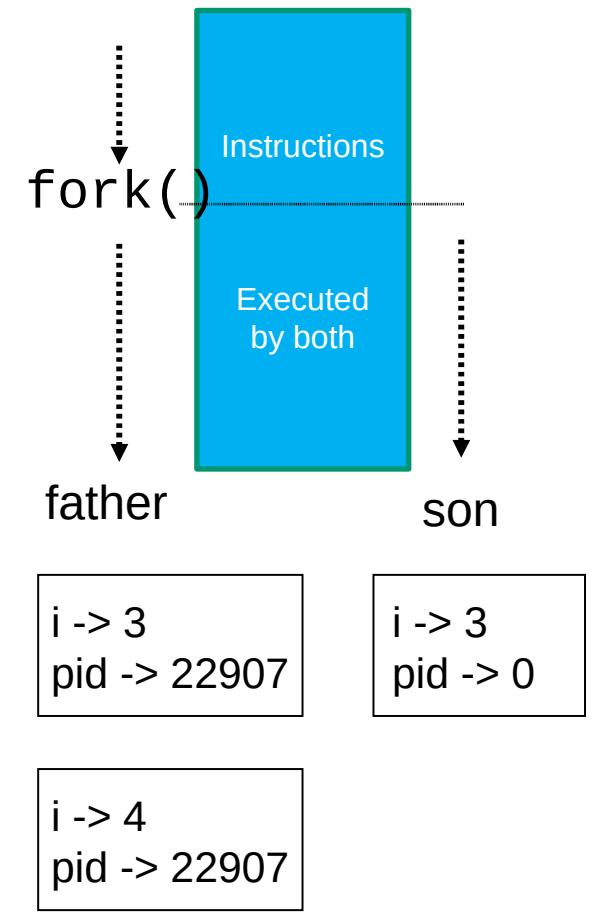
# Process creation

- To start a new process call:
  - **#include <unistd.h>**
  - **pid\_t fork();**
- A new process is created
- The new instruction to be executed is the one following the fork
  - On the father and son !!!!
- All variables are duplicated with the same value
  - After the fork all changes in values are local to each process.
- Fork is a system call that return two different values:
  - In the son return 0
  - In the father return the son PID

# Process creation

- Process creation pattern

- pid\_t pid;
- int i=3;
- pid = fork();
- if (pid==0) {
- /\* process son \*/
- }else {
- /\* process father \*/
- i++;
- }
- /\* both processes \*/



# New processes

- Inherit most of information
  - Memory space (variables allocated memory)
  - Code
  - Opened files (pipes, fifos, sockets)
- How to execute different programs?
  - Replace code after fork

# Implementing UNIX fork

## Steps to implement UNIX fork

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

# Execve

- execve() executes the program pointed to by filename.
  - int execve(const char \*filename, char \*const argv[],  
–                   char \*const envp[]);
    - Filename – program
    - Argv – program arguments
    - Envp – environment variables
    - Empty argv/ envp → array[0] = NULL !!!!
  - Only returns on error :/
- Replaces process image
  - Code, data, files, pipes, socket, mqueus, mmap, timers, signals, ...
  - All previous state is lost

# Implementing UNIX exec

- Steps to implement UNIX fork
  - Load the program into the current address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''

# Questions

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately? Why?

# System

- Some times it is necessary to execute other program
  - Without loosing everything (execve)
- Solution:
  - if(fork() ==0)
  - execve(program, ..., ...);
  - else
  - Wait for children termination();
- Other solution
  - int system(const char \*command);

# System

- Man system
  - `int system(const char *command);`
- Launches a process and waits for its termination
- The system function executes the following system calls:
- Forks a new process
- The child replaces the program (with execve)
- The parent process waits for the child termination
- System returns the child exit code

# Command line

- How processes are started on the command line?
  - The program file is read from the keyboard
  - The command interpreter looks for the program in the PATH
  - A new process is created
  - The new process replaces himself by the selected program
  - The parent process (command line) blocks waiting for the termination of the child
- The wait can be done in the background if
  - The user terminated the command with &
  - The notification of conclusion is printed on the screen

# Process termination

- When a process executes the `exit(int)` function:
  - The calling process is terminated "immediately".
  - Any open file descriptors belonging to the process are closed;
  - any children of the process are inherited by process 1,
  - init, and the process's parent is sent a `SIGCHLD` signal.
  - Man 2 exit / man 3 exit
- What happens to the return code?

# Process termination

- When a process terminates UNIX maintain some information regarding the process
  - Until the parent is notified of its dead
  - During this period the process is considered “zombie”.
- A process return code is to be received by the parent
  - Child → parent communication
  - This reception can be done asynchronously
- The OS should maintain some information regarding
  - the process that terminated
  - Its return code
  - The id of the parent that should receive the return code

# Reception of return code

- If parent need the child return code:

- It must wait for its dead:
  - Call wait/waitpid function

- `pid_t waitpid(pid_t pid, int *status, int options);`

- Process waits for a specific child termination.

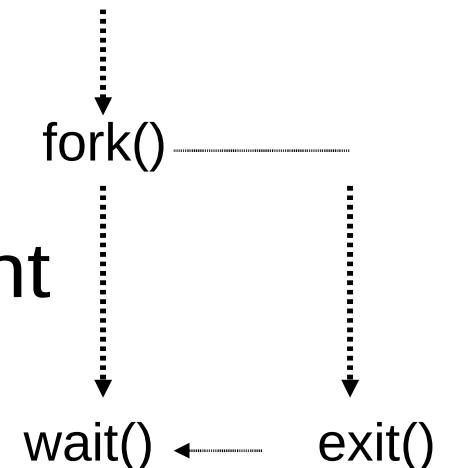
- 1<sup>st</sup> argument – ID of child (-1 any process)

- 2<sup>nd</sup> argument – child status

- 3<sup>rd</sup> argument

- WNOHANG: return immediately if no child has exited

- WUNTRACED: also return if a child has stopped



# Reception of return code

- `pid_t wait(int *);`
  - Process waits for any child
  - 1<sup>st</sup> argument status code
- `wait(&status) <=> waitpid(-1, &status, 0)`
- `wait():`
  - on success, returns the process ID of the terminated child
  - error, -1 is returned.
- `waitpid():`
  - on success, returns the process ID of the child whose state has changed;
  - if WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state,
    - then 0 is returned.
  - On error, -1 is returned.

# Reception of return code

- **WIFEXITED(status)**
  - returns true if the child terminated normally, that is, by calling exit(3) or \_exit(2), or by returning from main().
- **WEXITSTATUS(status)**
  - returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit(3) or \_exit(2) or as the argument for a return statement in main().
  - This macro should be employed only if WIFEXITED returned true.
- **WIFSIGNALED(status)**
  - returns true if the child process was terminated by a signal.
  - **WTERMSIG(status)**
    - returns the number of the signal that caused the child process to terminate.
    - This macro should be employed only if WIFSIGNALED returned true.
- **WIFSTOPPED(status)**
  - returns true if the child process was stopped by delivery of a signal; this is possible only if the call was done using WUNTRACED or when the child is being traced (see ptrace(2)).
  - **WSTOPSIG(status)**
    - returns the number of the signal which caused the child to stop.
    - This macro should be employed only if WIFSTOPPED returned true.

# Killing processes

- A process running in the command line can be killed:
  - Issuing the CTRL-C command
  - Executing the command **kill [-s signal] PID**
    - If PID equals 0 all process from the groups are killed
- CTRL-Z stops (suspends a process)
  - The user can then issue:
    - **bg**: the suspended process resumes in the “background”
    - **fg** : the suspended process resumes in the “foreground”
    - **kill %**: the process is killed

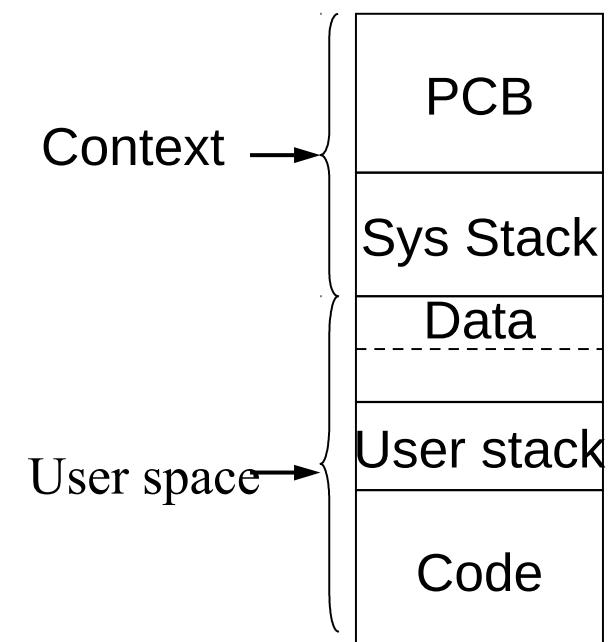
# Killing processes

- `int kill(pid_t pid, int sig)`
  - send signal to a process
  - 1<sup>st</sup> argument – PID of the process
    - 0 → all processes from the groups are signaled
  - 2<sup>nd</sup> argument the signal identifier
    - SIGTERM – same as the CTRL-C
    - SIGKILL – Kills the process
    - SIGTERM - same as the CTRL-Z
    - SIGCONT – same as fg or bg



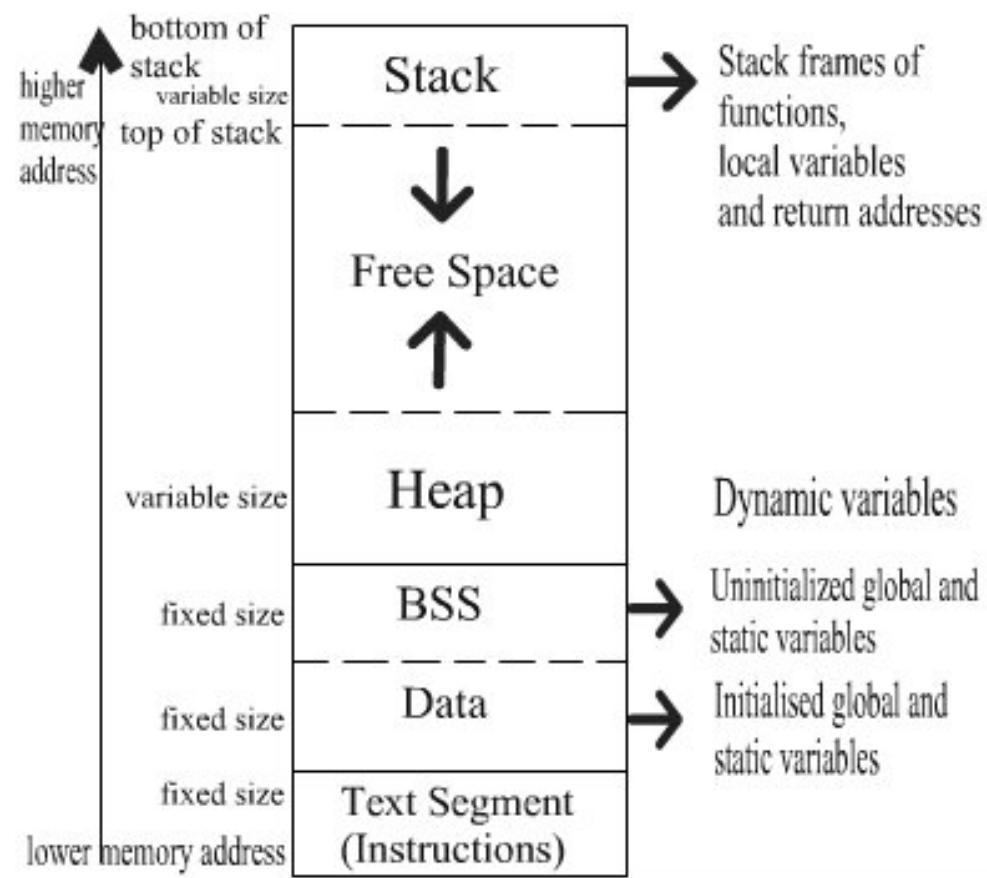
# Processes

- PID / Process ID
  - Number that identifies each process
  - PID is of type `pid_t` (tipicamente `int`).
- The maximum value is usually 32768 (short),
  - `/proc/sys/kernel/pid_max`
- Each process has 2 parts
  - User space
  - Context.



# Process

- User addressable space contains
  - Program data
    - User stack
      - local variables and parameters
    - Heap
      - mallocs
    - Shared memory
      - mmap
    - Global variables.
  - Program code
- size
  - Returns program size
- Top
  - Show process size



# Process context

- Stored in kernel memory
- Kernel stack
  - Needed by system calls
- PCB “Process Control Block”
  - Information needed for process management
    - Multiprocessing
    - Memory/devices management
  - An entry in a kernel table
  - Contains everything a process needs to run.
  - Stores process state between times it is running
  - PCB is reflected in the pseudo-filesystem /proc

# Process context

- PCB - Process Control Block
  - Process identification data
    - Process ID, ID of parent process, user ID of owner
  - Processor state data
    - Program counter
    - Registers
  - Process control data
    - Process state
    - Priority
    - Execution times (since last context switch)
    - memory map information
    - Open I/O devices and files
    - Events waiting for processing

# Management

- Fork
  - creates a new process
- Exec
  - Replaces execution by a new program
- wait / waitpid
  - blocks process until child exits
- exit()
  - terminates program execution
- kill
  - sends signal to process
- getpid
  - returns process identifier

# Processes organization

- In unix the first process to be created is called init/systemd/launchd
  - Its PID is 1
- This process is responsible for the unix initialization
  - It executes the /etc/rc.\* files to start services
- When a user logins
  - A hierarchy of processes is created
  - Last one to be created is bash
    - less /proc/2571/status
    - less /proc/2563/status
    - less /proc/1/status
- In unix all processes are created by another one
  - Not exactly :)
- There is a tree hierarchy that starts in init/systemd/launchd

# Process identification

- All processes are identified by its PID
  - ps / top /proc/...
- To know is identifier a process can call the
  - getpid system call
- To know the identification of its parent
  - getppid
- Man getpid
  - #include <sys/types.h>
  - #include <unistd.h>
  - pid\_t getpid(void);
  - pid\_t getppid(void);

# Signals/ Events

- Process interact with kernel using system calls
- In computer system special events require the kernel to interact with user level processes
- In user these events are called signals
  - Exceptions
    - illegal memory access, division by zero, ...
  - User generated
    - process abortion , Ctrl-C
  - generated by system calls
    - son process termination, timer, kill

# How to handle signals

- Polling
  - All processes poll the kernel to exsiting events
  - Not efficient
  - Not correct
    - programmer shoiuld implement this “system”feature”
- kernel can mange notification (and scheduling)
  - of processes requesting notification

# Upcall: User-level event delivery

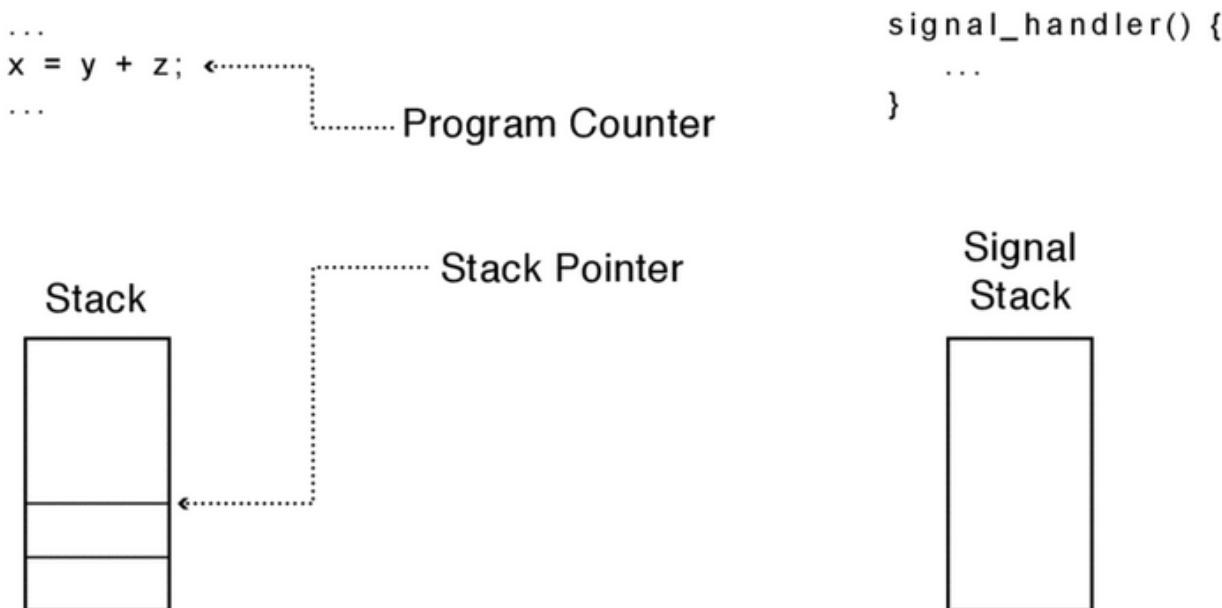
- Notify user process of some event that needs to be handled right away
  - Time expiration
    - Real-time user interface
    - Time-slice for user-level thread manager
  - Interrupt delivery for VM player
  - Asynchronous I/O completion (async/await)
- UNIX signal

# Upcalls vs Interrupts

- Signal handlers = interrupt vector
- Signal stack = interrupt stack
- Automatic save/restore registers = transparent resume
- Signal masking: signals disabled while in signal handler

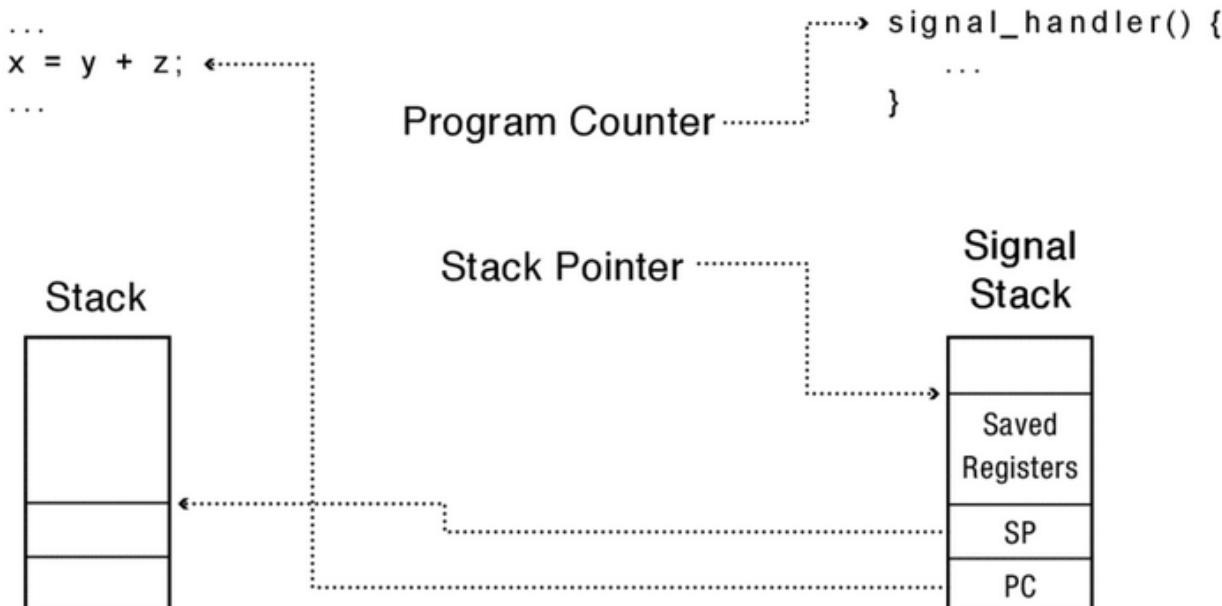
# Upcall: Before

- Process previously register a signal handler
  - will execute in user level
  - kernel will redirect execution to it
- Kernel was notified of the event
  - verifies that process registered and handler



# Upcall: During

- In kernel mode
  - created signal stack
    - correct signal return
  - updated SP and PC
    - for process to execute handler
- In user mode
  - execution of handler
  - Return to resume



# Signal handling in UNIX

- Default handling
  - Kernel handles the signal
  - Process is not notified
    - Most of times it is terminated
- Ignore the signal
- handle the event with a user level handler
- Two signals can not be handled or ignored:
  - SIGKILL
  - SIGSTOP

# Unix signals

- man 7 signal

Signal	Code	Default action	Cause
SIGHUP	1	Terminates process	Terminal line hangup
<b>SIGINT</b>	<b>2</b>	<b>Terminates process</b>	<b>interrupt program</b>
SIGQUIT	3	Terminates process + core dumps	Quit program
<b>SIGKILL</b>	<b>9</b>	<b>Terminates process</b>	<b>Kill program</b>
SIGALRM	14	Terminates process	Timer expired
<b>SIGCHLD</b>	<b>20</b>	<b>Ignored</b>	<b>child status has changed</b>
SIGSTOP	17	Stop process	Stop signal generated from keyboard
SIGCONT	19	Continue if stoped	Continue after stop
SIGUSR1	31	Terminates process + core dumps	User defined signal 1

# Signal handler

- `void (*sighandler_t)(int);`
- Registration of signal handler
  - `#include <signal.h>`
  - `sighandler_t signal(int, sighandler_t);`
  - 1<sup>st</sup> parameter
    - signal code (SIGXXX)
  - 2<sup>nd</sup> parameter
    - pointer to function
    - or SIG\_IGN
    - or SIG\_DFL
- Returns the previous signal handler
- Different OS implement signals in different ways
  - signal handler should be re-assigned

# Signal handler

- `int sigaction(int, const struct sigaction * newhandler, struct sigaction * old);`
  - System call to configure system handling
  - Same behavior in all systems
  - 1<sup>st</sup> parameter : specifies a signal number.
  - 2<sup>nd</sup> parameter: new handler
  - 3<sup>rd</sup> parameter: old handler
- Signal handler configuration - `struct sigaction`
  - `void (*sa_handler)()`
    - signal handler function
  - `void (*sa_sigaction)(int signum, siginfo_t *siginfo, void *uctx);`
    - alternative signal handler function
  - `sigset_t sa_mask`
    - signals that should be blocked during signal handler
  - `int as_flags`
    - signal handling configuration
  - `void (*sa_restorer)();`
-

# Signal creation

- int kill(pid\_t pid, int sig);
  - #include <sys/types.h>
  - #include <signal.h>
  - 1<sup>st</sup> parameter
    - pid of process to receive signal
    - -1 all processes
  - 2<sup>nd</sup> parameter
    - Signal number
  - Returns 0 in case of success
    - -1

# Process management

- kill [signal] PID
- CTRL+C
  - SIGINT (interrompe processo)
- CTRL+Z
  - SIGSTOP (suspende processo)
- bg [%job]
  - processo suspenso retorna no fundo (“background”)
- fg [%job]
  - processo suspenso retorna em primeiro plano
- Nota: por omissão, comandos bg,fg referem último processo suspenso.

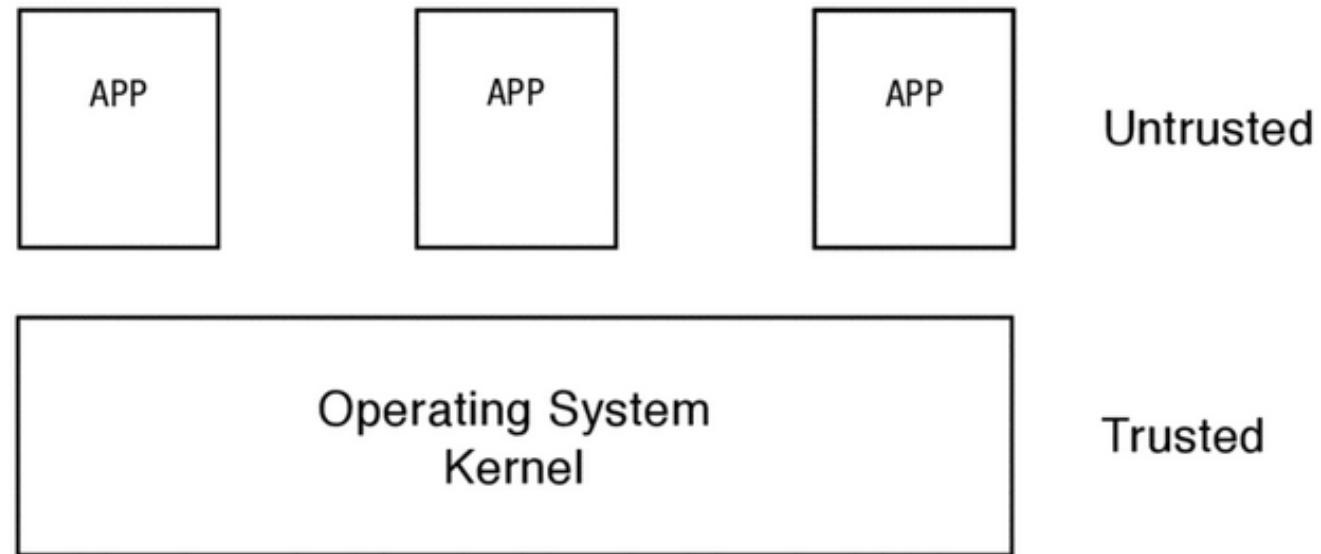


# Protection

- OS central role
  - isolation of misbehaving applications
- Fundamental to other OS goals
  - Reliability
  - Security
  - Privacy
  - fairness
- OS kernel
  - implements protection
  - lowest level SW running on the system
-

# (Un)trusted code

- Applications
  - untrusted

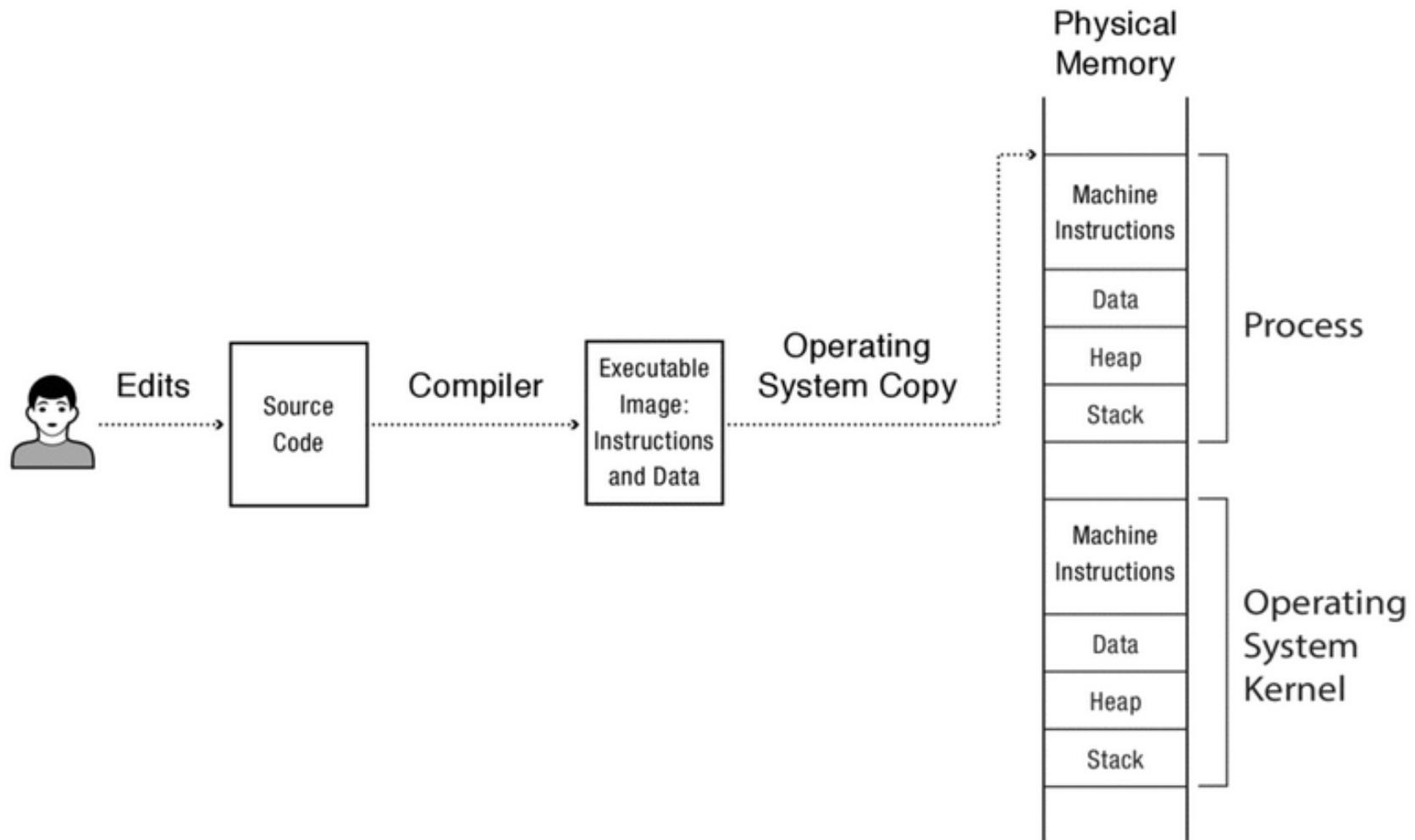


- Process
  - execution of application program with restricted rights
  - Needs permission
    - from OS kernel
    - to access resources (memory from other processes, I/O. ...)
    -

# Challenge: Protection

- How do we execute code with restricted privileges?
  - because the code is buggy
  - because it might be malicious
- Some examples:
  - A script running in a web browser
  - A program you just downloaded off the Internet
  - A program you just wrote that you haven't tested yet

# A Problem



# Process Abstraction

- Process: an *instance* of a program, running with limited rights
  - Thread: a sequence of instructions within a process
    - Potentially many threads per process (for now 1:1)
  - Address space: set of rights of a process
    - Memory that the process can access
    - Other permissions the process has (e.g., which system calls it can make, what files it can access)

# Hardware Support

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - To prevent user code from overwriting the kernel
- Timer
  - To regain control from a user program in a loop
- Safe way to change context
- Safe way to switch from user mode to kernel mode, and vice versa

# How to guarantee protection?

- How can we implement execution with limited privilege?
  - Execute each program instruction in a simulator
  - If the instruction is permitted, do the instruction
  - Otherwise, stop the process
  - Basic model in Javascript and other interpreted languages
- How do we go faster?
  - Run the unprivileged code directly on the CPU
  - control privileged code
  - limit privileged code

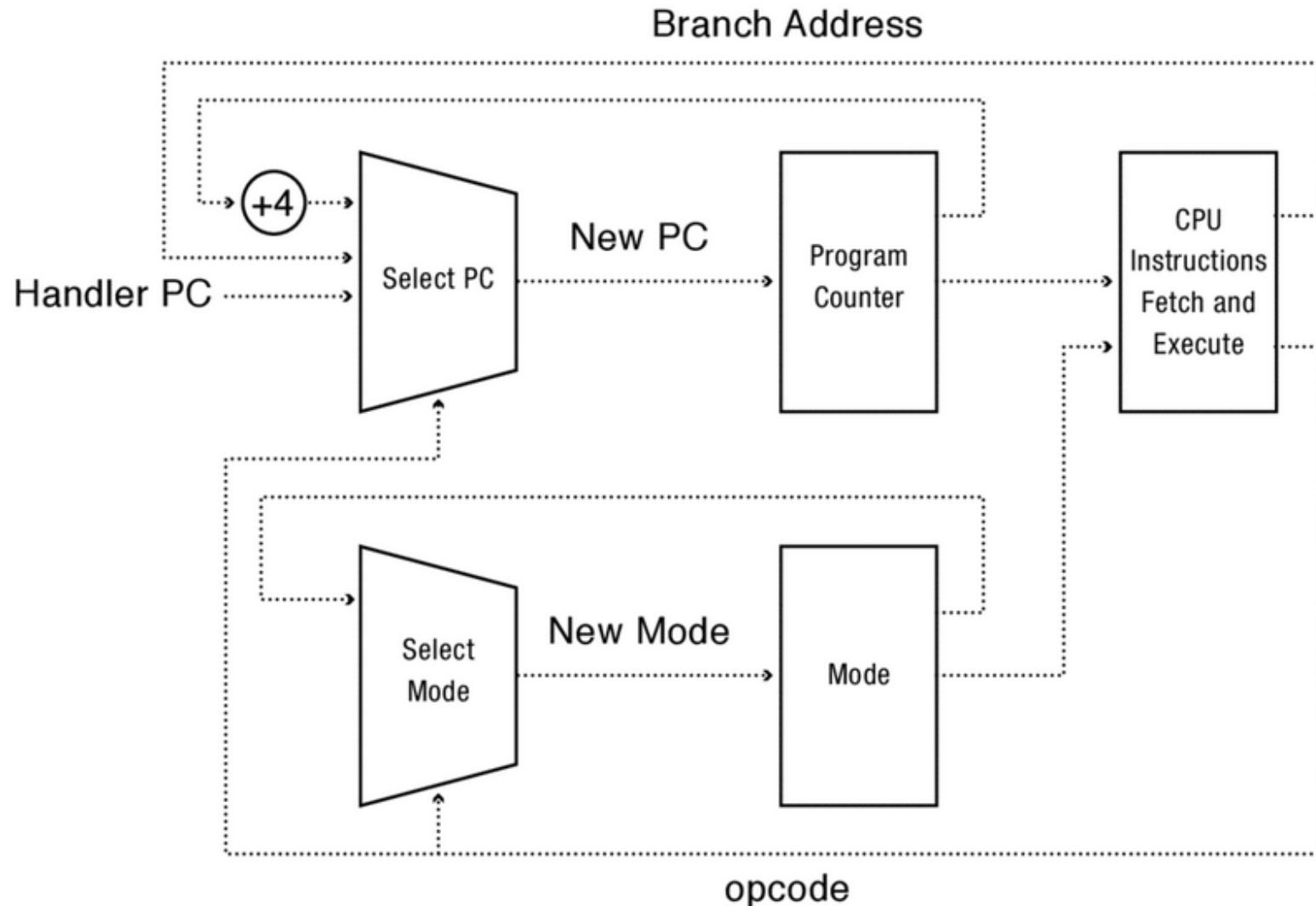
# Hardware Support: Dual-Mode Operation

- Kernel mode
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- User mode
  - Limited privileges
  - Only those granted by the operating system kernel

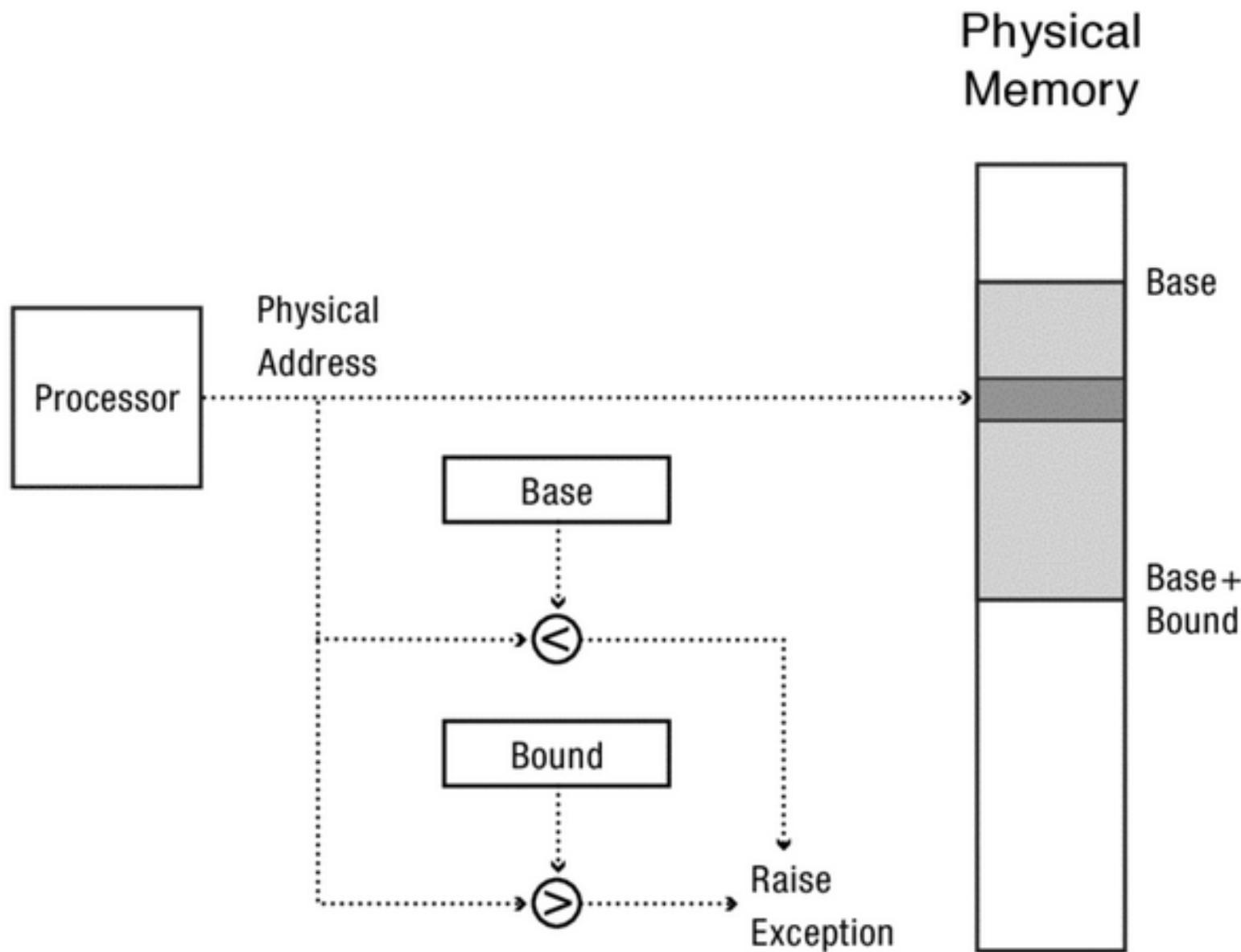
# Hardware Support: Dual-Mode Operation

- On the x86, mode stored in EFLAGS register
- On the MIPS, mode in the status register
- 
- Safe control transfer
  - How do we switch from one mode to the other?

# A CPU with Dual-Mode Operation



# Simple Memory Protection



# Simple Memory Protection

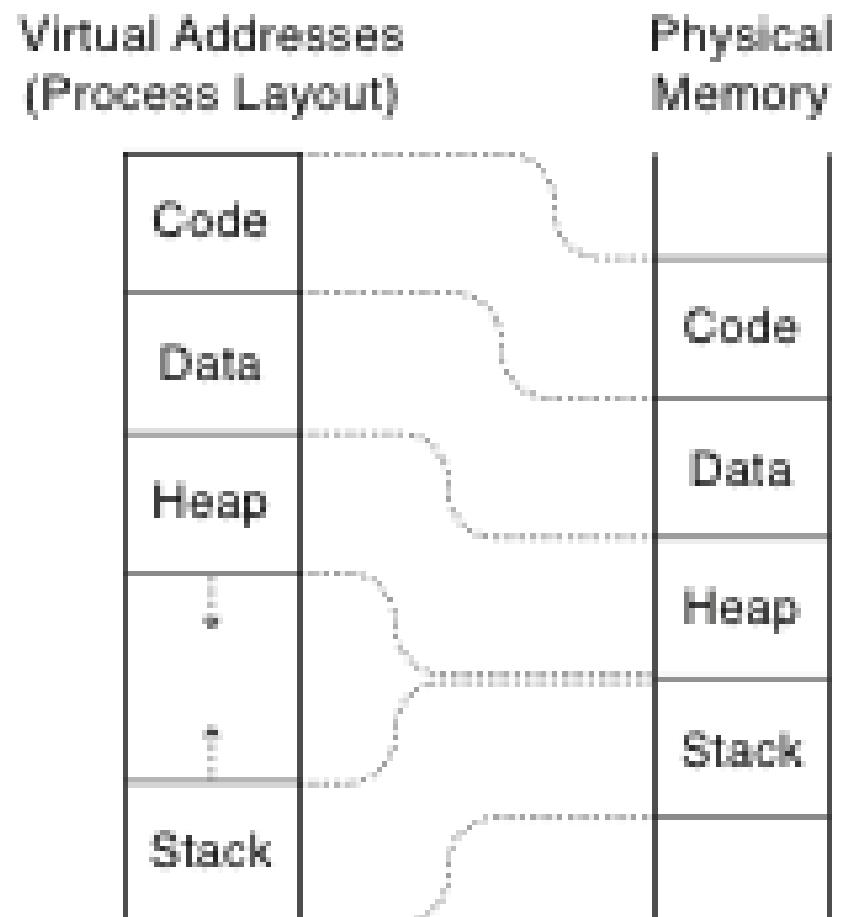
- Kernel
  - executes without base and bound registers
  - full access to all memory
- User level process
  - different base and bound registers
  - disjoint memory areas

# Simple Memory Protection

- Expandable heap and stack
- Memory sharing
- Physical memory addresses
- Memory fragmentation
- 
- Virtual memory

# Virtual Addresses

- Translation done in hardware, using a table
- Table set up by operating system kernel



# Example

```
int staticVar = 0;    // a static variable
main() {
    staticVar += 1;
    sleep(10); // sleep for x seconds
    printf ("static address: %x, value: %d\n", &staticVar,
    staticVar);
}
```

What happens if we run two instances of this program at the same time?

What if we took the address of a procedure local variable in two copies of the same program running at the same time?

# Hardware Timer

- Hardware device that periodically interrupts the processor
  - Returns control to the kernel handler
  - Interrupt frequency set by the kernel
    - Not by user code!
  - Interrupts can be temporarily deferred
    - Not by user code!
    - Interrupt deferral crucial for implementing mutual exclusion
- Scheduling/ multiprocessing
  - Assign PCU to processes in round-robin
  - Switch processes at high frequency

# Mode Switch

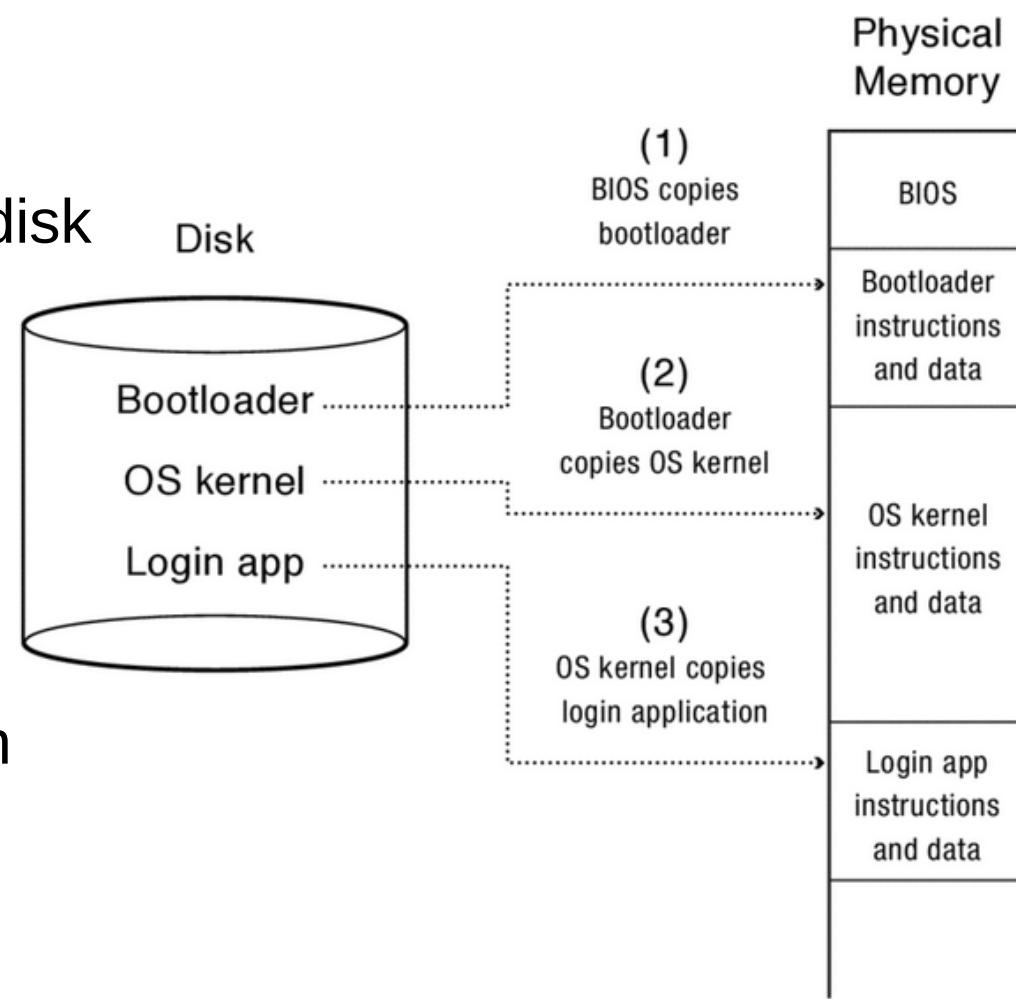
- From user mode to kernel mode
  - Interrupts
    - Triggered by timer and I/O devices
  - Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (aka protected procedure call)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

# Mode Switch

- From kernel mode to user mode
  - After booting
  - Resume after interrupt, process exception, system call
  - New process
  - Switch to a different process
  - User level-upcall

# OS boot

- PC is powered on
- BIOS is executed
  - Bios copies bootloader from disk
- Bootloader executes
  - Os kernel is copied
- OS kernel is executed
  - configures INT table
  - configures memory protection
- Executed services
- Executes login app

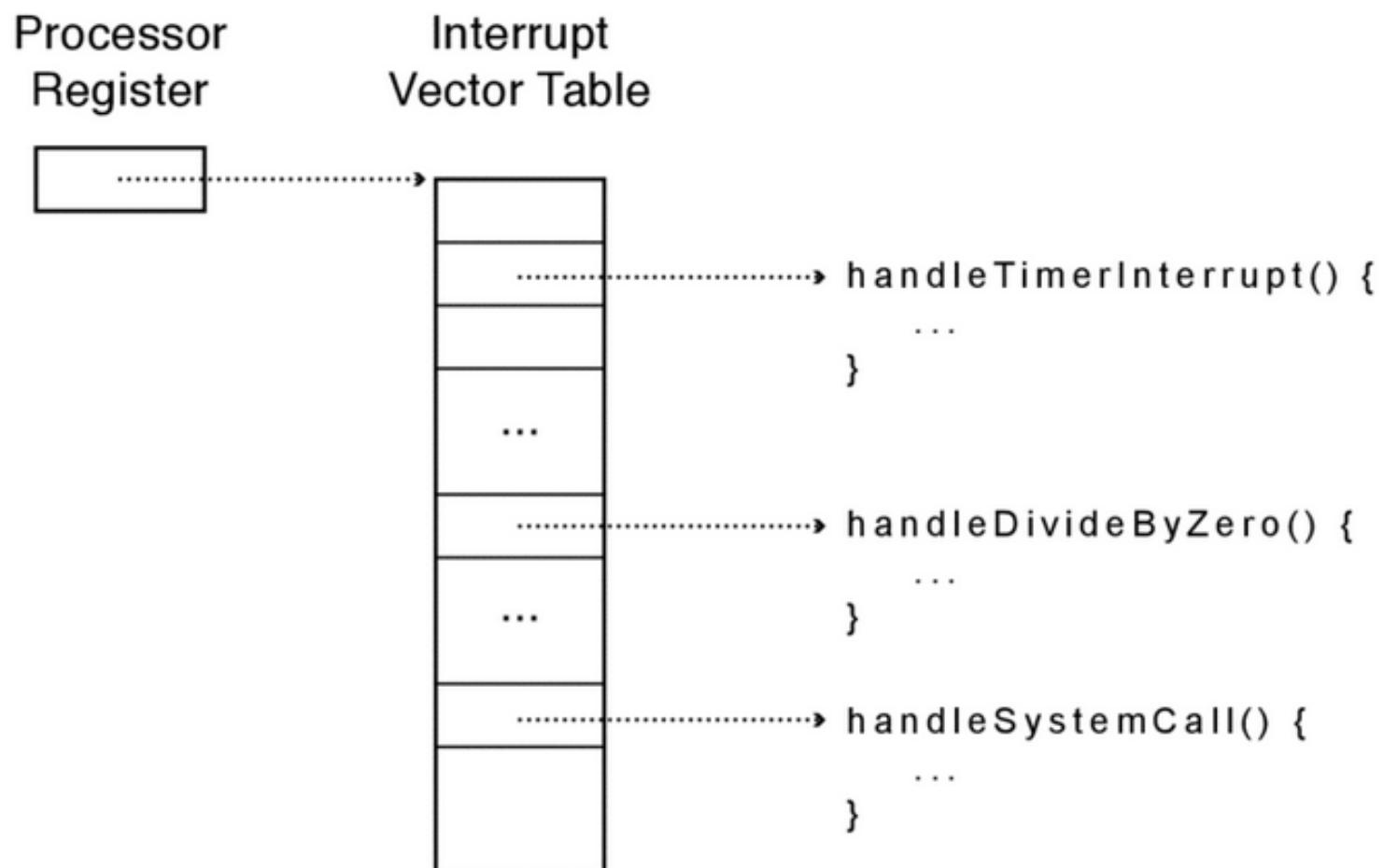


# How interrupt safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Atomic transfer of control
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

# Interrupt Vector

- Table set up by OS kernel; pointers to code to run on different events

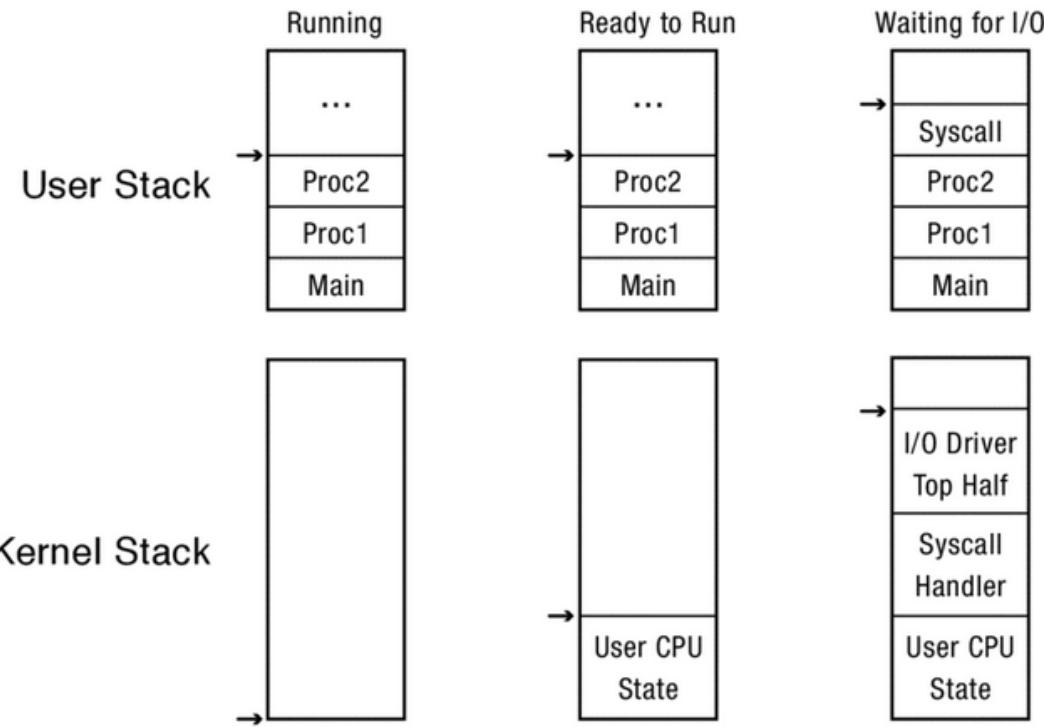


# Interrupt Stack

- Per-processor, located in kernel (not user) memory
  - Usually a process/thread has both: kernel and user stack
- Kernel stack
  - reliability /security
    - insecure code should not modify kernel stack (execution)
- Multiprocessors
  - Multiple stacks <= Multiple syscalls

# Two stacks per process

- Running process - user mode
  - no kernel stack
- Running process - kernel mode
  - kernel stack
- Ready to running process
  - CPU state in kernel stack
- Waiting for I/O
  - kernel execution state
    - stored in stack



# Interrupt Masking

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - On x86
    - CLI: disable interrupts
    - STI: enable interrupts
    - Only applies to the current CPU (on a multicore)
- We'll need this to implement synchronization in chapter 5

# Interrupt Handlers

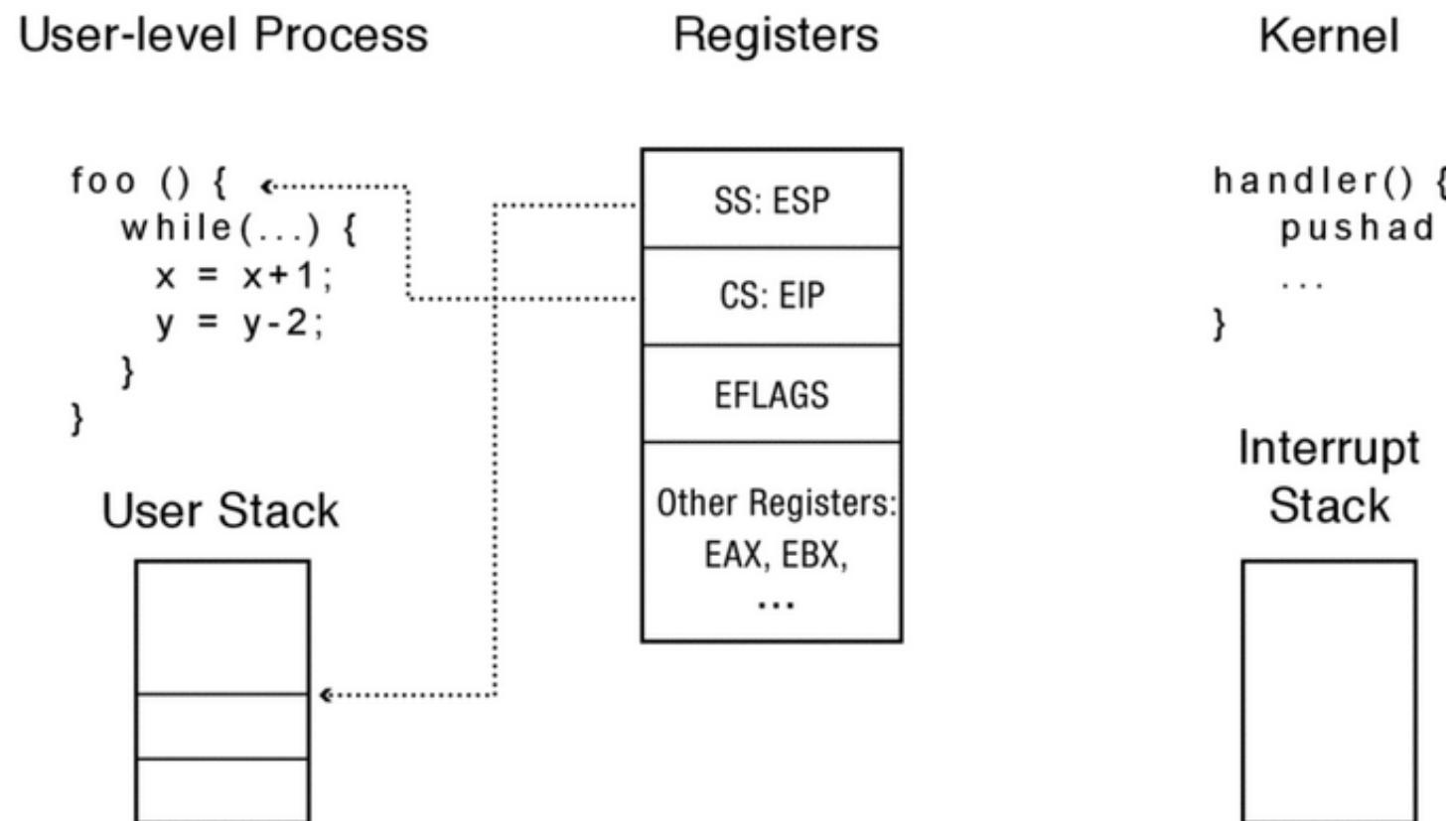
- Non-blocking, run to completion
  - Minimum necessary to allow device to take next interrupt
  - Any waiting must be limited duration
  - Wake up other threads to do any real work
    - Linux: semaphore
- Rest of device driver runs as a kernel thread

# Case Study: x86 Interrupt

- before interrupt handler code execution
  - Save current stack pointer
  - Save current program counter
  - Save current processor status word (condition codes)
  - Switch to kernel stack; put SP, PC, PSW on stack
  - Switch to kernel mode
  - Vector through interrupt table
- Interrupt handler saves registers it might clobber
- Execution of handler code
- Return / Restoring of process state ( IRET X86 instruction)

# Before Interrupt

- pointers: segment base+offest
- Current instructions: CS+EIP
- Current privilege: low-order bit of the ESP



# During Interrupt

1) Mask interrupts

2) Save to temporary HW registers

- SS:ESP
- Execution Flags
- CS:EIP

3) Switch to kernel mode

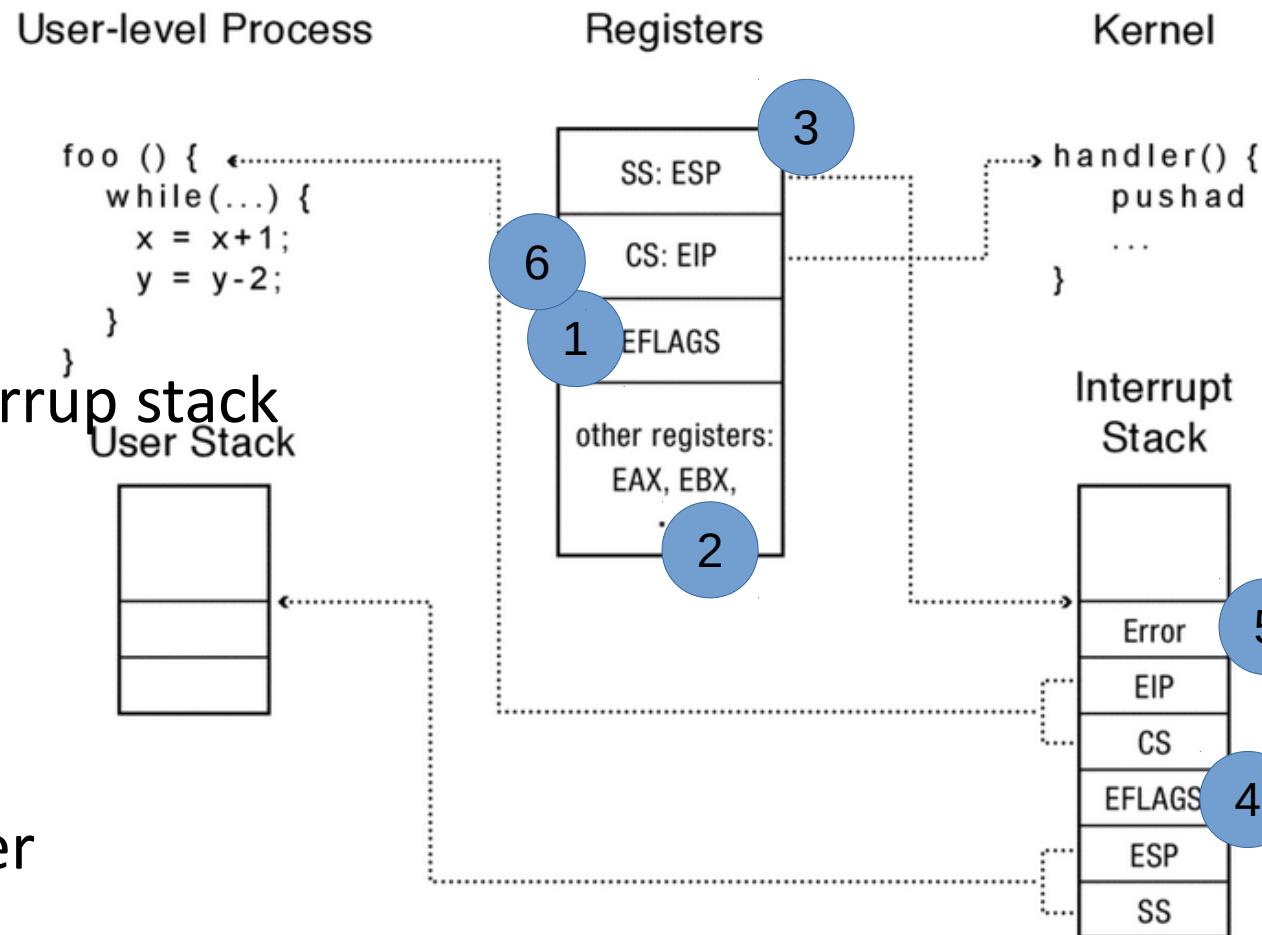
4) Switch onto kernel interrupt stack

5) Push 3 values

- to stack

6) Save error code

7) Invoke interrupt handler



# After Interrupt

- Stack also contains saved registers

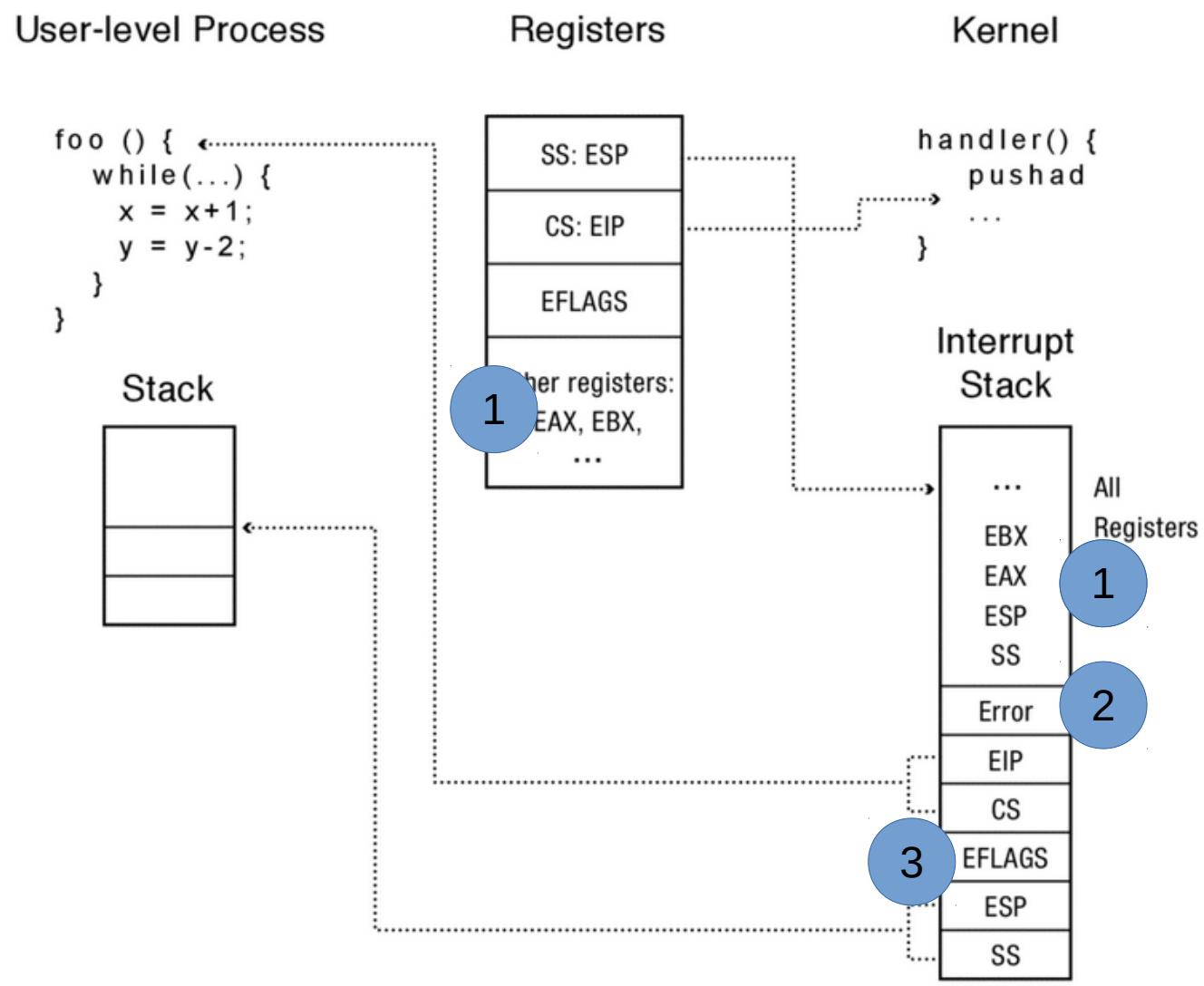
## 1)POPs all registers

- restore CPU state

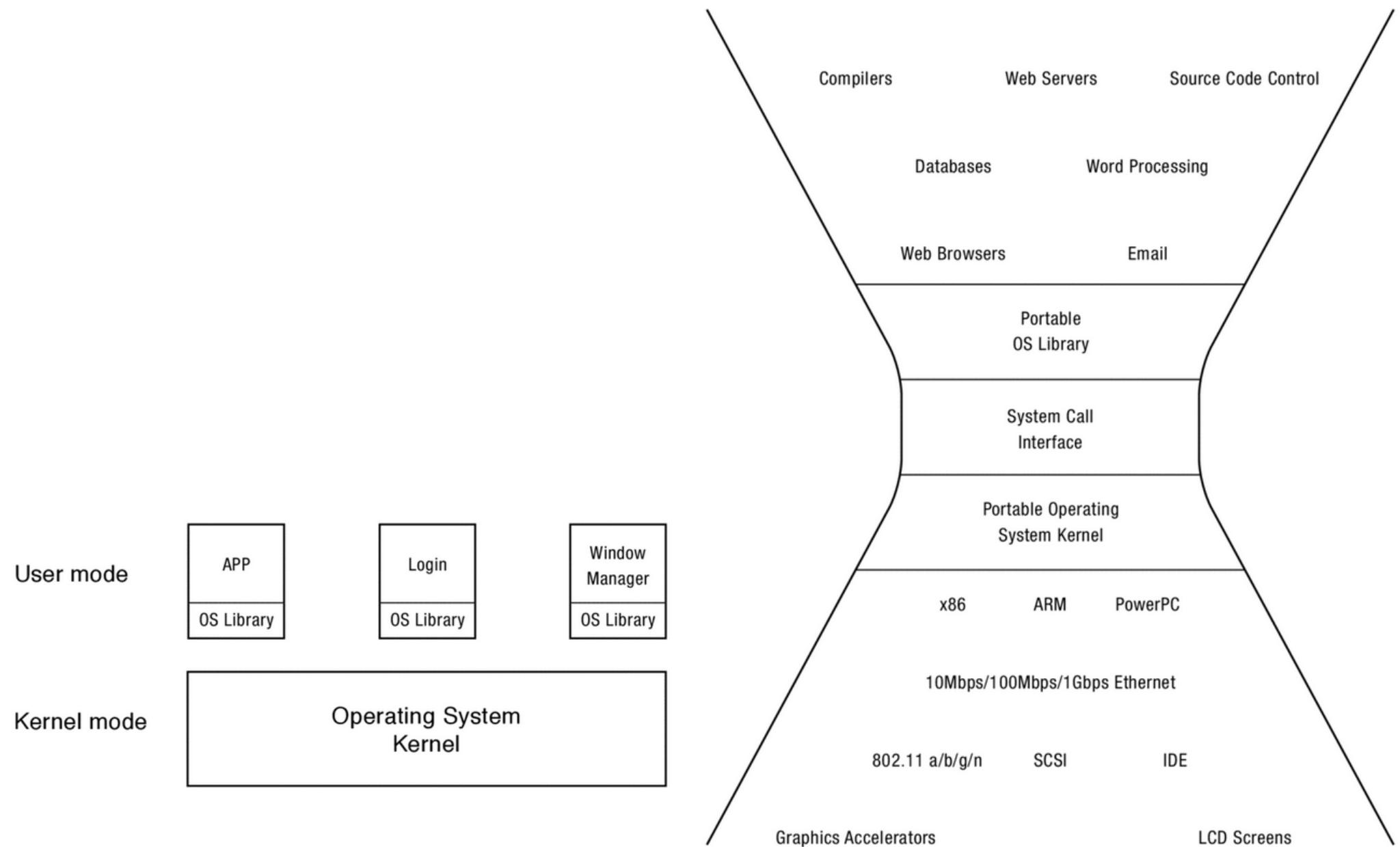
## 2)POP error code

## 3)IRET

- Restores
  - SS:ESP
  - Execution Flags
  - CS:EIP
- returns



# System calls



# System Calls

- Application interact with the OS
  - by system calls
- Some system calls can be invoked by the user from a C program
  - `count=read(fd,buffer,nbytes)`
- Or from the command line
  - `read [-u fd] [-n nbytes] [-a fname] [name1] ...`

# System Calls

- In Linux system calls are grouped in groups:
  - Process control: fork, execute, wait, ...
  - File management: open, read, set, ...
  - Device management: request, read, ...
  - Information maintenance: date, ps , ...
  - Communication: send, ...
- First unix version
  - 60 system calls
- Current Linux version
  - more than 300

# System Calls

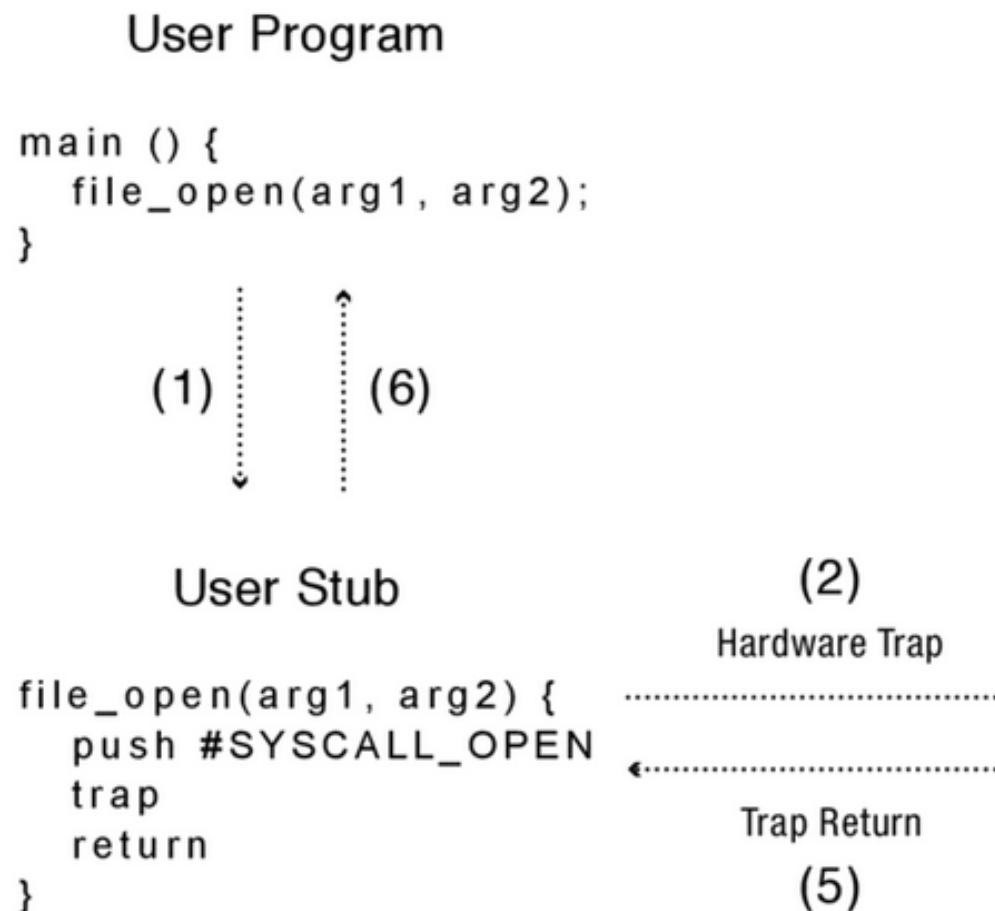
- Microsoft offers the **Windows API** .
- Consists of the following functional categories:
  - Administration and Management
  - Diagnostics
  - Graphics and Multimedia
  - Networking
  - Security
  - **System Services**
  - Windows User Interface
- <https://msdn.microsoft.com/en-us/library/aa383723>

# System call vs Library functions

- System calls are provided by the system and are executed (mostly) in the system kernel.
  - They are entry points into the kernel and are therefore NOT linked into your program.
  - The source code is not portable
  - The API is portable
- Library calls include the ANSI C standard library and are therefore portable.
  - These functions are linked into your program.
- man read
- man fread

# Kernel System Call

- Stub call (regular C)
- Stubb fills “syscall arguments”
- Trap is generated
  - similar in INTR
- Kernel code executed
- handle returns



# Kernel System Call Handler

- Kernel stub

- Locate arguments
  - In registers or on user stack
  - Translate user addresses into kernel addresses
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code

- Executes handler

- Kernel Stub

- Copy results back into user memory
- Translate kernel addresses into user addresses

