INTRODUCTION

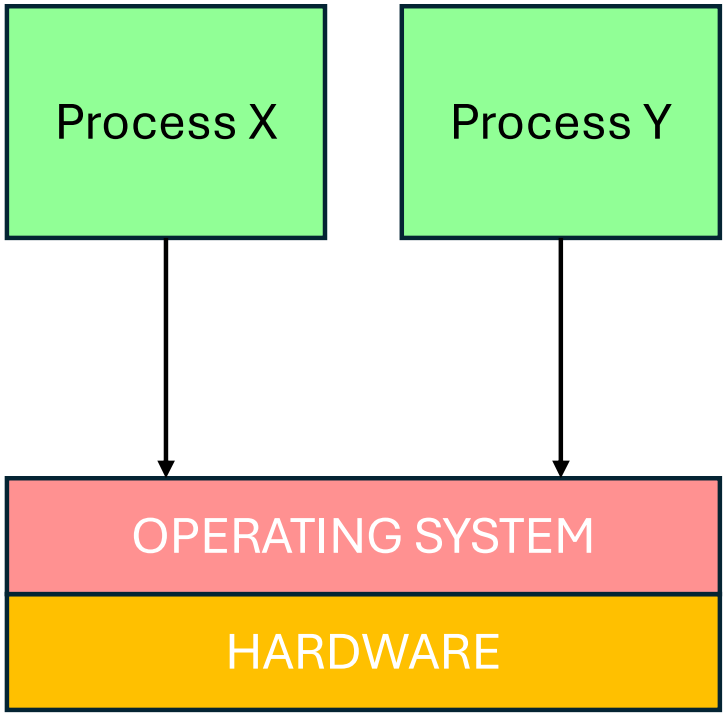
Alex Pajuelo and Juan Jose Costa

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
BarcelonaTech

Operating system

The operating system (OS) is a passive software layer that abstracts and manages the underlying hardware and provides services to execute the user code

Execution stack

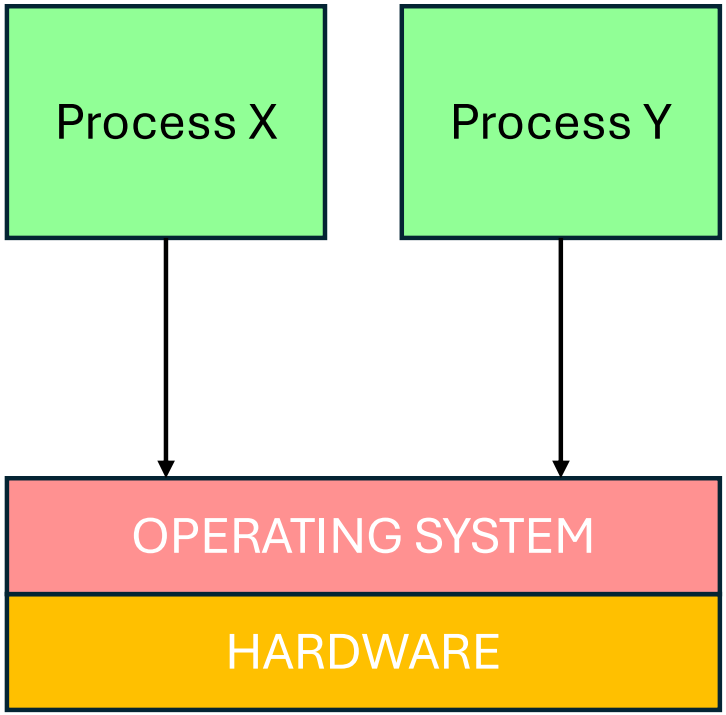


The operating system is responsible of:

- Isolate de hardware
- Manage the hardware
- Abstract the hardware
- Virtualize the hardware to processes
- Security
- Protection

The OS must be the only one to access the hardware. Some mechanism must be provided to prevent the user processes from accessing the hardware.

Execution stack



The processes:

- Cannot access the hardware
- Must invoke the OS for resources

The OS must be the only one to access the hardware. Some mechanism must be provided to prevent the user processes from accessing the hardware.

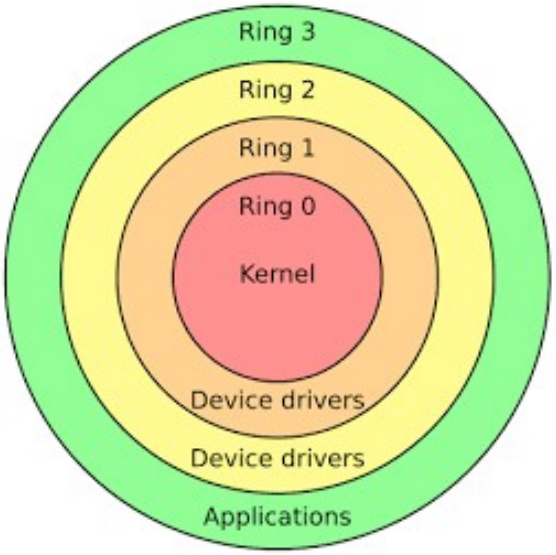
How a device is accessed

- To access a device, some special assembler instructions are provided:
  - inb
  - outb
- To prevent the user processes from accessing a device, those assembler instructions cannot be executed by user code
- The processor must check if those instructions are executed by user code and notify the OS in that case
- Solution: Privileged levels (PL)

Architecture Fundamentals

Privileged levels

- An x86 processor defines 4 levels of privilege



MIN

MAX

RING 3: Minimum level of privileges. Access to a limited amount of memory and cannot access hardware.

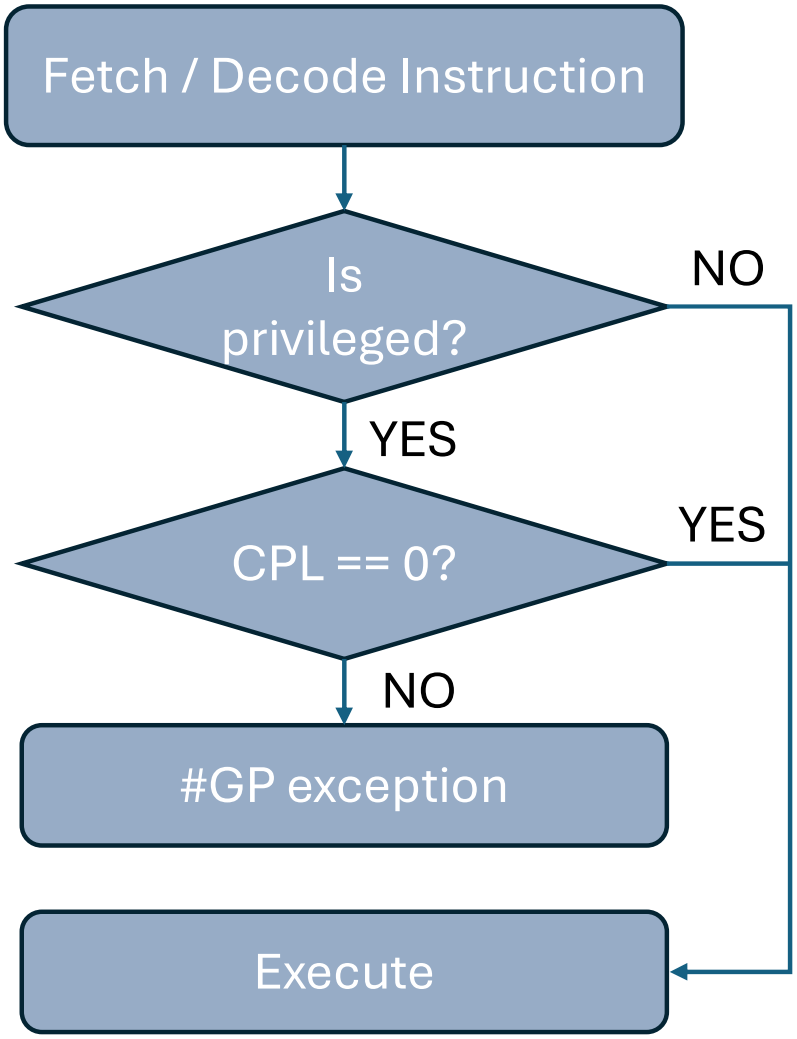
RING 2: Unused

RING 1: Unused

RING 0 : Maximum level of privileges. Unlimited access to hardware and memory

How privileged levels work

- The processor provides a CPL (Current Privileged Level) register (bits 1-0 of the CS register) to indicate in which level instructions are executing
- Whenever a new instruction is fetched and decoded, the processor determines which is the Privileged Level (PL) this instruction must be executed
  - For example: MOV can be executed in any PL, but OUTB can only be executed with CPL == 0
- If the instruction is a privileged instruction and the CPL == 0, the instruction can be executed
- If the instruction is a privileged instruction but the CPL is not 0, a #GP exception is raised



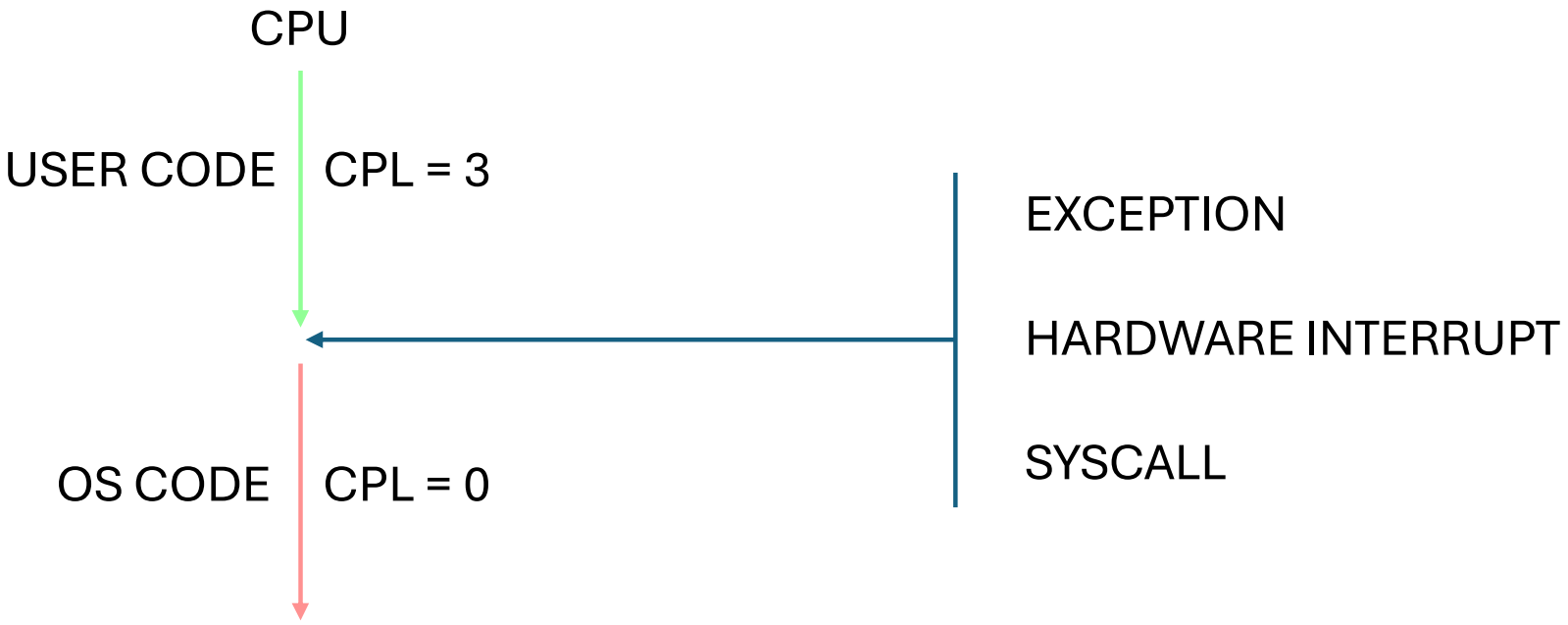
Why this works

- Since user processes are executed in Ring 3 (PL=3 or user mode), they cannot execute assembler instructions to access the hardware

- Since the OS is executed in Ring 0 (PL=0 or OS mode), it can execute privileged instructions to access the hardware

Changing the privilege level

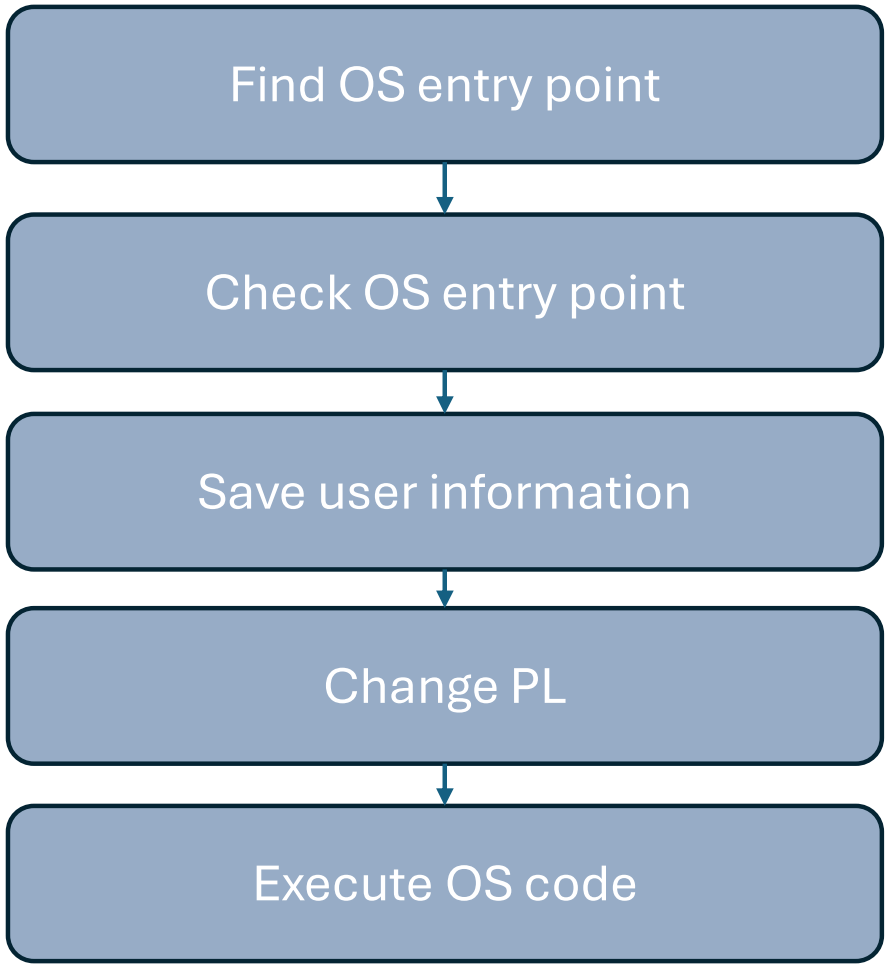
- The OS must be executed from time to time, but the CPL must be 0
- Some mechanism must be provided by the processor to change the privileged level from PL = 3 to PL = 0



Mechanism to change PL

- Exceptions: raised by the CPU when an abnormal execution of the user code is detected: page fault, protections, division by 0, ...
- Hardware interrupts: triggered by some device to notify to the CPU that some management action must be performed or that some event has happened: timer interrupt, keyboard interrupt, harddrive interrupt, ...
- Software interrupts: the user code invokes directly the operating system: typically, syscalls

Steps to change PL



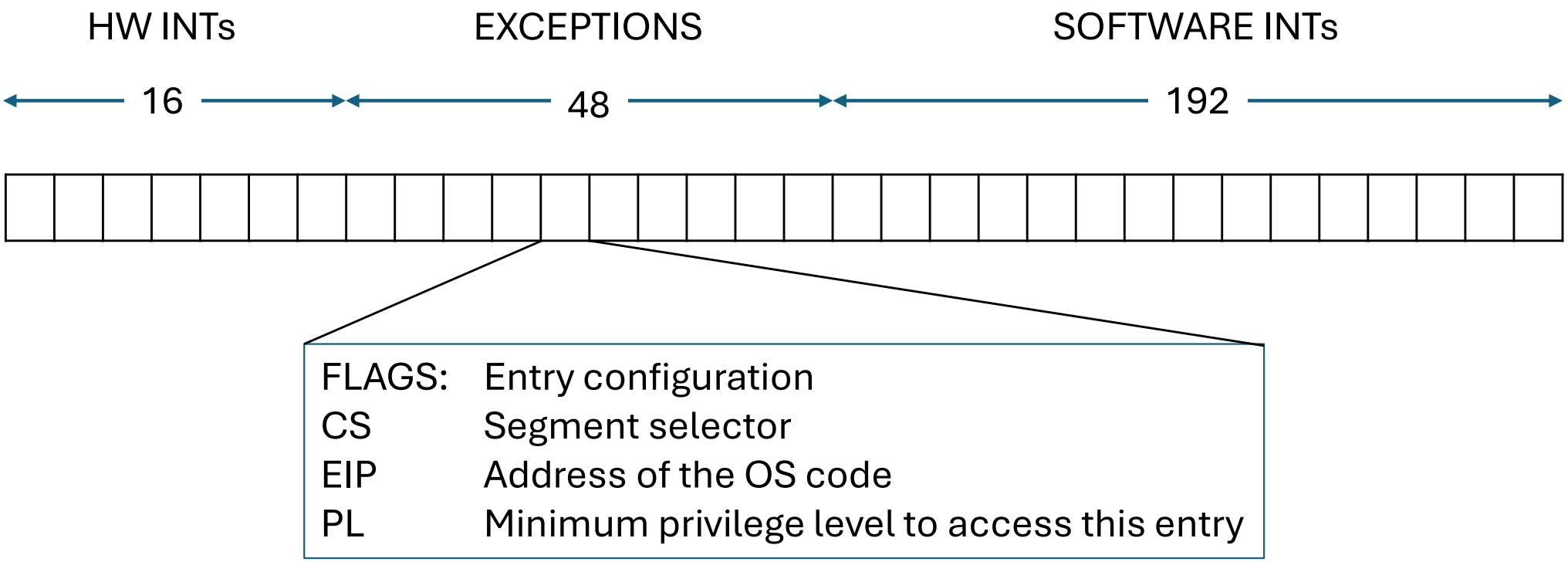
- All steps are performed automatically by the CPU
- No OS intervention is required
- Hardware structures are directly accessed by the CPU
- OS is responsible to fill data in the hardware structures (boot time)

Find OS entry point

- The processor access the IDT (Interrupt Descriptor Table) to obtain the address of the OS code that must be executed
- The IDT is a hardware structure that contains, for every possible hardware interrupt, exception o syscall, the address of the OS service routine to be executed
- Depending on the nature of the event, a particular part of the IDT is accessed

IDT (Interrupt Descriptor Table)

- Array of 256 entries, allocated in memory



- PL: to prevent the user from executing OS code related to HW Ints or exceptions
- Initialized in Boot Time by the OS

Check OS entry point

- After accessing the IDT, the processor accesses the GDT (Global Descriptor Table) to check the correctness of the field CS and EIP
- This is done to ensure than nobody has modified the IDT after boot (security)
- The GDT is way too complex and out of the scope of the course

Save user information

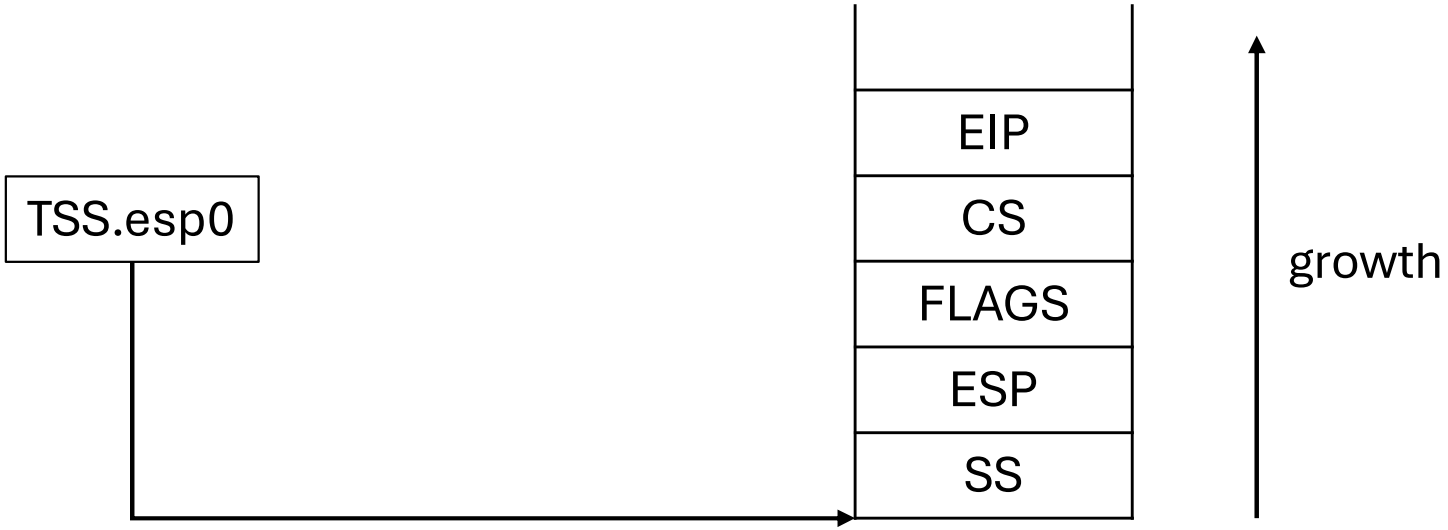
- When everything looks OK, the processor saves the required information to return to user code
- This information is:
  - CS: user code segment
  - EIP: address of return to user code
  - FLAGS: processor flags in user mode
  - SS: user stack segment
  - ESP: address of the top of the user code
- This information is called hardware context and is the minimum information, from the processor point of view, to keep on executing the user code.
- This information is pushed in the System stack

System stack

- When some code is executed, a stack is needed to save the information of local variables, parameter passing, ...
- When executing in user mode, a user stack is used (this is the one that you know)
- When executing in system mode, a system stack is used (and cannot be the same as the user stack)
- The system stack is used due to security reasons: never rely on any data coming from the user
- There is one system stack per thread
- Later we will see where the system stack is allocated

System stack (and II)

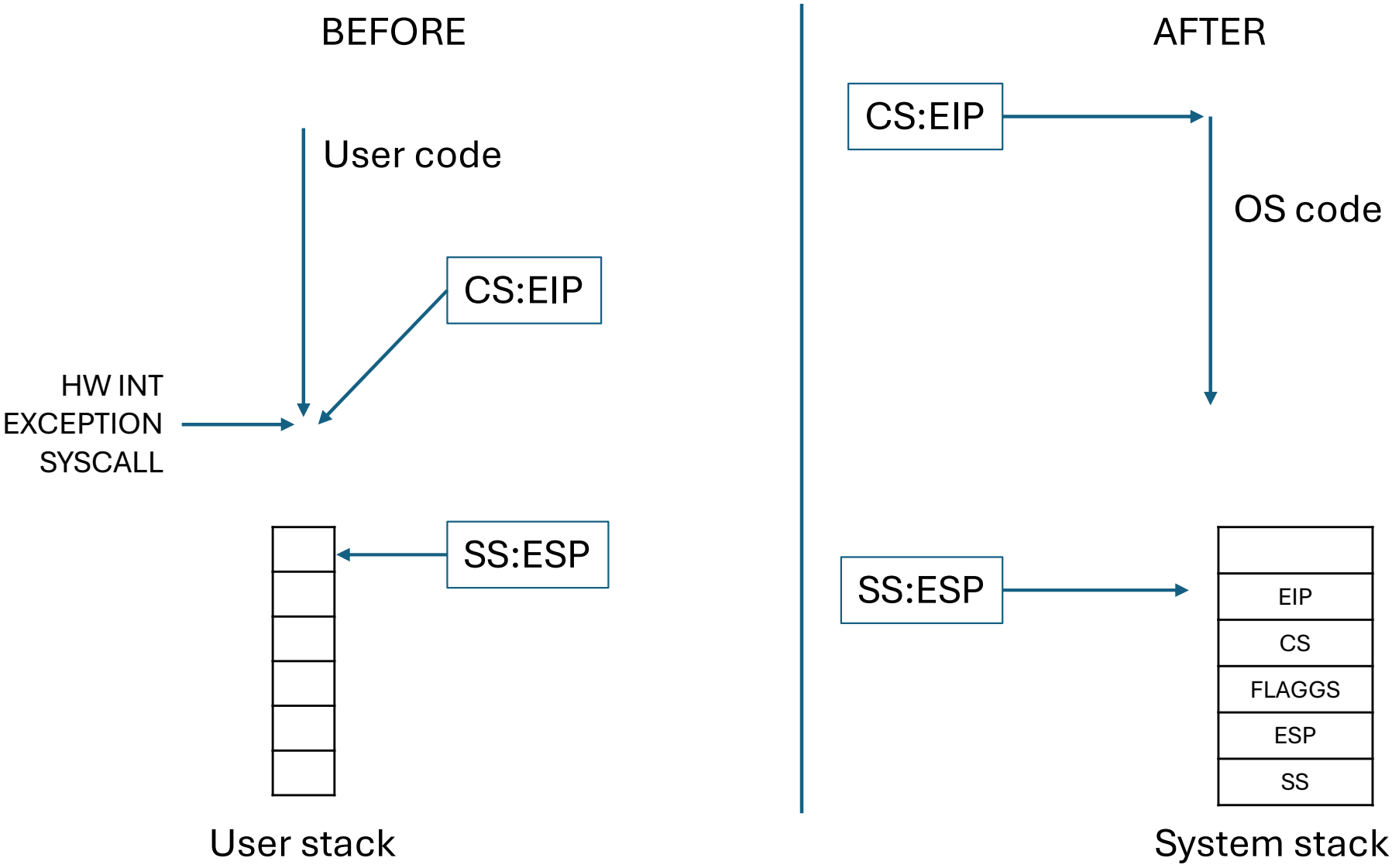
- The address of the system stack is stored in the esp0 field of the TSS (Task State Segment) hardware structure.
  - The TSS holds information of the thread that is currently executing
- The processor, through the TSS.esp0 field, obtains the address of the system stack of the current thread and pushes the hardware context



Change PL

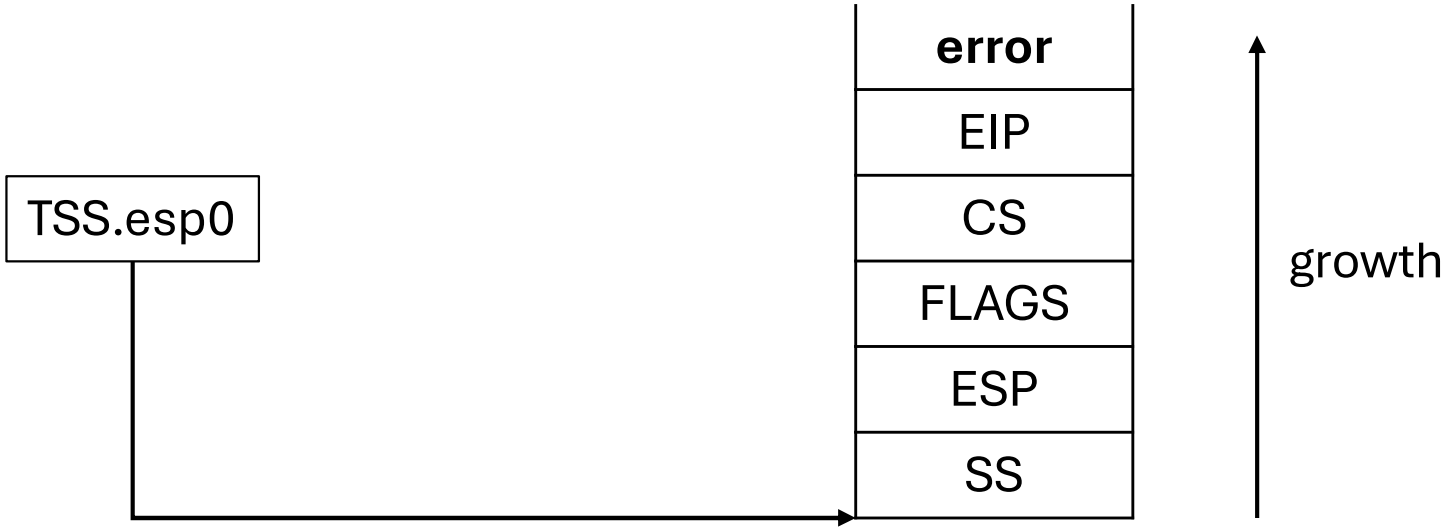
- Once the processor has pushed in the system stack the hardware context, it's time to set up everything to execute the OS code
- Now:
  - CS:EIP points to the next instruction in user code
  - SS:ESP points to the user stack
- What must be done:
  - CS:EIP must point to the first OS instruction
  - SS:ESP must point to the top of the system stack

Privilege level change



Exceptions

- For some exceptions, extra information is pushed in the system stack



Software interrupts (syscalls)

- To raise a software interrupt (syscall) some particular assembler instructions are provided:
  - INT
  - SYSENTER
  - SYSCALL (out of scope)
- They all change the PL but have different characteristics

INT assembler instruction

- First provided instruction in x86
- It has a parameter:
  - INT x
- The parameter indicates which IDT entry must be used to obtain the address of the OS code
- The behavior is similar to the one described before in this presentation (IDT, GDT, TSS, push hardware context in the system stack, change SS:ESP and CS:EIP)

SYSENTER assembler instruction

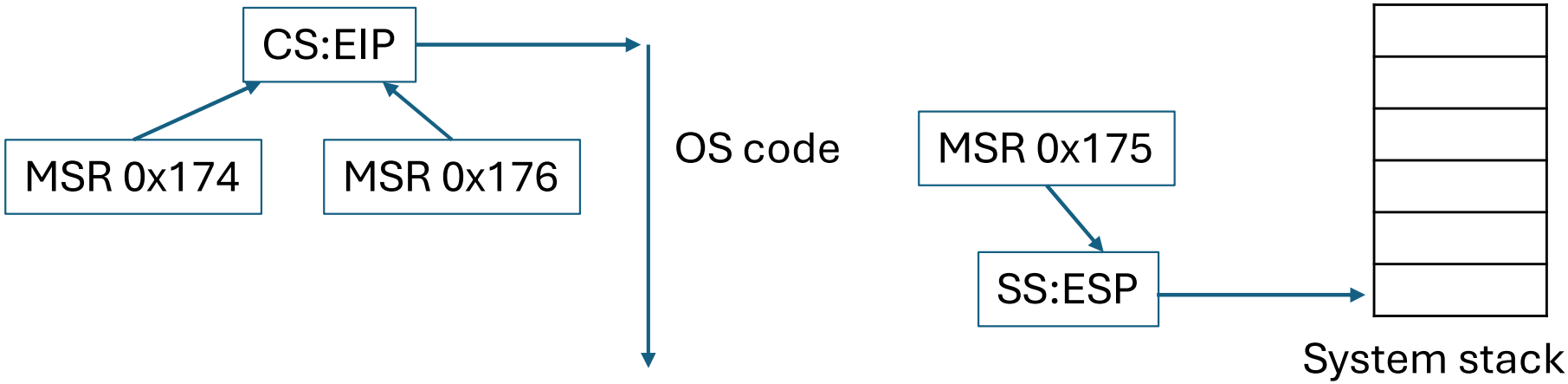
- INT is slow since many memory accesses are performed
- Sysenter goal: change the PL faster
- For this reason, changing the PL with sysenter is called a Fast system call
- Sysenter obtains the data from MSRs (Model Specific Registers)

MSR	VALUE
0x174	OS CS (Code segment descriptor)
0x175	System stack address
0x176	OS entry point address

- MSRs can only be modified in Ring 0

Sysenter (and II)

- Sysenter DOES NOT push the hardware context in the stack. This is a responsibility of the OS.



Compiler Fundamentals

Compiler

- A compiler translate the high-level code to low-level code (typically assembler)
- Implementation of a calling convention
  - Not all architectures and OS follow the same convention
- We will use x86 calling convention for Linux-based OSes

Calling convention

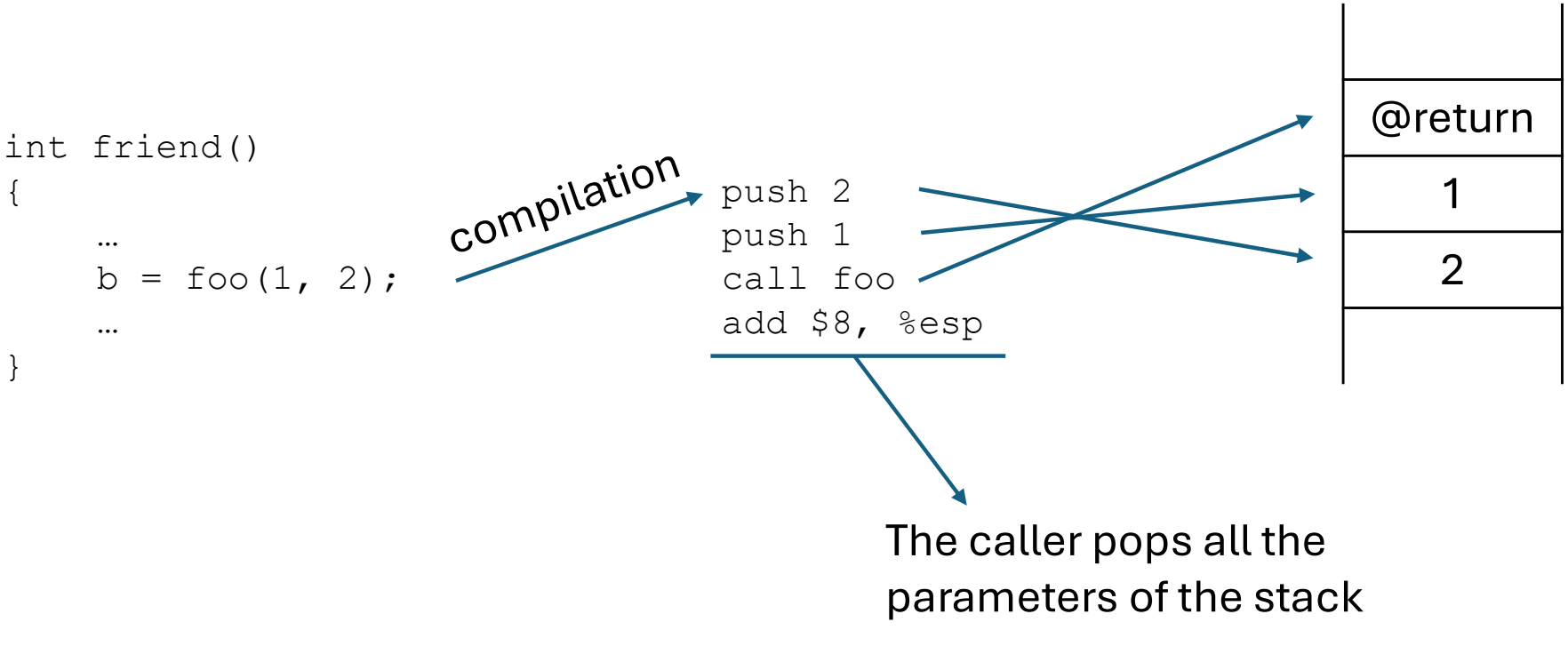
- Defines how to call a function
- In particular:
  - How parameters are passed to the called function (callee)
  - How parameters are accessed in the callee
  - How results are returned from the callee
- Example:

```
int friend()
{
     …
     b = foo(1, 2);
     …
}
```

```
int foo(int p1, int p2)
{
     return p1 + p2;
}
```

Parameter passing

- Parameters are always pushed in the stack
- They are always pushed from left to right
- Leftmost is bottommost in the stack



Parameter access

- Register EBP is used as pivot in the stack to access the parameters of a function
- When a function is compiled, a preamble is always added to setup the EBP register
- EBP is always restored with its previous value (pivot in the caller) at the end of function
- EBP plus positive offsets are used to access parameters
  - First parameter (leftmost): ebp + 8
  - Second parameter: ebp + 12
- EBP plus negative offsets are used to access local variables

EBP as frame pointer

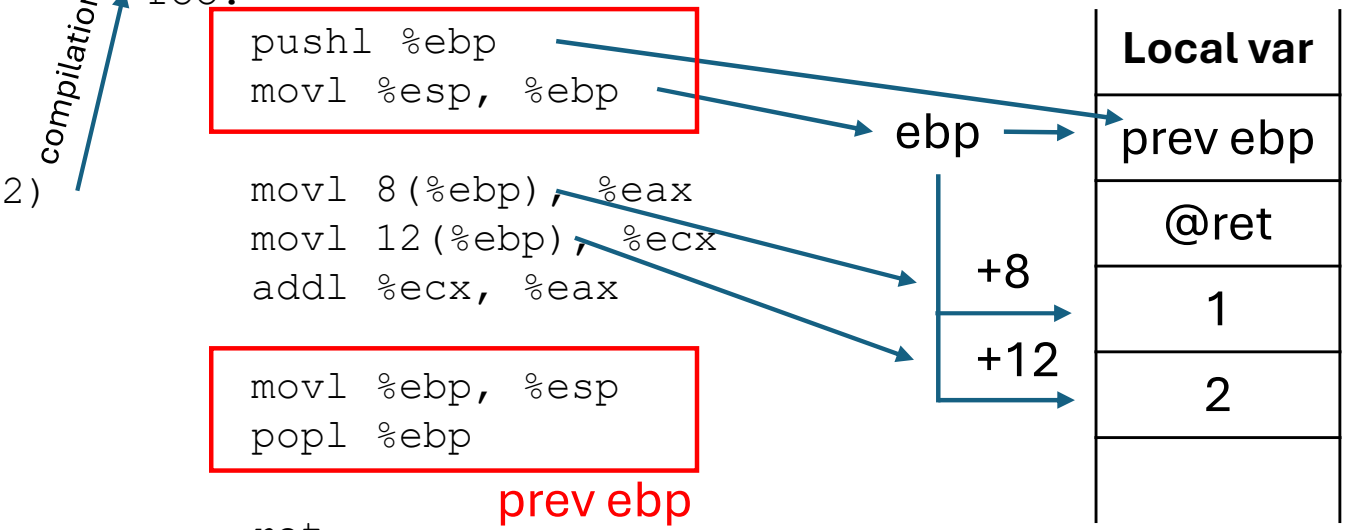
```
int foo(int p1, int p2)
{
     return p1 + p2;
}
```

compilation

```
foo:
    pushl %ebp
    movl %esp, %ebp

    movl 8(%ebp), %eax
    movl 12(%ebp), %ecx
    addl %ecx, %eax

    movl %ebp, %esp
    popl %ebp

    ret
```

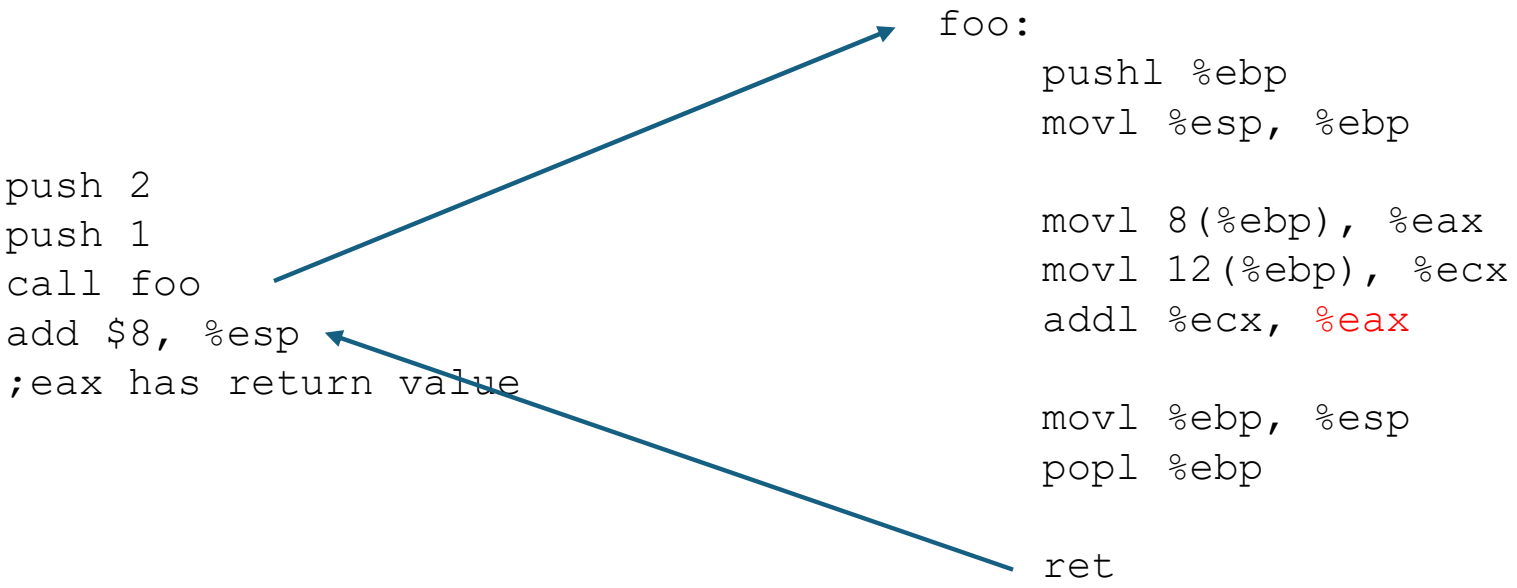
preamble

prev ebp restoration



Results

- Results are always returned using the EAX register



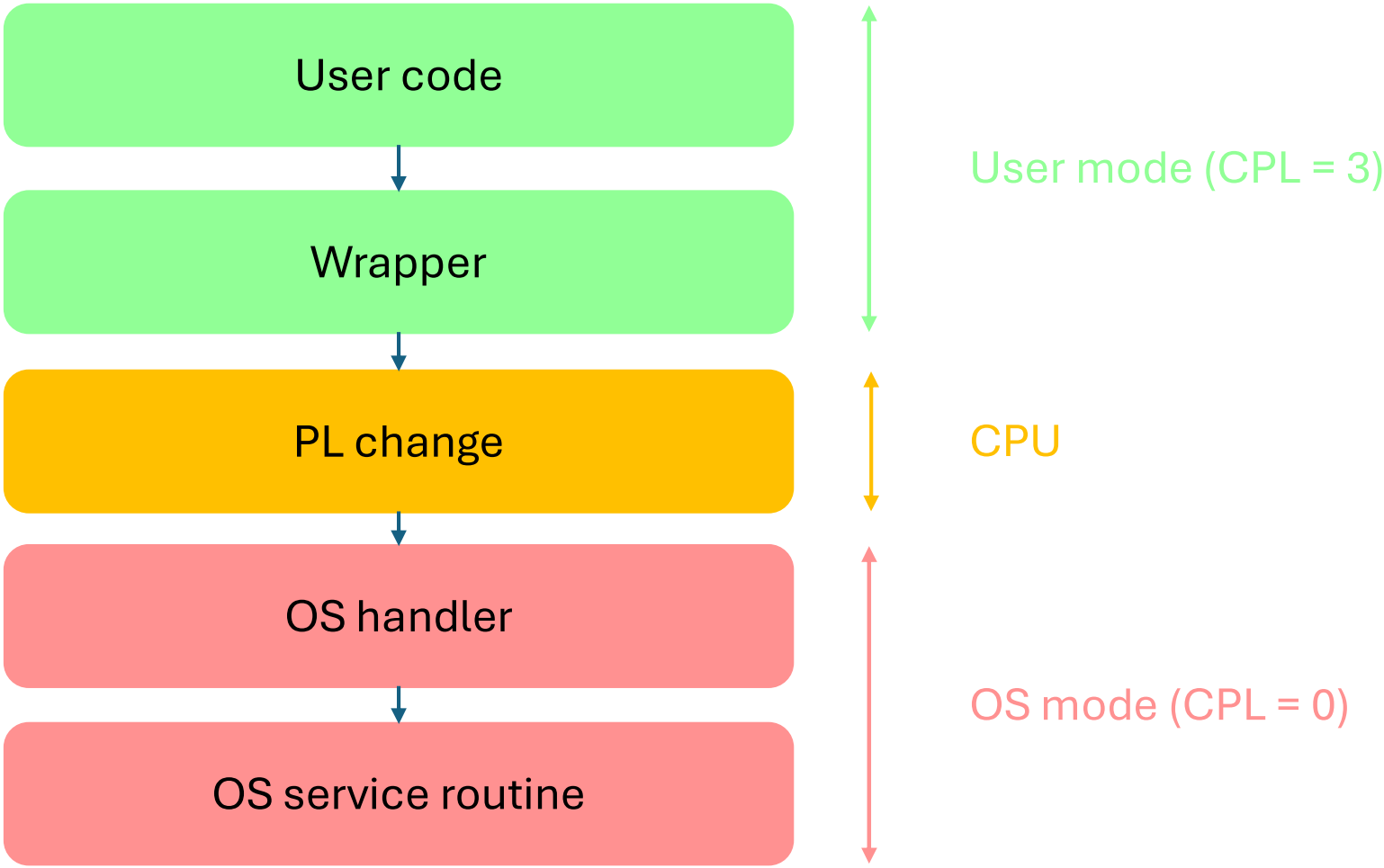
Building the OS

Building the OS

- Now that we know:
  - What privilege levels are
  - How the processor works with privilege levels
  - How privilege levels are changed
  - How the compiler works when calling functions
- It is time to build the OS

Flow diagram of a syscall

- When a thread requires something from the OS, the following flow diagram is executed:



User code

- User code never access a hardware device
- In addition, it should never invoke a system call directly
- Instead, it calls a function, called Wrapper, that will invoke the operating system and hide to the user all required details

Wrappers

- A wrapper is a function that implements all the needed logic to invoke a system call
- They ensure portability of the user code since they expose a standard interface to the user code (Posix for Linux)
- A wrapper is tightly coupled to the operating system
  - Depending on how the system works, the wrapper is implemented
- Wrappers are provided by the OS programmer in libraries (libso.so for Linux or ntdll.dll in Windows)
- Usually written in assembler

User code and wrappers

```
user code
…
if (open("hola.txt", O_RDONLY) == -1)
…
```

Wrapper library

```
…

int open(char *name, int flags [, int mode])
{
…
OS Invocation
…
}

int read(int fd, void *buffer, size_t size)
{
…
OS Invocation
…
}

…
```

- Standard interface (POSIX):
  - The header of the wrapper always is the same for any POSIX-compliant OS
  - Portability

How wrappers are implemented

- Since wrappers transform a standard interface to a particular way of invoking the OS, some decisions must be taken:
  - How parameters are passed to the OS
  - How the OS is invoked
  - How results are returned from the OS

Parameter passing

- Nowadays, parameter passing to the OS is implemented in three different ways:
  - Using CPU registers (Linux)
  - Using a buffer (Windows)
  - Using both (OSes in x64 architectures)

Parameter passing using CPU registers

- Map every parameter in a CPU register
- Typically:

```
int syscall(int par1, int par2, int par3, int par4, int par5, int par6);
              ↓       ↓       ↓       ↓       ↓       ↓
             ebx     ecx     edx     esi     edi     ebp
```

```
int read(int fd, void *buffer, size_t size)
     push %ebp
     movl %esp, %ebp
     pushl %ebx     // Safe register
     movl 8(%ebp), %ebx
     movl 12(%ebp), %ecx
     movl 16(%ebp), %edx
…
```

- Pros: fast and easy
- Cons: wrappers have a limited number of parameters
  - As many parameters as available CPU registers

Parameter passing using a buffer

- Parameters are stored in a user buffer
- The pointer to that buffer is passed to the OS
- Typically: the buffer is the user stack
- Ebx is assigned the value of EBP that points to the activation frame of the wrapper
- Pros: the wrapper must do nothing
- Cons: the service routine must copy the user buffer to an OS buffer and unscramble the parameters

Parameter passing using both

- To overcome the limitation on the number of CPU registers when passing parameters mapped to registers, a mixed approach is implemented:
  - The first 6 parameters are passed using CPU registers
  - The remaining parameters are passed using a pointer to the user stack
- This is a typical calling convention in x64 architectures

OS invocation from Wrappers

- As seen before, a wrapper can use different assembler instruction to perform a system call:
  - INT
  - SYSENTER
  - SYSCALL
- The chosen one depends on how the OS expects to be invoked
- More than one can be used

OS invocation using INT

- Remember that INT needs a parameter
  - This parameter represents the IDT entry to be used to find the OS entry point
- There are a total of 192 entries in the IDT for software interrupts (syscalls)
- Three different ways of working with that parameter have been implemented:
  - 1 IDT entry for OS service
  - 1 IDT entry for group of related OS services
  - Only 1 IDT entry (OS entry point)

1 IDT entry for OS service

- A service is mapped in one entry of the IDT

Entry	25	26	27	28	29	30
Service	read	write	close	dup	dup2	fork

- Wrappers are aware of that and invoke the INT instruction with the corresponding IDT entry number

```
int read(int fd, void *buffer, size_t size)
{
…
      int $25
…
}
```

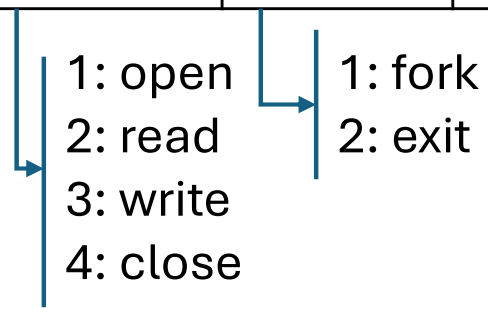
```
int write(int fd, void *buffer, size_t size)
{
…
      int $26
…
}
```

- Used in embedded OSes with few services
- Pros: Wrappers are easy to implement
- Cons: Limited number of services (as many as IDT entries)

1 IDT entry for group of OS services

- A group of services are mapped in one entry of the IDT

Entry	25	26	27	28	29	30
Group	filesystem	Process	threads	memory	io	ctrl



- EAX is used as a service selector inside the group

```
int read(int fd, void *buffer, size_t size)
{
…
      movl $2, %eax
      int $25
…
}
```

```
int fork()
{
…
      movl $1, %eax
      int $26
…
}
```

1 IDT entry for group of OS services (and II)

- Pros: overcomes the limitation on the number of services. Now the number of OS services are nearly unlimited (192 IDT entries * 2^32 services per entry)
- Cons: as in the previous one, many handlers (explained later) are needed for syscalls. This reduces code maintainability and reduces security

Only 1 IDT entry for all OS services

- All services are offered using only one entry of the IDT
  - Linux: 0x80
  - Windows: 0x2E
- Eax register used as a service selector

```
int read(int fd, void *buffer, size_t size)
{
…
      movl $2, %eax
      int $0x80
…
}
```

```
int fork()
{
…
      movl $16, %eax
      int $0x80
…
}
```

- Pros: enhances code maintainability and security
- All current general-purpose OSes implement this

Returning results

- The OS returns the result of an operation in EAX (as functions do)
- But:
  - Returns EAX >= 0 if OK
  - Returns EAX = -ERRNO if failed
- The user expects (following POSIX)
  - Returns EAX >= 0 if OK
  - Returns EAX = -1 and errno set to error code if failed
- So, the wrapper must transform the return value to the format the user expects

A whole wrapper with INT

```
int read(int fd, void *buffer, size_t size)
    pushl %ebp
    movl %esp, %ebp

    pushl %ebx
    movl 8(%ebp), %ebx
    movl 12(%ebp), %ecx
    movl 16(%ebp), %edx

    movl $4, %eax
    int $0x80

    popl %ebx

    test %eax, %eax
    jns ok

    negl %eax
    movl %eax, errno
    movl $-1, %eax
ok:
    movl %ebp, %esp
    popl %ebp

    ret
```

preamble

parameter passing

OS invocation

stack restoration

return value transformation

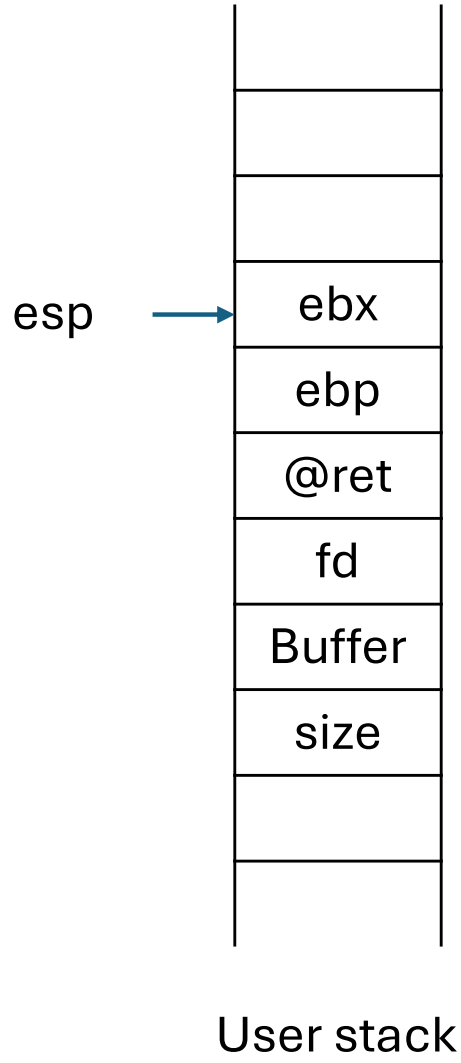
previous ebp restoration

Using sysenter instead of INT

- Remember that sysenter does not push the hardware context in the stack
- So, the system stack is different if invoking the OS with sysenter than that using int
- In addition, the OS needs the user stack address and the user code address to return to user mode
- Solution: pass user esp through register ebp and user return address using the user stack

A wrapper with sysenter

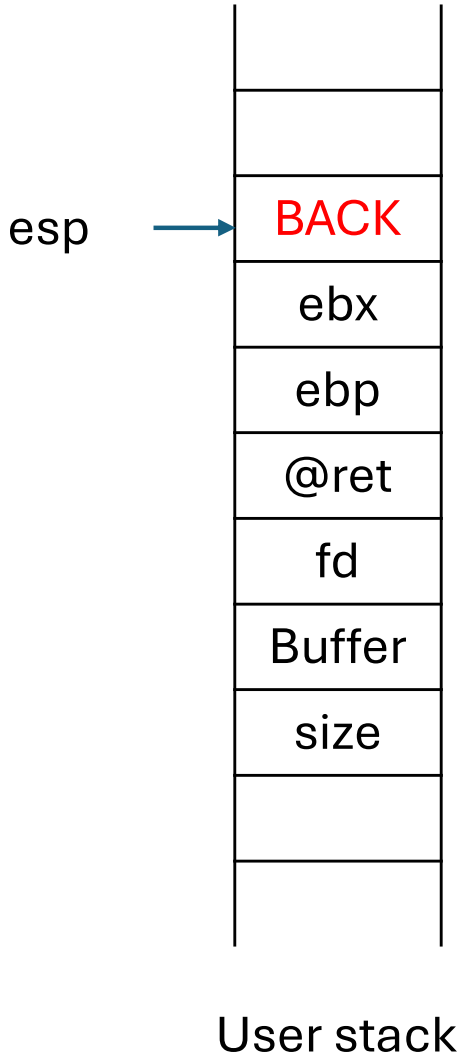
```
int read(int fd, void *buffer, size_t size)
     pushl %ebp
     movl %esp, %ebp

     pushl %ebx
     movl 8(%ebp), %ebx
     movl 12(%ebp), %ecx
     movl 16(%ebp), %edx
```



A wrapper with sysenter

```
int read(int fd, void *buffer, size_t size)
     pushl %ebp
     movl %esp, %ebp

     pushl %ebx
     movl 8(%ebp), %ebx
     movl 12(%ebp), %ecx
     movl 16(%ebp), %edx

     pushl $BACK

     sysenter

BACK:
```

Push the return address to user code

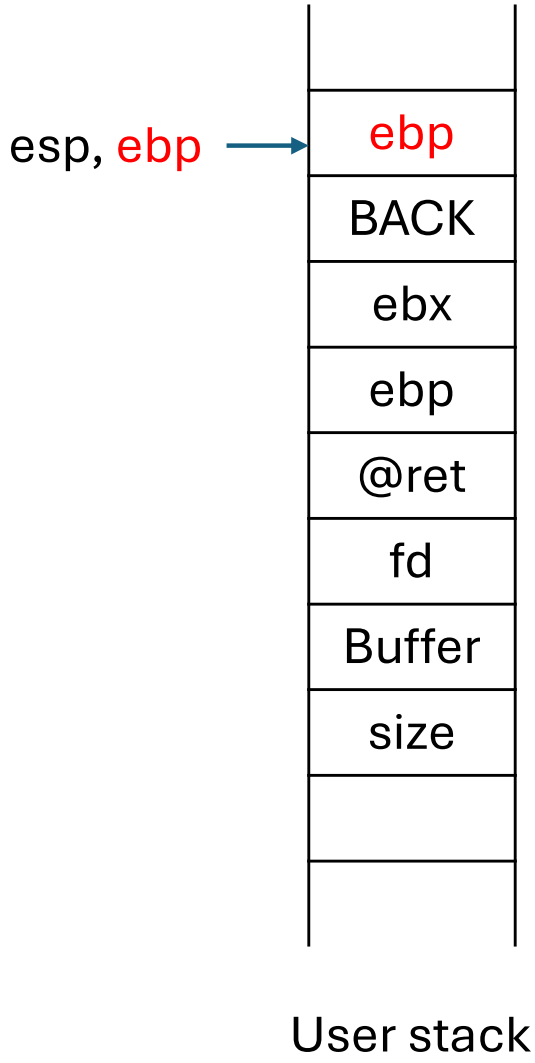


User stack

A wrapper with sysenter

```
int read(int fd, void *buffer, size_t size)
     pushl %ebp
     movl %esp, %ebp

     pushl %ebx
     movl 8(%ebp), %ebx
     movl 12(%ebp), %ecx
     movl 16(%ebp), %edx

     pushl $BACK

     pushl %ebp
     movl %esp, %ebp

     sysenter

BACK:
     popl %ebp
```

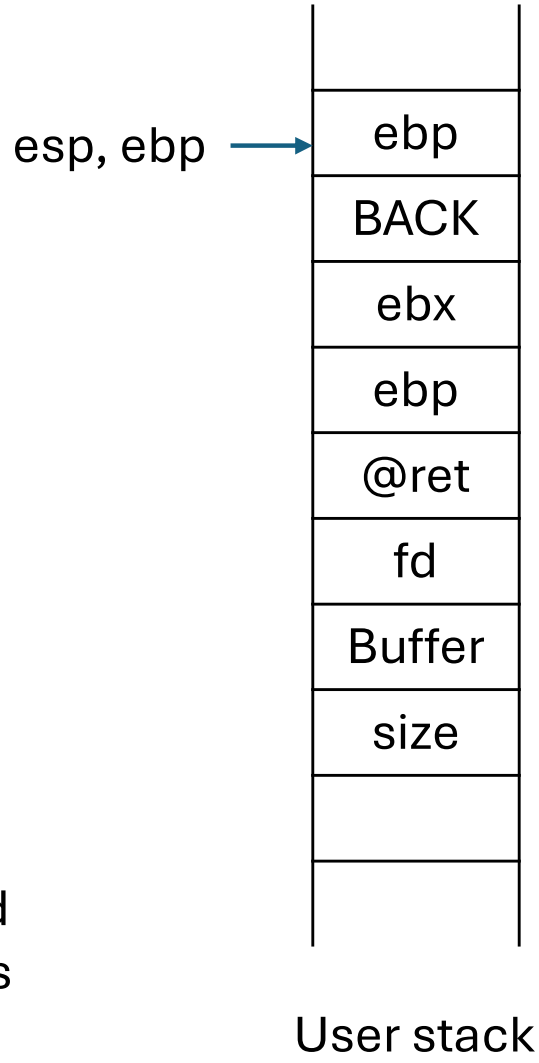
Previous ebp is pushed in the user stack and assign esp to ebp



User stack

A wrapper with sysenter

```
int read(int fd, void *buffer, size_t size)
     pushl %ebp
     movl %esp, %ebp

     pushl %ebx
     movl 8(%ebp), %ebx
     movl 12(%ebp), %ecx
     movl 16(%ebp), %edx

     pushl $BACK

     pushl %ebp
     movl %esp, %ebp

     sysenter

BACK:
     popl %ebp
```



User stack

Now, ebp holds the user stack address and
*(ebp + 4) has the user code return address

OS Handlers

- It's the first executed code in OS mode
- There is one handler per used IDT entry
  - 16 handlers of hardware interrupts
  - 48 handlers of exceptions
  - 1 handler of the unique OS entry point
- Basically, for context management: software context
- Written in assembler
- It calls a high-level C function (service routine) where the behavior of the service is implemented

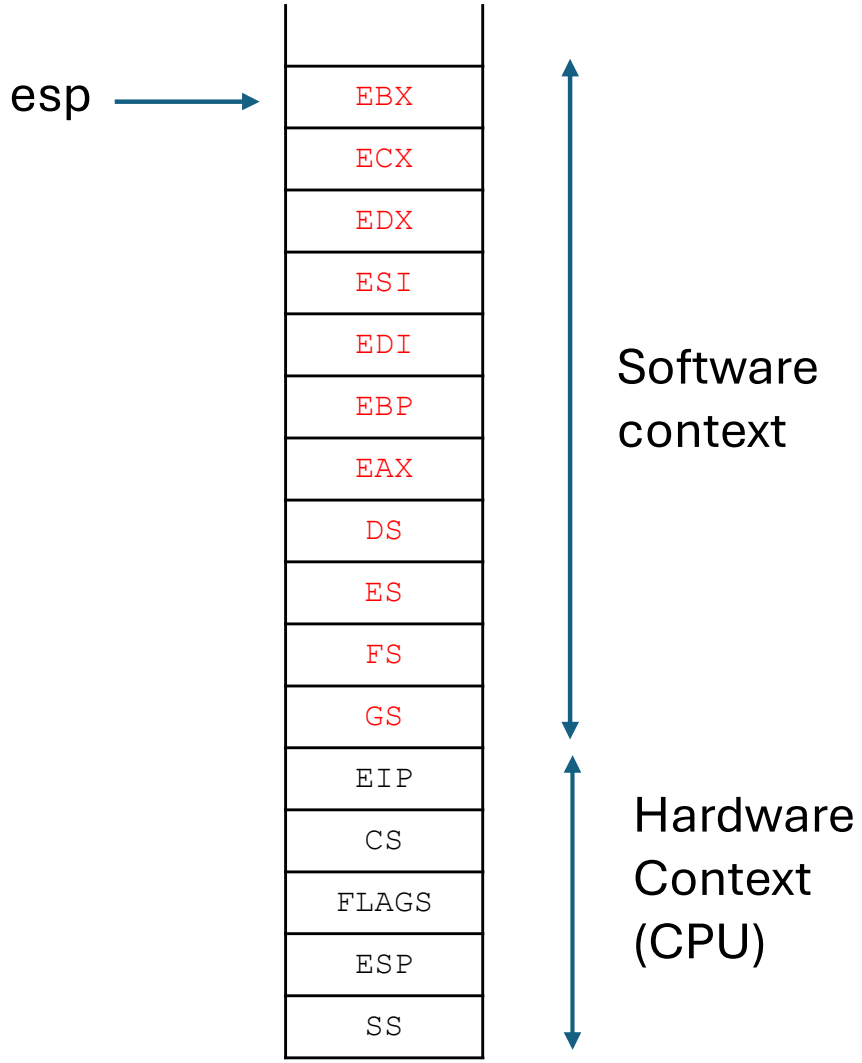
Software context

- It is the minimal information required for the OS to return to user mode
- It if the first action performed inside the handler
- Implemented as the macro SAVE_ALL

SAVE_ALL macro

- Pushes the required CPU registers in the system stack

```
#define SAVE_ALL \
        pushl %gs; \
        pushl %fs; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx;
```



Calling the service routine

- Once the software context has been pushed into the system stack, the handler must call the service routine
- There is one service routine for hardware interrupt and exception, so, the handler calls directly this service routine

```
clock_handler:
    SAVE_ALL
    call clock_routine
```

- For syscalls, it is somewhat more complicated

Calling a syscall service routine

- Remember that, from the wrapper, and having a unique OS entry point, the register EAX works as a service selector
- It is not advisable to have a huge switch…case statement inside the handler to call the appropriate service
  - A current OS can have more than 1000 services
- Solution: syscall_table + index-scaled indirect call

Syscall table

- In OSes with a unique entry point
- It is an array, with as many entries as services the OS has, and in every entry, the address of a service routine is stored
- In x86, every entry is 4 bytes.

Entry	1	2	3	4	5	6
Service	open	read	write	close	dup	...

- Initialized in boot time

Index-scaled indirect call

- X86 provides an instruction that is able to:
  - Access an array (base_address)
  - Get the value of one position of that array given the number of position (eax) and the size of every entry of the array (4 bytes)
  - Make a call having as address the value obtained from the array

```
call *base_address(, %eax, 4)
```

- It is equivalent to:

```
address = base_address + %eax * 4
value = *address
call value
```

All together

- Having into account the system_call table and the index-scaled indirect call, calling a service routine inside the handler is as easy as doing:

```
call *syscall_table(, %eax, 4)
```

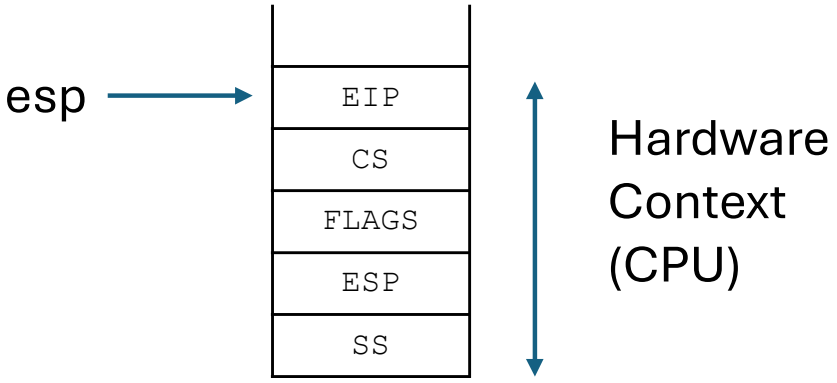
Returning to user mode

- Once the service routine has finished, it is time to return to user mode
- To do this two actions must be performed
  - Restore the software context
  - Return to user mode

Restore the software context

- Remember that the handler has pushed the software context in the system stack
- To restore the context, a new macro called RESTORE_ALL is executed
- RESTORE_ALL pops all CPU registers that SAVE_ALL pushed in the system stack

```
#define RESTORE_ALL \
      popl %ebx;    \
      popl %ecx;    \
      popl %edx;    \
      popl %esi;    \
      popl %edi;    \
      popl %ebp;    \
      popl %eax;    \
      popl %ds;     \
      popl %es;     \
      popl %fs;     \
      popl %gs;
```



System stack after RESTORE_ALL

Returning to user mode

- X86 provides an instruction to return to user mode changing the privilege level: IRET
- IRET performs the opposite work of INT:
  - Pops the hardware context from the system stack
  - Points ss:esp and cs:eip to the user stack and code
  - Restores the processor flags from the stack

Returning to user mode from HW INT

- In OS mode, HW interrupts are disabled
- This means that when executing in OS mode no HW int will be processed by the CPU
- Before executing RESTORE_ALL in the handler of a HW interrupt, it is necessary to notify the PIC to start sending again more HW interrupts to the processor
- This is done using the macro EOI

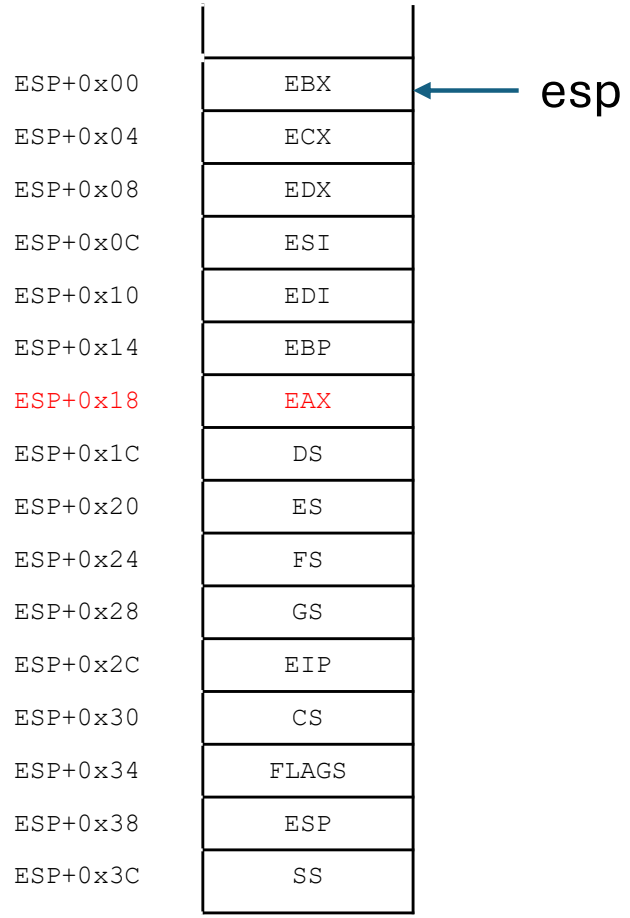
```
#define EOI \
        movb $0x20, %al; \
        outb %al, $0x20;
```

Preserving the service routine return value

- A service routine, if needed, will return a result using the EAX register
- If you notice, this value will be lost since the handler will execute a RESTORE_ALL that, among others, restores the previous value of the EAX register from the software context
- To prevent that, after calling the service routine, the handler will smash the value of the EAX register in the software context for later recover it using RESTORE_ALL

Preserving the service routine return value (and II)

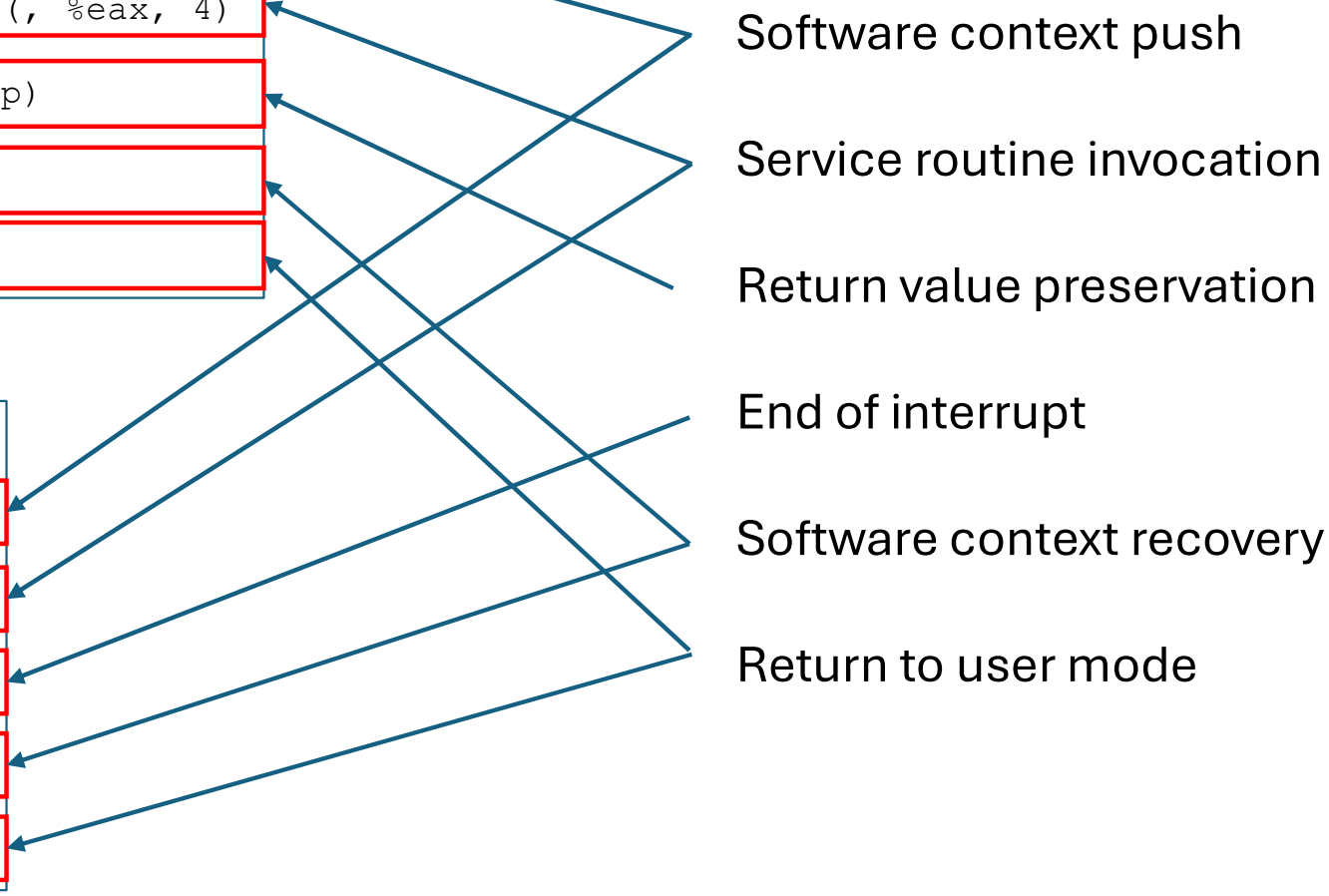
```
call *syscall_table(, %eax, 4)
movl %eax, 0x18(%esp)
RESTORE_ALL ;Recovers the correct value
```



A whole handler

```
syscall_handler:
    SAVE_ALL
    call *syscall_table(, %eax, 4)
    movl %eax, 0x18(%esp)
    RESTORE_ALL
    IRET
```

```
clock_handler:
    SAVE_ALL
    call clock_routine
    EOI
    RESTORE_ALL
    IRET
```



Software context push

Service routine invocation

Return value preservation

End of interrupt

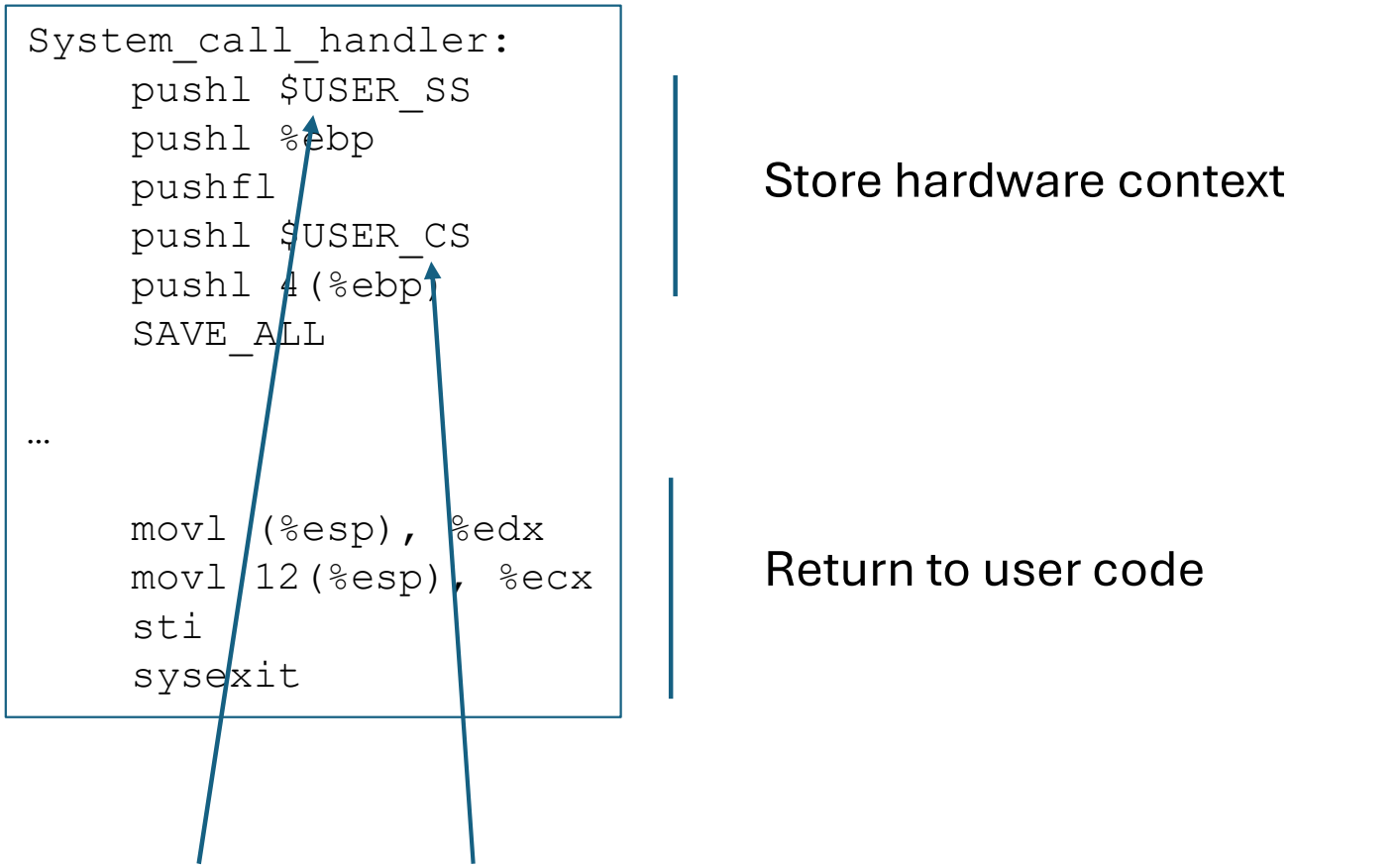
Software context recovery

Return to user mode

System call handler with sysenter

- Remember that sysenter does not push the hardware context in the stack
- So, the handler is responsible to push it
- Remember that a wrapper with sysenter is using register ebp to point to the user stack and *(ebp + 4) holds the user code return address
- In addition, to return to user code, the hander must use the sysexit assembler instruction
  - Uses the value in %edx as return address
  - Uses the value in %ecx as user stack address
  - HW Interrupts must be reenabled using sti

A whole handler with sysenter



USER_SS and USER_CS are constants known by the kernel
PUSHFL pushes the CPU flags

Service routine

- The service routine implements the real behavior of a system service
- Written in C
- Nomenclature is important:
  - Service routines for HW interrupts and exceptions are named with the suffix _routine: clock_routine
  - Service routines for syscalls are named with the prefix sys_: sys_read, sys_write, ...
- Follows a lazy coding pattern:
  - Don’t do anything if you cannot do it

Service routine structure

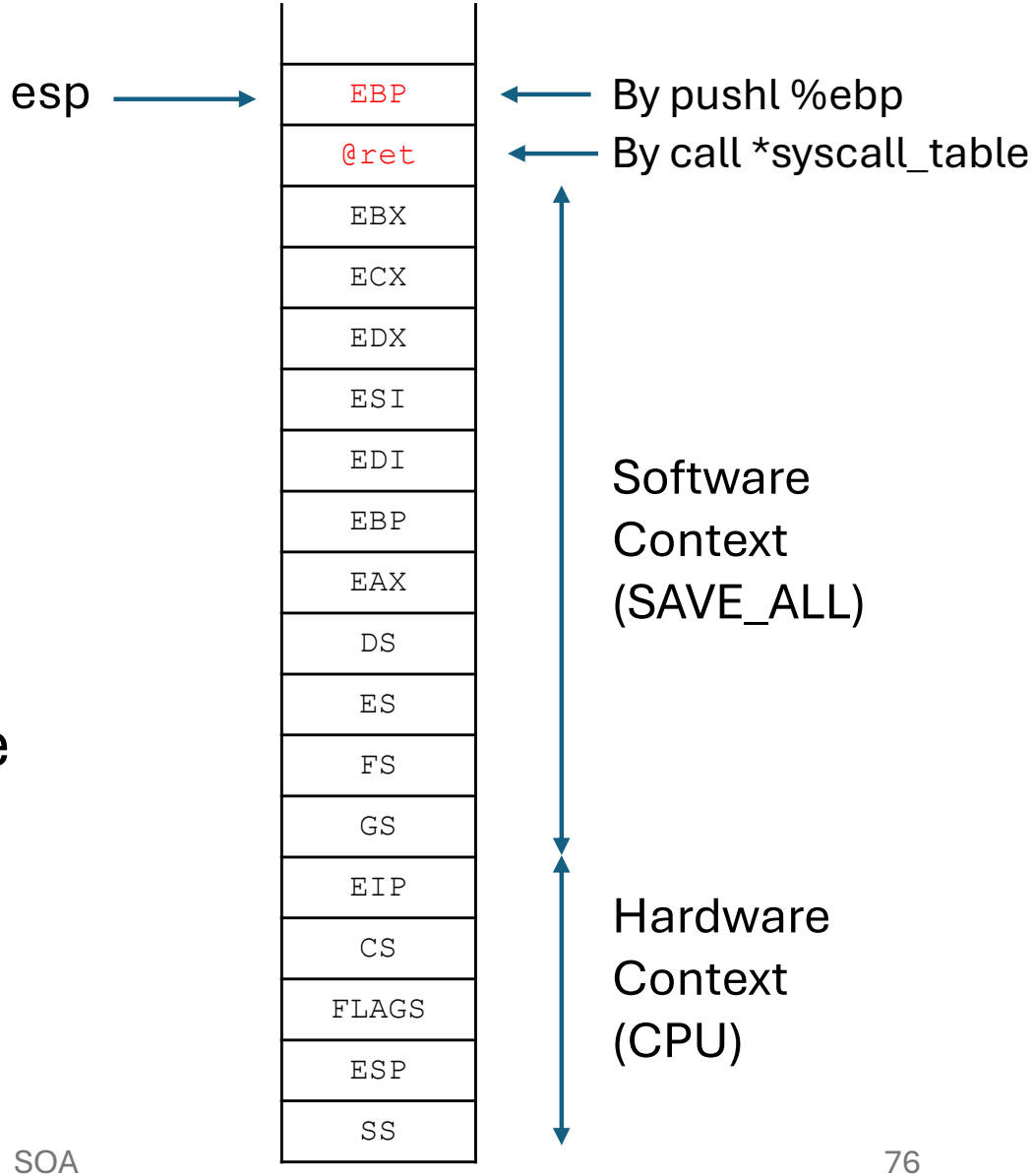
- The structure of a service routine is:

```
int sys_whatever(list_of_parameters)
{
     parameter checking;       // If fails, return error
     resource allocation;      // If fails, deallocate resources and return error
     do_work;
     return result;
}
```

- Remember that return value follows:
  - If OK return a value >= 0
  - If fail return –error_code
- Do not forget to deallocate any allocated resource if fail
  - If not done, memory leaks

System stack in the service routine

- Before executing the first high-level instruction of the service routine, the system stack is as follows
- Remember that:
  - The call to the service routine pushes the return address in the stack
  - The compiler has added the preamble when translating the high-level code to assembler

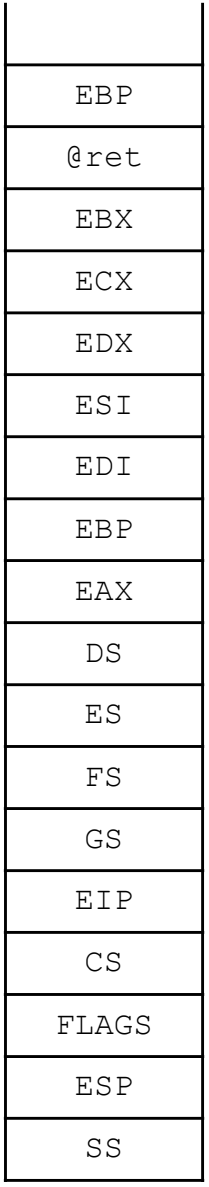


How parameters are accessed from the service routine

- Remember that parameters are passed from the wrapper to the service routine using either registers or the user stack or both
- When using registers, somehow these registers must be pushed to the system stack
- When using a buffer, the parameters must be recovered from that buffer

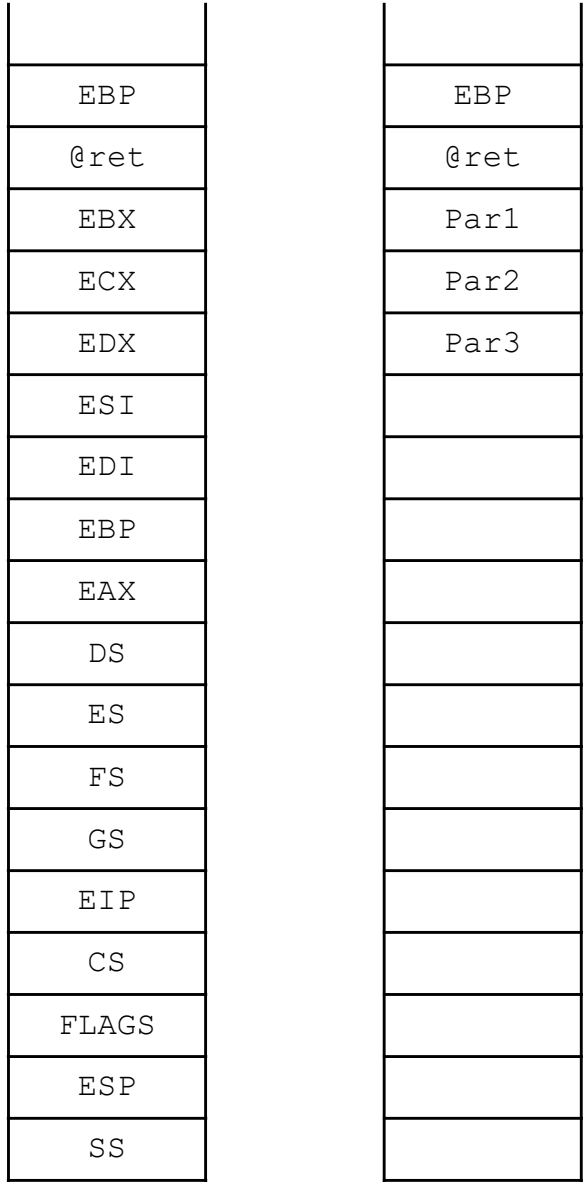
Recovering parameters passed as registers

- This is the system stack before executing the first high-level instruction of the service routine



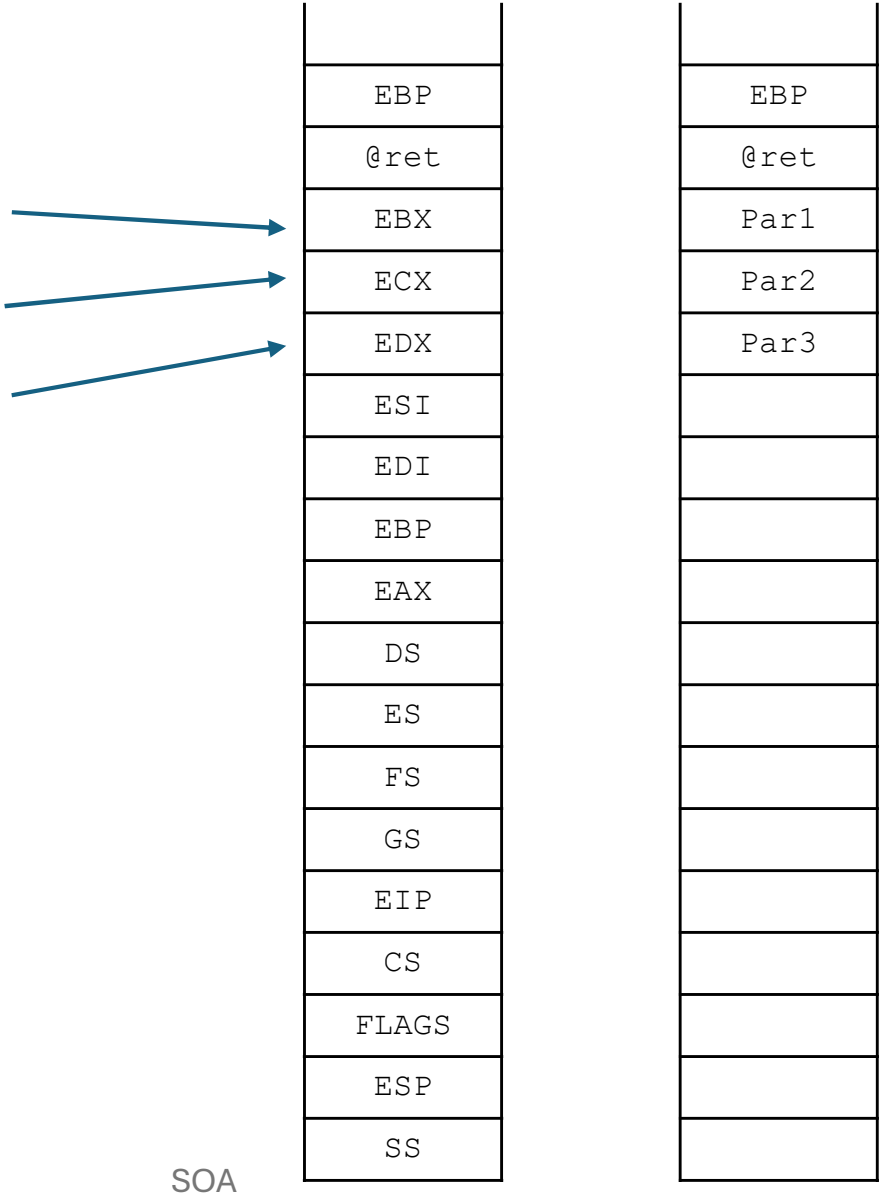
Recovering parameters passed as registers

- This is the activation frame of a function with three parameters



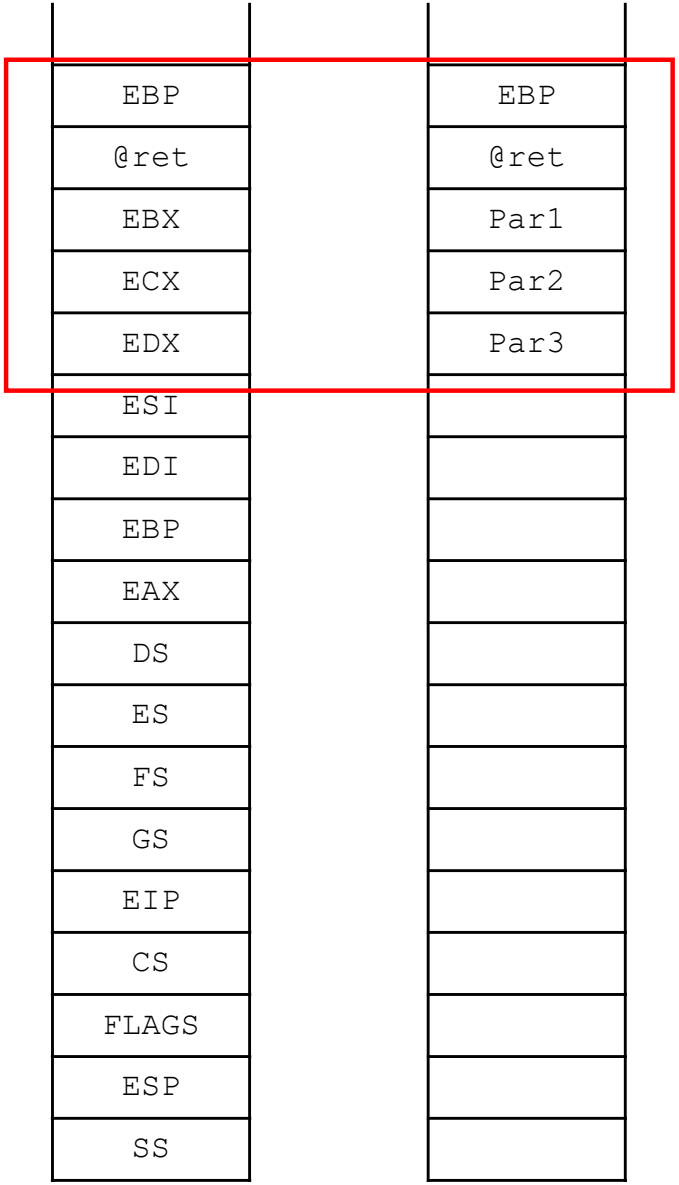
Recovering parameters passed as registers

- Remember that in the wrapper:
  - Par1 is passed in EBX
  - Par2 is passed in ECX
  - Par3 is passed in EDX



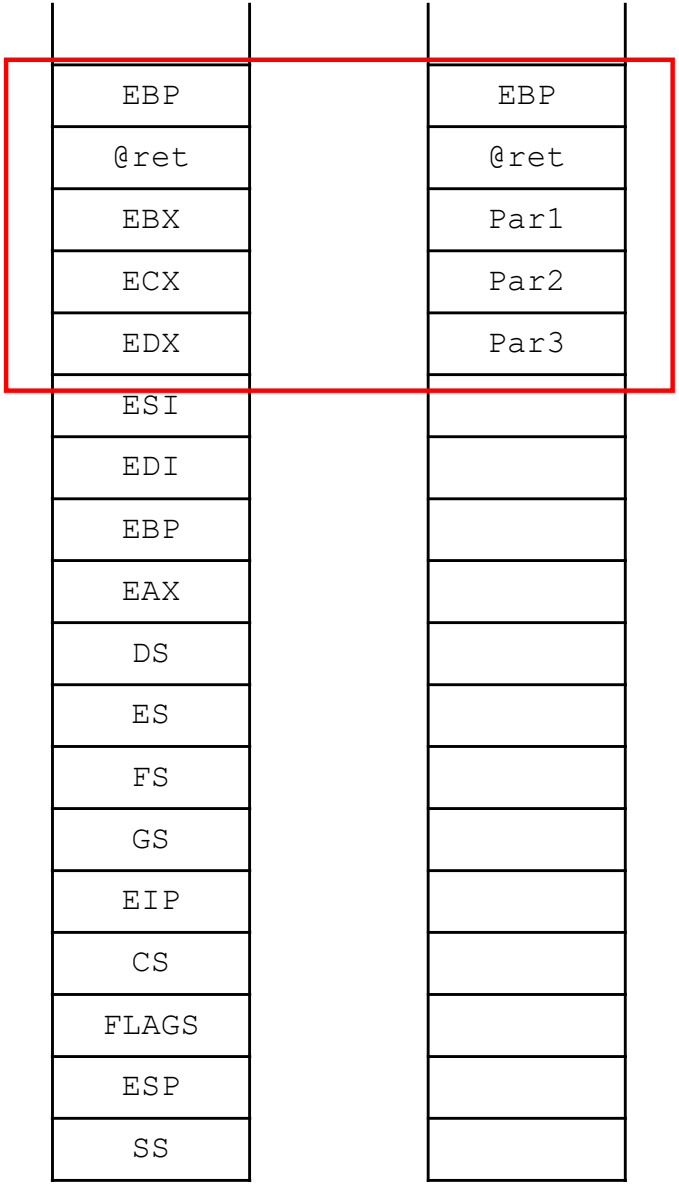
Recovering parameters passed as registers

- Since SAVE_ALL has pushed the registers in a particular order, the software context has become an activation frame for the service routine!!!!



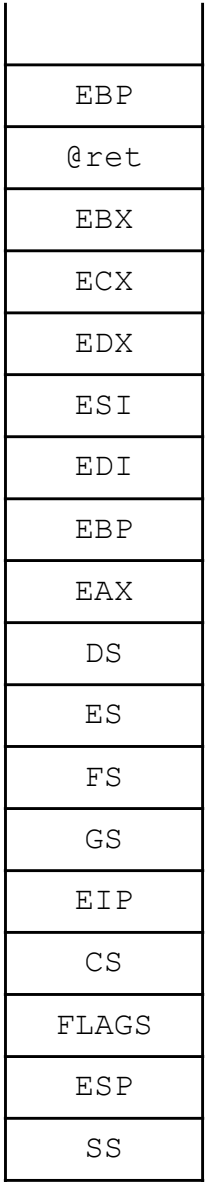
Recovering parameters passed as registers

- If the order of the registers in the wrapper is modified, the SAVE_ALL macro must be modified to maintain this behavior



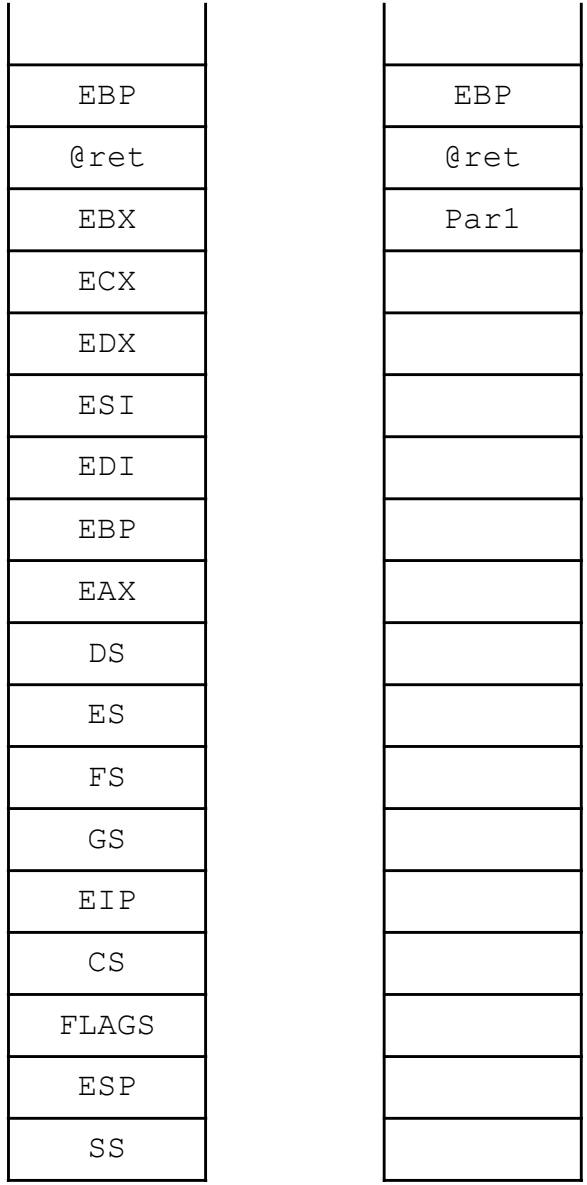
Recovering parameters passed as a user buffer

- This is the system stack before executing the first high-level instruction of the service routine



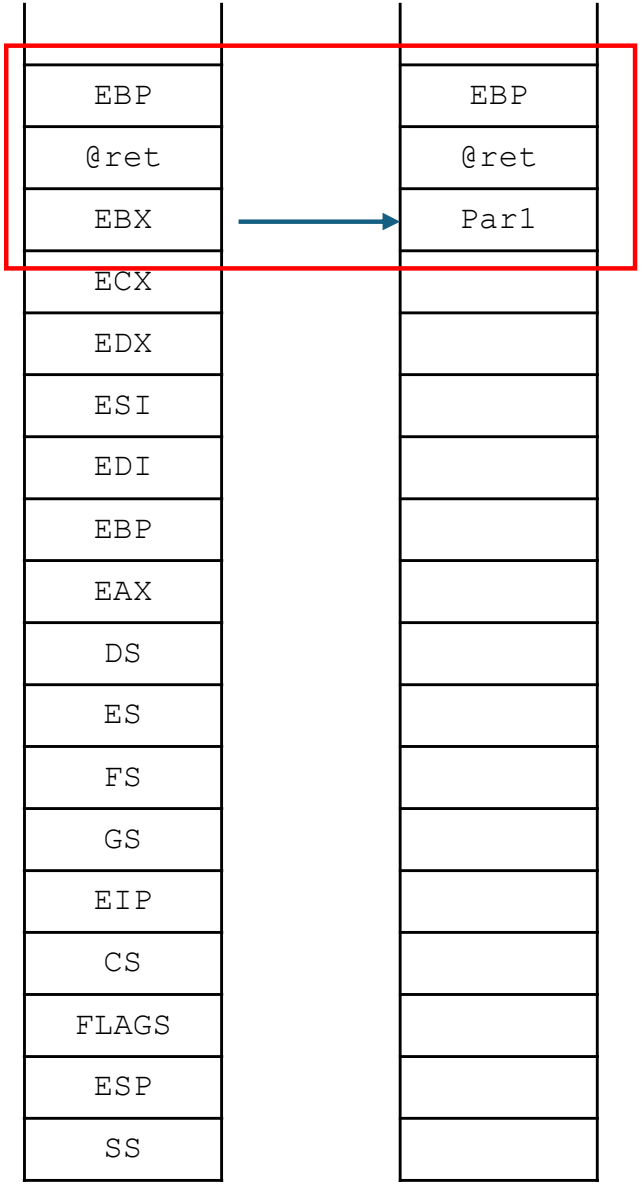
Recovering parameters passed as a user buffer

- This is the activation frame of a function with one parameter



Recovering parameters passed as a user buffer

- Remember that in the wrapper the register ebx is used to point to the activation frame of the wrapper in the user stack
- Here, this EBX, as before, is the first parameter of the service routine
- So, the service routine must be rewritten to have just one parameter and to copy the user buffer locally



Service routine when parameter passing is using a user buffer

```
struct read_params
{
      int fd;
      void *buffer;
      size_t size;
};
int sys_read(void *user_address)
{
      struct read_params params;

      if (!access_ok(user_address + 4)) return -EINVAL;

      CopyFromUser(&params, user_address, sizeof(struct read_params));
      …
      return whatever;
}
```

Local system buffer

Validate user address

Copy to local buffer

- There are more efficient ways to do this!!!

Last notes

- It is very important that you dominate the system stack:
  - What the contents is
  - How the system stack is constructed
  - What is pushed in the system stack and when

- If you cannot know in any moment what the contents of the system stack is, you will fail this part of the course