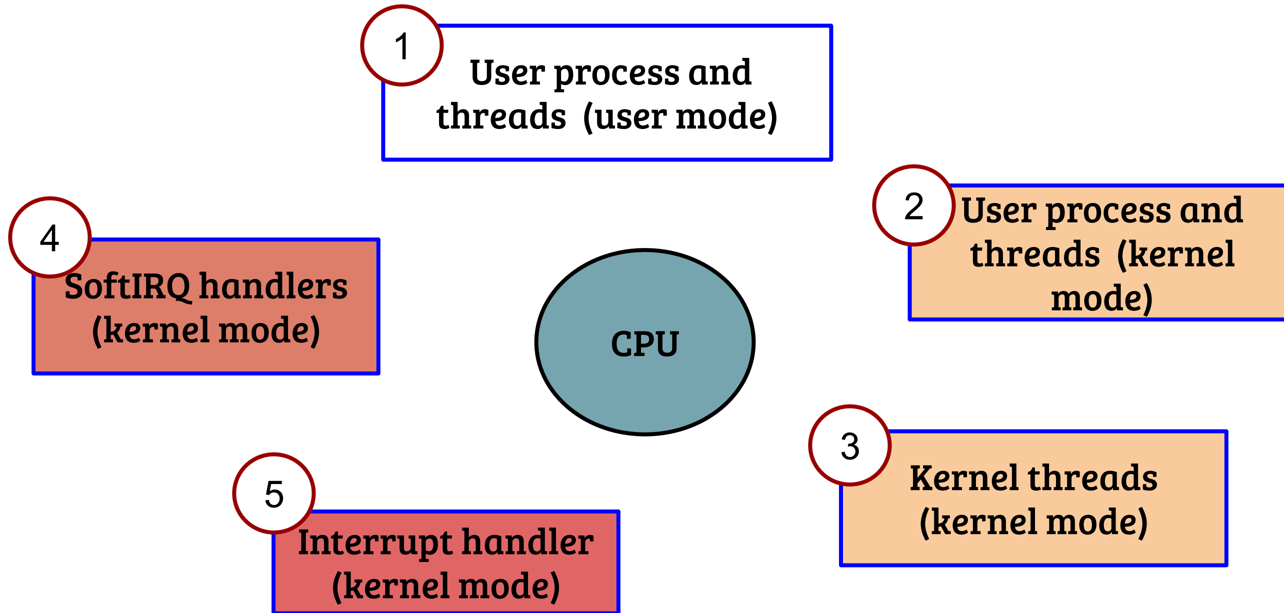
CS614: Linux Kernel Programming

Execution Contexts: User

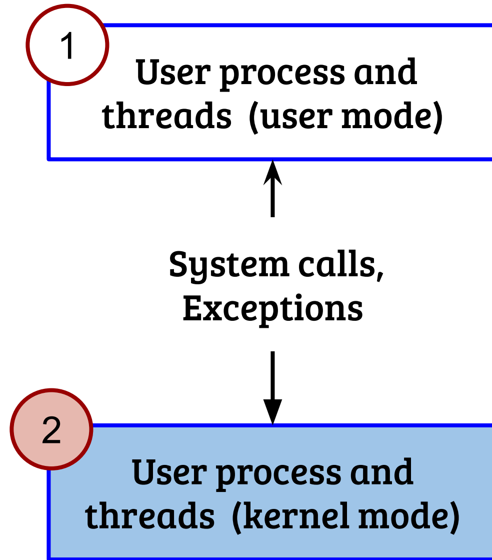
Debadatta Mishra, CSE, IIT Kanpur

Execution contexts in Linux

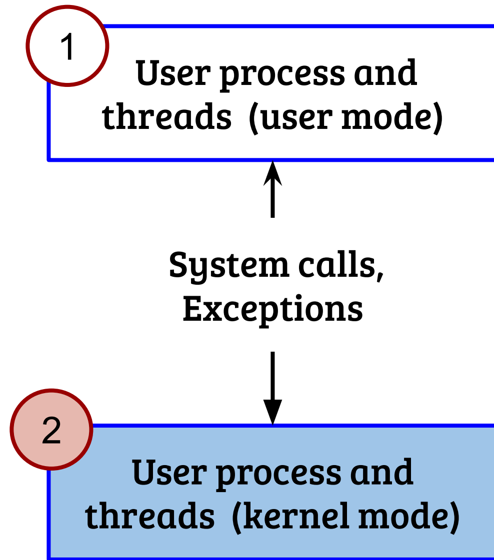


- In a linux system, the CPU can be executing in one of the above contexts
- For (3), (4) and (5), the context is not associated with any user process

User contexts

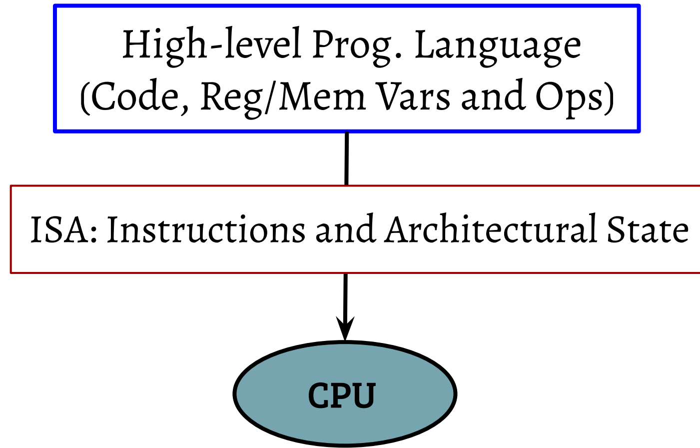


User contexts

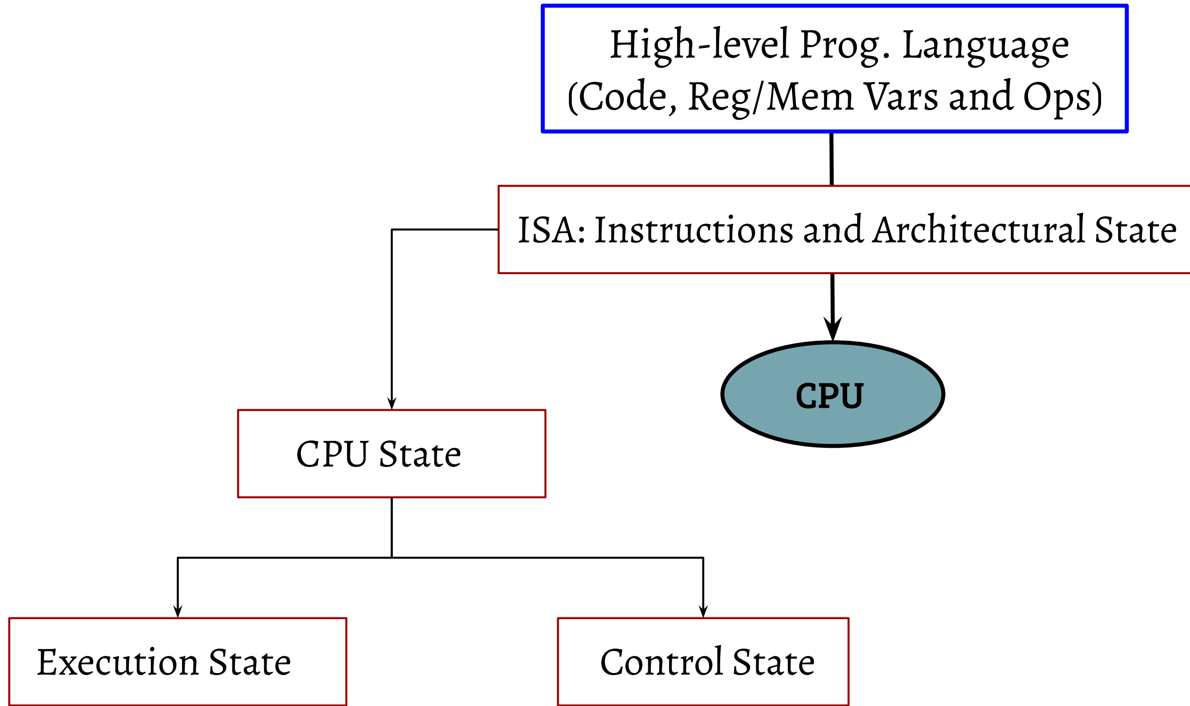


- What are the differences in execution states?
- What is the exact entry and exit mechanisms—user to kernel context switch and vice-a-versa?
- What is the need for save and restore of the execution states? How implemented in Linux kernel?
- Access/modification of user execution state from the kernel mode, how?
- How OS manages the user contexts?

Execution: Code to ISA (and hardware resources)

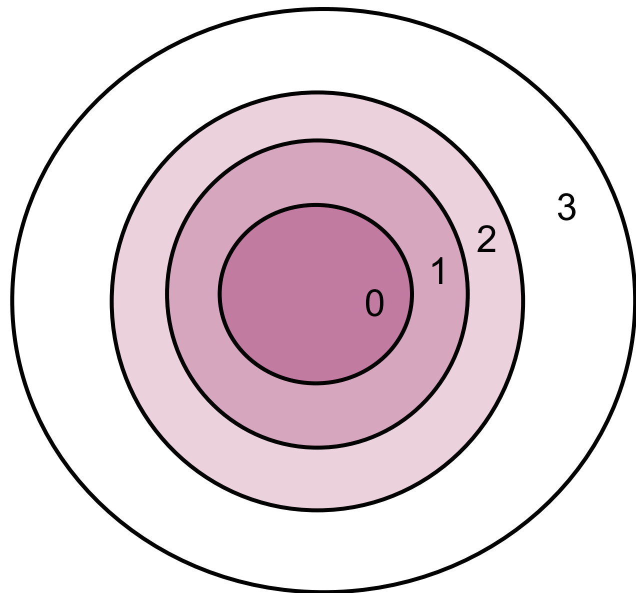


Execution: Code to ISA (and hardware resources)



- Instructions modifying the execution state is allowed from all mode of execution
- Operation on control registers are restricted using privilege support of the underlying ISA

X86: rings of protection



- 4 privilege levels: 0⟶ highest, 3⟶ lowest
- Some operations are allowed only in privilege level 0
- Most OSes use 0 (for kernel) and 3 (for user)
- Different kinds of privilege enforcement
  - Instruction is privileged
  - Operand is privileged

Privileged instruction: HLT (on Linux x86_64)

```
int main( )
{
  asm("hlt;");
}
```

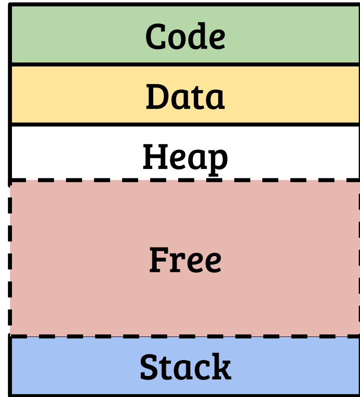
- HLT: Halt the CPU core till next external interrupt
- Executed from user space results in protection fault
- Action: Linux kernel kills the application

Privileged operation: Read CR3 (Linux x86_64)

```
#include<stdio.h>
int main( ){
  unsigned long cr3_val;
  asm volatile("mov %%cr3, %0;"
           : "=r" (cr3_val)
           :: );
 printf("%lx\n", cr3_val);
}
```

- CR3 register points to the address space translation information
- When executed from user space results in protection fault
- “mov” instruction is not privileged per se, but the operand is privileged

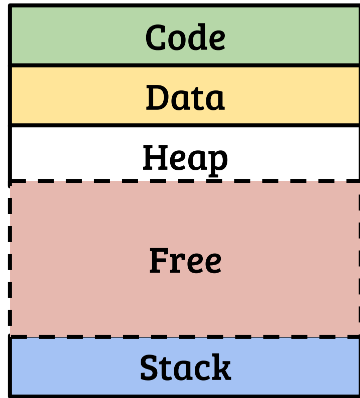
The OS address space



OS

Not only I have to enable address space for each process, I need an address space myself which is protected from the user processes. Design?

The OS address space

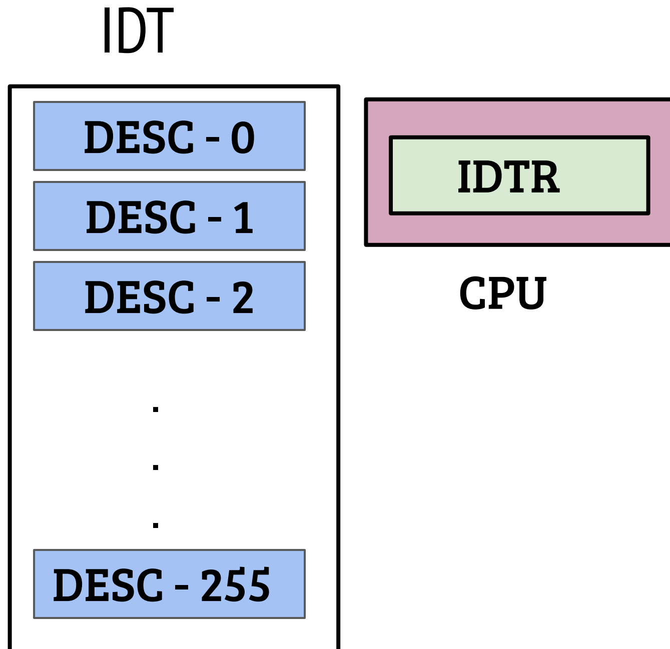


OS

Not only I have to enable address space for each process, I need an address space myself which is protected from the user processes. Design?

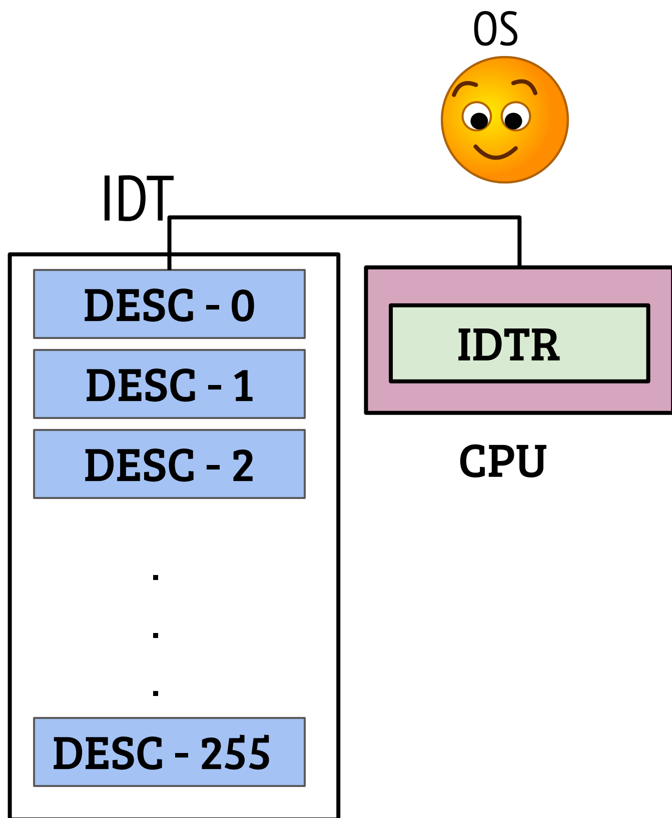
- Possible design approaches
  - Use a separate address space for the OS, change the translation information on every OS entry (inefficient, but strongly isolated)
  - Consume a part of the address space from all processes and protect the OS addresses using H/W assistance (was most commonly used)
  - Linux uses a hybrid approach (will discuss latter)

Interrupt Descriptor Table (IDT): gateway to handlers



- Interrupt descriptor table provides a way to define handlers for different events like external interrupts, faults and system calls by defining the descriptors
- Descriptors 0-31 are for predefined events e.g., 0 → Div-by-zero exception etc.
- Events 32-255 are user defined, can be used for h/w and s/w interrupt handling

Defining the descriptors (OS boot)



- Each descriptor contains information about handling the event
  - Privilege switch information
  - Handler address
- The OS defines the descriptors and loads the IDTR register with the address of the descriptor table (using *LIDT* instruction)

System call INT instruction (Conventional Method)

- INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
- Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates
- The generic system call handler invokes the appropriate handler function. How?

System call INT instruction (Conventional Method)

- INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
- Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates
- The generic system call handler invokes the appropriate handler function, How?
    - Every system call is associated with a number (defined by OS)
    - User process sends information like system call number, arguments through CPU registers which is used to invoke the actual handler

System call in Linux Kernel (using syscall inst.)

- X86 provides a fast system call method through the "syscall" instruction
- OS configures designated privileged registers with the entry address (and other information related to privilege change)
- The hardware saves the next instruction address (user return address) into RCX, change privilege levels and sets RIP to the syscall entry address. (SP and CR3 are not modified)
- Arguments and return value
    - RAX: System call # and return value
    - Arguments passed: RDI, RSI, RDX, R10, R8, R9

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning
- If OS has its own stack, who switches the stack on kernel entry?

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning
- If OS has its own stack, who switches the stack on kernel entry?
- On X86 systems, the hardware (or OS in case of “syscall”) switches the stack pointer to the stack address configured by the OS

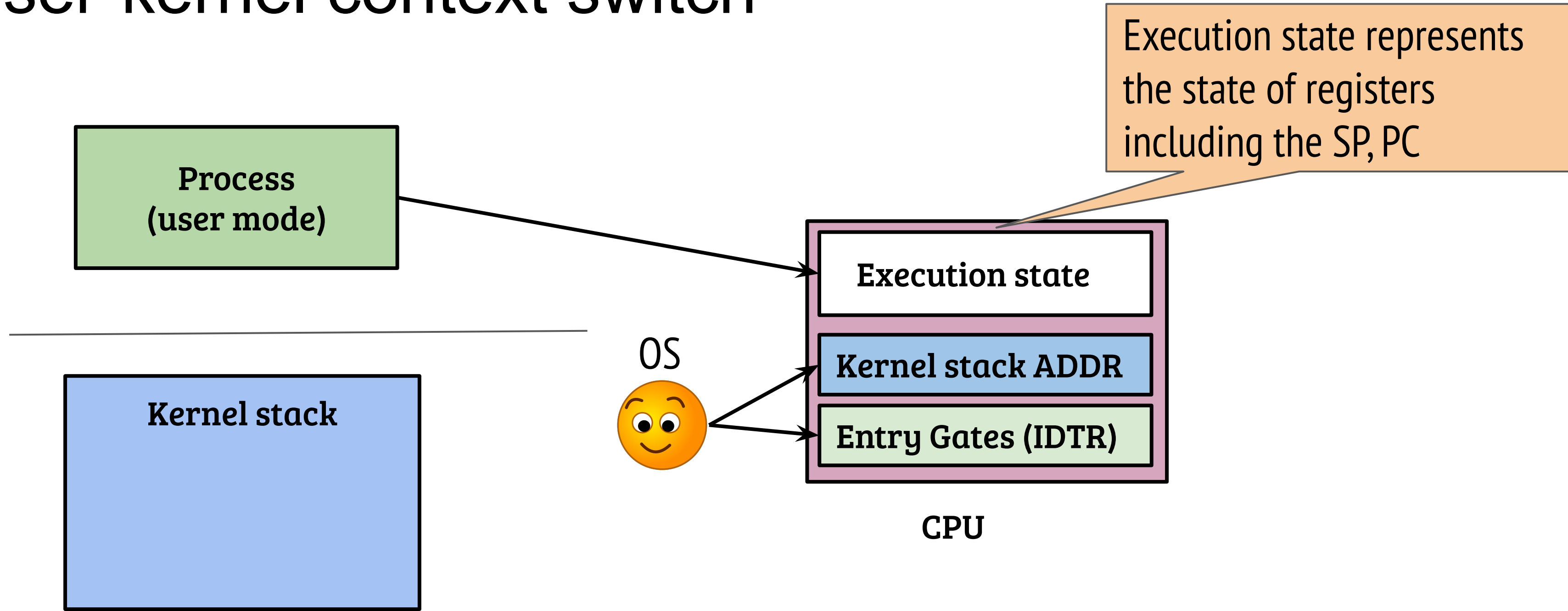
Management of (context) kernel stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working

Management of (context) kernel stacks

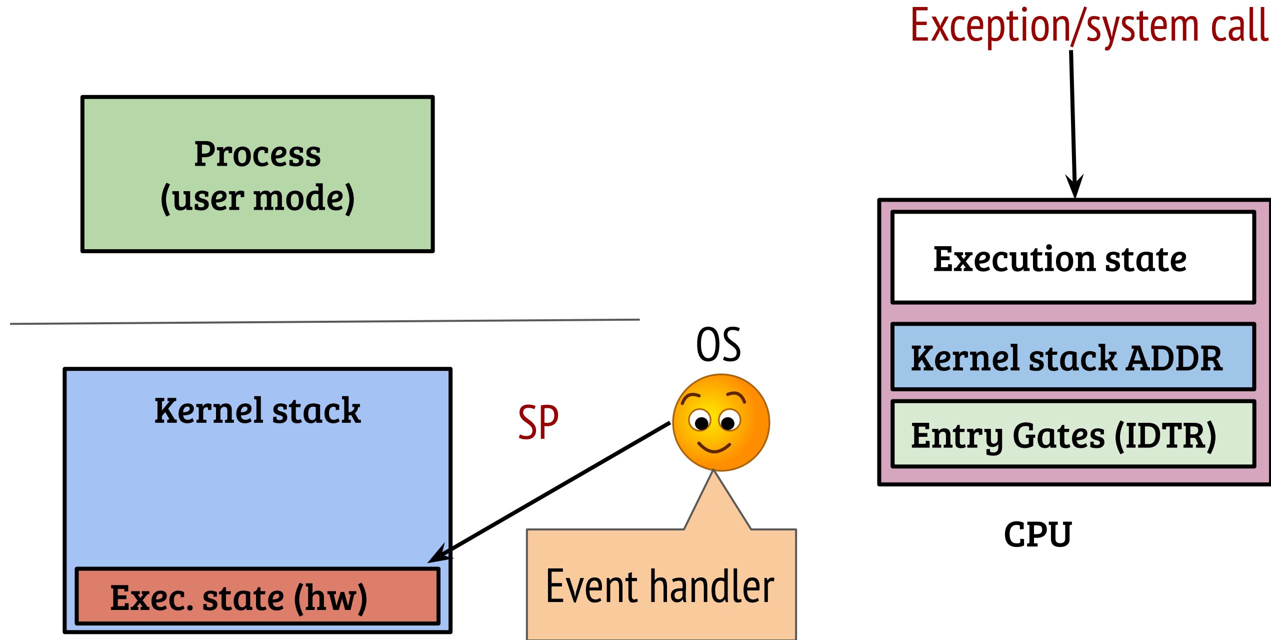
- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working
  - The OS configures the kernel stack address of the currently executing process in the hardware
  - For exceptions, the x86 hardware switches the stack pointer
  - For syscalls, the linux kernel entry handler switches the SP

User-kernel context switch



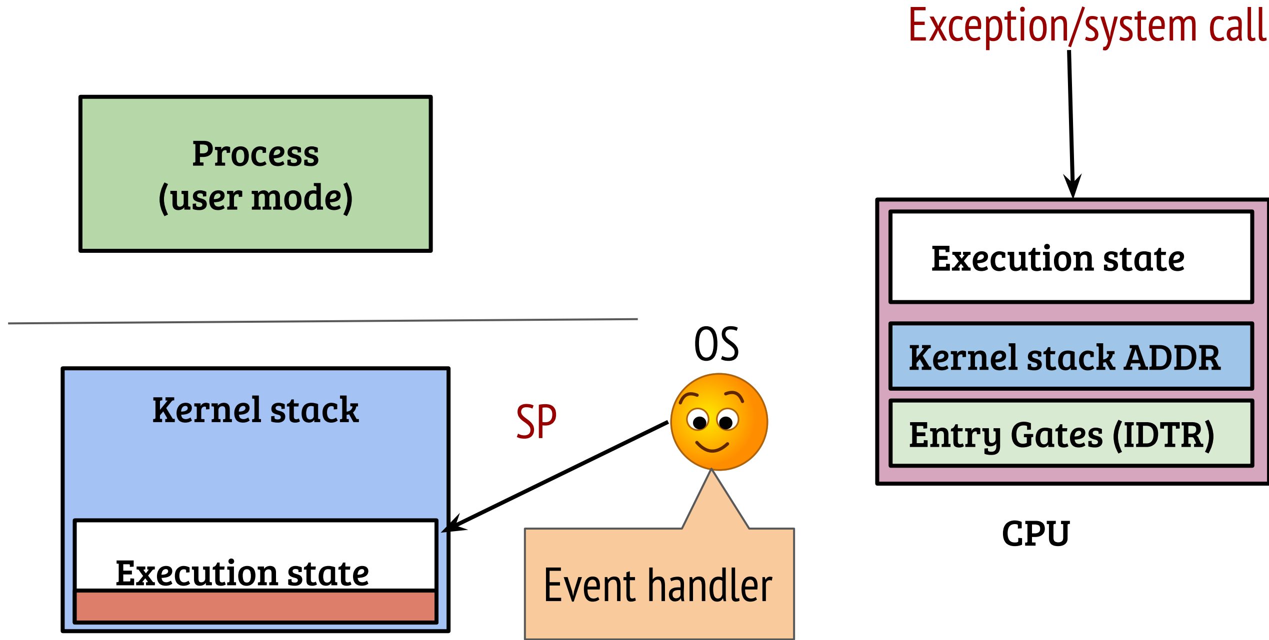
- The OS configures the kernel stack of the process before scheduling the process on the CPU

User-kernel context switch



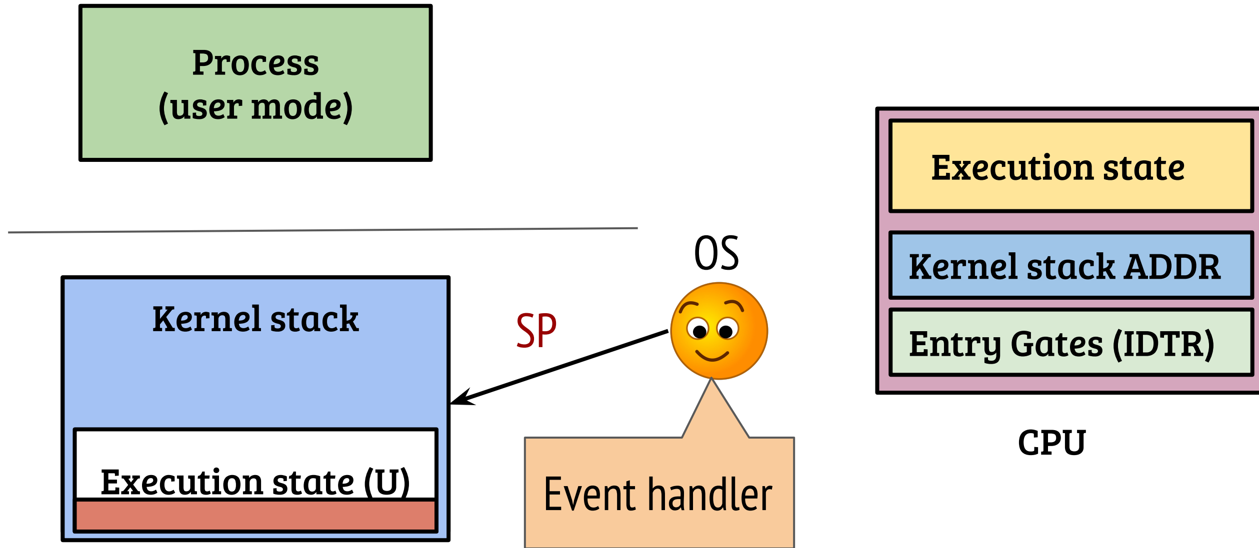
- The CPU saves a minimal execution state onto the kernel stack for entry though IDT defined events in x86

User-kernel context switch



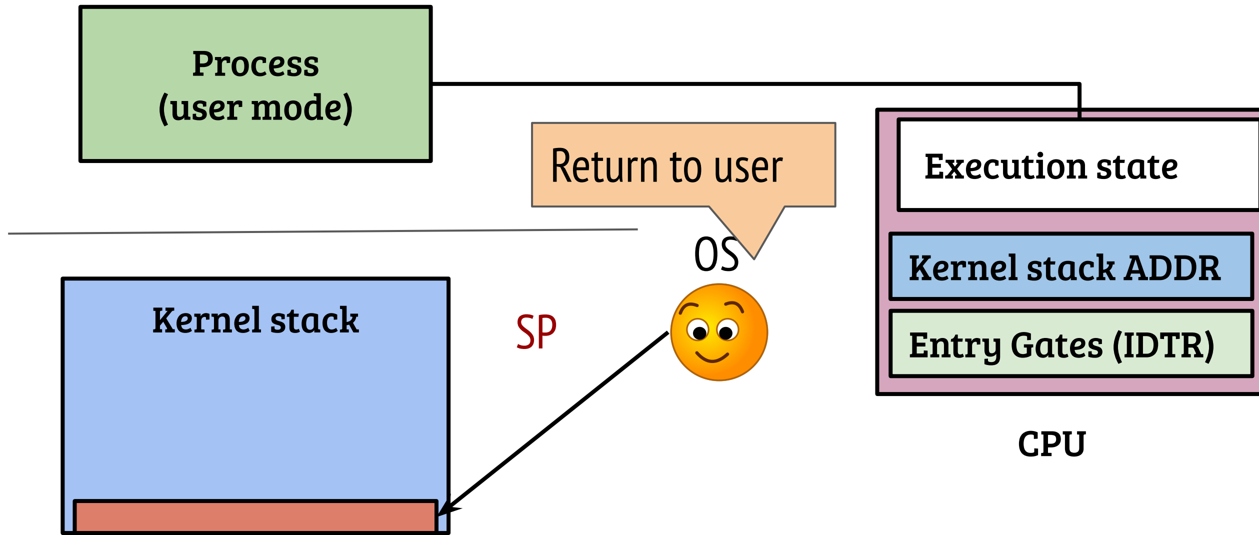
- The CPU (and/or OS) saves the execution state onto the kernel stack
- The kernel handler points a “struct pt_regs” type into the stack—can be accessed for any task using “task_pt_regs(task)”

User-kernel context switch



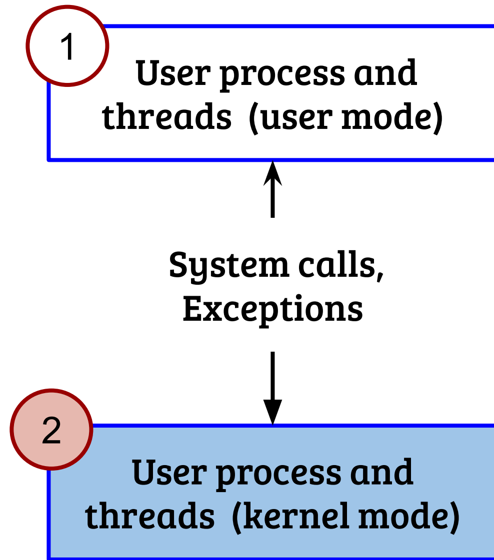
- The OS executes the event (syscall/exception) handler
  - Makes uses of the kernel stack
  - Execution state on CPU is of OS at this point

User-kernel context switch



- The kernel stack pointer should point to the expected position (in x86)
- CPU loads the user execution state (saved by it onto the kernel stack and resumes user execution)

User contexts

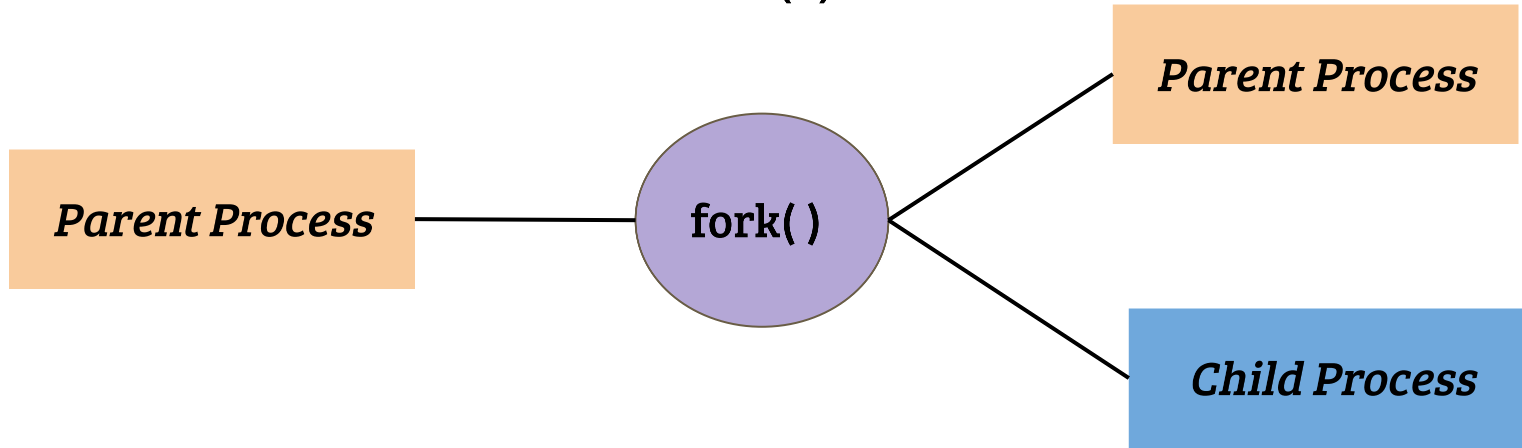


- What are the differences in execution states? Priv., stack, address space (partially or completely)
- What is the exact entry and exit mechanisms—user to kernel context switch and vice-a-versa? Handshake with hardware—IDT and syscall entry configurations
- What is the need for save and restore of the execution states? How implemented in Linux kernel? Correct execution, saved using the kernel stack
- Access/modification of user execution state from the kernel mode, how? Can be accessed from the kernel stack (memory state access require special care TBD)
- How the kernel manages the user contexts?

Quiz

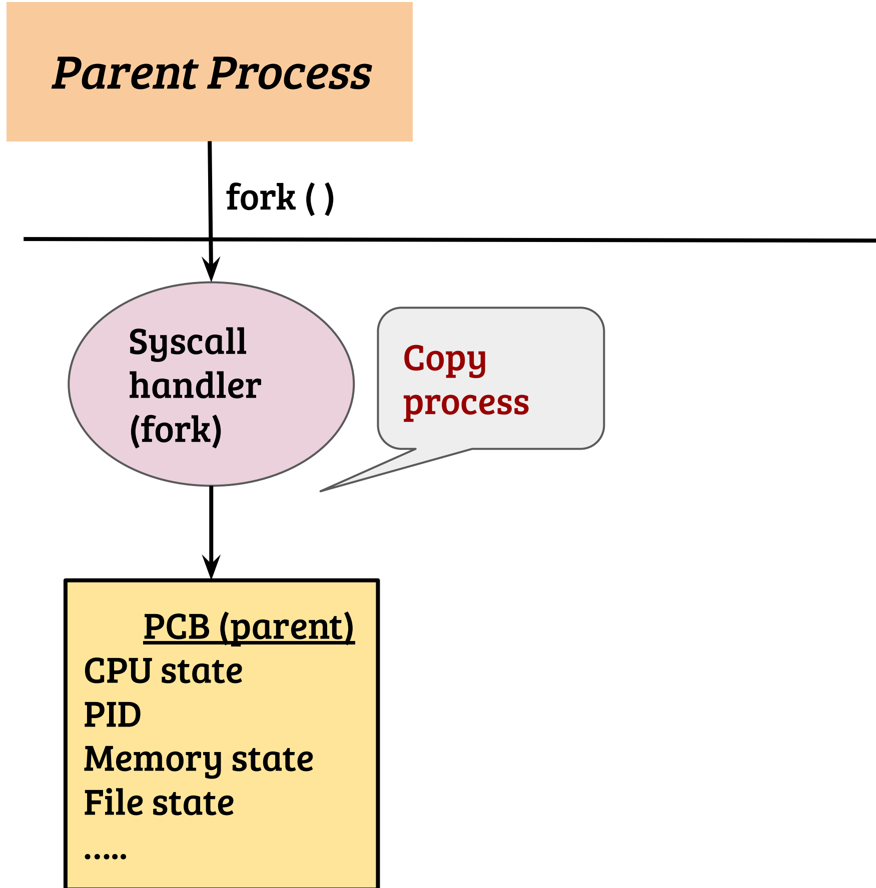
- Modifying the user state from kernel for fun!
- Download the quiz
  - Contains a module (simplecdev.c) and a user program (testcdev.c)
  - Run “make” to compile both
- Task 1: In line#43 of testcdev.c, depending on the assignment, the behavior of the program should change as follows,
  - If assigned to NULL, normal execution flow will occur
  - If assigned to “&only_one_read”, “Read successful” will not be printed
- Task 2: Will the solution work if replace the char device by a sysfs interface? Prove by evidence!

Process creation - fork( )

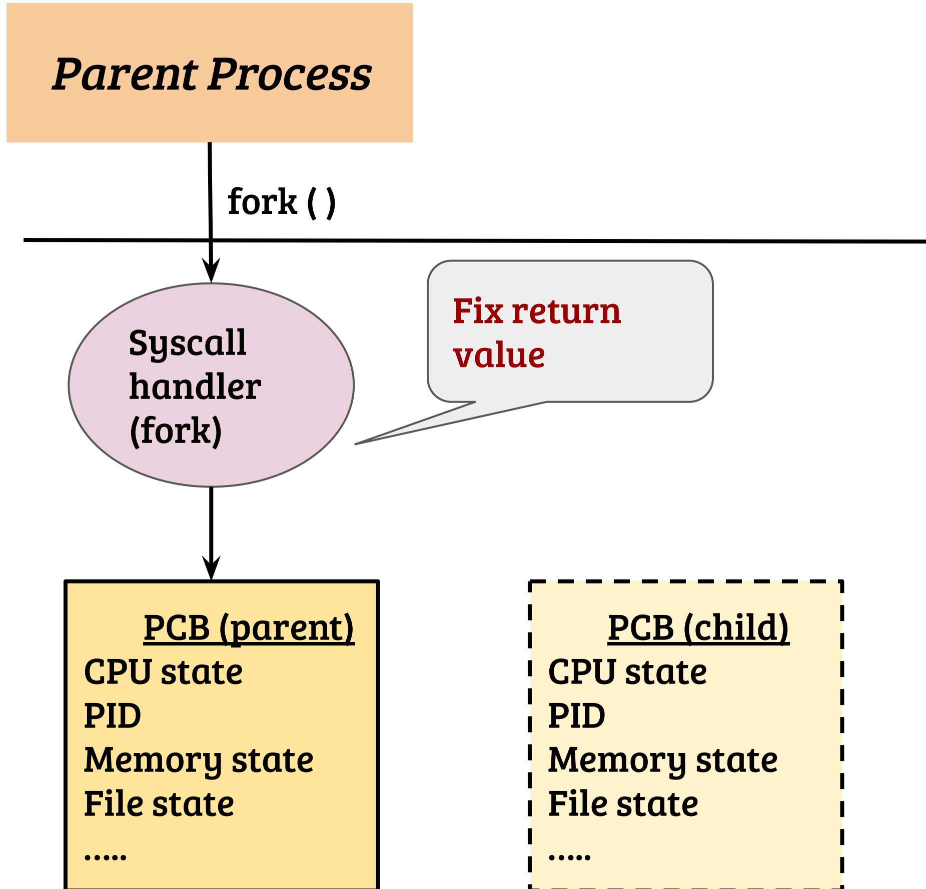


- fork( ) system call is weird; not a typical "privileged" function call
- fork( ) creates a new process; a *duplicate* of calling process
- On success, fork
  - Returns PID of child process to the caller (parent)
  - Returns 0 to the child

Typical implementation of fork

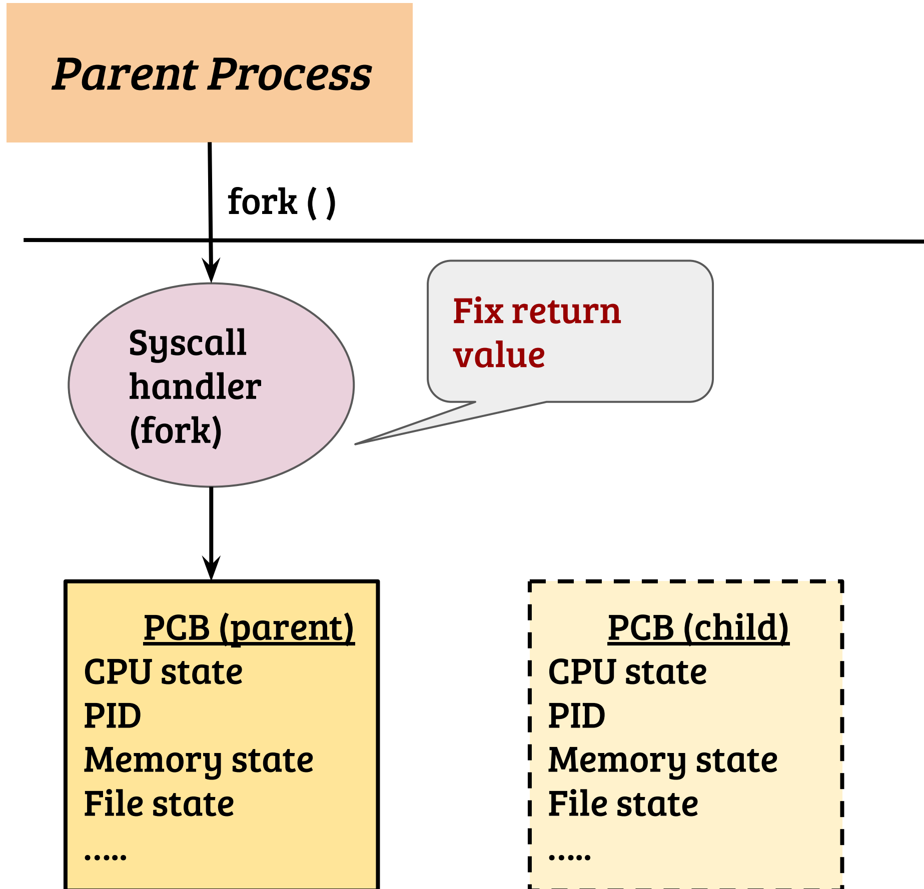


Typical implementation of fork



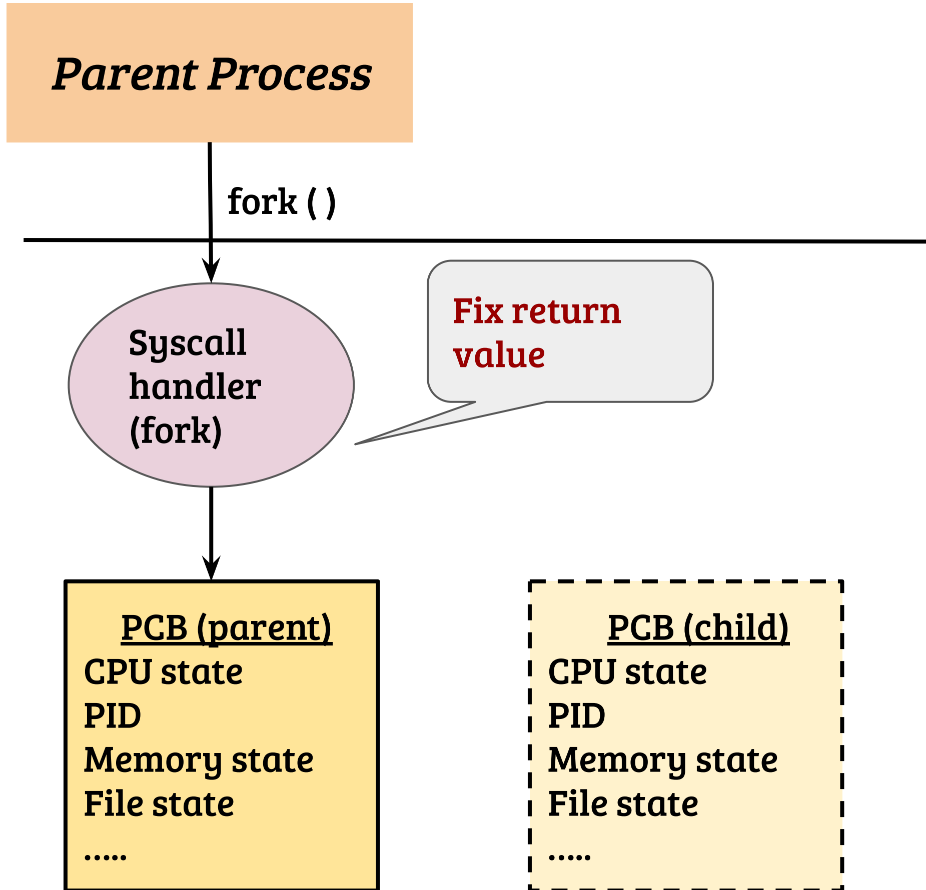
- Child should get '0' and parent gets PID of child as return value. How?

Typical implementation of fork



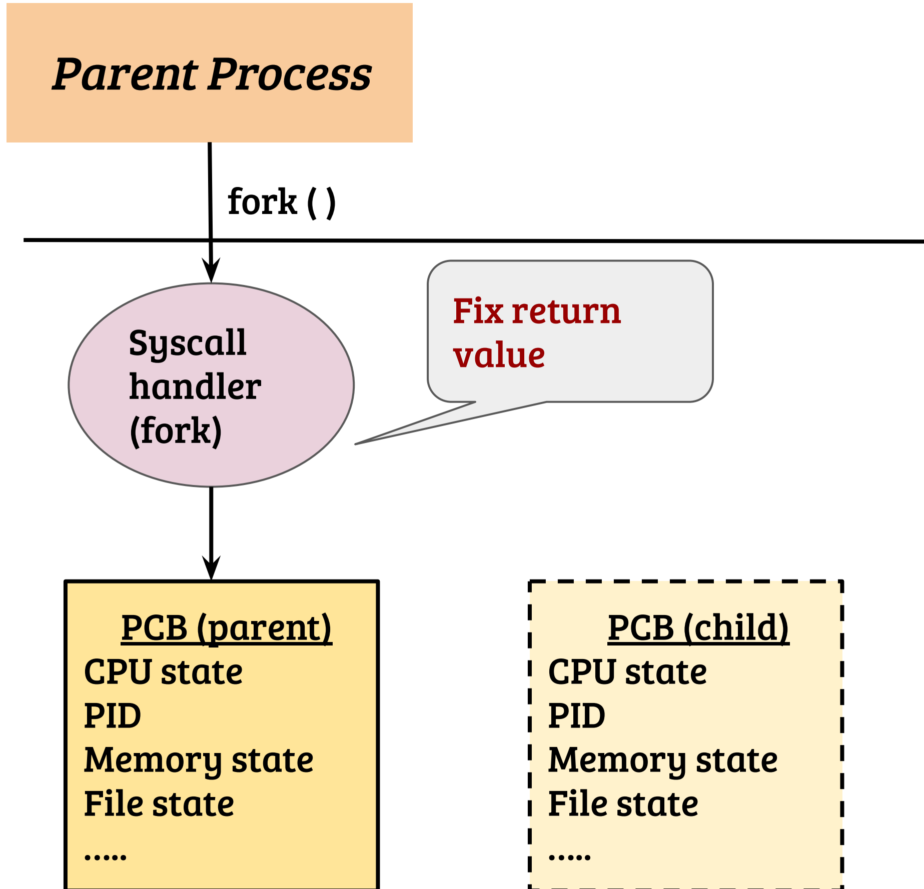
- Child should get ‘0’ and parent gets PID of child as return value. How?
- OS returns different values for parent and child

Typical implementation of fork



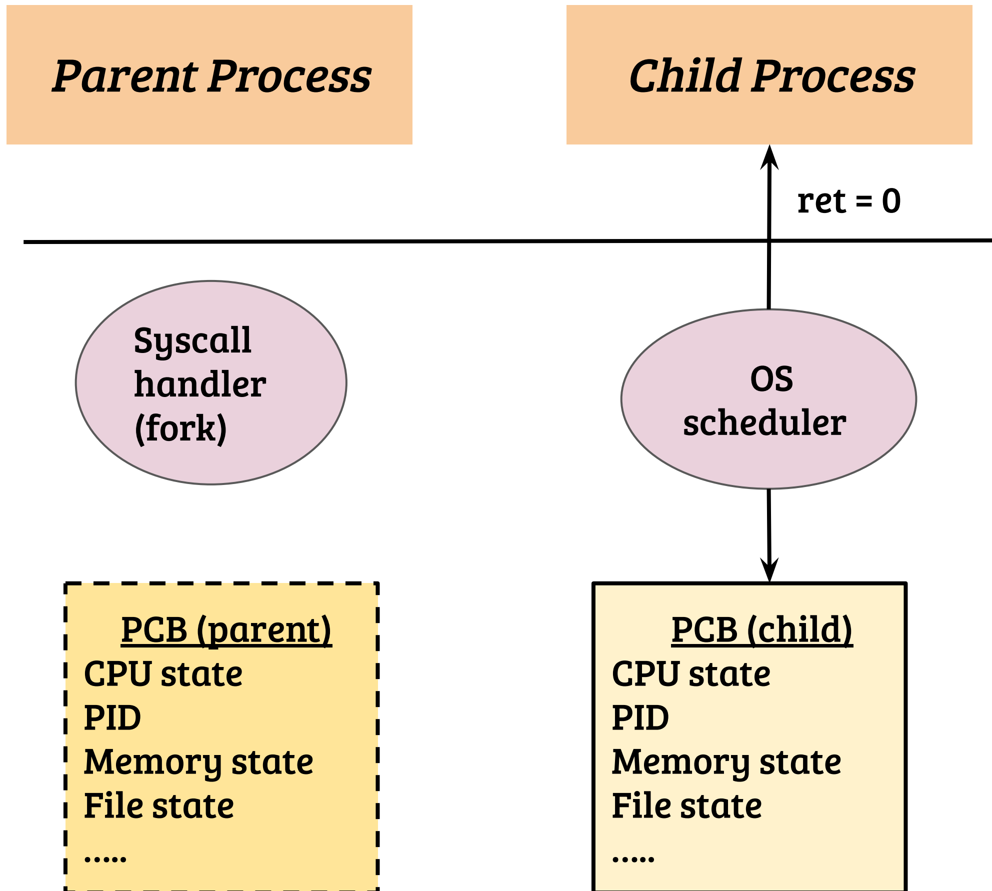
- Child should get '0' and parent gets PID of child as return value. How?
- OS returns different values for parent and child
- When does child execute?

Typical implementation of fork



- Child should get '0' and parent gets PID of child as return value. How?
- OS returns different values for parent and child
- When does child execute?
- When OS schedules the child process

Typical implementation of fork



- PC is next instruction after fork( ) syscall, for both parent and child
- Child memory is an exact copy of parent
- Parent and child diverge from this point

Fork and its variants

- The Linux kernel *fork* implementation is by default Copy-on-Write (CoW). In *CoW fork*, what all are copied from parent into the child task?
  - State of address space (Yes/No)
  - Page tables (Yes/No)
  - Physical frames (Yes/No)

Fork and its variants

- The Linux kernel *fork* implementation is by default Copy-on-Write (CoW). In *CoW fork*, what all are copied from parent into the child task?
  - State of address space (Yes/No) Yes, required to be copied (fork semantics)
  - Page tables (Yes/No) Yes, to allow parent and child operate independently
  - Physical frames (Yes/No) No, both page table entries marked read-only
- Linux provides *vfork( )* to reduce meta-data overheads when used with exec
  - Complete memory state remains shared till the child exits or execs
  - Useful to launch new executables efficiently
  - What is the catch?

Fork and its variants

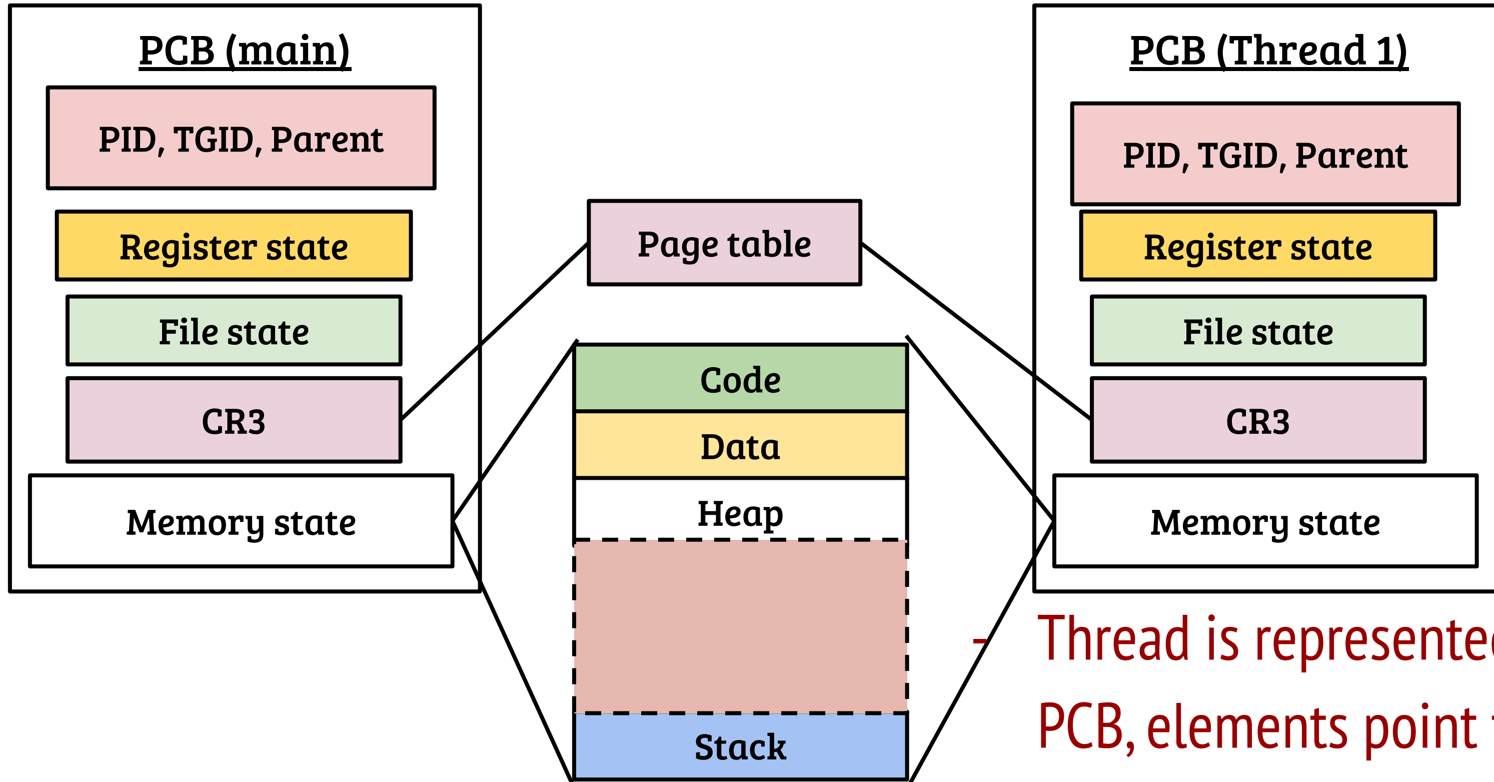
- The Linux kernel *fork* implementation is by default Copy-on-Write (CoW). In *CoW fork*, what all are copied from parent into the child task?
  - State of address space (Yes/No) Yes, required to be copied (fork semantics)
  - Page tables (Yes/No) Yes, to allow parent and child operate independently
  - Physical frames (Yes/No) No, both page table entries marked read-only
- Linux provides *vfork( )* to reduce meta-data overheads when used with exec
  - Complete memory state remains shared till the child exits or execs
  - Useful to launch new executables efficiently
  - What is the catch? The user space should be careful not to allow the parent memory to be corrupted, especially the user space stack can be tricky!

User threads using posix thread API

```
int pthread_create( pthead_t *tid, pthread_attr_t *attr,
                    void * (*thfunc) (void*), void *arg);
```

- Creates a thread with "tid" as its handle and the thread starts executing the function pointed to by the "thfunc" argument
- A single argument (of type void *) can be passed to the thread
- Thread attribute can be used to control the thread behavior e.g., stack size, stack address etc. Passing NULL sets the defaults
- Returns 0 on success.
- Thread termination: return from thfunc, pthread_exit( ) or pthread_cancel( )
- In Linux, pthread_create and fork implemented using clone( ) system call

PCB of a multithreaded process (Linux)



- Thread is represented by a separate PCB, elements point to the structure containing subsystem level info.

The clone system call

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...)
```

- Parent can control the execution of new process (execution and stack)
- Provides flexibility to the parent to share parts of its execution context in a selective manner
- Examples flags:
    - CLONE_FILES: Share files between parent and new process
    - CLONE_VM: Share the address space
    - CLONE_VFORK: Execution of parent process is suspended

Clone: Implementation in Linux kernel

- Syscall handler for clone should provide flexible sharing. Implementation?

Clone: Implementation in Linux kernel

- Syscall handler for clone should provide flexible sharing. Implementation?
  - Syscall Handler → Kernel clone → Copy process
  - Depending on flags, different subsystems are copied or shared
- Depending on the usage, the saved user state is required to be changed. Why? How implemented?

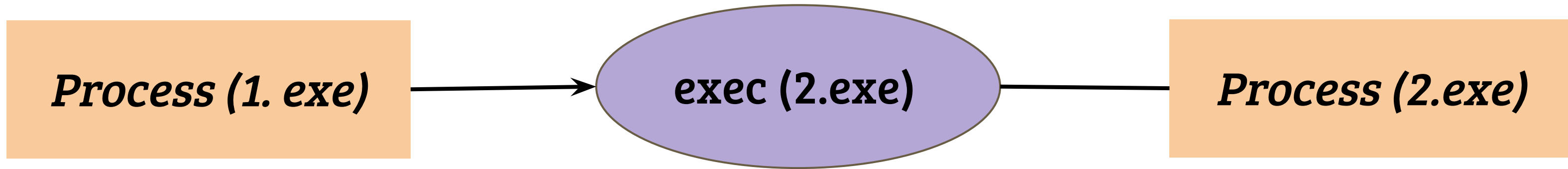
Clone: Implementation in Linux kernel

- Syscall handler for clone should provide flexible sharing. Implementation?
  - Syscall Handler → Kernel clone → Copy process
  - Depending on flags, different subsystems are copied or shared
- Depending on the usage, the saved user state is required to be changed. Why? How implemented?
  - For pthreads, the SP and RIP need to be changed
  - Change the register states during CPU thread copy
- Changes to the kernel space of newly created execution context required. Why? How implemented?

Clone: Implementation in Linux kernel

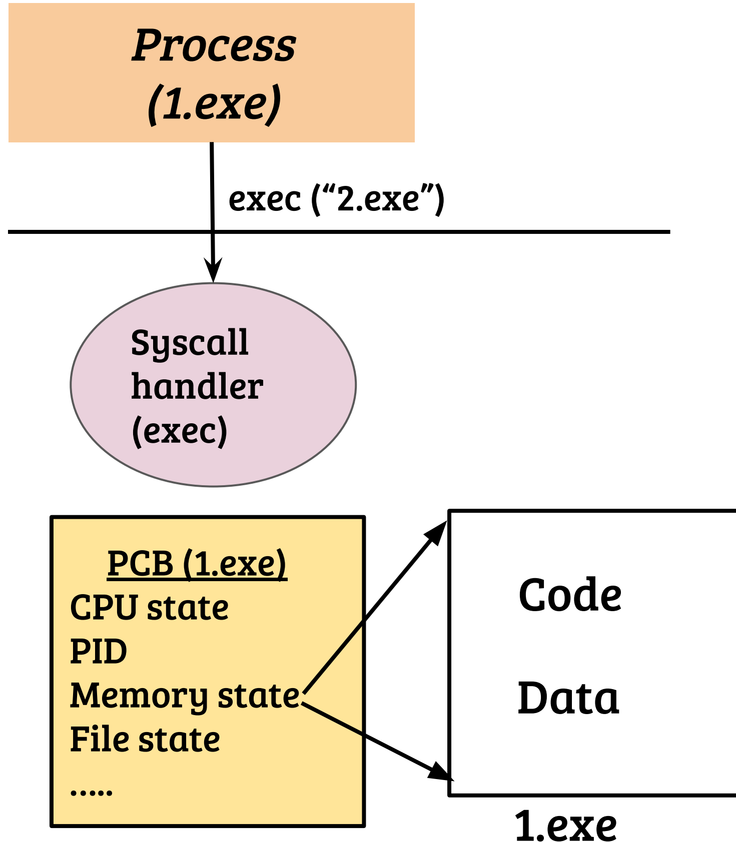
- Syscall handler for clone should provide flexible sharing. Implementation?
  - Syscall Handler $\longrightarrow$ Kernel clone $\longrightarrow$ Copy process
  - Depending on flags, different subsystems are copied or shared
- Depending on the usage, the saved user state is required to be changed. Why? How implemented?
  - For pthreads, the SP and RIP need to be changed
  - Change the register states during CPU thread copy
- Changes to the kernel space of newly created execution context required. Why? How implemented?
  - Child can not return in the same path, returns through a special stub

Load a new binary - exec( )



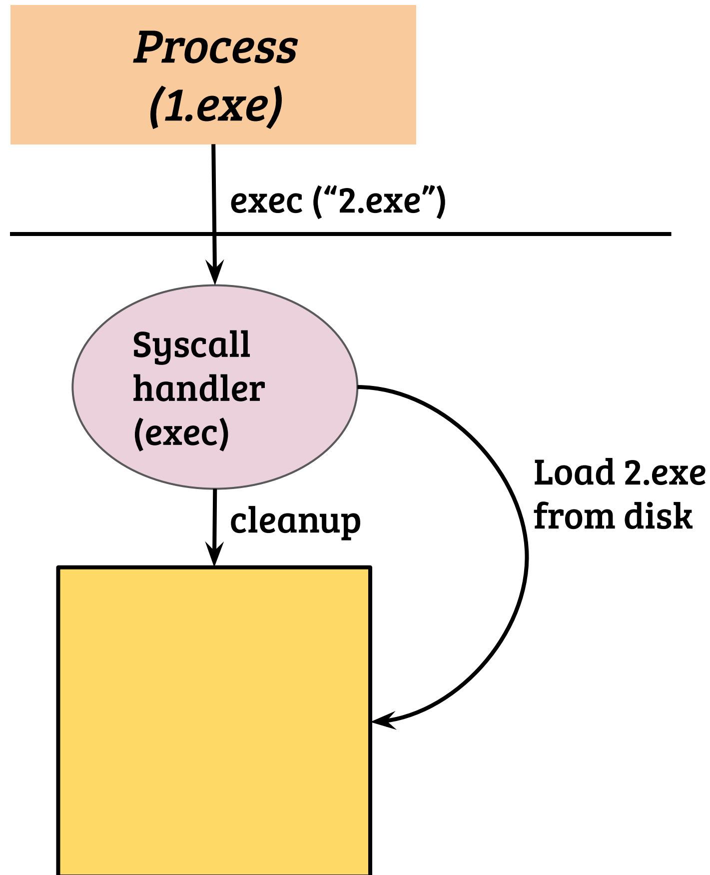
- Replace the calling process by a new executable
  - Code, data etc. are replaced by the new process
  - Usually, open files remain open

Typical implementation of exec



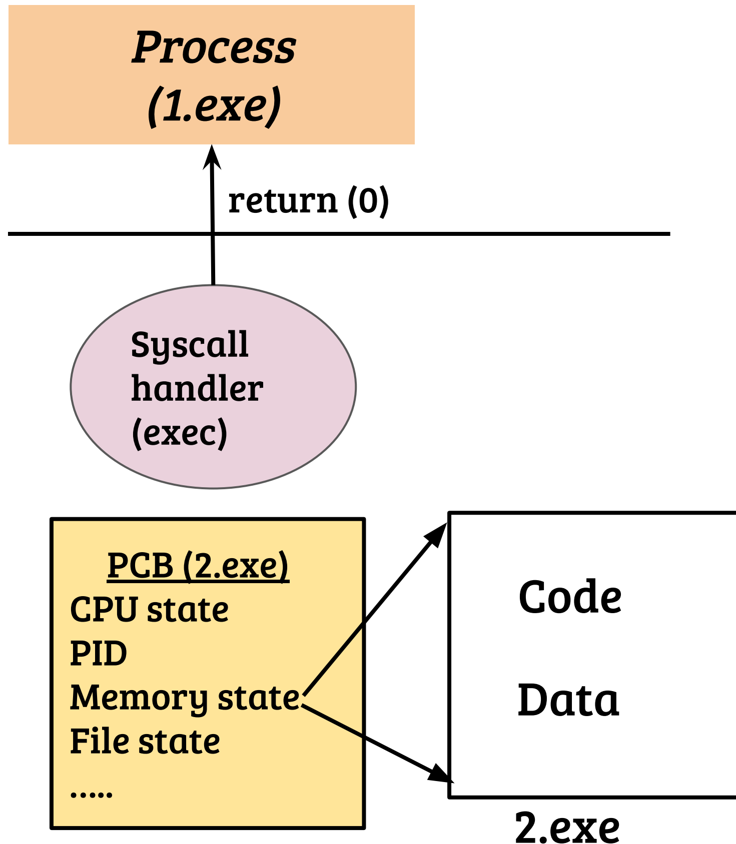
- The calling process commits self destruction! (almost)

Typical implementation of exec



- The calling process commits self destruction! (almost)
- The calling process is cleaned up and replaced by the new executable
- PID remains the same

Typical implementation of exec



- The calling process commits self destruction! (almost)
- The calling process is cleaned up and replaced by the new executable
- PID remains the same
- On return, new executable starts execution
- PC is loaded with the starting address of the newly loaded binary

Exec: Implementation in Linux kernel

- When should the self destruction of address space take place? What are the design choices?

Exec: Implementation in Linux kernel

- When should the self destruction of address space take place? What are the design choices?
    - Can not destroy until validity is checked; validity check not complete until the binary/arguments are examined
    - Duplicated processing vs. working with a fresh (discardable) space
    - There would be a point of no return, delayed is better
- How does the kernel parse the binary (and deduce entry address)? What about command line arguments?

Exec: Implementation in Linux kernel

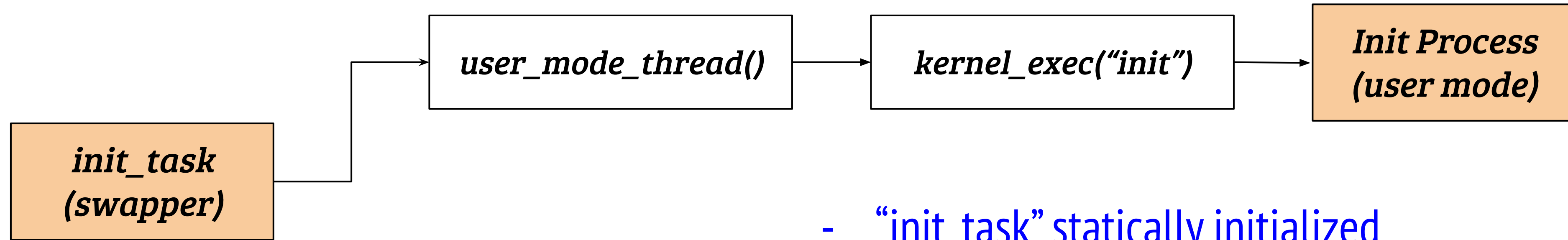
- When should the self destruction of address space take place? What are the design choices?
    - Can not destroy until validity is checked; validity check not complete until the binary/arguments are examined
    - Duplicated processing vs. working with a fresh (discardable) space
    - There would be a point of no return, delayed is better
- How does the kernel parse the binary (and deduce entry address)? What about command line arguments?
    - Basic binary parsing for ELF (and other types) e.g., load_elf_binary ( )
    - Command line arguments are placed in the stack

The first process

- What is the first execution entity in Linux?

The first process

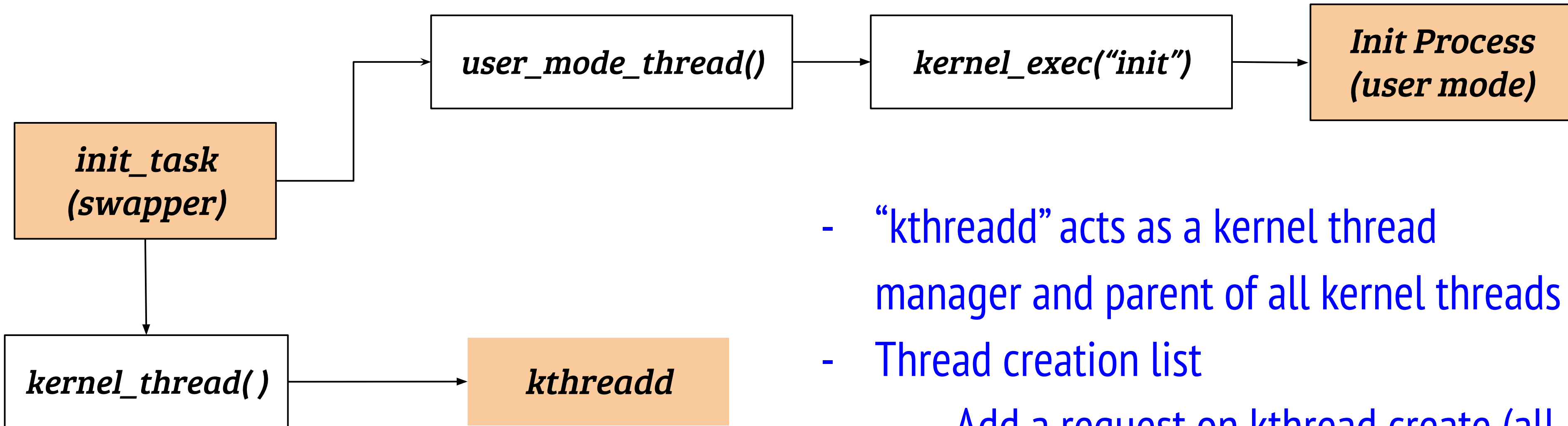
- What is the first execution entity in Linux?



- "init_task" statically initialized
- A special "clone" call from the kernel mode to create a thread of execution in kernel till actual init is executed
- Executes user space init based on configuration and default paths

The first process

- What is the first execution entity in Linux?



- "kthreadd" acts as a kernel thread manager and parent of all kernel threads
- Thread creation list
    - Add a request on kthread create (all types of kernel threads)
    - Wakeup kthreadd
    - Kthreadd ⟶ kernel_thread( )