Context execution

Alex Pajuelo and Juan Jose Costa

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

Monoprogrammed OS

- A monoprogrammed OS will only execute one program at the same time
- Processes and threads do not exist, only a program in execution
- Example: MSDOS
- Pros: Simple
- Cons: not able to fully exploit the hardware resources
- Solution: multiprocess/multithreading

Multiprogrammed OS

- A multiprogrammed OS can have several processes and threads alive at the same time
- It is not the same “alive” and “in execution”
  - “Alive” means that it exists
  - “In execution” means that it is alive and using the CPU
- The OS maintains structures to hold information of the processes and threads alive
- From the point of view of the OS, a process is a Process Control Block (PCB)
- From the point of view of the OS, a thread is a Thread Control Block (TCB)

Definitions

- Thread: basic scheduling unit
  - It is the code in execution
- Process: container of resources
  - A thread asks the OS for resources, but the OS allocates those resources to the process the thread belongs to
- A process can have more than one thread
- A thread can only belong to one process
- Processes do not share threads and threads do not move from one process to another

Processes as a resource container

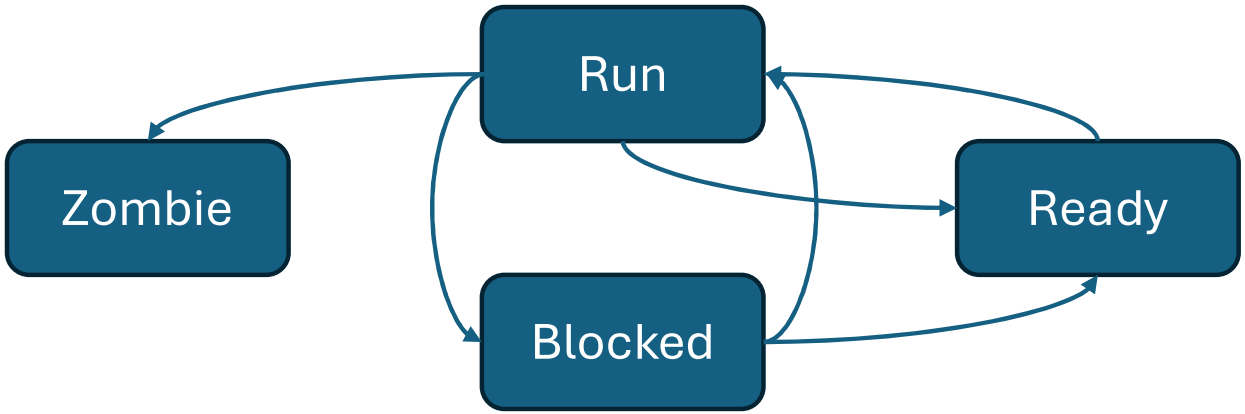
- All resources are assigned to processes
- Possible resources:
  - PID: Process IDentifier
  - Memory: the PT is assigned to the process
  - File descriptors: the File Descriptor Table is assigned to the process (review OS course)
  - Signals: (review OS course)
  - Threads
  - …
- All resources are shared among threads belonging to the same process

Threads as execution units

- Threads only need to store the information required to be executed:
  - TID: Thread IDentifier
  - Execution context: explained later
  - User stack
  - System stack
  - Thread Local Storage (TLS): small memory assigned to a thread to store particular data. In our case: errno
- Errno is local to a thread because threads could execute system calls and, in case of error, they may check the error code

Life cycle

- To have more than one thread alive and/or in execution, the OS defines a life cycle
- The life cycle represents in what states a thread can be when alive
- Very dependent on the OS features
- Linux life cycle:



Life cycle (and II)

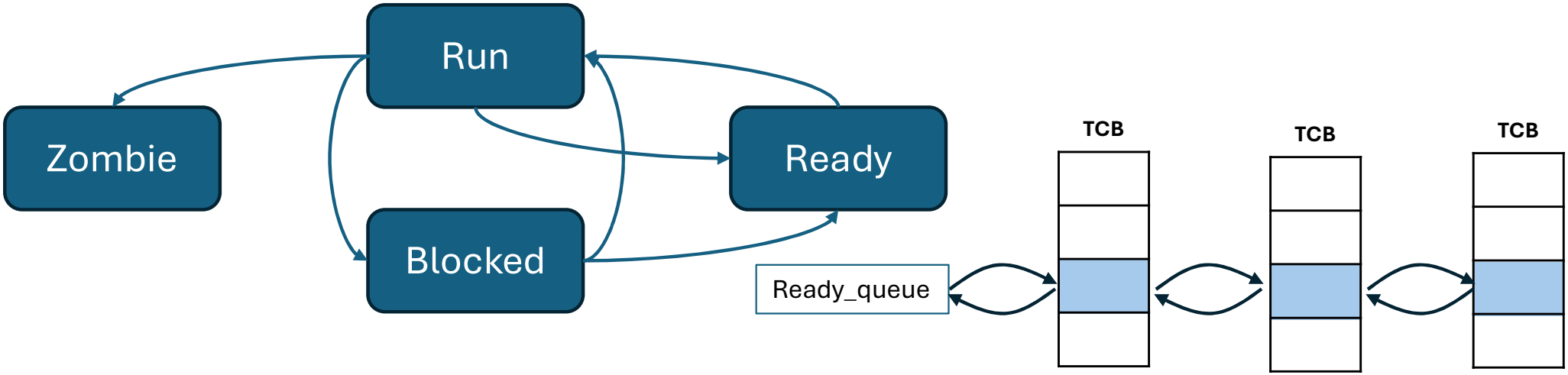
- Usually, states are for threads not for processes
- A thread can be only in one state but there can be more than one thread in the same state
- Transitions between states are defined from OS features
- We will work with 4 states:
  - Ready
  - Run
  - Blocked
  - Zombie
- All states are represented by queues
- This means that we know that a thread is in a particular state because its TCB is enqueued in a queue
  - Except for the RUN state

Ready state

- When a thread is in the Ready state it means that it can continue execution
  - There will be no difference, from the OS point of view, between a thread that CAN CONTINUE execution and a thread that CAN START execution
- The ready state is represented by the Ready queue
- A thread is in the Ready state when its TCB is enqueued in the Ready queue
- Only threads enqueued in the Ready queue are candidates to execute in the CPU (Run state)

Ready queue

- The ready queue is the structure of the OS in which all the TCBs of threads that can continue execution are enqueued
- The short-term scheduler only analyzes the ready queue when looking for a thread to pass to the RUN state
- There can be more than one in the OS

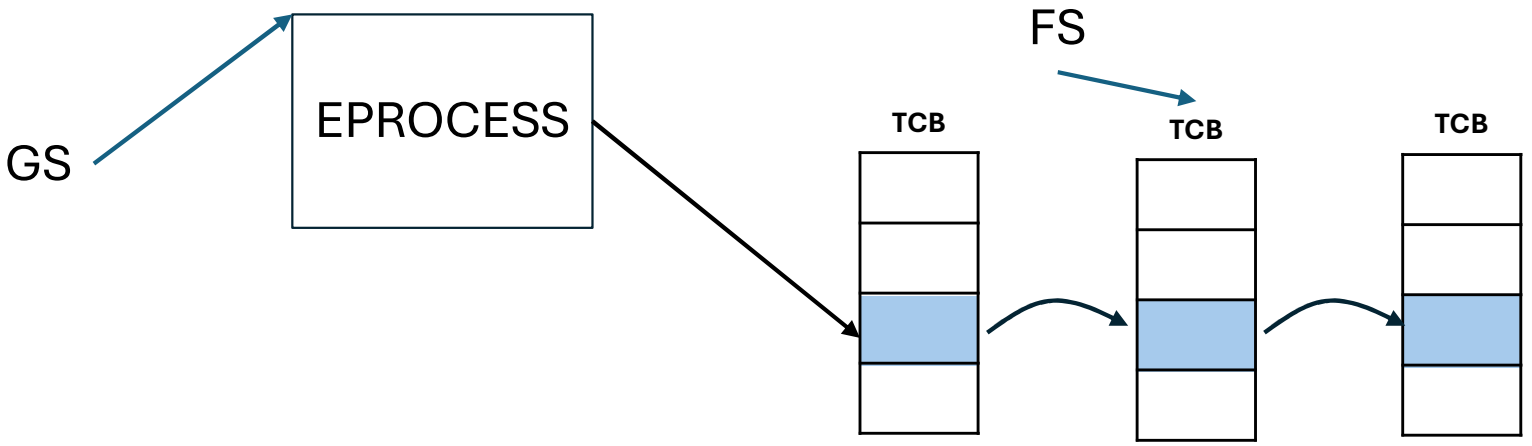


Run state

- A thread that is executing (using the CPU) is in the Run state
- There can be as many threads in the Run state as CPUs are present in the hardware
- The TCB of a thread in the Run state is not queued anywhere and is called current
- But it must be accessed in OS mode to get information of that thread
- Discussion: how to set up structures to access a TCB efficiently

Windows PCB and TCB

- In Windows, there is an explicit relationship between PCBs (called EPROCESS) and TCBs (called ETHREADs)
- The PCB includes a list of TCBs



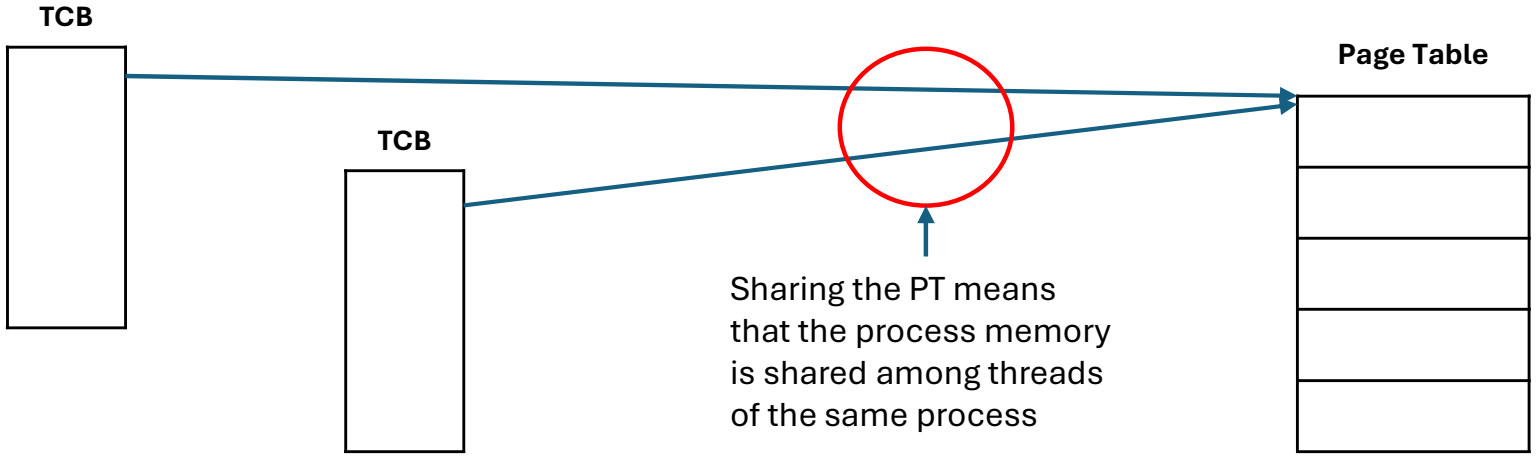
- The GS register always points to the current EPROCESS
- The FS register always points to the current ETHREAD

Accessing Windows PCBs and TCBs

- To access a field inside those structures, use a direct offset:
  - GS:[0]
  - FS:[0x34e]
- The Visual Studio compiler will never use the GS and FS registers when generating assembler code

Linux PCB and TCB

- In Linux, there is not any structure to hold the information of a process. In fact, a process is an abstraction.
- The TCB is called struct task_struct and holds the information of the thread and the process it belongs to
- Threads of the same process share the structures that hold information about resources of the process. This is how the abstraction of a process is created.



Accessing Linux TCBs

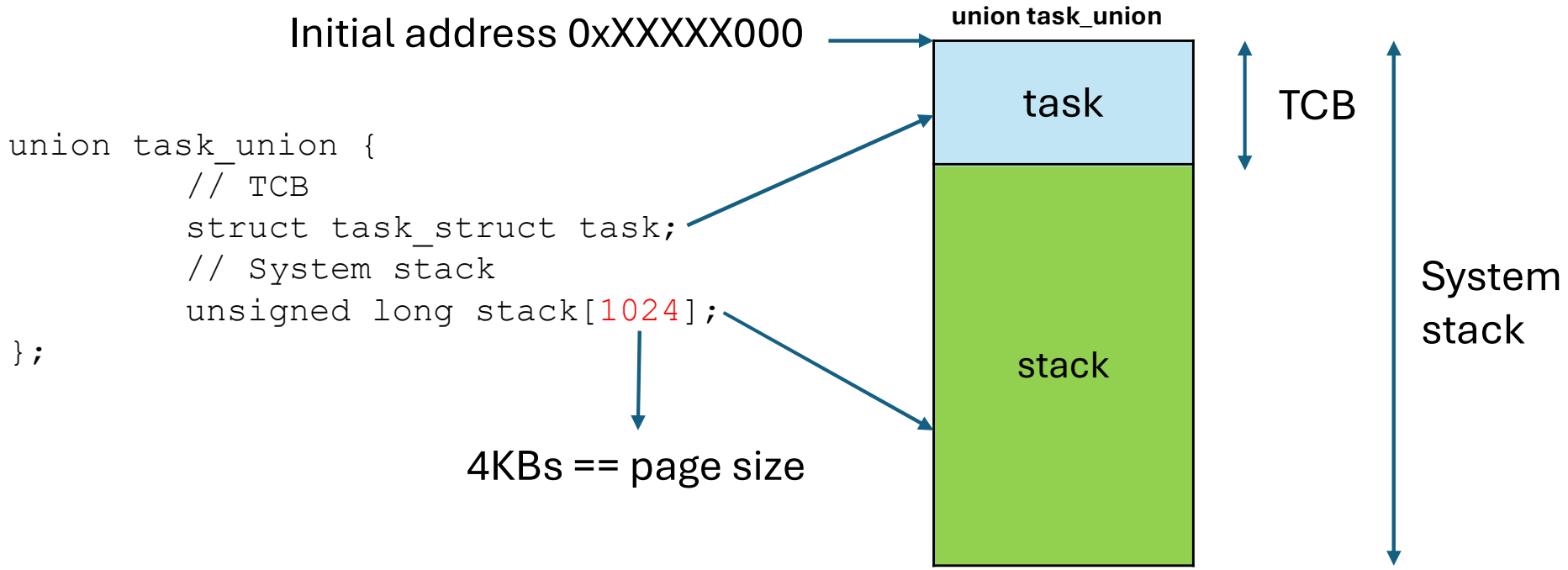
- The philosophy behind the access to the struct task_struct of current is somehow more elaborated that that in Windows but the access time is nearly the same
- The typical definition of a struct task_struct:

```
struct task_struct {
        int PID;
        int TID;
        pagetable_entry *TP;
        …
};
```

- Remember that one thread has its own system stack

The TCB and the system stack

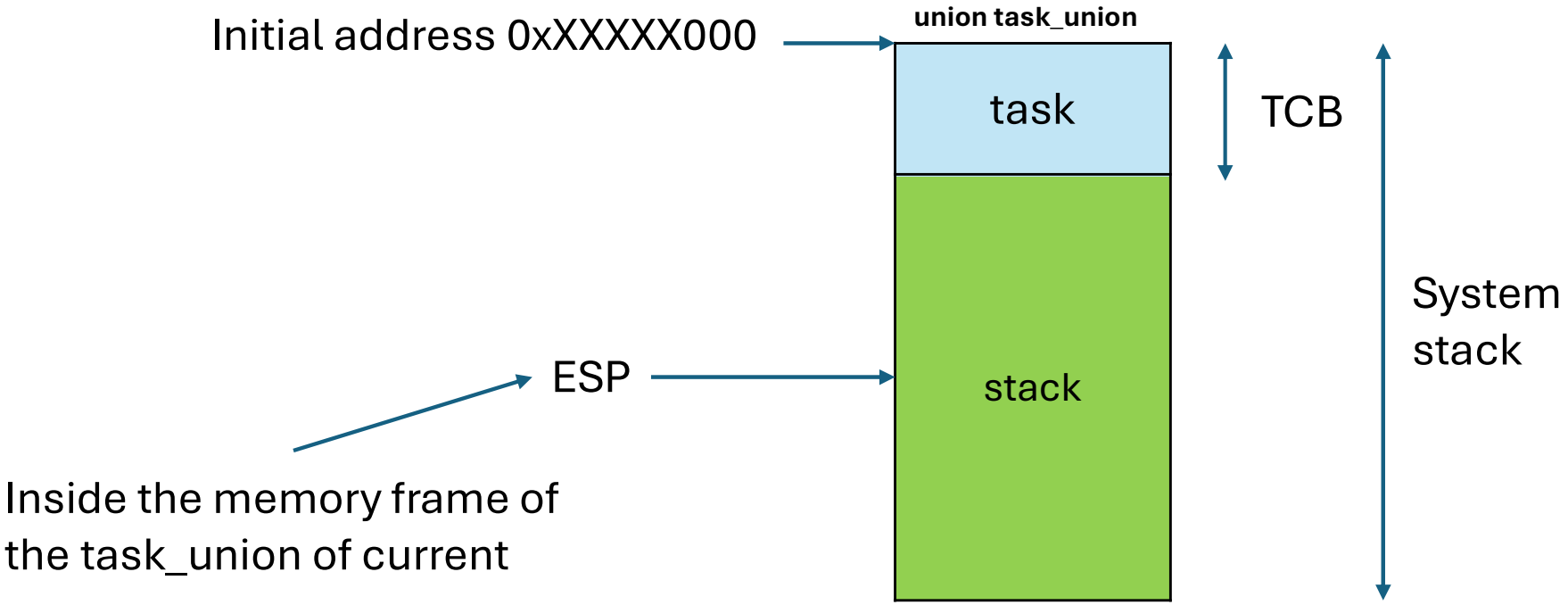
- In Linux, the TCB and the system stack are overlapped in memory in the union task_union structure:



- In addition, that union is allocated at the beginning of a memory page. Its initial address is always 0xXXXXX000

Finding current

- In addition, the ESP register is always pointing somewhere in the system stack of current



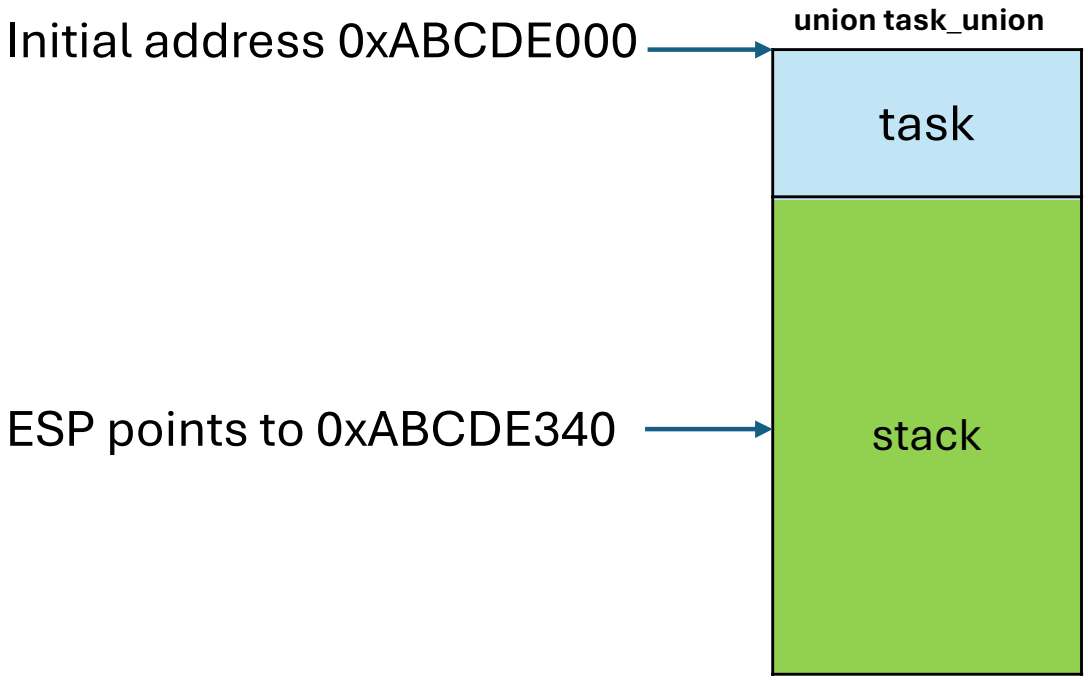
- To find the struct task_struct of current, you only need to clear the 12 least significant bits of ESP

Function struct task_struct *current()

- To find the address of the struct task_struct of current, implement the following function that returns the address of ESP aligned to the beginning of a memory frame (where the struct task_struct begins):

```
// struct task_struct *current()
current:
        movl %esp, %eax
        andl $0xFFFFF000, %eax
        ret
```

Example of finding the struct task_struct




```
ESP = 0xABCDE340
ESP & 0xFFFFF000 = 0xABCDE000
```

Casting union task_union and task_struct

- Notice that the union task_union and the struct task_struct, start in the same memory address
- To find the union task_union from the struct task_struct:

```
// Given that struct task_struct *t points to a struct task_struct
union task_union *u = (union task_union*)t;
```

- To find the struct task_struct from the union task_union:

```
// Given that union task_union* u points to a union task_union
struct task_struct *t = (struct task_struct*)u;
```

unsigned long stack[1024];

- The stack field of the union task_union provides a high-level view of the system stack as an array
- The system stack size must be always multiple of page size and aligned to the start of a page
- Remember that a stack starts from the bottom and finished at the top
  - So, the first used position of the system stack is [1023]
- There is not any mechanism to detect when the system stack overwrites the struct task_struct
  - To avoid this case, the OS programmer must always know the contents of the stack and how it grows up

Blocked state

- A thread in Run can pass to blocked state whenever it executes a long latency operation
  - I/O
  - Synchronization
  - Pause
  - Sleep
- The blocked state will be explained later when describing device access

Zombie state

- A thread passes to the Zombie state whenever it dies, or it is killed by the OS
- In Zombie, all resources allocated for the thread (or the process if the last thread of the process dies) are freed
- Only the TCB (or the TCB + PCB in Windows) remains
- The internals of this state will be explained later

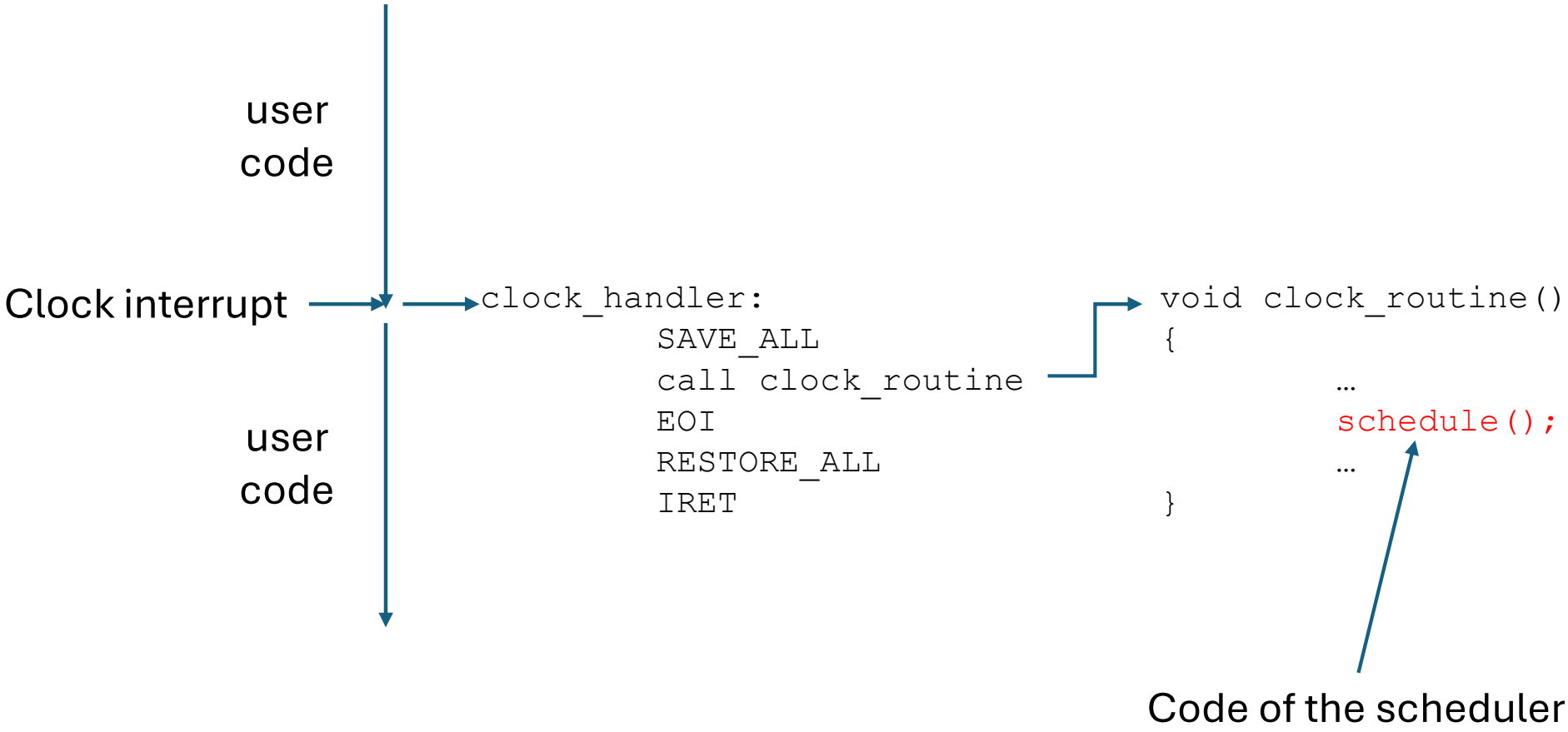
Short term scheduler

- The scheduler is responsible of moving threads from the Ready state to the Run state
- It decides when, how and who
  - Scheduler policy: defines when a thread passes to Run
  - Scheduler algorithm: define which thread passes to Run
- The scheduler can only pass to Run a ready thread
- Notice that when a thread passes to the Run state, in parallel a thread leaves the Run state
- So, moving one thread to Run always involves two threads
  - The thread that leaves Run is current
  - The thread that enters Run is new

Calling the short-term scheduler

- The short-term scheduler is called:
  - Every clock interrupt to update data
  - When a thread leaves the Run state
- Every clock interrupt, the scheduler is called to update current thread scheduling data and, if needed, preempt the CPU (put another thread in the CPU)
- When a thread leaves the Run state, the scheduler must be called to select another thread and move it to the Run state

Calling the short-term scheduler



Short-term scheduler code

- Even if it depends on the scheduling policy and algorithm, the usual code of the scheduler is:

```
void schedule()
{
        update_sched_data();
        if (needs_sched())
        {
                update_thread_state(current(), &ready_queue);
                sched_next();
        }
}
```

Updating scheduling data

- Every clock interrupt, the scheduling data of the current thread is update
- This is normal for a scheduling algorithm that defines CPU bursts (quantum) per thread
- A CPU burst defines how long a thread can stay in the CPU state before preempted
- Round-robin-based scheduling algorithms

```
void updata_sched_data()
{
        current()->remaining_quantum--;
}
```

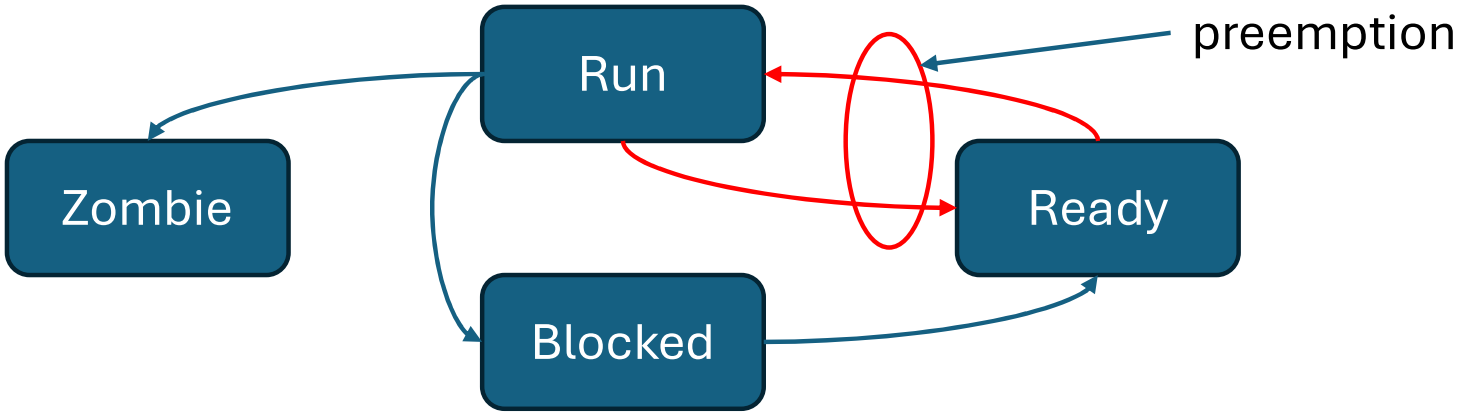
Checking scheduling data

- After updating the scheduling data, the scheduler checks if the current thread must be preempted
- For that, the scheduler checks if a given condition has been fulfilled
- For example, in Round Robin, if the quantum has expired

```
int needs_sched()
{
        return (current()->remaining_quantum == 0;
}
```

Preempting the CPU

- If the quantum of the current thread is expired, the OS preempts the CPU
- Preemption means that the current thread is forced to leave the CPU and a ready thread will move to Run state
- The thread leaving the CPU (current) is enqueued in the ready queue to keep on executing later



Preempting the CPU

- The update_thread_state function enqueues a thread (first parameter) in a given queue (second parameter)

```
void update_thread_state(struct task_struct *thread, struct list_head *queue);
```

- This function is executed when the state of a thread must be changed

- In this case:

```
update_thread_state(current(), &Ready_queue);
```

Preempting the CPU

- Even if the current thread is enqueued in the Ready queue, it is still current
- To make the current thread leave the CPU, the scheduler call sched_next
- Sched_next is responsible of:
  - Selecting the new thread to execute (new)
  - Switch to that thread (task_switch)
- Sched_next is the function that must be called when the current thread leaves the CPU
- Task_switch changes the current execution context

Selecting the next thread

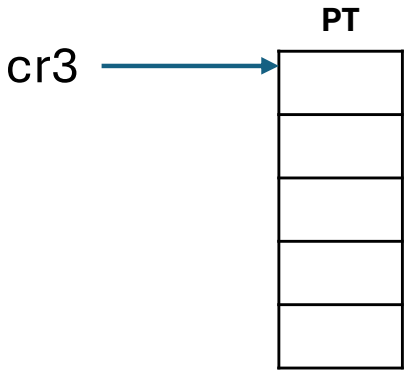
- Inside sched_next, a thread to enter into the CPU next must be selected
- For that, the scheduler selects one thread enqueued in the Ready queue
- Once selected, the struct task_struct of that thread is deleted from the Ready queue
- The selected thread depends on the scheduling policy and algorithm.

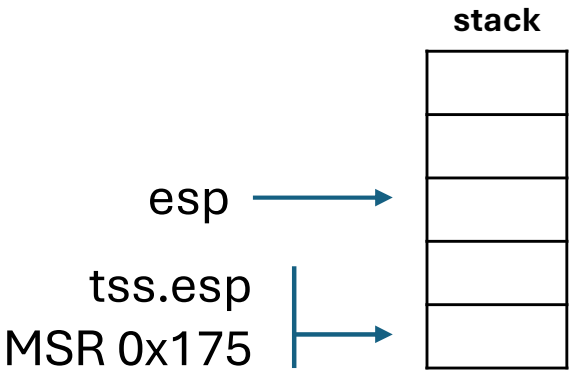
Switching the execution context

- Inside sched_next the execution context is switched to the one of thread new (the thread that will execute next)
- This execution context is performed in a function called task_switch
- Task_switch is the only function able to pass a thread to the Run state.
- All threads are moved to the Run state using task_switch
- A context switch is performed hundreds of times per second, so it must be very efficient!!!

Execution context

- When in privileged mode, the OS uses the execution context of the current thread
- This means that:
  - The address translation is performed with the PT of the current process (cr3 points to that PT)
  - The system stack in use is the one of current (ESP points to that system stack)
  - When there is a mode switch, from user to OS, tss.esp0 and MSR 0x175 point to the system stack of current





Function task_switch

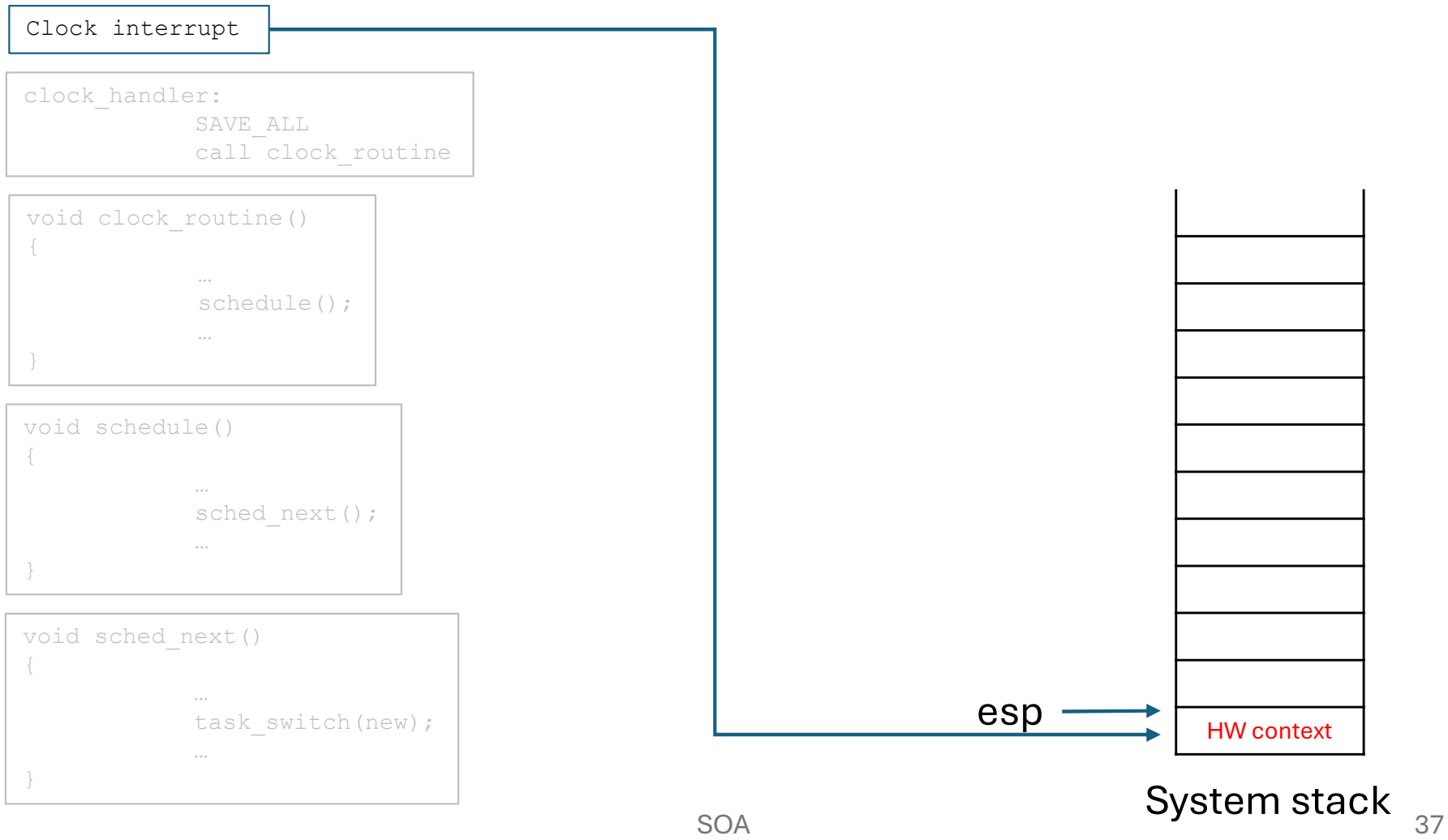
- Basically task_switch:
  - Stores the needed information to restore the execution context of new
  - Changes cr3 to point to the PT of the process the thread new belongs to
  - Tss.esp and MSR 0x175 must point to the system stack of thread new
  - ESP must point to the system stack of thread new
- Its header is:

```
void task_switch(struct task_struct *new);
```

- Where new is the struct task_struct * to the thread that enters the CPU
- It is important to know the execution flow until task_switch

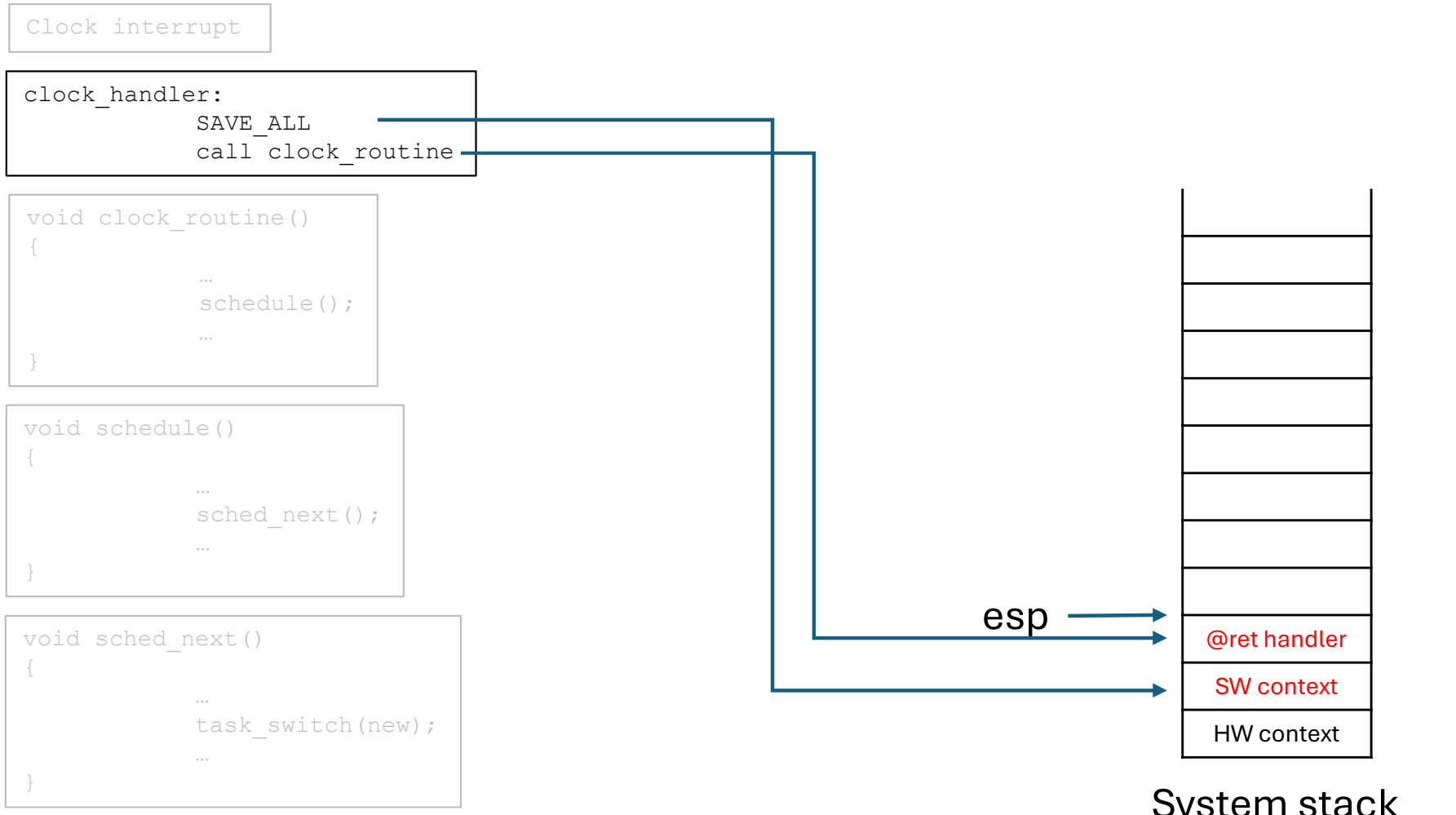
Execution flow until task_switch

- Remember that the scheduler is executed every clock interrupt



Execution flow until task_switch

- The handler pushes the software context and calls the clock_routine



Execution flow until task_switch

- The clock routine pushes ebp (frame pointer) and calls the scheduler

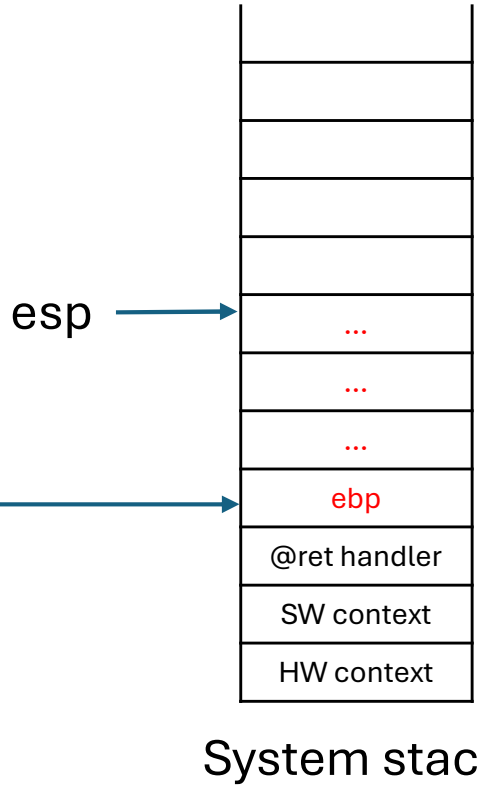
```
Clock interrupt
```

```
clock_handler:
            SAVE_ALL
            call clock_routine
```

```
void clock_routine()
{
            …
            schedule();
            …
}
```

```
void schedule()
{
            …
            sched_next();
            …
}
```

```
void sched_next()
{
            …
            task_switch(new);
            …
}
```



Execution flow until task_switch

- Finally, the task_switch is called. Task_switch pushes ebp (frame pointer) and moves esp to ebp

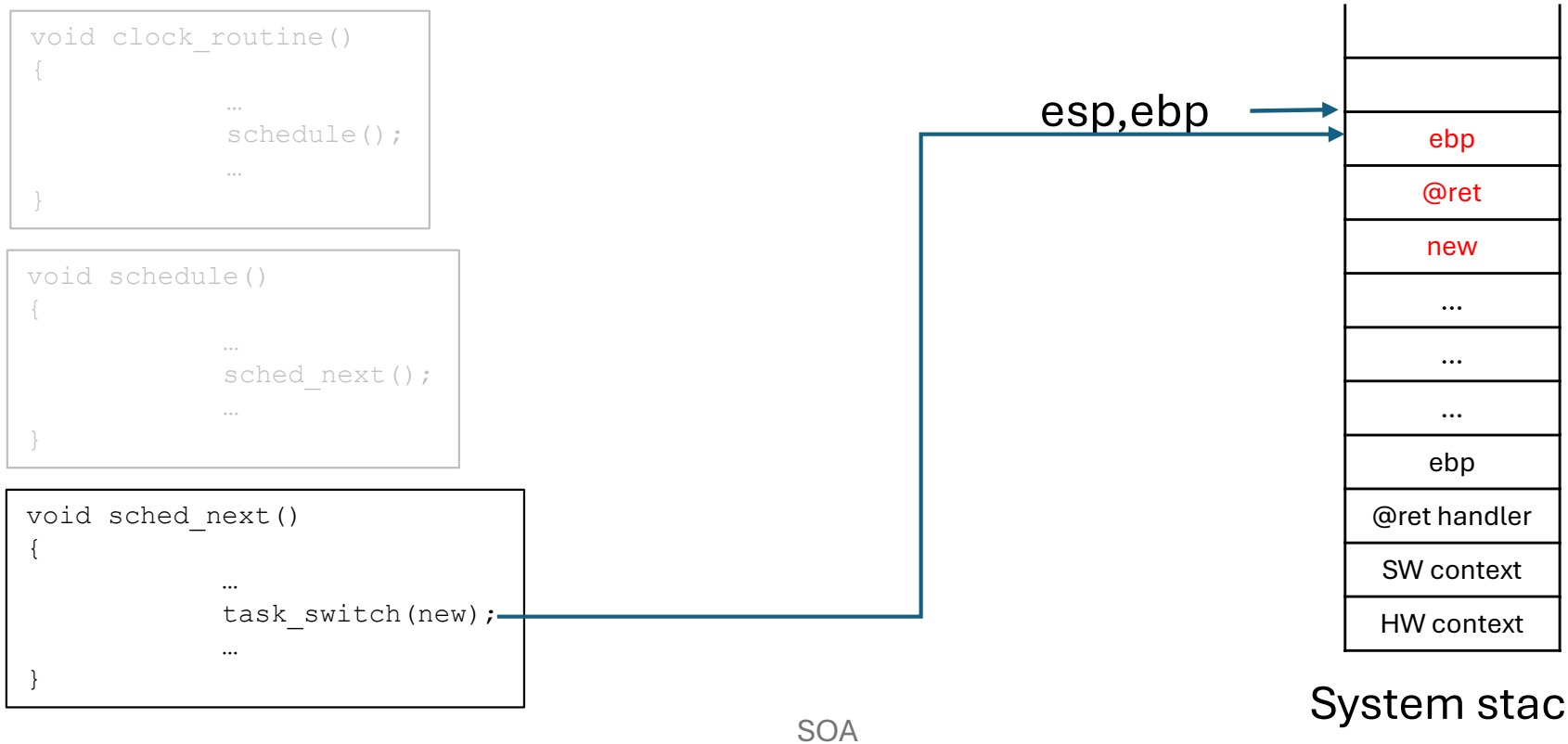
```
Clock interrupt
```

```
clock_handler:
            SAVE_ALL
            call clock_routine
```

```
void clock_routine()
{
            …
            schedule();
            …
}
```

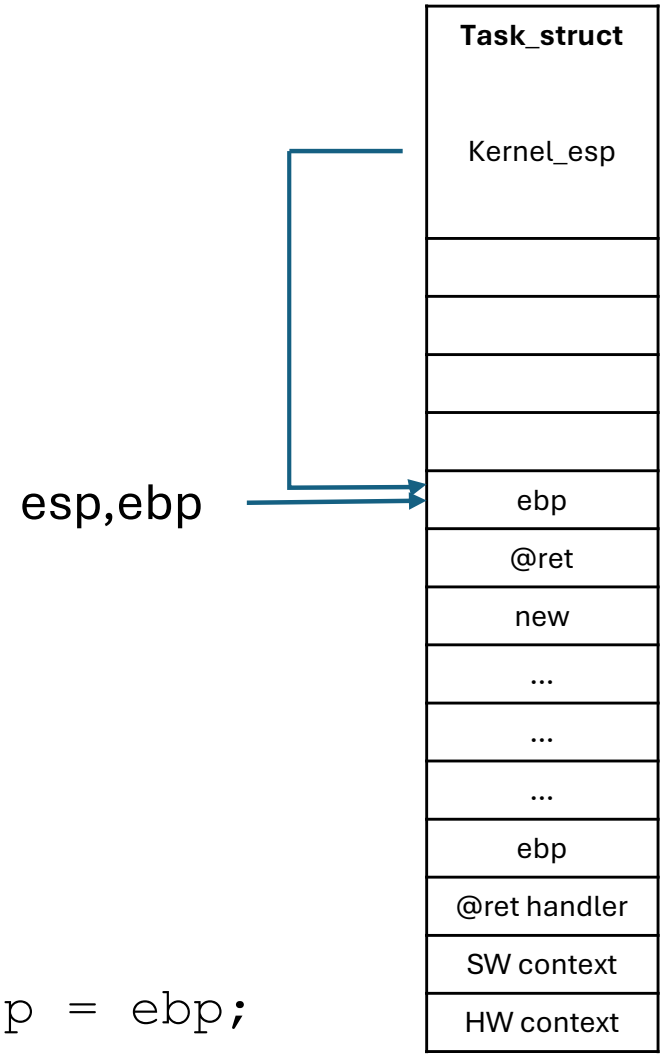
```
void schedule()
{
            …
            sched_next();
            …
}
```

```
void sched_next()
{
            …
            task_switch(new);
            …
}
```



void task_switch(struct task_struct *new)

- The first thing task_switch does is storing the needed information to restore the context execution of the current thread
- For that, a new variable called kernel_esp is added to the struct task_struct to save the value of the ebp register when task_switch is executing




```
current()->kernel_esp = ebp;
```

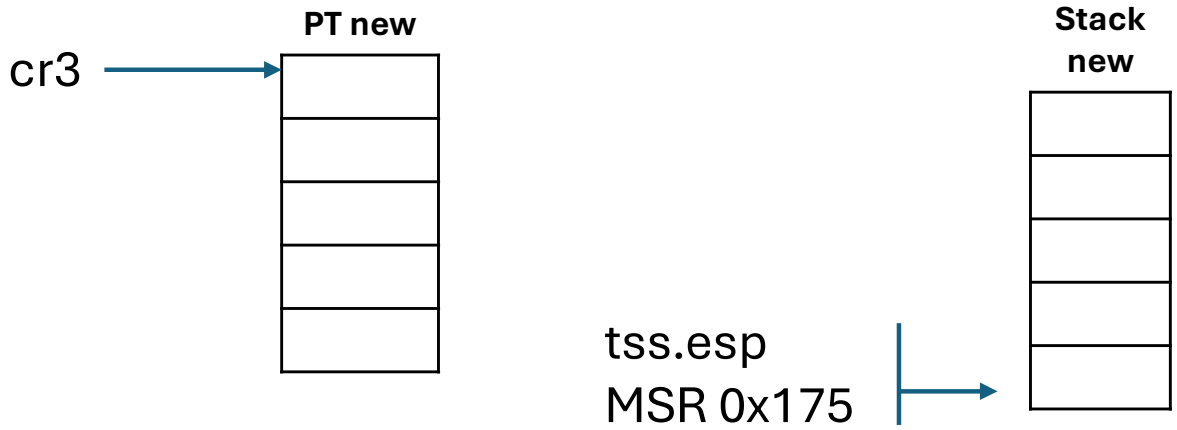
long *kernel_esp;

- Field of the struct task_struct that points to the ebp the function task_switch has pushed in the stack when executed
- Its value is only valid when the thread is in the ready state
- For the rest of states, its value is undetermined

void task_switch(struct task_struct *new)

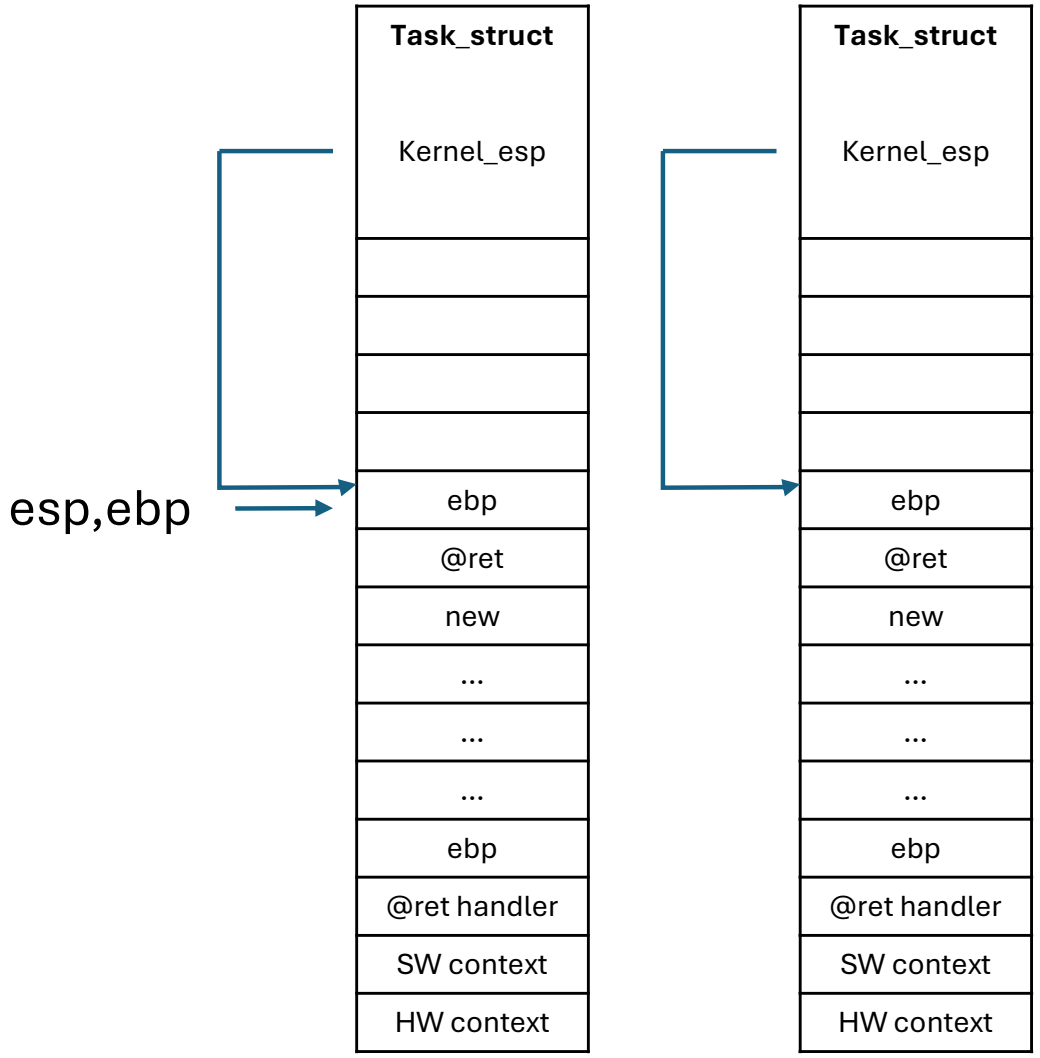
- After that, the task_switch function changes:
  - cr3 to point to the PT of the process the thread new belongs to
  - tss.esp and MSR 0x175 (if sysenter is used) to point to the system stack of the thread new

```
set_cr3(new->PT);
tss.esp0 = &((union task_union*)new)->stack[1024];
write_msr(0x175, tss.esp0);
```



void task_switch(struct task_struct *new)

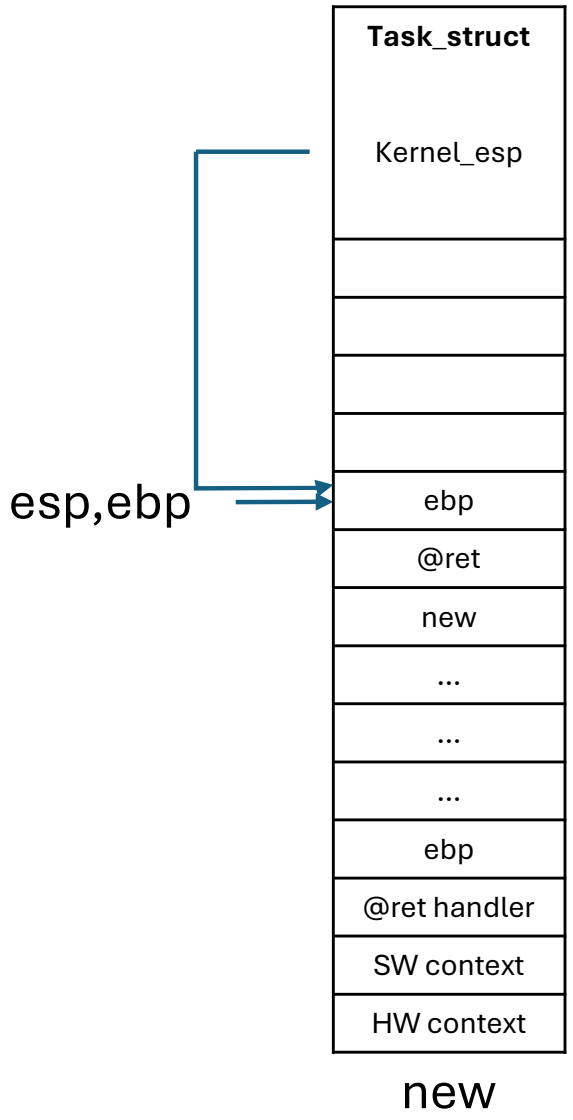
- Finally, task_switch must change ESP to point to the system stack of thread new
- The structure of the system stack of the thread new is similar to the system stack of the thread current and the kernel_esp of new points to the ebp of task_switch when new was current (i.e. when new was preempted)




```
esp = new->kernel_esp;
```

esp = new->kernel_esp;

- After executing this instruction, the thread new is current since the system stack of the thread new is in use by the CPU

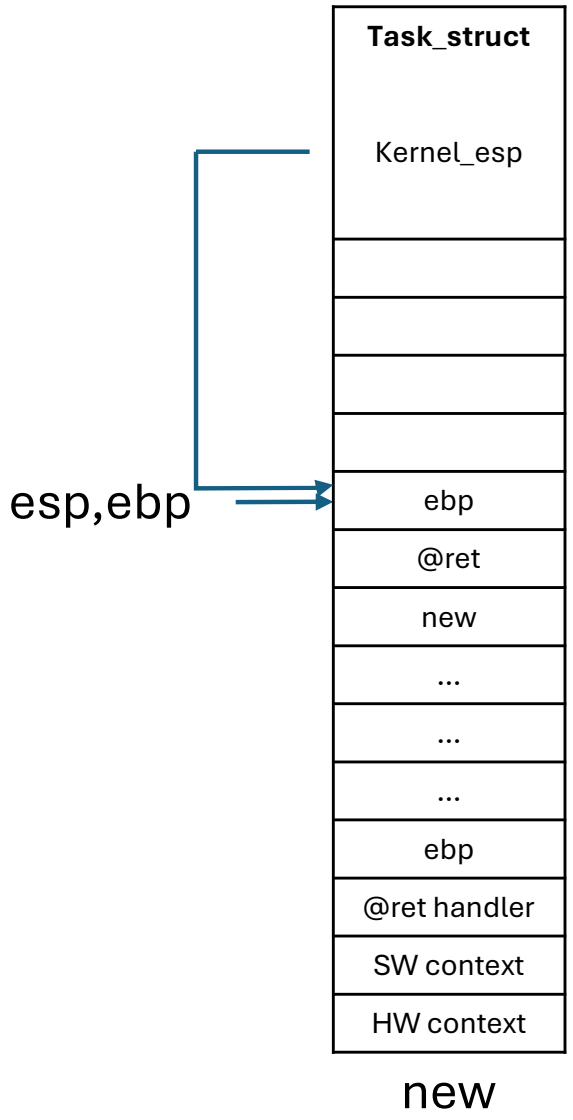


Finalizing task_switch

- Finally, task_switch must return to user mode
- For that, task_switch executes:

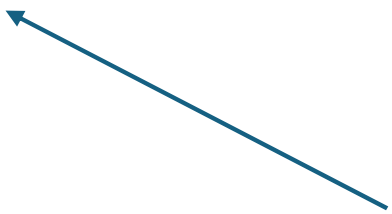
```
popl %ebp
ret
```

- Notice that since the system stack of new is in use, its execution context is completely recovered when returning from functions and from the clock interrupt!!



Task_switch full code

```
void task_switch(struct task_struct *new)
{
        current()->kernel_esp = ebp;
        set_cr3(new->PT);
        tss.esp0 = &((union task_union*)new)->stack[1024];
        write_msr(0x175, tss.esp0);
        esp = new->kernel_esp;
        popl %ebp
        ret
}
```



From this point onwards, new is current

Switchable thread

- Notice that task_switch finishes with:

```
popl %ebp
ret
```

- This means that a thread, for being able to enter into the CPU, needs that kernel_esp points to a value for ebp followed by a return address

Task_switch execution order

- The order of instructions is not mandatory (except for popl ebp, ret)
- But remember that after changing the system stack, new becomes current

```
void task_switch(struct task_struct *new)
{
        current()->kernel_esp = ebp;
        esp = new->kernel_esp;
        set_cr3(current()->PT);
        tss.esp0 = &((union task_union*)current())->stack[1024];
        write_msr(0x175, tss.esp0);
        popl %ebp
        ret
}
```

From this point onwards, new is current

Task_switch performance

- Task_switch is very fast since only CPU registers are set and only one value is stored in memory
- Unfortunately, setting cr3 to change the address space flushes the TLB
- A flush in the TLB causes a lot of TLB miss bursts that decrease the performance
- Improvement: if switching to a thread new that belongs to the same process as the thread current, do not change cr3
- To even improve more this, schedulers always try to switch to a thread of the same process

Task_switch improvement

```
void task_switch(struct task_struct *new)
{
        current()->kernel_esp = ebp;
        if (current()->PID != new->PID)
                set_cr3(new->PT);
        tss.esp0 = &((union task_union*)new)->stack[1024];
        write_msr(0x175, tss.esp0);
        esp = new->kernel_esp;
        popl %ebp
        ret
}
```

Process creation

- In a multiprocess/multithreaded OS, the OS must provide services to create process and threads
- When creating a process, the OS will automatically create 1 thread for that process
  - 1 process + 1thread
- Three different cases:
  - Init process
  - Idle process
  - Fork
- But, before that, we must know how the kernel allocates OS structures

Structure allocation

- To simplify structure management, the OS always uses structures with a size equal to a page size (4KBs)
- And uses identity mapping to access these structures
- Up to know, we have seen the following OS structures:
  - Page table
  - Union task_union
- Remember that before accessing an allocated frame, it must be mapped in the OS section of the PT

Allocating a structure

- To allocate a frame:

```
int frameID = alloc_frame();
```

- Giving that OSPT is the PT where OS structures must be mapped, map the allocated frame with identity mapping:

```
set_ss_pag(OSPT, frameID, frameID);
union task_union *t = (union task_union*)(frameID << 12);
```

- Since the logical and the physical frame coincide, the OS does not have to translate physical addresses when accessing structures
- To ensure this, the OS reserves a zone in the OS PT (called hyperspace) and the corresponding physical pages to store the identity maps.

Init process creation

- Init is the first process executed in user mode after the boot of the OS finalizes
- It is created in boot time
- The image of a process in memory is defined when creating the Init process
- It is created in real mode but executed in protected mode
  - It is created without logical to physical address translation
  - It is executed with logical to physical address translation
- It is the only process that, when executed for its first time, does not enter Run state through task_switch

Creating the Init process

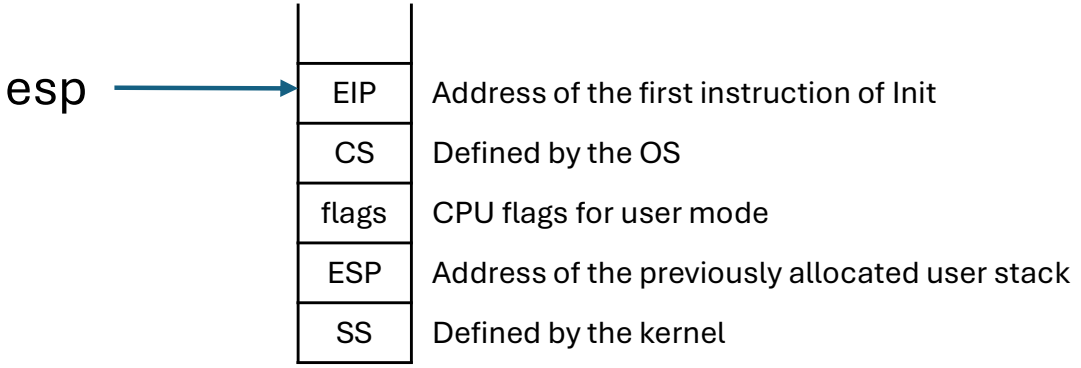
- First, allocate a PT
- After that, allocate and initialize a union task_union
  - The PID of Init is always 1
- Map the OS code and data in the PT of Init
  - This is very dependable of the OS
- Map the user code, data in the PT of Init
- Allocate and map a user stack
- Make Init become current
- Return to user mode executing the first instruction of Init and using its user stack

Making init become current

- After allocating all resources for Init and before switching to user mode, Init must become current
- For that:
  - cr3 must point to the allocated PT of Init
  - tss.esp0 and MSR 0x175 must point to the system stack of Init
- It is similar to a task_switch to Init but in this case, it must be done manually during boot time

Starting Init

- To start executing Init, the OS simulates a return from an interrupt
- For that, the OS creates a hardware context in the Init system stack
- And sets ESP to point to that hardware context
- Finally, the instruction IRET is executed to switch to user mode, using the user stack of Init and having as return address, the first instruction of Init
- This is called a return_gate



Idle process

- The idle process is a system process that executes when the current thread leaves the CPU and there are no more threads in the ready queue
- Since the CPU cannot be halted temporarily, sched_next selects this process when the ready queue is empty
- It is created in boot time
- It never executes in user mode, but always in OS mode
- It is responsible of performing OS maintenance and it can execute the midterm scheduler
- Usually is an idle loop. It waits for clock interrupts to execute the scheduler and in case the ready queue is not empty, it is instantly preempted
- The idle process cannot be enqueued in the Ready queue but its struct task_struct is accessed using a global pointer

Creating the Idle process

- To create the Idle process, first, allocate a union task_union and initialize its struct task struct
- After that, allocate PT
- Since it will never execute in user mode, the OS only maps the OS code and data in the PT
- Initialize its system stack for Idle to be able to execute throught task_switch

Initializing the Idle process system stack

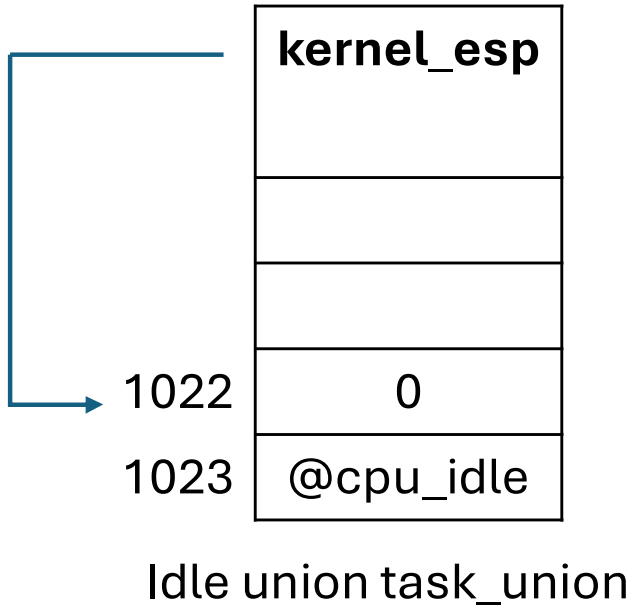
- Remember that for a process to move to Run state, kernel_esp must point to an ebp value followed by a return address
- Idle process will never execute in user mode. This means that it does not need to have a hardware context nor a software context in the system stack
- The code of idle is in the cpu_idle function:

```
void cpu_idle()
{
        sti;            // Allow interrupts in OS mode
        while (1);      // Infinite loop
}
```

Initializing the Idle process system stack

- So, the only information the Idle's system stack has when created is an ebp and a return address and kernel_esp points to that ebp

```
// Return address
idle->stack[1023] = cpu_idle;
// Fake ebp
idle->stack[1022] = 0;
idle->task.kernel_esp = &idle->stack[1022]
```



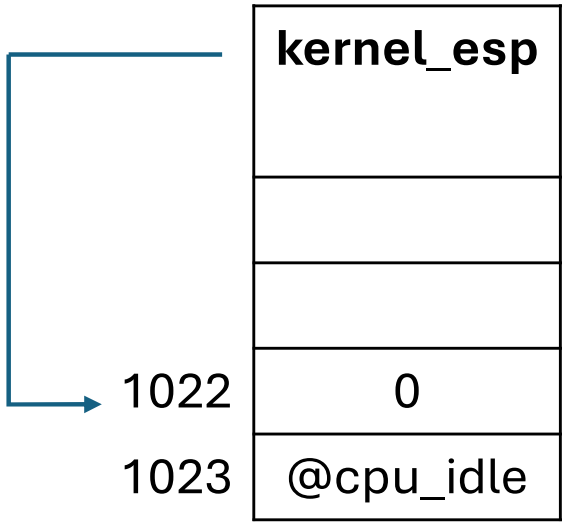
Idle union task_union

- When task_switch switches to idle, it will execute cpu_idle

Switching to idle

- When task_switch is executed having idle as new, it first, stores the context of current and sets cr3, tss.esp0 and MSR 0x175 to those of idle

```
void task_switch(struct task_struct *new)
{
            current()->kernel_esp = ebp;
            set_cr3(new->PT);
            tss.esp0 = &((union task_union*)new)->stack[1024];
            write_msr(0x175, tss.esp0);
            esp = new->kernel_esp;
            popl %ebp
            ret
}
```

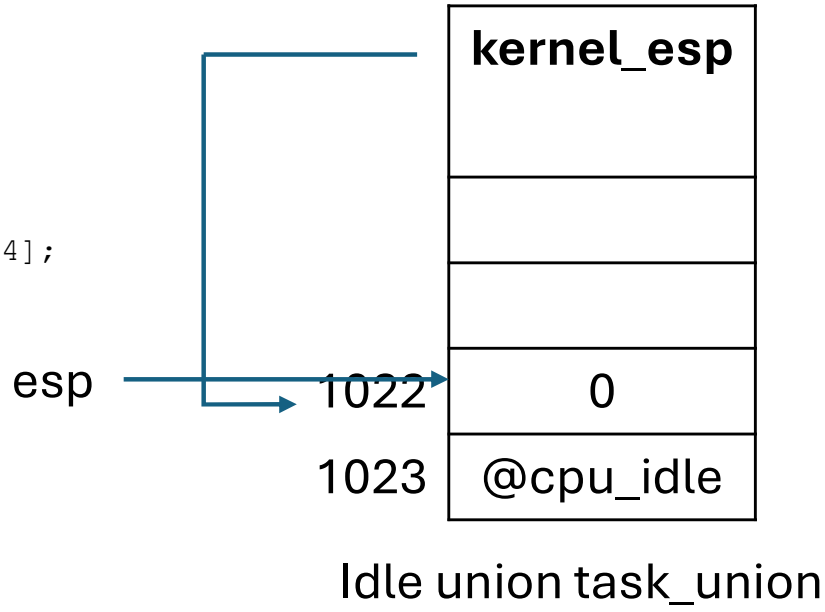


Idle union task_union

Switching to idle

- After that, modified esp with new->kernel_esp to point to the system stack of idle

```
void task_switch(struct task_struct *new)
{
            current()->kernel_esp = ebp;
            set_cr3(new->PT);
            tss.esp0 = &((union task_union*)new)->stack[1024];
            write_msr(0x175, tss.esp0);
            esp = new->kernel_esp;
            popl %ebp
            ret
}
```

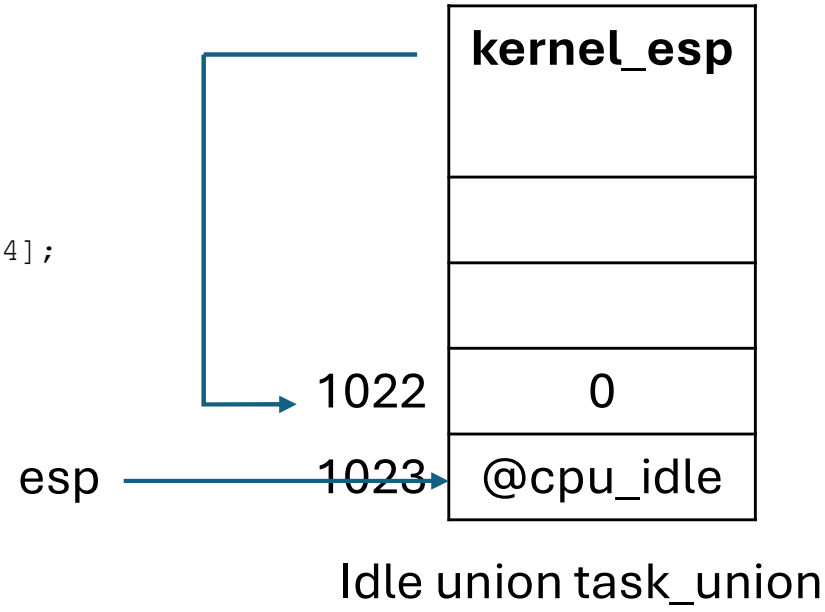


Idle union task_union

Switching to idle

- Then, recovers the previous value of ebp. In this case, it is a 0 but it could be any value since it will not be used

```
void task_switch(struct task_struct *new)
{
            current()->kernel_esp = ebp;
            set_cr3(new->PT);
            tss.esp0 = &((union task_union*)new)->stack[1024];
            write_msr(0x175, tss.esp0);
            esp = new->kernel_esp;
            popl %ebp
            ret
}
```

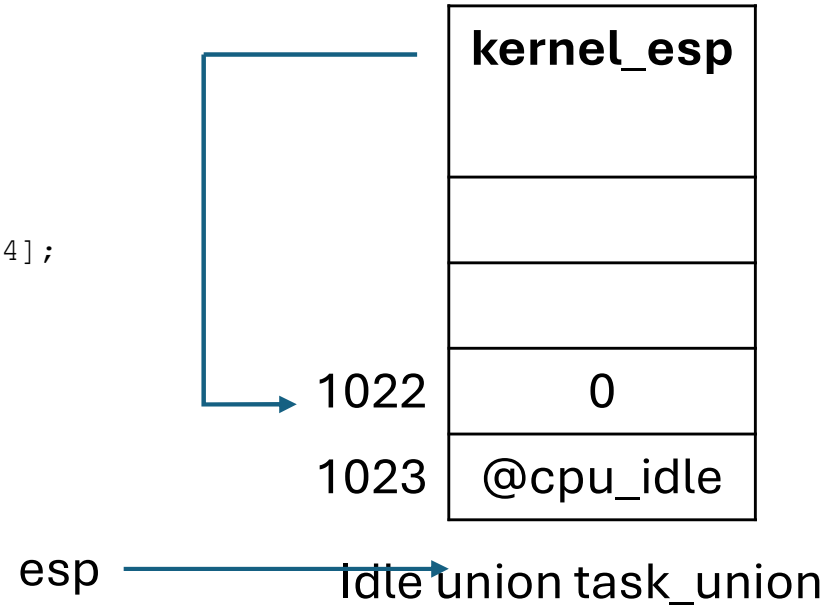


Idle union task_union

Switching to idle

- Finally, task_switch returns with the address of cpu_idle, what means that task_switch returns to cpu_idle, executing the first instruction of the process

```
void task_switch(struct task_struct *new)
{
            current()->kernel_esp = ebp;
            set_cr3(new->PT);
            tss.esp0 = &((union task_union*)new)->stack[1024];
            write_msr(0x175, tss.esp0);
            esp = new->kernel_esp;
            popl %ebp
            ret
}
```



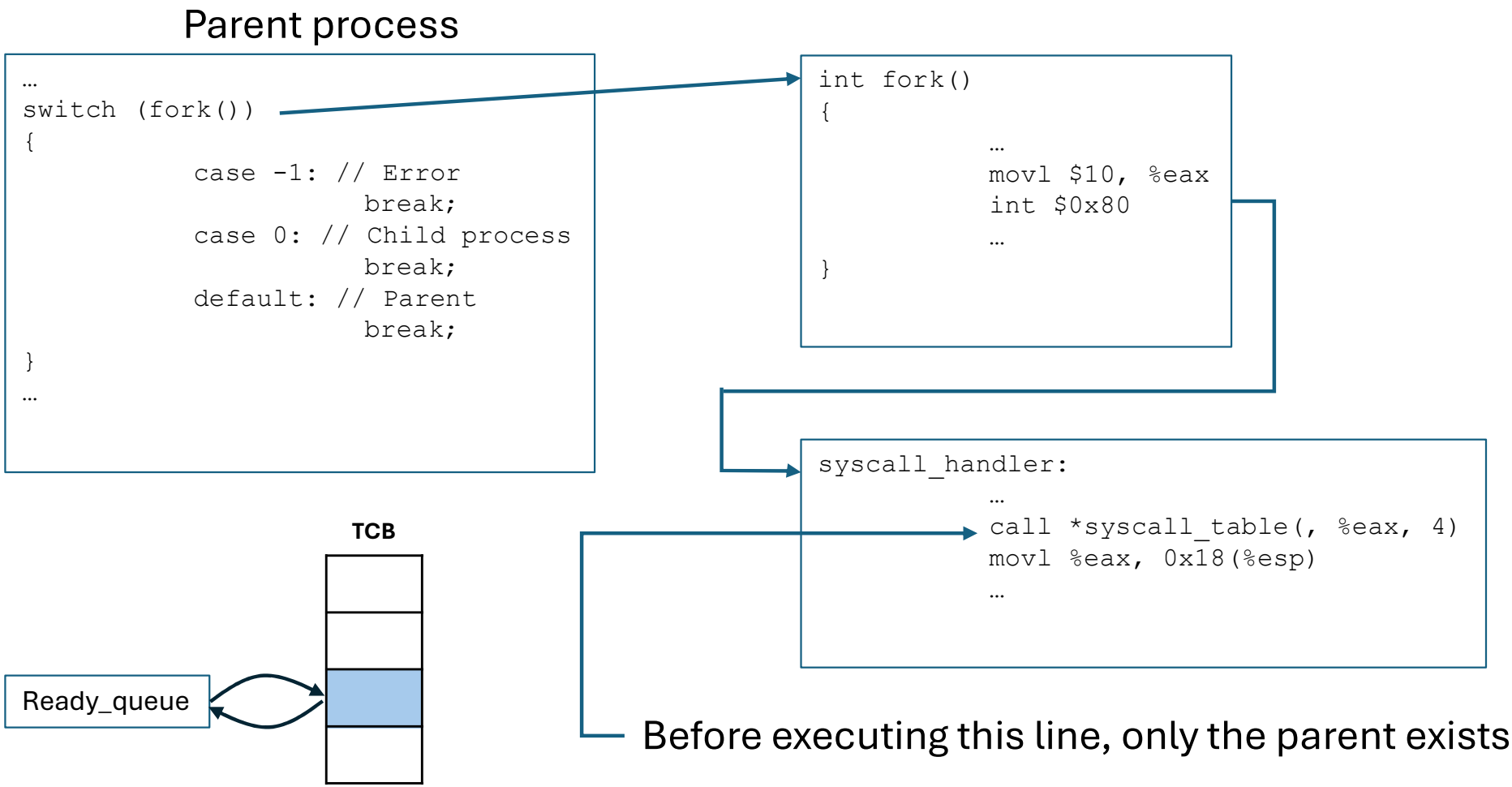
Process creation

- Sys_fork is used to create processes in Linux
- The process that calls sys_fork is named parent
- The new process is called child
- Sys_fork creates a new process by copying the user part of the parent process

```
// Return –errno if something fails
// return 0 to the new process
// return the PID of the new process to the parent process
int sys_fork();
```

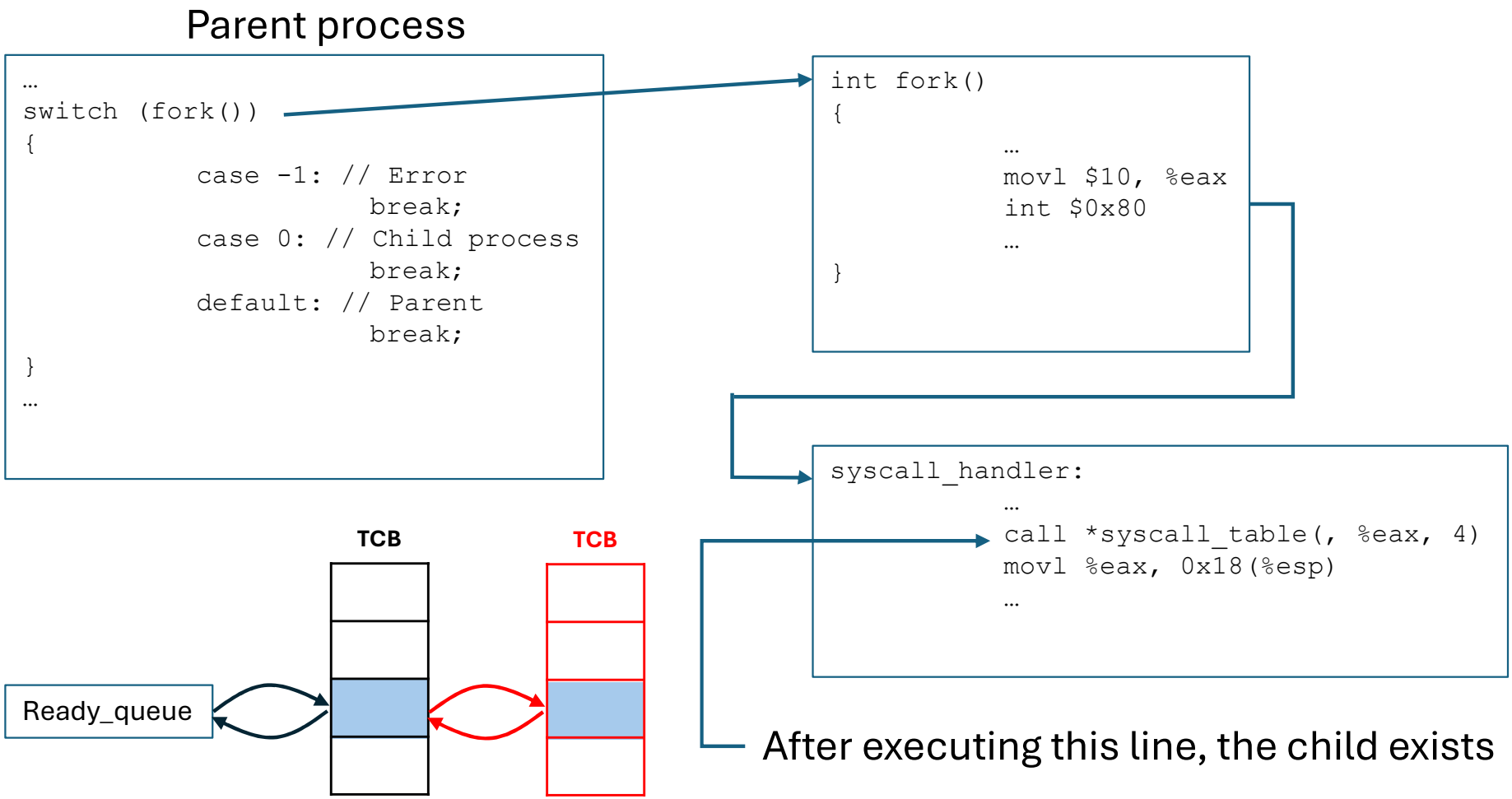
Using fork()

- The current process wants to create a new process



Using fork()

- The current process wants to create a new process



Using fork()

- The return value of fork is used to specialize code

```
…
switch (fork())
{
        case -1: // Error
                break;
        case 0: // Child process
                break;
        default: // Parent
                break;
}
…
```

This code will only be executed by the child process

This code will only be executed by the parent process

From this point onwards, the code will be executed by both

sys_fork implementation

1. Allocate a union task_union
2. Allocate a PT
3. Assign a PID to the child process
4. Initialize the child's task_union
5. Copy parent's memory to child
6. Initialize the rest of the OS structures of the child process
7. Prepare the child's system stack
8. Enqueue child's task_struct in the Ready queue
9. Return child's PID

Assigning PIDs

- PIDs must be unique in the OS
- Several strategies can be implemented
- The PID cannot reveal internal information of the OS
- Monotonical increment:
  - PID = ++global_PID;
  - Warning with the overflow
  - Must check for collisions

```
int PID = ++global_PID;
while (PID_exists(PID)) PID++;
```

Assigning PIDs (and II)

- Return as PID the address of the task_struct
  - It can be a security hole since the address of the task_struct is known from user space

```
int frameID = alloc_frame();
set_ss_pag(PT, frameID, frameID);
struct task_struct *task = frameID << 12;
int PID = (int)task;
```

- Generate a pseudorandom number
  - Best strategy since no information is reveal
  - Warning with collisions

```
int PID = rand();
while (PID_exists(PID)) PID++;
```

Initializing the child's task_union

- Notice that the parent's is a valid one since it has the data of an alive and executing process
- In addition, its system stack has all the necessary context to return to user mode
- So, the fastest way to initialize the child's task_union is to copy that of the parent to the child's

```
union task_union *Parent; // Parent's task_union
union task_union *Child; // Child's task_union
copy_data(Parent, Child, sizeof (union task_union));
```

- After that, all the particular fields for the child must be initialized (PID, PT, ...)

Copying memory from parent to child

- Copying the memory from parent to child involves two steps
- First, the OS memory must be shared with the child
- Second, the copy of the user part of the memory (code, data and stacks) from parent to child

Sharing the OS memory with the child

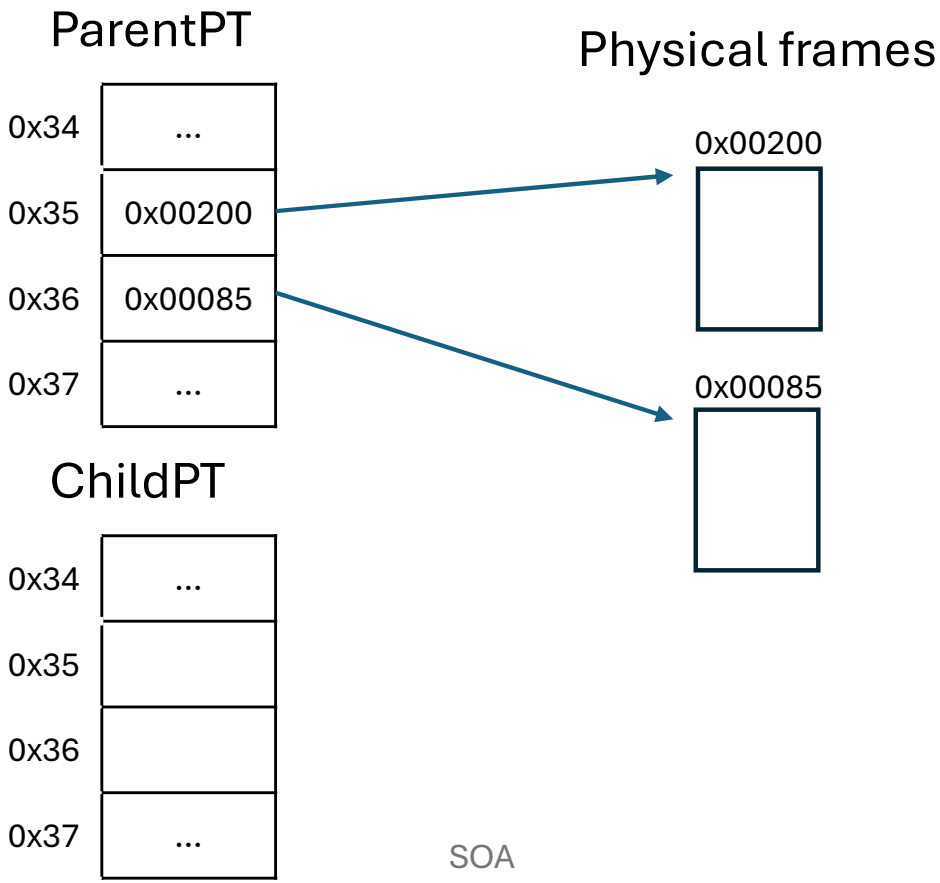
- Remember that all processes in the OS, share the OS memory (code and data)
- This means that all processes in their PT have the same mapping between logical and physical frames for the OS
- To share the OS memory, the PT entries corresponding to OS memory must be copied from parent to child

```
int frameID;
for (frameID = KERNEL_STAR_PAGE; frameID <= KERNEL_END_PAGE; frameID++)
{
        // Obtain the physical frame associated to a logical frame
        int map = get_frame(ParentPT, frameID);
        // Map the frame in the same entry of the child's PT
        set_ss_frame(ChildPT, frameID, map);
}
```

Sharing the OS memory example

```
int frameID;
for (frameID = KERNEL_STAR_PAGE; frameID <= KERNEL_END_PAGE; frameID++)
{
            // Obtain the physical frame associated to a logical frame
            int map = get_frame(ParentPT, frameID);
            // Map the frame in the same entry of the child's PT
            set_ss_frame(ChildPT, frameID, map);
}
```

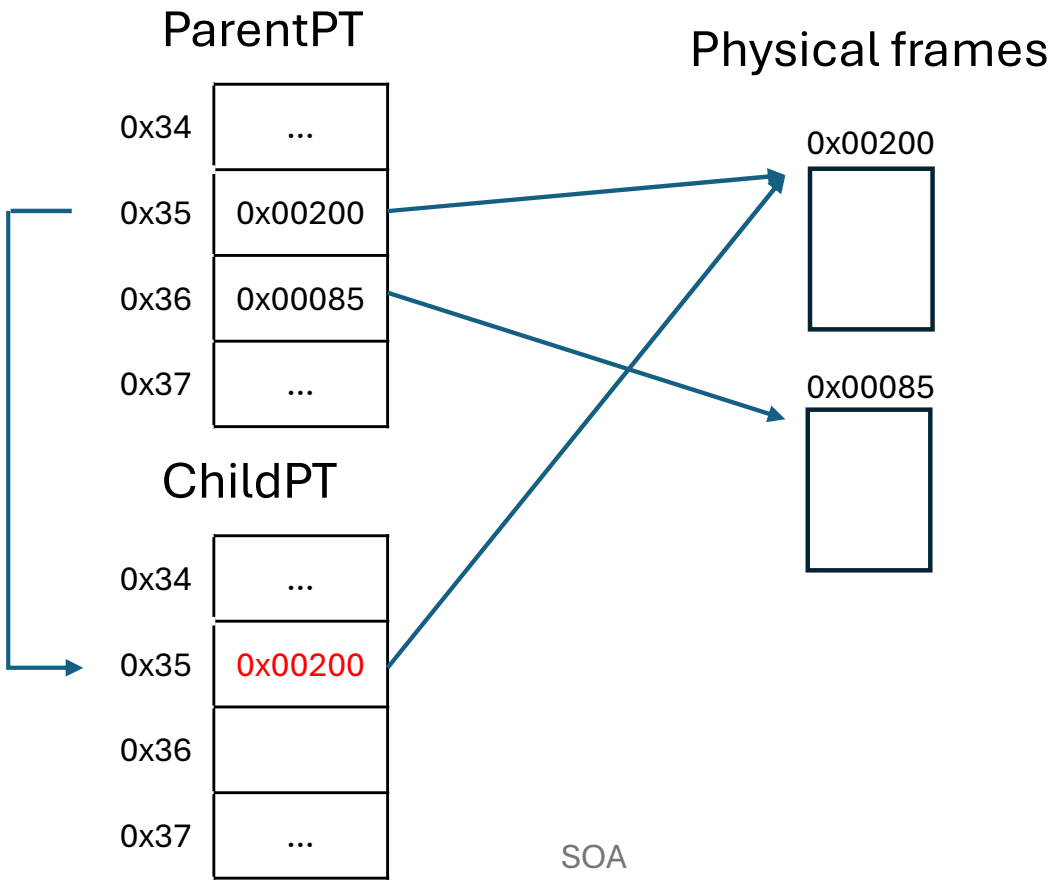
frameID	map
0x35	0x00200



Sharing the OS memory example

```
int frameID;
for (frameID = KERNEL_STAR_PAGE; frameID <= KERNEL_END_PAGE; frameID++)
{
            // Obtain the physical frame associated to a logical frame
            int map = get_frame(ParentPT, frameID);
            // Map the frame in the same entry of the child's PT
            set_ss_frame(ChildPT, frameID, map);
}
```

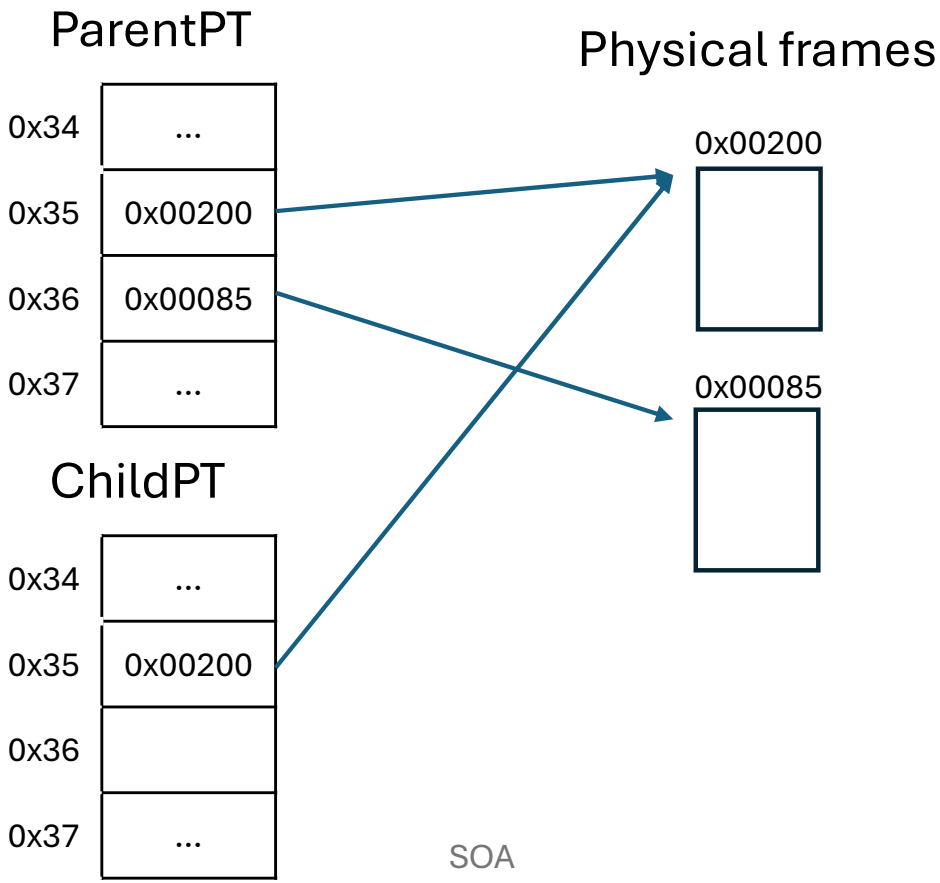
frameID	map
0x35	0x00200



Sharing the OS memory example

```
int frameID;
for (frameID = KERNEL_STAR_PAGE; frameID <= KERNEL_END_PAGE; frameID++)
{
            // Obtain the physical frame associated to a logical frame
            int map = get_frame(ParentPT, frameID);
            // Map the frame in the same entry of the child's PT
            set_ss_frame(ChildPT, frameID, map);
}
```

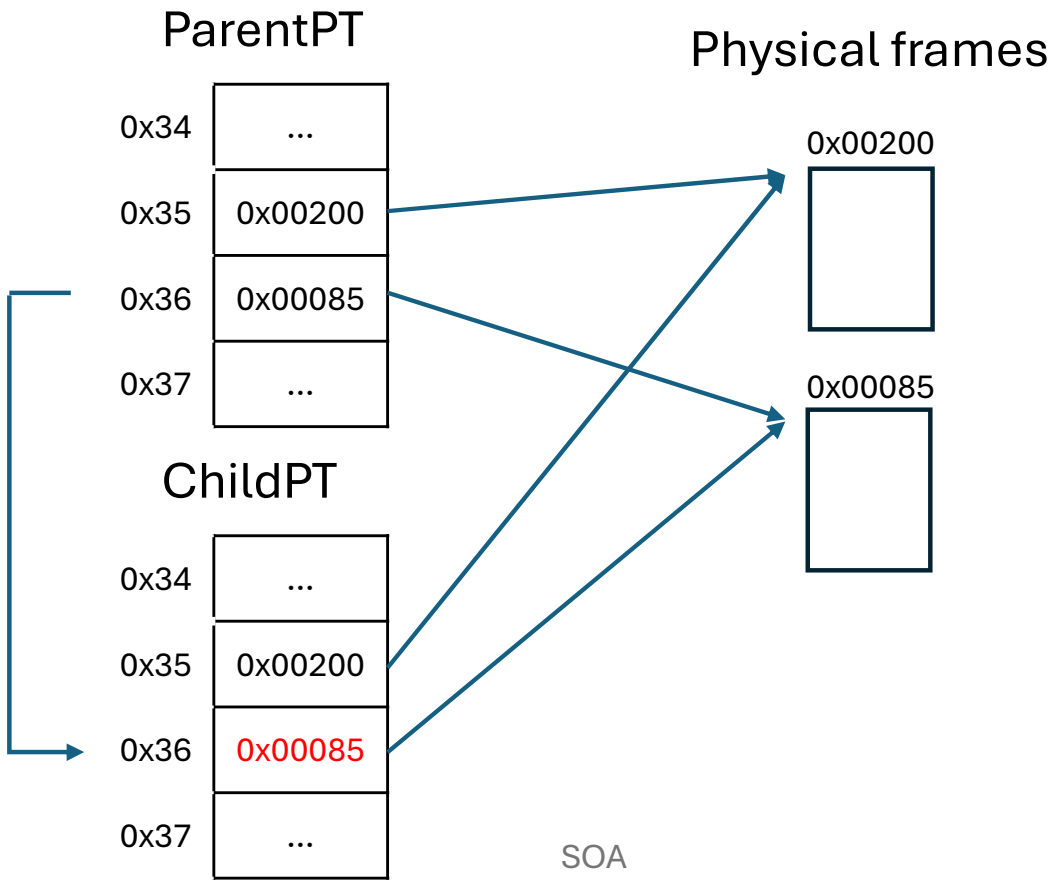
frameID	map
0x36	0x00085



Sharing the OS memory example

```
int frameID;
for (frameID = KERNEL_STAR_PAGE; frameID <= KERNEL_END_PAGE; frameID++)
{
            // Obtain the physical frame associated to a logical frame
            int map = get_frame(ParentPT, frameID);
            // Map the frame in the same entry of the child's PT
            set_ss_frame(ChildPT, frameID, map);
}
```

frameID	map
0x36	0x00085

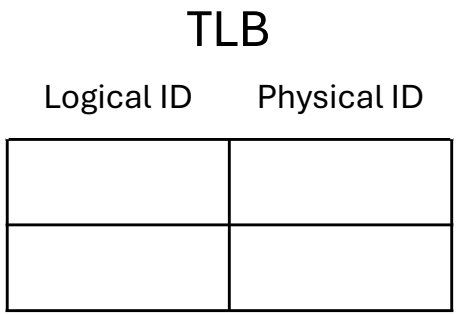


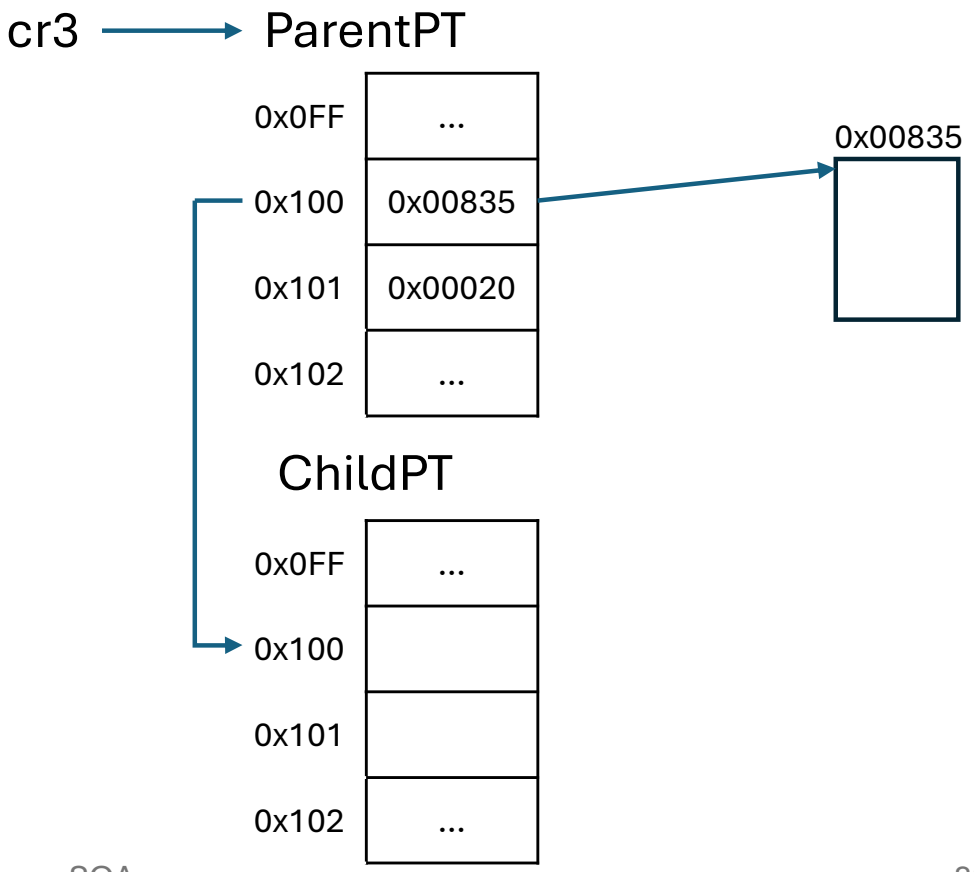
Copying the user memory from parent to child

- Remember that processes do not share memory
- Every process has its own user code, user data and user stacks and are not shared among processes
- This means that, when copy the user part of a process, the contents of the physical frames must be copied
- Also remember that when in OS mode, the OS uses the execution context of the current process
- In this case, the cr3 register points to the parent's PT
- Problem: the physical frames of the child process cannot be accessed

Problem of copying the contents of one physical frame from Parent to Child

- Imagine that the contents of the physical frame mapped to logical frame 0x100 in the Parent PT must be copied to the physical frame mapped to logical frame 0x100 but in the ChildPT

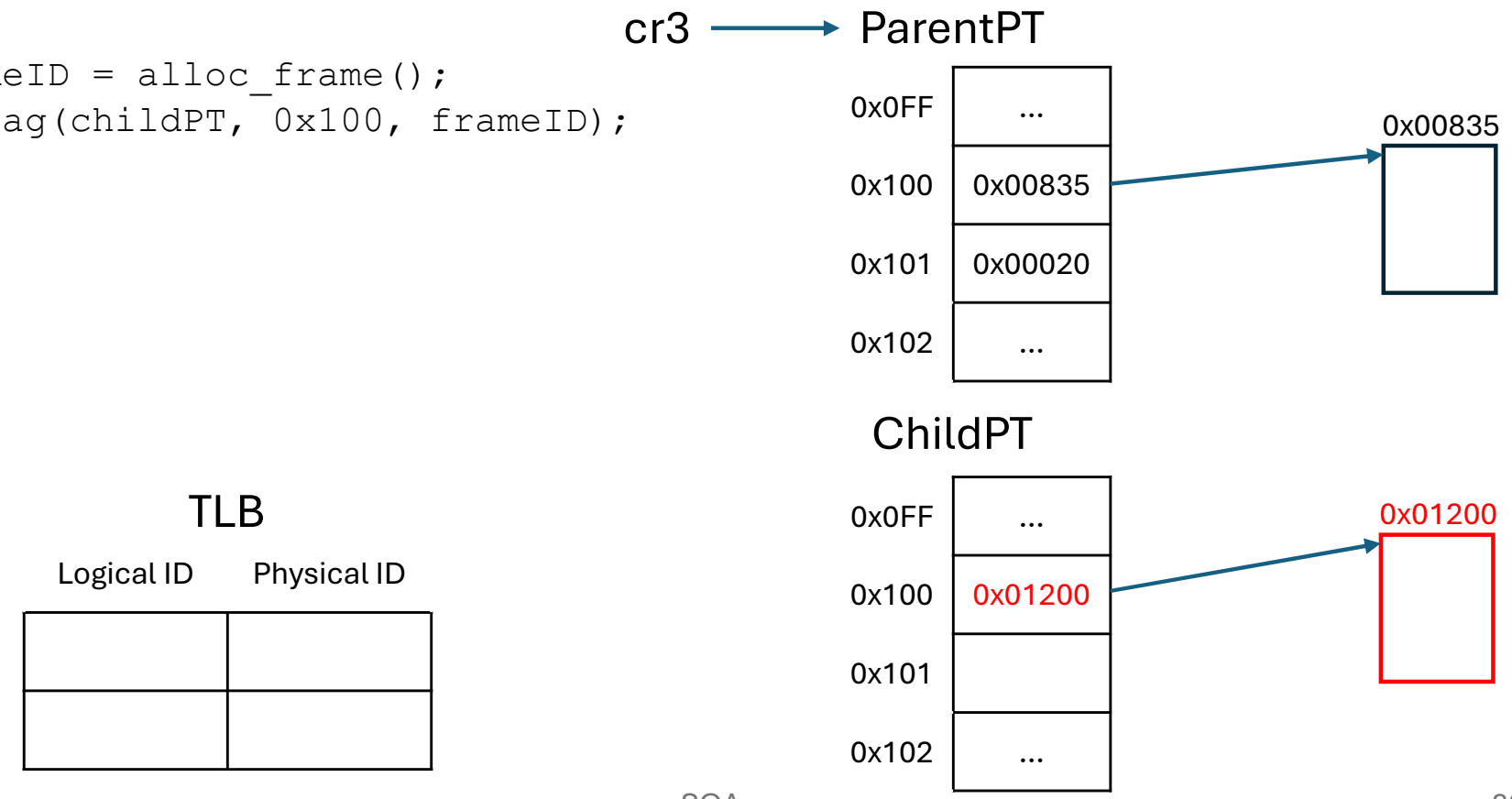




Problem of copying the contents of one physical frame from Parent to Child

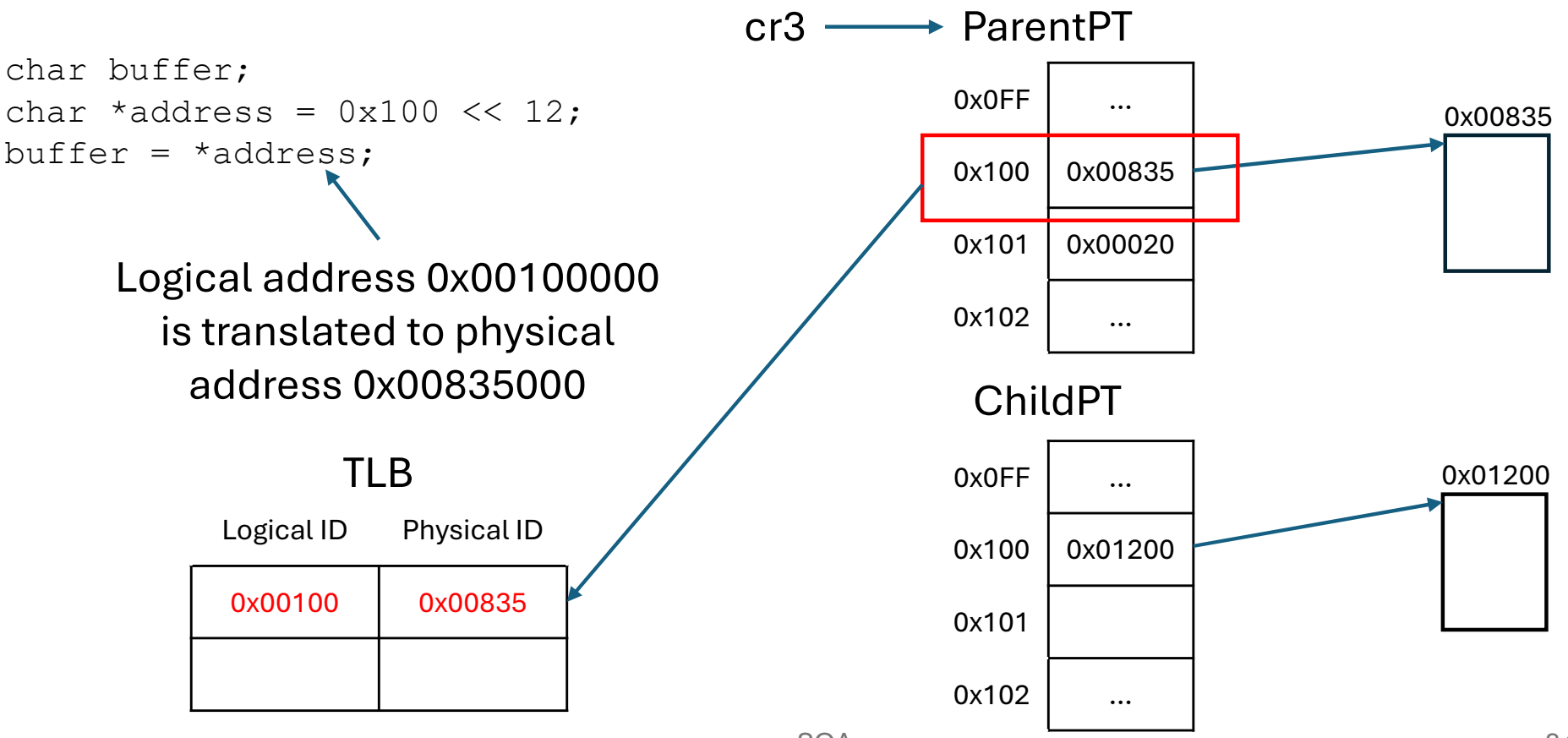
- Since the contents of a physical frame must be copy to the child, first, allocate and map a physical frame in entry 0x100 of the child PT

```
int frameID = alloc_frame();
set_ss_pag(childPT, 0x100, frameID);
```



Problem of copying the contents of one physical frame from Parent to Child

- Now, read the contents of logical frame 0x100. Since there is a TLB miss, the TLB loads the <logicalID, physicalID> couple from parent's PT.



Problem of copying the contents of one physical frame from Parent to Child

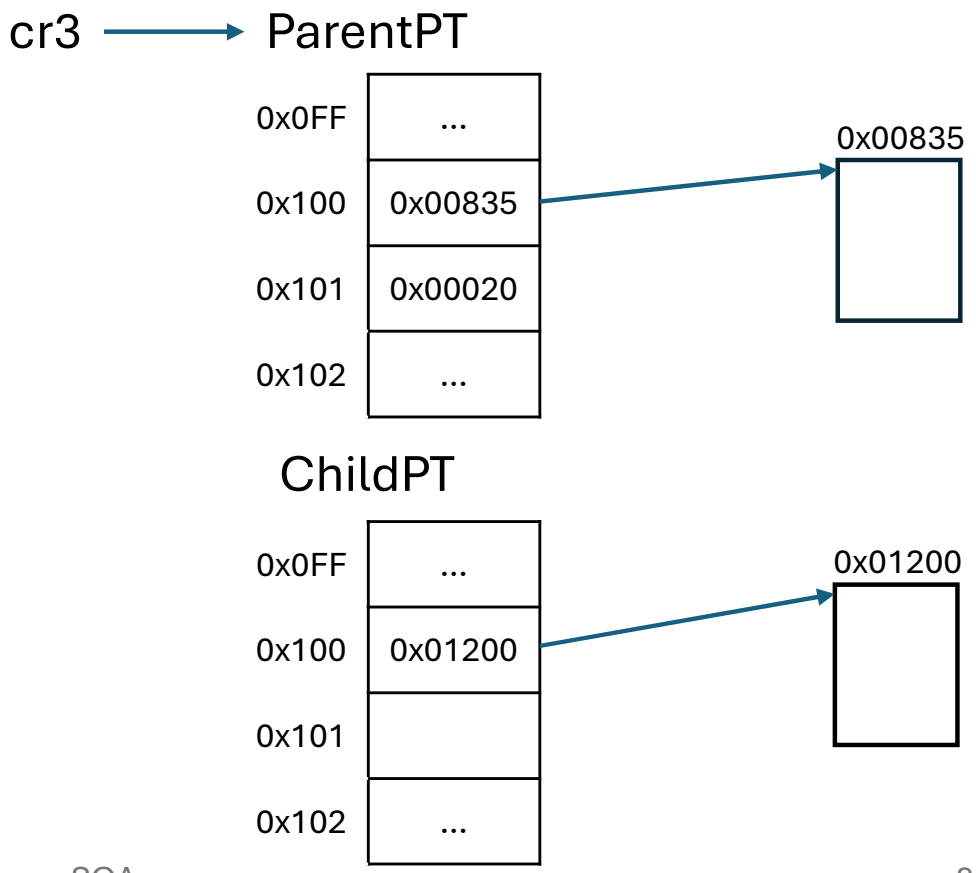
- When writing to the same logical frame for the child process, there is a TLB hit and the physical frame associated is 0x00835 when we want 0x01200

```
char buffer;
char *child_address = 0x100 << 12;
*child_address = buffer;
```

Logical address 0x00100000 is translated to physical address 0x00835000

TLB

Logical ID	Physical ID
0x00100	0x00835



Problem of copying the contents of one physical frame from Parent to Child

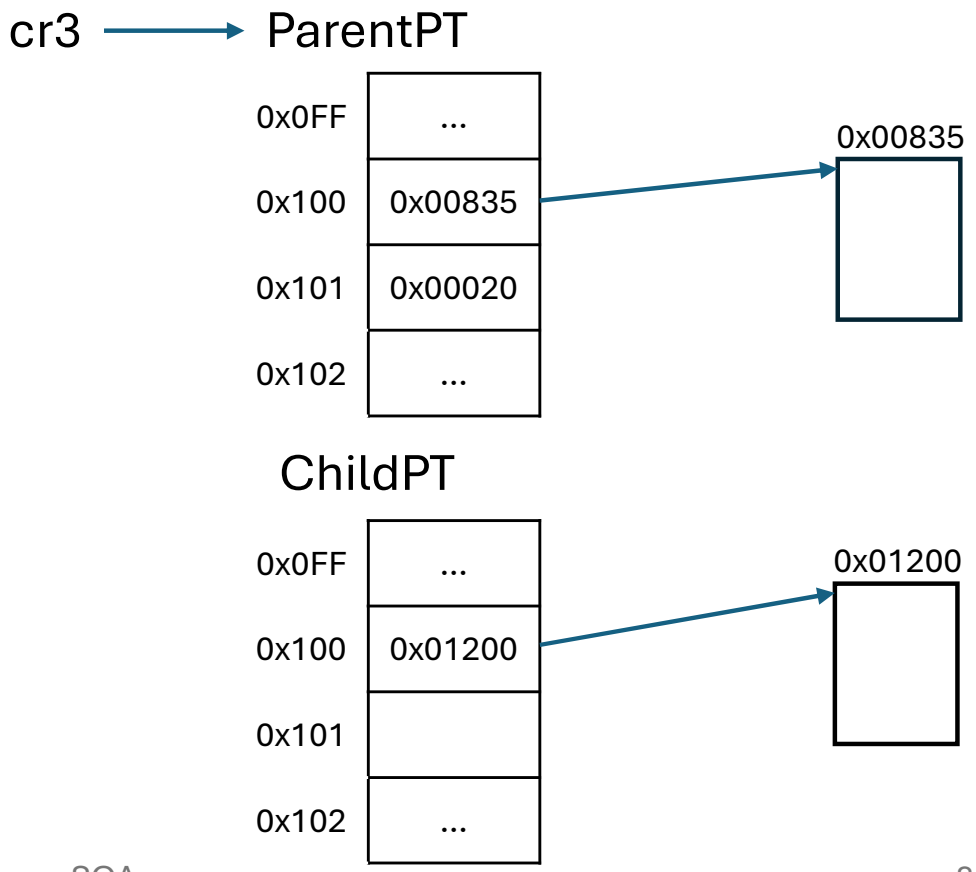
- Physical frames mapped in the ChildPT cannot be accessed since TLB generates translations with the PT of the parent process

```
char buffer;
char *child_address = 0x100 << 12;
*child_address = buffer;
```

Logical address 0x00100000 is translated to physical address 0x00835000

TLB

Logical ID	Physical ID
0x00100	0x00835

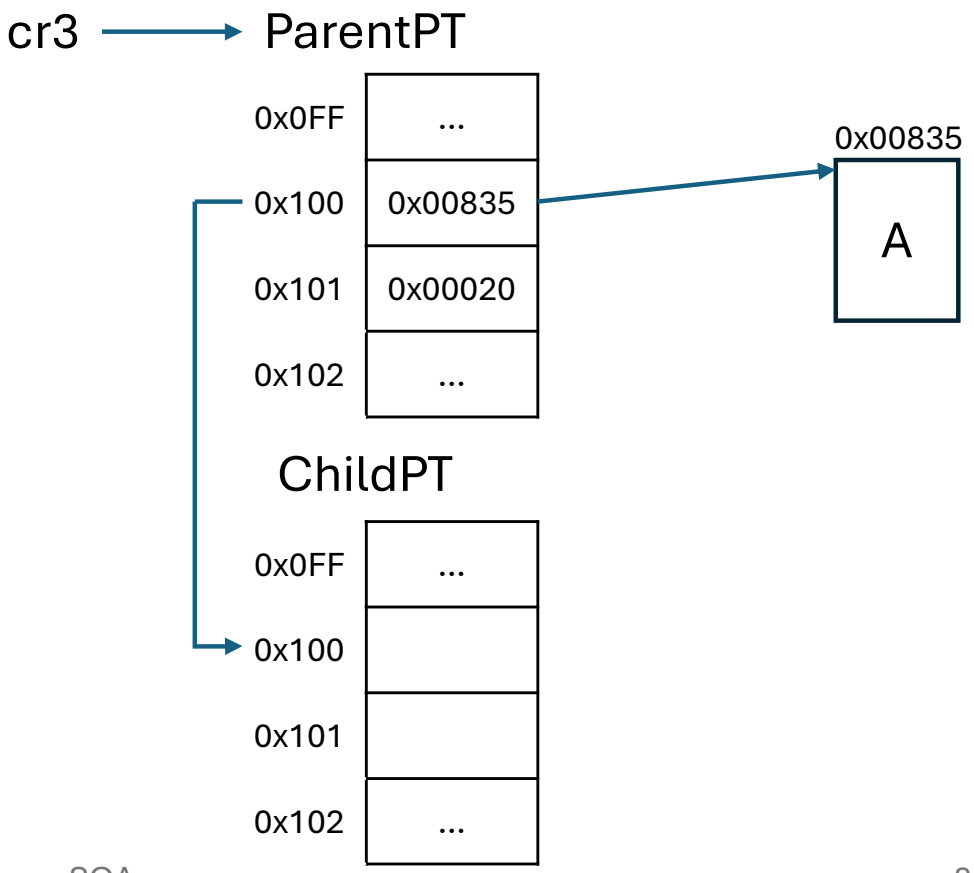


Solution for copy memory from parent to child

- The basis of the problem is that the TLB is translating addresses with the parent's PT
- Since the child physical frames are mapped in the child's PT, they cannot be accessed
- Solution: create temporal maps of the physical frames of the child process in the parent's PT
- For that, look for empty entries in the parent's PT and create maps for the physical frames of the child process
- So, the child's physical frame have logical addresses mapped in the address space of the parent
- Remember: since processes do not share memory, those temporal maps must be removed from the parent's PT

Solution for copying memory from parent to child

- As before, the contents of the physical frame mapped to logical frame 0x100 in the parent's PT must be copied to the physical frame mapped to logical frame 0x100 but in the child's PT



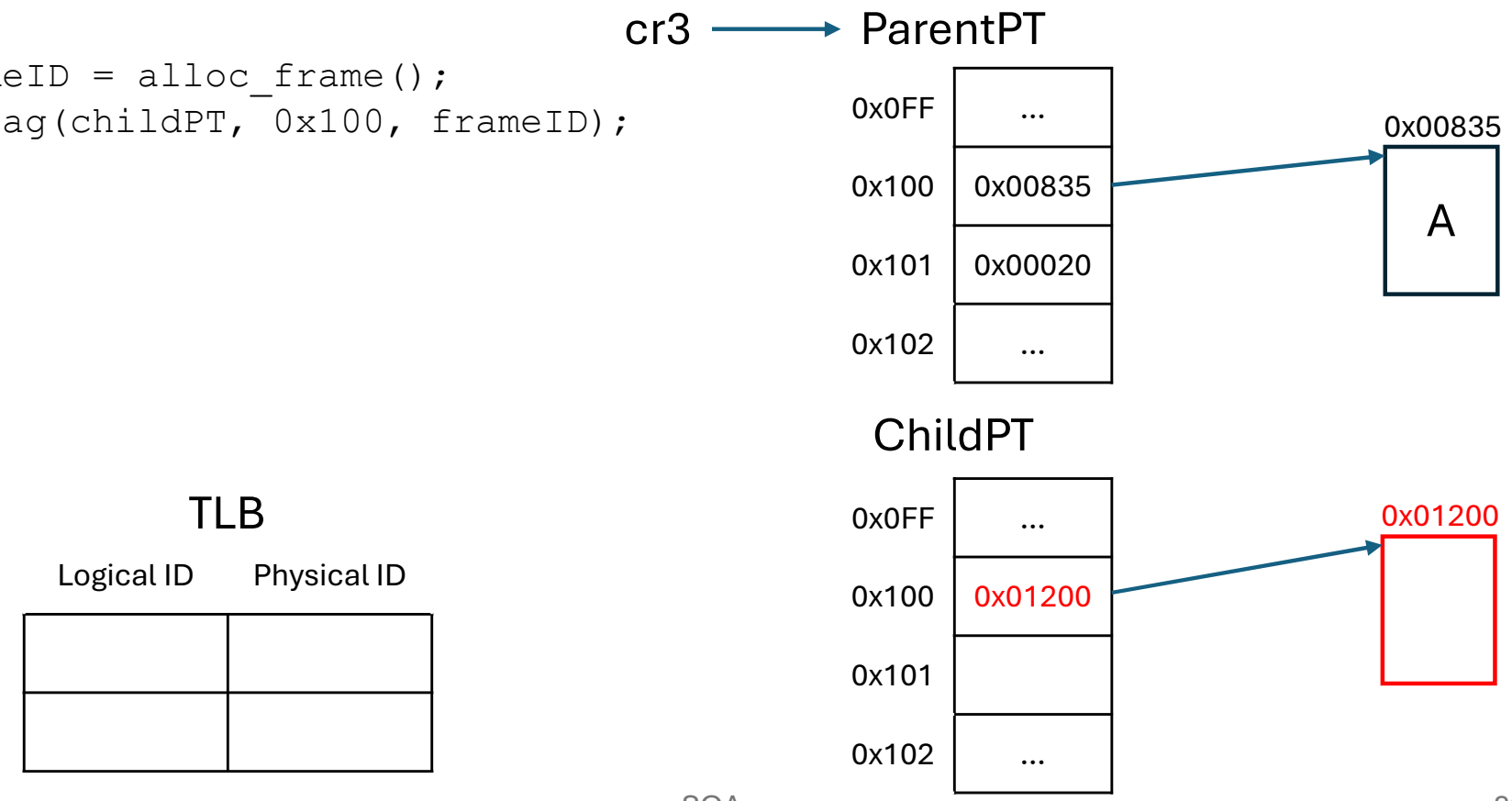
TLB

Logical ID	Physical ID

Solution for copying memory from parent to child

- As before, since the contents of a physical frame must be copy to the child, first, allocate and map a physical frame in entry 0x100 of the child's PT

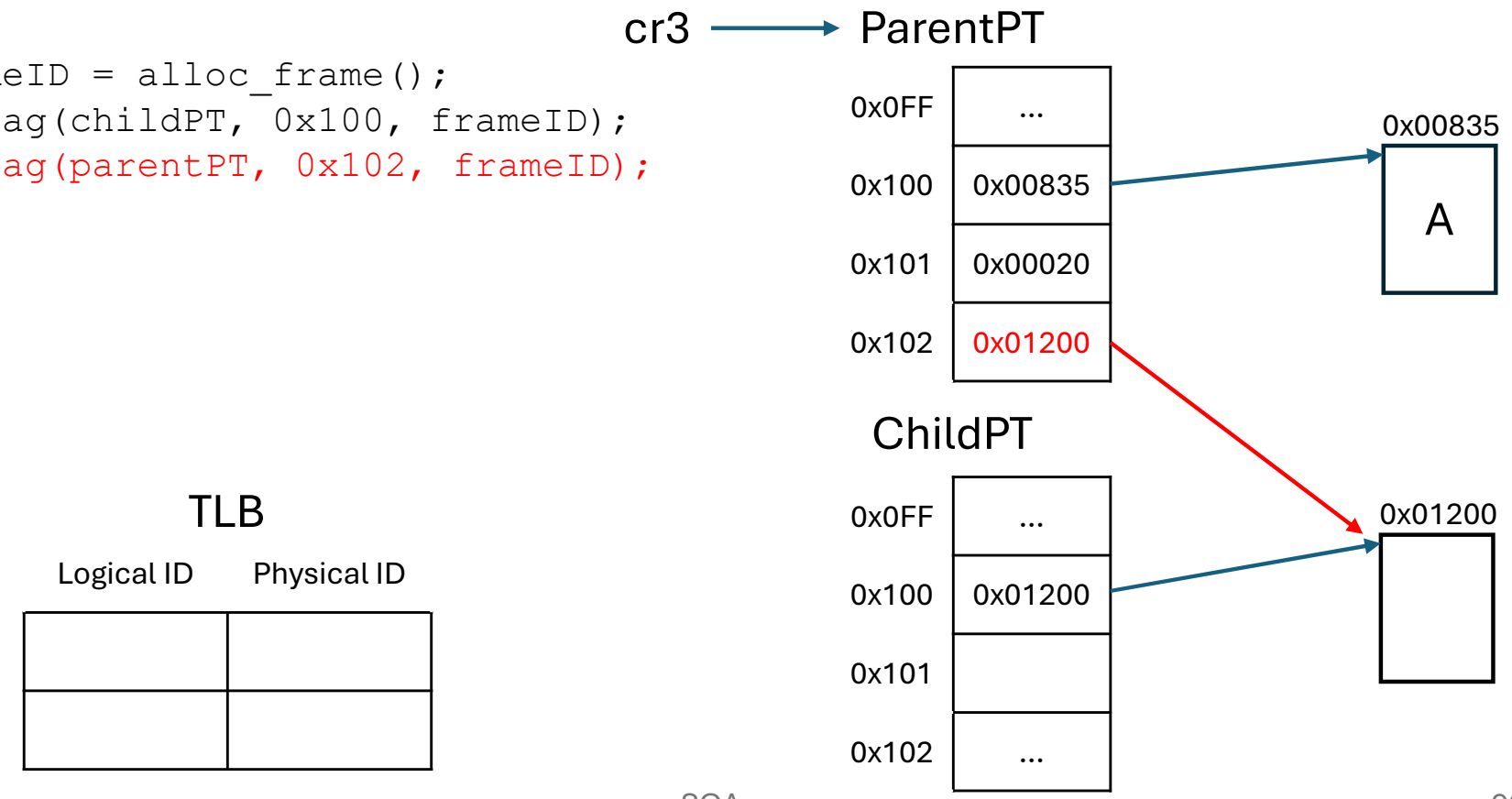
```
int frameID = alloc_frame();
set_ss_pag(childPT, 0x100, frameID);
```



Solution for copying memory from parent to child

- But now, create a temporal map of the child's physical frame in parent's PT. Suppose entry 0x102 in the parent's PT is empty.

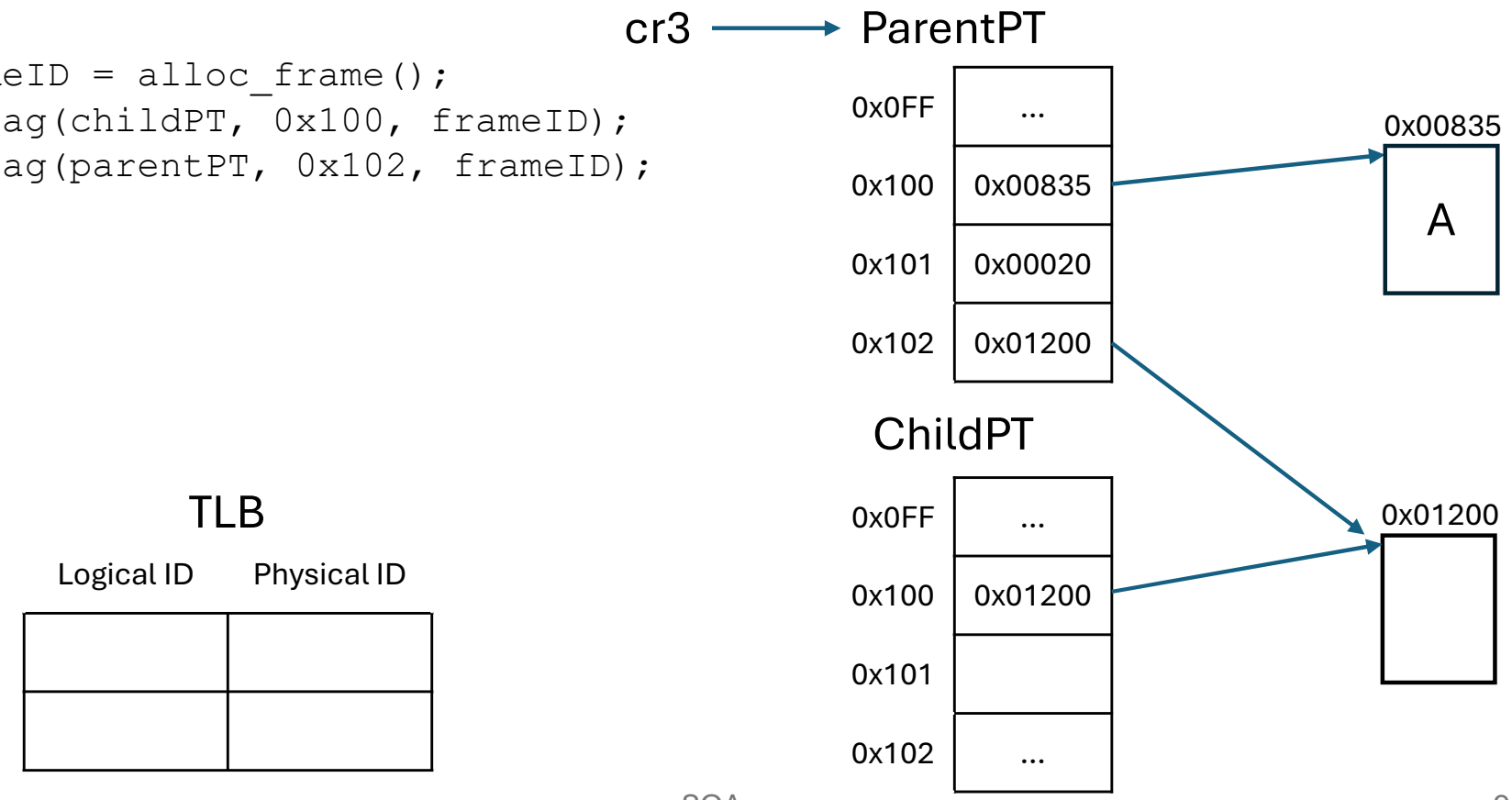
```
int frameID = alloc_frame();
set_ss_pag(childPT, 0x100, frameID);
set_ss_pag(parentPT, 0x102, frameID);
```



Solution for copying memory from parent to child

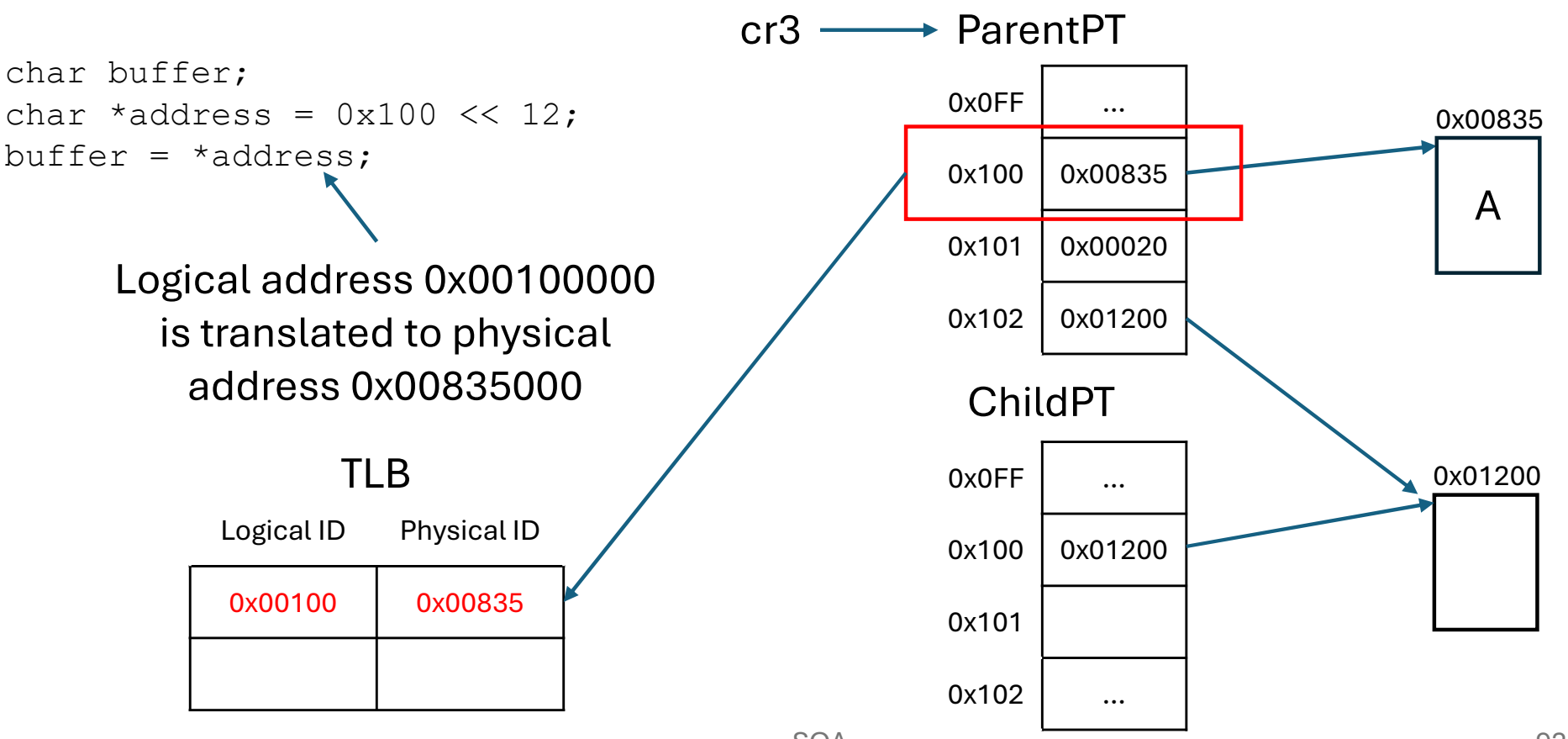
- Now, logical frame 0x100 in child's PT is seen as logical frame 0x102 in parent's PT

```
int frameID = alloc_frame();
set_ss_pag(childPT, 0x100, frameID);
set_ss_pag(parentPT, 0x102, frameID);
```



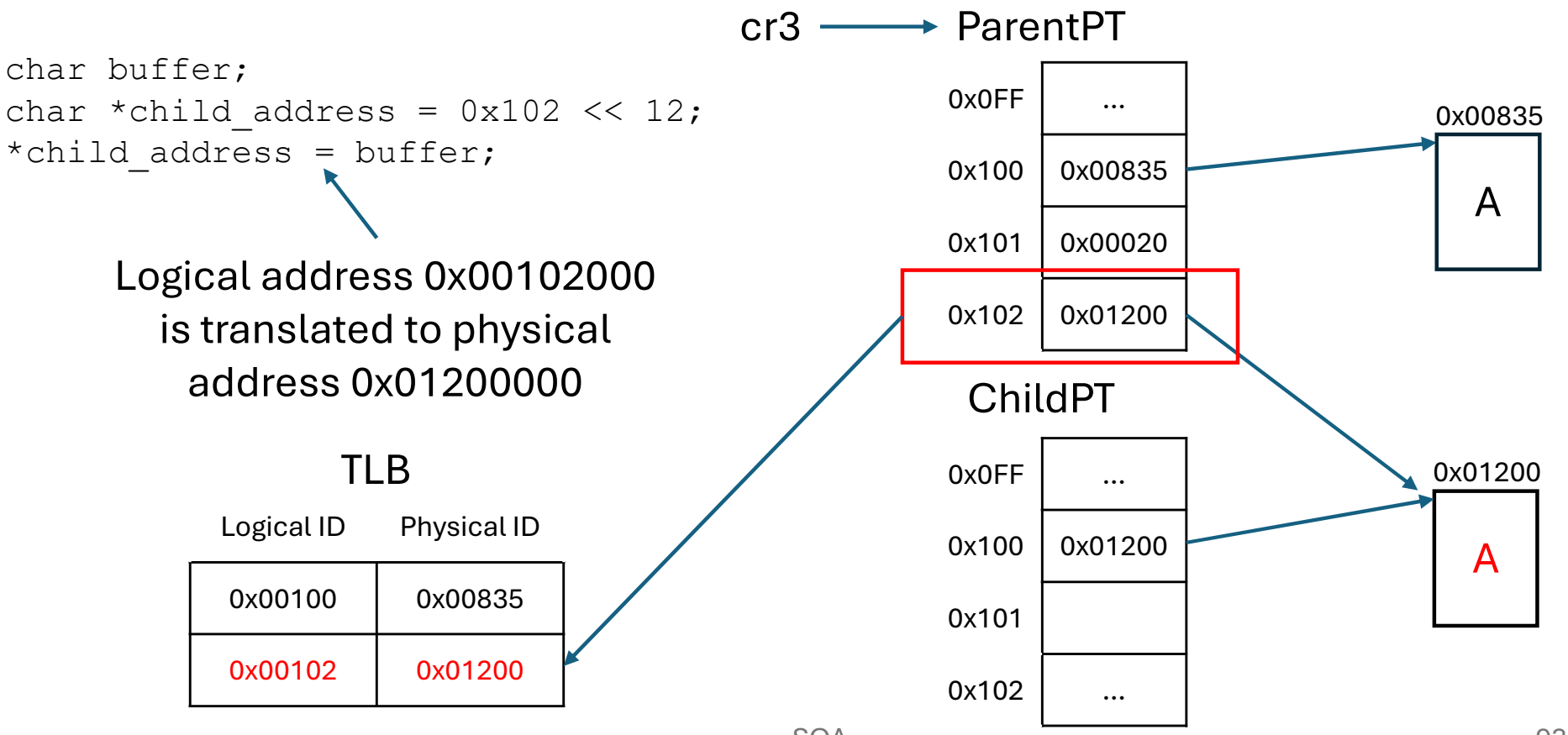
Solution for copying memory from parent to child

- Access the contents of the physical frame mapped to logical frame 0x100 in parent's PT. There is a TLB miss and TLB loads the data from parent's PT



Solution for copying memory from parent to child

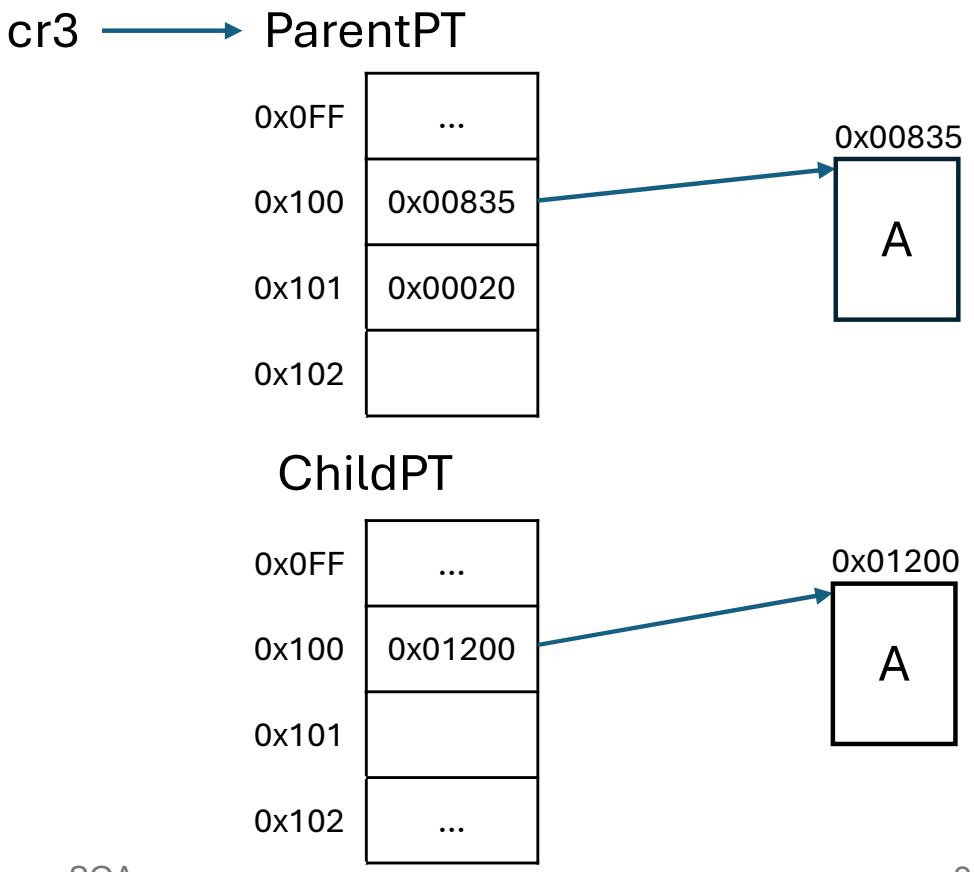
- Now, the data can be written to child's physical page using the temporal map in the parent's PT. There is a TLB miss and the TLB loads the data from parent's PT



Solution for copying memory from parent to child

- When the copy is finished, remove the temporal map from parent's PT and flush the TLB to synchronize the hardware

```
del_ss_page(parentPT, 0x102);
set_cr3(parentPT);
```



TLB

Logical ID	Physical ID

Copying the whole user memory

- This procedure must be performed for all the pages belonging to user code, data and stack
- A typical loop to copy all the user code looks like:

```
int free_logical_frame = find_free_logical_frame(parentPT);

for (int i = 0; i < USER_CODE_PAGES; i++)
{
        int frameID = alloc_frame();
        set_ss_pag(childPT, FIRST_USER_CODE_PAGE + i, frameID);
        set_ss_pag(parentPT, free_logical_frame , frameID);
        copy_data((FIRST_USER_CODE_PAGE + i) << 12,
                free_logical_frame << 12,
                PAGE_SIZE);
        del_ss_pag(parentPT, free_logical_frame);
        set_cr3(parentPT);
}
```

Copying the whole user memory

- But that loop has problem: the TLB is flushed for every frame what produces a lot of TLB miss bursts and degrades performance.
- Solution: find a gap large enough to copy several pages

```
int free_logical_gap = find_free_logical_gap(parentPT, USER_CODE_PAGES);

for (int i = 0; i < USER_CODE_PAGES; i++)
{
        int frameID = alloc_frame();
        set_ss_pag(childPT, FIRST_USER_CODE_PAGE + i, frameID);
        set_ss_pag(parentPT, free_logical_gap + i , frameID);
        copy_data((FIRST_USER_CODE_PAGE + i) << 12,
                (free_logical_gap + i) << 12,
                PAGE_SIZE);
        del_ss_pag(parentPT, free_logical_gap + i);
}
set_cr3(parentPT);
```

Creating a gap in parent's PT

- By construction, the OS always creates the image of a process with gaps
- But if not, sys_fork must create a gap
- For that, first backup a part of the parent's PT in the system stack

```
int gap[USER_CODE_PAGES];
for (int i = 0; i < USER_CODE_PAGES; i++)
{
        gap[i] = get_frame(parentPT, FIRST_GAP_PAGE + i);
        del_ss_pag(parentPT, FIRST_GAP_PAGE + i);
}
set_cr3(parentPT);
```

Restoring a gap in parent's PT

- And later, recover the original maps in the gap

```
for (int i = 0; i < USER_CODE_PAGES; i++)
{
        set_ss_frame(parentPT, FIRST_GAP_PAGE + i, gap[i]);
}
set_cr3(parentPT);
```

- Remember to flush the TLB always

Continuing with sys_fork

- After copying the memory, sys_fork must copy the rest of the OS structures of the parent process to the child process:
  - Field descriptor table (review OS course)
  - Signals (review OS course)
- The copy and initialization of any of these structures is defined with the behavior of the feature
  - For example: pending signals are not inherited but service routine for signals are

Preparing the child's system stack

- The child process (its threads) after creation, will sooner or later pass to the Run state through task_switch
- Since the parent task_union has been copied to the child's task_union, the contents of the system stack of the child process is well-known
- In addition, it is a system stack valid to return to user mode in the child process

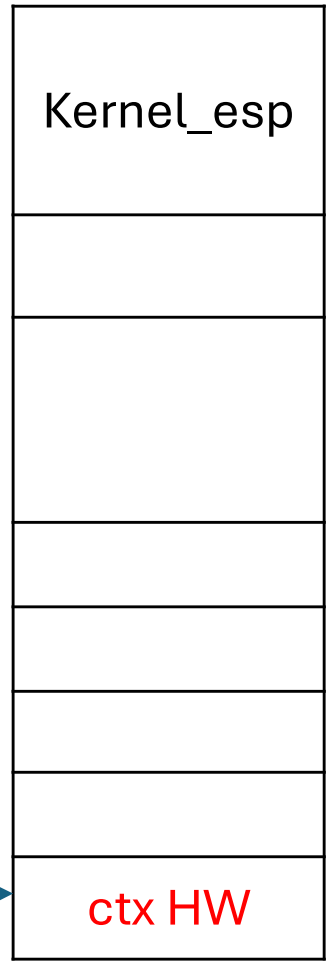
child's system stack

- The system stack before executing the first high-level instruction of sys_fork is:

```
int fork()
{
…
            movl $10, %eax
            int %0x80
…
}
```

```
syscall_handler:
            SAVE_ALL
            …
            call *syscall_table(, %eax, 4)
```

```
int sys_fork()
{
            pushl %ebp
            movl %esp, %ebp
            …
}
```



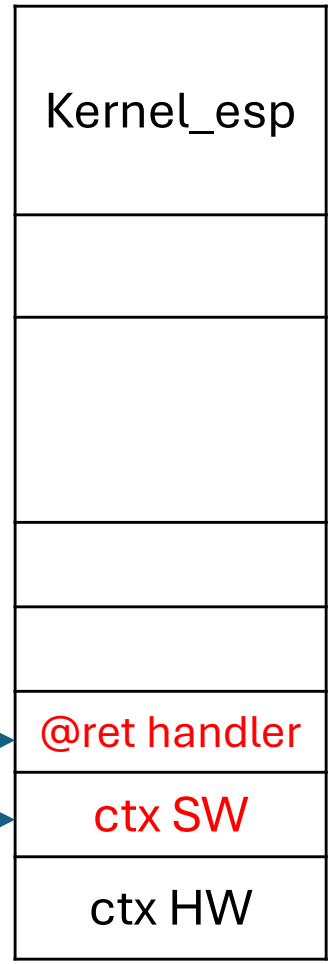
child's system stack

- The system stack before executing the first high-level instruction of sys_fork is:

```
int fork()
{
…
        movl $10, %eax
        int %0x80
…
}
```

```
syscall_handler:
        SAVE_ALL
        …
        call *syscall_table(, %eax, 4)
```

```
int sys_fork()
{
        pushl %ebp
        movl %esp, %ebp
        …
}
```



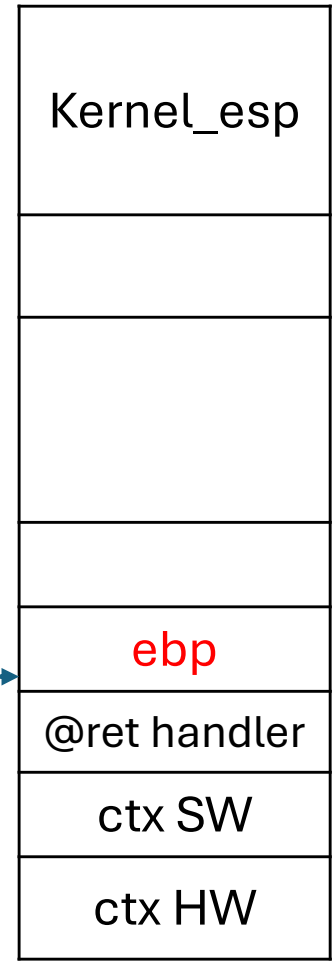
child's system stack

- The system stack before executing the first high-level instruction of sys_fork is:

```
int fork()
{
…
        movl $10, %eax
        int %0x80
…
}
```

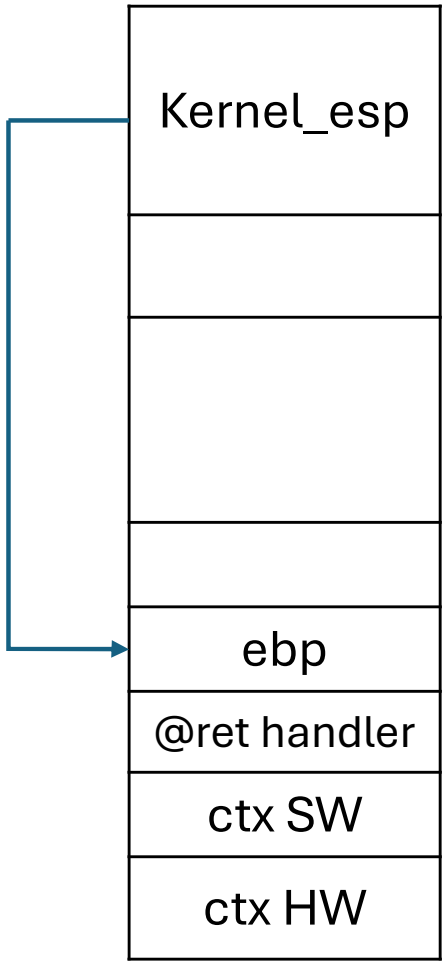
```
syscall_handler:
        SAVE_ALL
        …
        call *syscall_table(, %eax, 4)
```

```
int sys_fork()
{
        pushl %ebp
        movl %esp, %ebp
        …
}
```



child's system stack

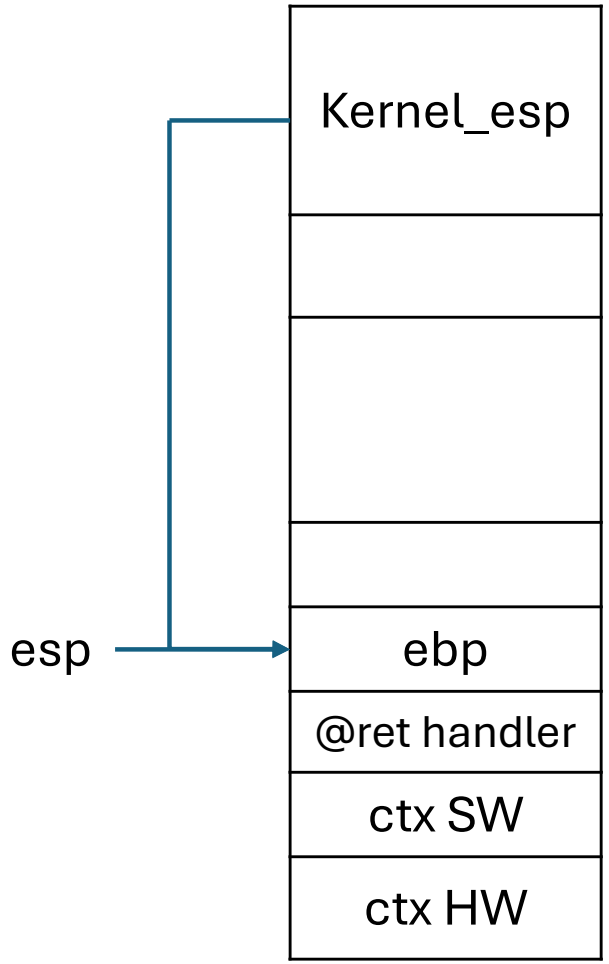
- Pointing child's kernel_esp point to the ebp of sys_fork is enough to have a switchable system stack



child's system stack

- So, when the scheduler selects the child process to execute, task_switch will be able to execute it

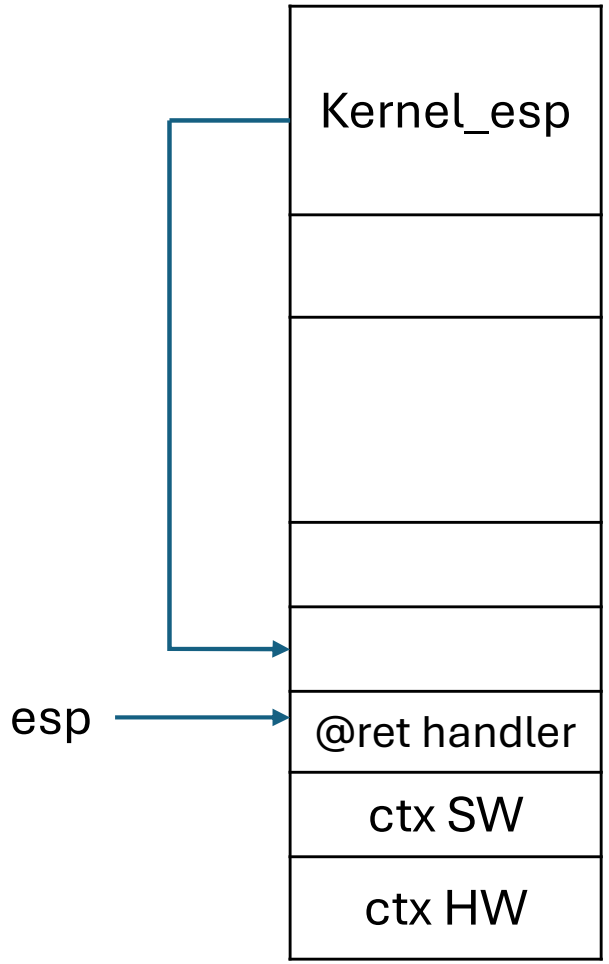
```
int task_switch(struct task_struct *new)
{
…
        esp = new->kernel_esp;
        popl %ebp
        ret
}
```



child's system stack

- So, when the scheduler selects the child process to execute, task_switch will be able to execute it

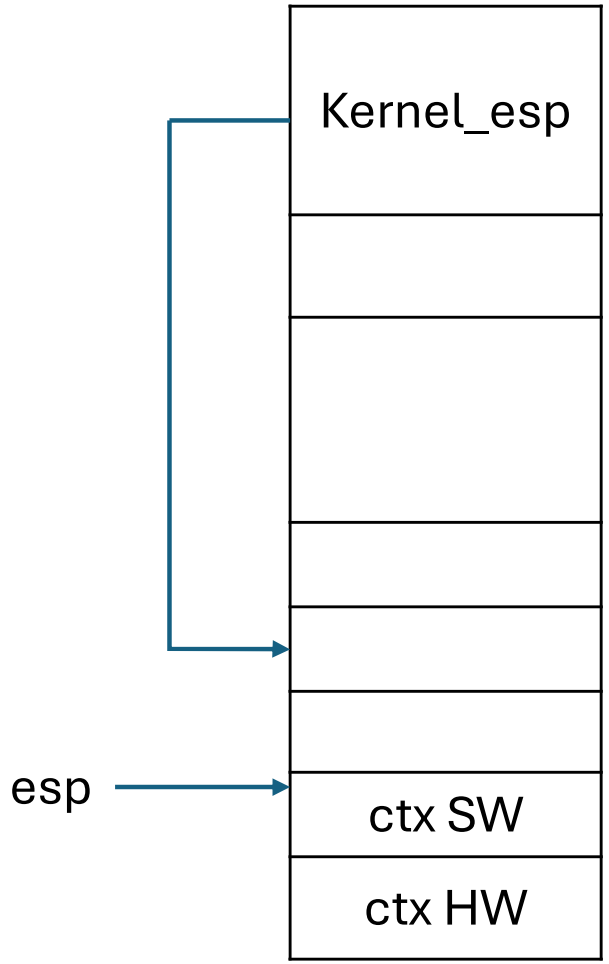
```
int task_switch(struct task_struct *new)
{
…
        esp = new->kernel_esp;
        popl %ebp
        ret
}
```



child's system stack

- The ret will return to the syscall_handler and, after that, to user mode in the child process

```
int task_switch(struct task_struct *new)
{
…
        esp = new->kernel_esp;
        popl %ebp
        ret
}
```



Problems of the solution

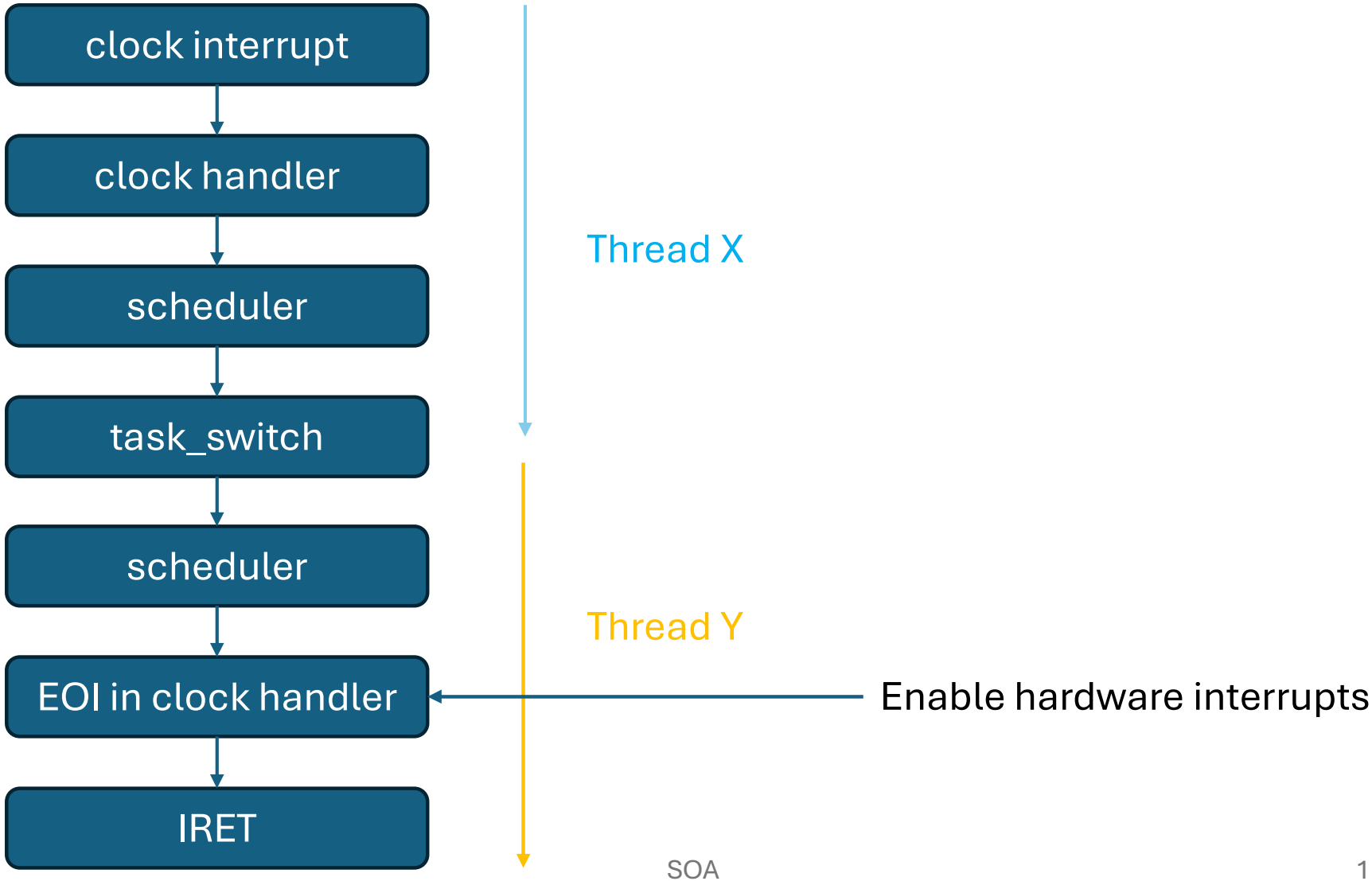
- Even if the solution seems straightforward, there are 3 problems:
- Problem 1: sys_fork does not return 0 to the child process
- Problem 2: hardware interrupts are not enabled
- Problem 3: in modern OSes, part of the initialization of a process is performed the first time that process becomes current

Problem 1 description

- Notice that when the child process passes to Run state it returns directly to the system call handler
- The first instruction that executes is movl %eax, 0x18(%esp) that was used to smash the value of eax in the software context
- But at that point, the value of eax cannot be determined and, of course, it could not be 0

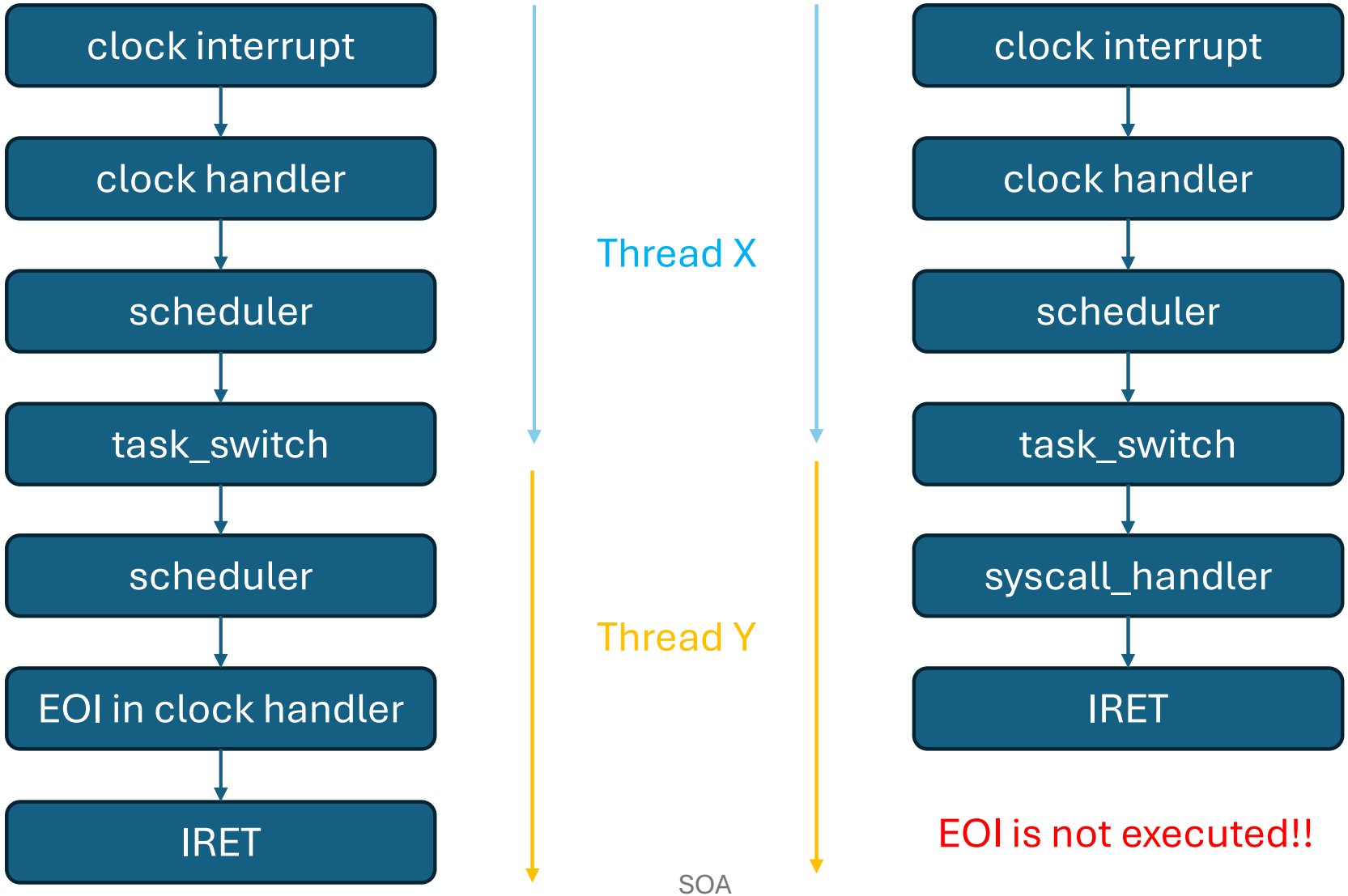
Problem 2 description

- Remember that the code path to switch to a task is:



Problem 2 description

- But for a new process, the path is:



Problem 2 description

- Remember that the in the child's system stack, the return address to the handler is to the syscall_handler because it has been copied from the parent's system stack inside a syscall (sys_fork)
- So, the first time the child process enters in the Run state, it returns to user mode through the syscall_handler which has no EOI (as expected)
- This makes that hardware interrupts remain disabled making the hardware not working

Problem 3 description

- Modern operating systems finalize the creation of a process when this process becomes current for the first time
- Notice that that system stack makes the process return to the system call handler and, after that, to user mode
- So, no finalization code can be executed as the system stack is now

Solution for the problems

- Modify the system stack for executing a specialized code the first time the process passes to Run and return to the clock interrupt handler with the appropriate value of EAX in the software context
- That specialized code will be contained in:

```
void ret_from_fork();
```

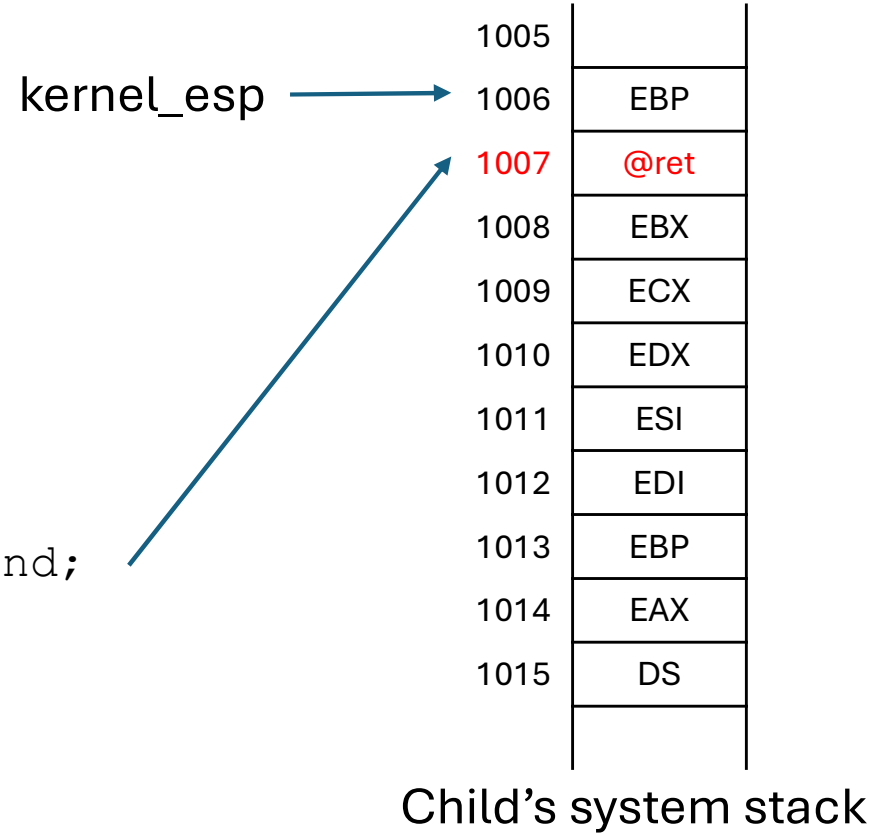
- We do not expect that you are able to find this solution by yourself, but you must understand how it works

Solution step 1

- First, modify the handler return address in the system stack by the address of the following instruction after the call clock_routine in the clock_handler in sys_fork

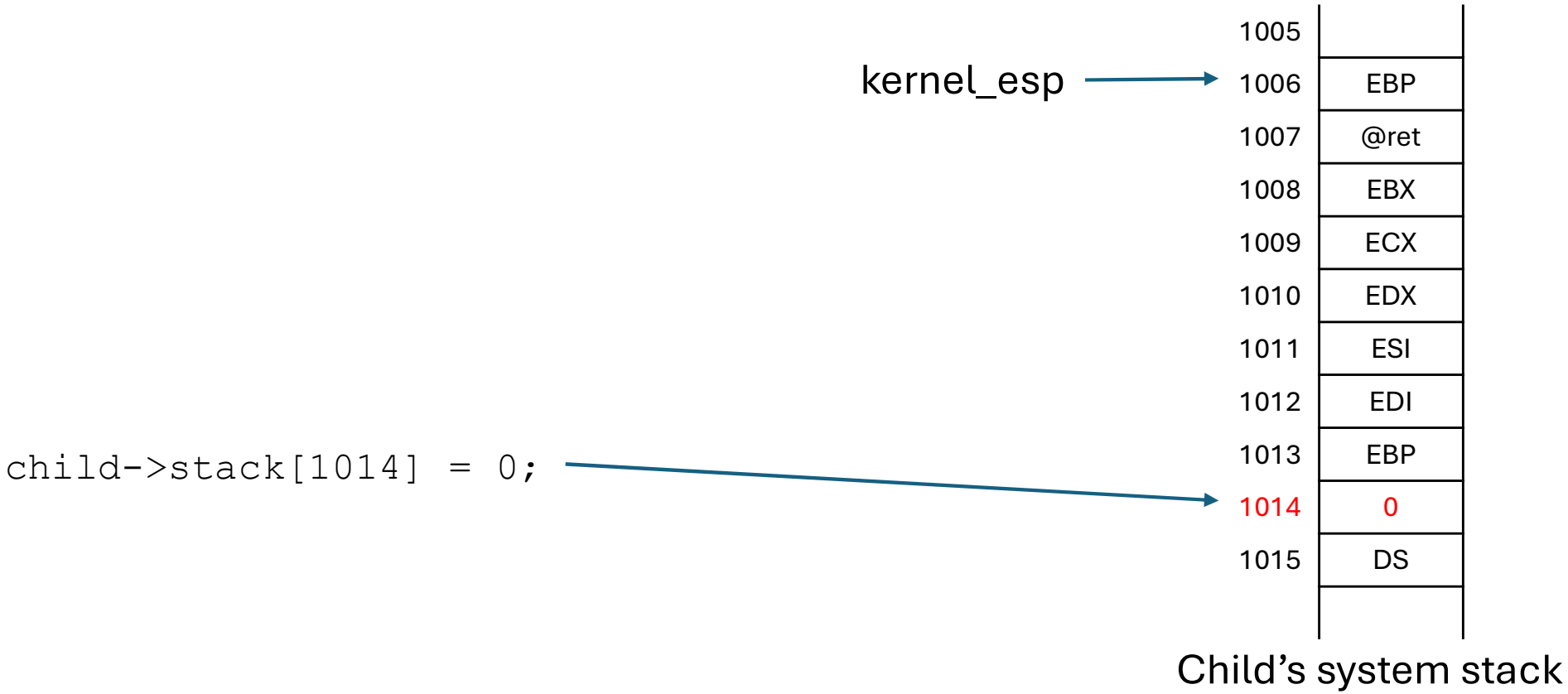
```
clock_handler:
          SAVE_ALL
          call clock_routine
clock_handler_end:
          EOI
          RESTORE_ALL
          IRET
```

```
child->stack[1007] = clock_handler_end;
```



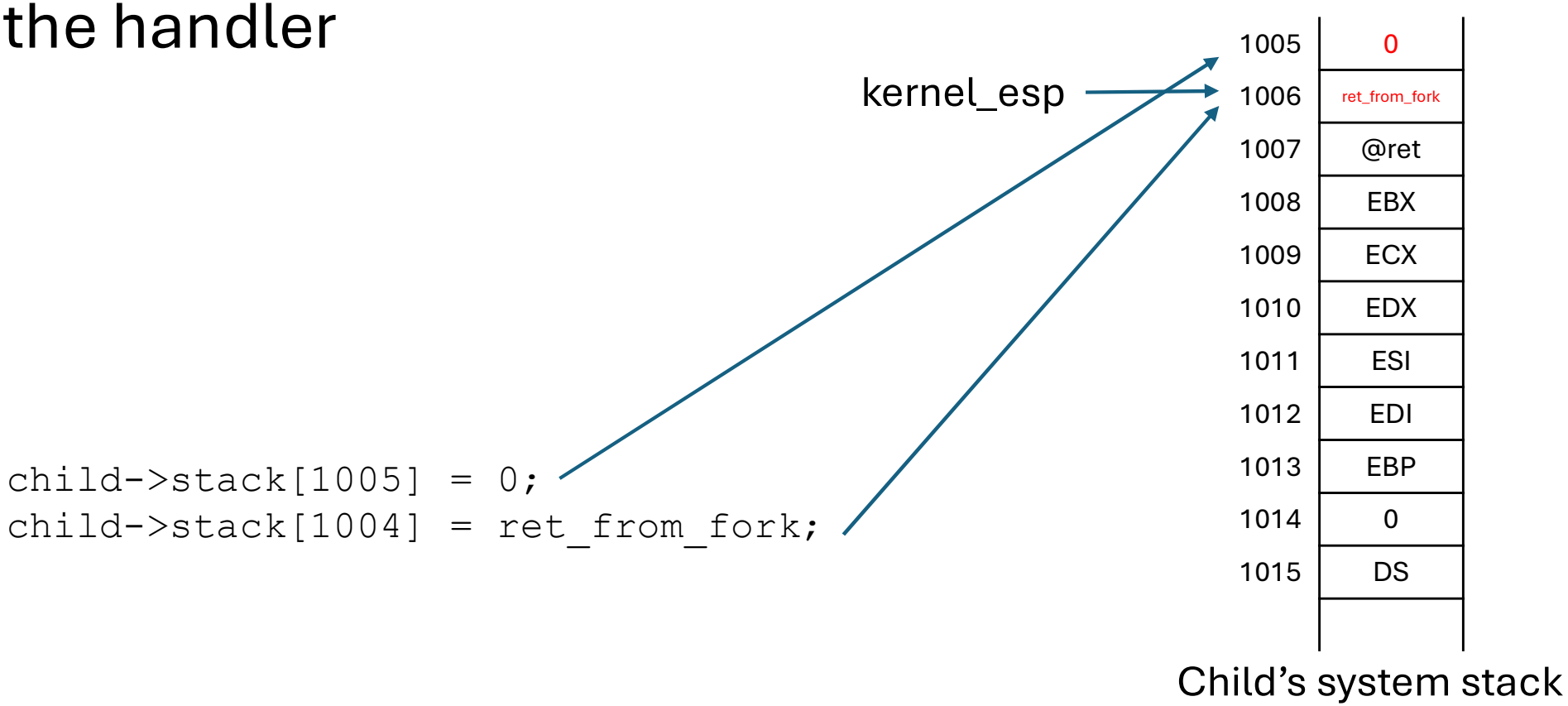
Solution step 2

- Modify the value of EAX with a 0 in the child's system stack in sys_fork



Solution step 3

- Since task_switch will use the EBP and @ret pointed by kernel_esp, those positions will be modified for, first, return to ret_from_fork and, in ret_from_fork, return to the handler

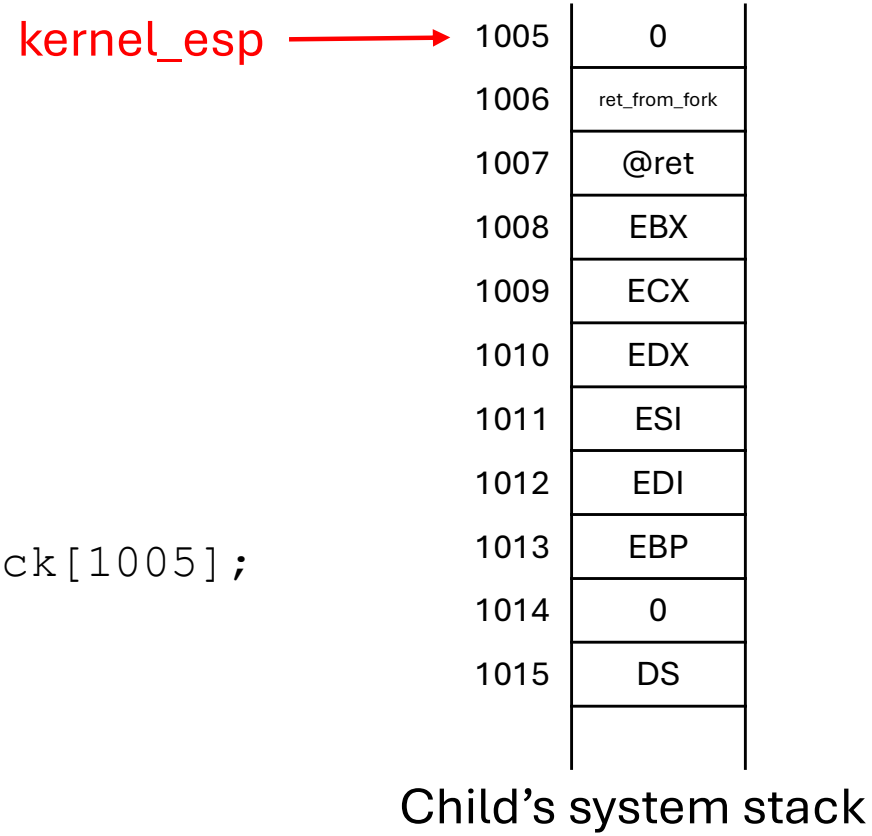



```
child->stack[1005] = 0;
child->stack[1004] = ret_from_fork;
```

Solution step 4

- And make child's kernel_esp point to that 0

```
child->task.kernel_esp = &child->stack[1005];
```

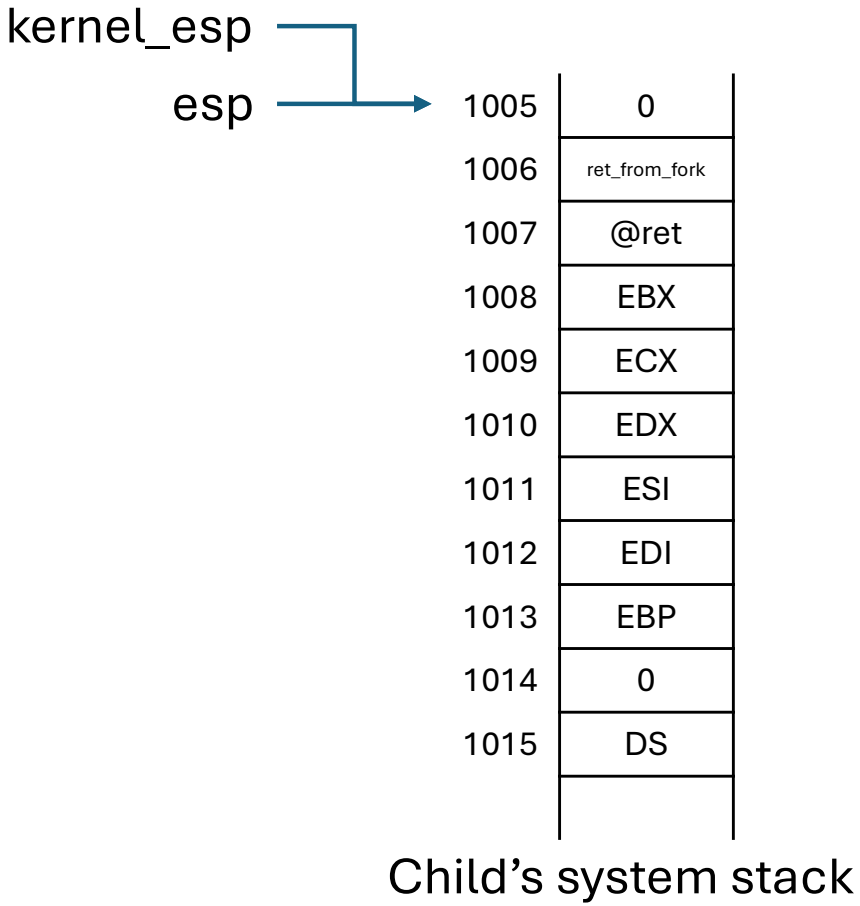


Child's system stack

How the solution works

- Whenever the scheduler selects the child process to execute, task_switch will put it in the CPU

```
void task_switch(struct task_struct *new)
{
…
            esp = new->kernel_esp;
            popl %ebp
            ret
}
```

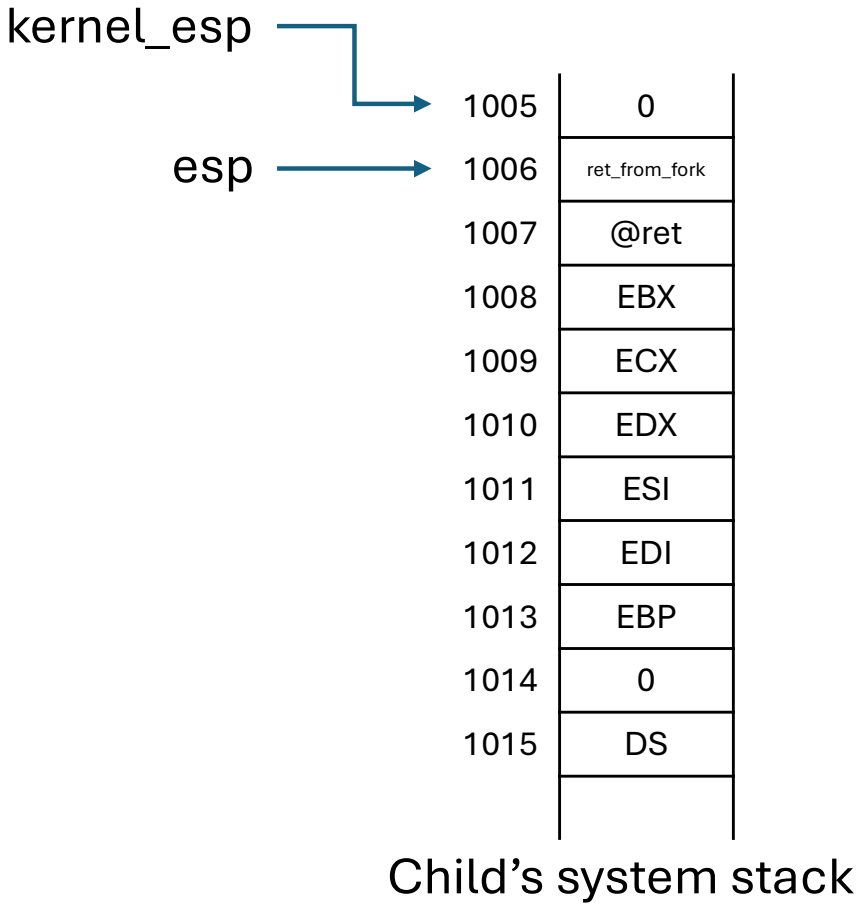


How the solution works

- Whenever the scheduler selects the child process to execute, task_switch will put it in the CPU

```
void task_switch(struct task_struct *new)
{
…
            esp = new->kernel_esp;
            popl %ebp
            ret
}
```

This will put 0 in ebp but this value will never be used



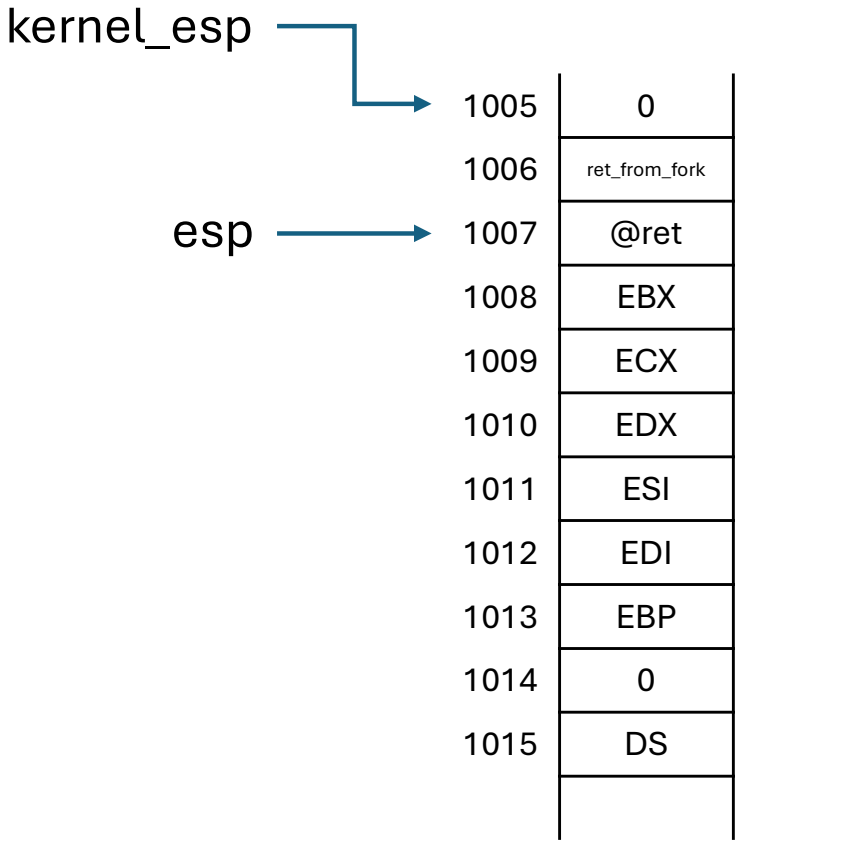
Child's system stack

How the solution works

- Whenever the scheduler selects the child process to execute, task_switch will put it in the CPU

```
void task_switch(struct task_struct *new)
{
…
          esp = new->kernel_esp;
          popl %ebp
          ret
}
```

This will return to ret_from_fork

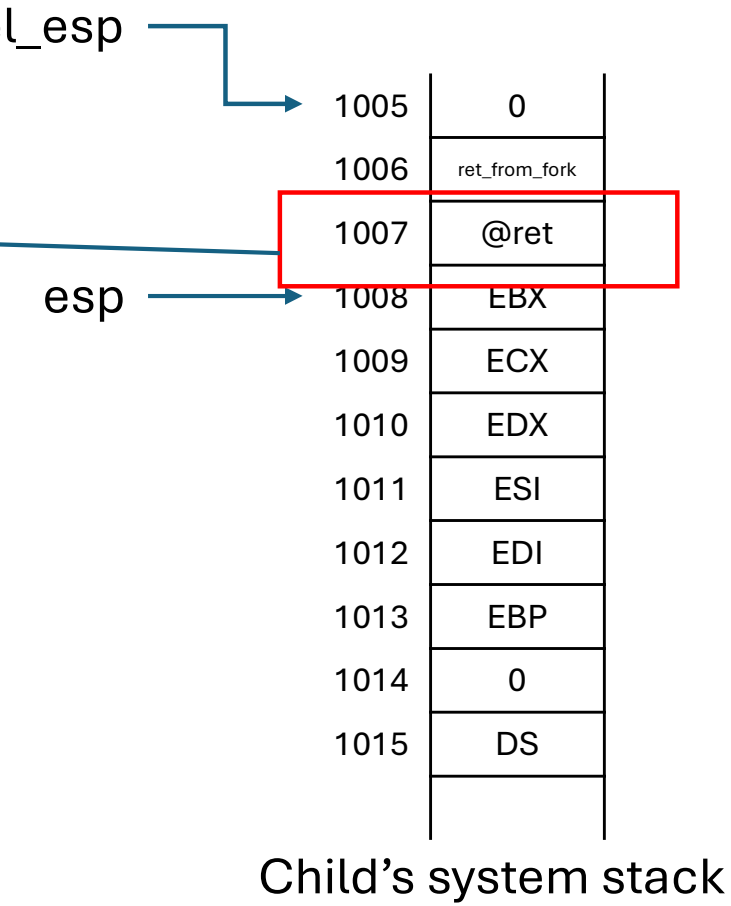


Child's system stack

How the solution works

- The first line executed in the clock_handler is EOI, reenabling the hardware interrupts

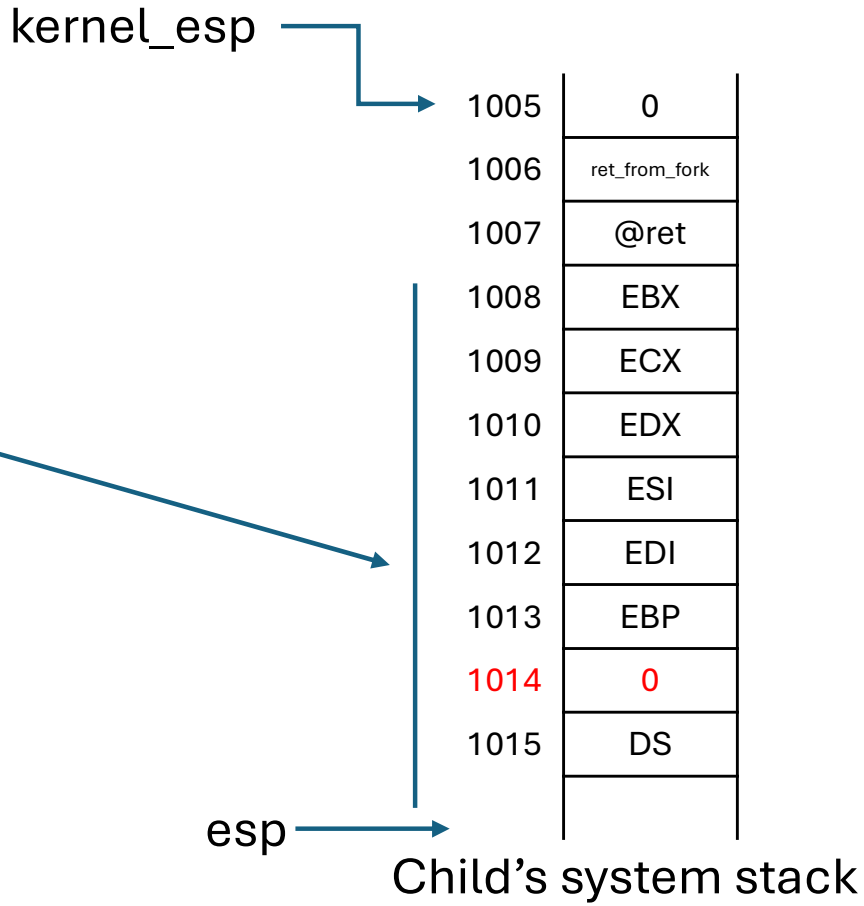
```
clock_handler:
          SAVE_ALL
          call clock_routine
clock_handler_end:
          EOI
          RESTORE_ALL
          IRET
```



How the solution works

- Finally, the clock_handler will execute RESTORE_ALL recovering the software context with the correct value of EAX

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



Finishing sys_fork

- After correctly setting up the system stack of the child process, sys_fork performs 2 more actions
- Enqueue the child process in the ready queue for being selectable by the scheduler
- Returns the PID of the child process
- After that, the syscall finishes returning to user mode

Zombie state

- When a process is in Run state, it can die
- For that, it executes:

```
int exit(int status);
```

- What sys_exit does:
  - Kills all threads in the process
  - Frees al resources:
    - Task_unions of the other threads in the process
    - All memory
    - Other structures
  - Pass the process to Zombie

Process hierarchy

- Every process has a pointer to its parent's task_struct
- Every process has two list of children:
  - Alive: list of children that are alive
  - Zombie: list of children that are zombie
- In sys_fork, all these fields are updated to be consistent:

```
…
child->parent = current();
list_add_tail(&child->task.anchor, &current()->alive);
…
```

- This creates an explicit hierarchy of processes in which a process knows its children and its parent

Passing a process to zombie

- So, in sys_exit a process must update its state in its parent's lists and launch the scheduler to abandon the CPU
- Notice that the union task_union of the thread that has called sys_exit is not deallocated, but it remains in the zombie state to allow the parent process to read its finalization status (parameter of the exit syscall)

```
…
current()->finalization_status = status;
list_del(&current()->task.anchor);
list_add(&current()->task.anchor, &parent->zombie);
sched_next();
…
```

Freeing a zombie process

- A zombie can deallocate the task_union of one of its children in zombie with the wait syscall:

```
// Returns -1 if the process has no children
// Returns the PID of one of the children
     process in Zombie
// If no children is in Zombie, blocks the
     process until one of its children
     processes becomes a Zombie
int wait(int *status);
```

Checking for child processes in sys_wait

- Sys_wait checks if process has any child process. If not, it returns –ENOCHLD:

```
int sys_wait(int *status)
{
        if ((list_empty(&current()->zombie)) &&
            (list_empty(&current()->alive)))
               return -ENOCHLD;
        …
}
```

Checking for child zombie processes in sys_wait

- If the process has any child process, sys_waits checks if any child process is in the zombie state and if it is the case, returns its finalization status and deallocates its union task_union

```
int sys_wait(int *status)
{
        if (!list_empty(&current()->zombie)
        {
                int PID;
                struct task_struct *t;
                // Must use list_head_to_task_struct
                t = lh2ts(list_first(&current()->zombie));
                list_del(&t->anchor);
                *status = t->status;
                PID = t->PID;
                free_frame(current()->PT, get_frame(current()->PT, t >> 12));
                del_ss_pag(current()->PT, t >> 12);
                set_cr3(current()->PT);
                return PID;
        }
}
```

- At this point, you should understand this code!!!

Blocking the parent process in sys_wait

- Finally, if the process has children but they are not in zombie, the process must block
- A process is blocked when:
  - It is alive
  - It is neither in RUN nor in READY
- Basically, this means that the process is not using the CPU and the process cannot be selected by the scheduler to pass to the CPU
- Even if the blocking state will be explained later, a simple blocking mechanism follows

A basic blocking mechanism

- In sys_wait, when the process must block waiting for any of its child process become a zombie, it must:
  - Mark this situation
  - Call the scheduler to abandon the CPU
- To mark that the process is blocked, a new field in its task_struct is used: blocked
- So, in sys_wait, the code to block a process is:

```
int sys_wait(int *status)
{
        …
        if (list_empty(&current()->zombie))
        {
                current()->blocked = 1;
                sched_next();
        }
        …
}
```

A basic unblocking mechanism

- A blocked process, waiting for its children to become zombie, will remain in that state until one of its children finish.
- So, the process will be unblocked by one of its children when executing sys_exit
- Unblocking means that the process can resume its execution

```
int sys_exit(int status)
{
        …
        if (current()->parent->blocked == 1)
        {
                current->parent->blocked = 0;
                list_add_tail(&current()->parent, &Readyqueue);
        }
        …
}
```

Threads

- Up to now, the OS is only able to create processes with one thread
- So, speaking about processes or threads is irrelevant because, remember, in Linux is nearly the same at this point
- Now, threads will be added to the OS
- You must review what resources belong to processes and what to threads

Preferring threads

- The use of threads is preferable to execute parallel tasks
- The creation of a thread is faster than the creation of a process
- The destruction of a thread is faster than the destruction of a process
- And threads can communicate and synchronize using the shared resources of a process, mainly, memory
  - No OS intervention is needed what improves the parallelism and reduces overhead of privilege level changing

Creating a thread

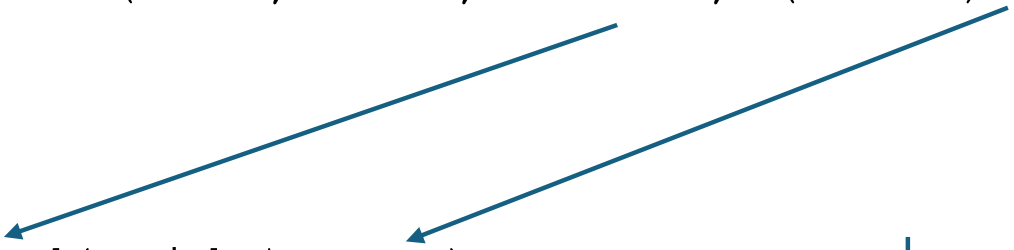
- The creation of a thread has to major different from creating a process:
  - It does not copy those resources belonging to the process, mainly, memory
  - The user must indicate where the code of the thread is
- The system call to create a thread is:

```
int pthread_create(pthread_t *id, pthread_attr_t *attr,
        void * (*function)(void *param), void *param);
```

- In which:
  - function: is the name of the function where the code of the thread is
  - arg: is the argument passed to the function of the thread when it is executed

Using pthread_create

```
pthread_create(NULL, NULL, thread, (void*)8);
```



```
void *thread(void *param)
{
        int value = (int)param;
        // Do some interesting stuff
        pthread_exit(0);
}
```

Thread function that will be executed in parallel with the code of the other threads in the process

pthread_create

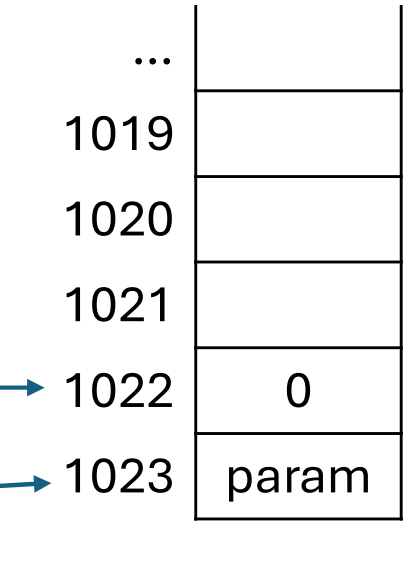
- To create a thread, pthread_create must:
- Allocate a union task_union (similar to sys_fork)
- Initialize that task_union (by copying the current task_union, as in sys_fork)
- Assign a Thread IDentifier (TID)
- Allocate a user stack
- Initialize the user stack
- Allocate the Thread Local Storage
- Prepare the system stack
- Enqueue the thread in the ready queue
- Return 0

Creating the user stack

- Once all OS resources has been allocated for the new thread, pthread_create must allocate and initialize the user stack
- For that, it allocates one physical frame and maps it in an empty entry of the PT of the process
- After that, a normal activation frame for a function with one parameter must be created manually
- By now, the return address in the activation frame of the function will be 0 (it will be changed later)

Creating the user stack

```
int syspthread_create(pthread_t *id, pthread_attr_t *attr,
        void* (*function)(void*), void *param)
{
…
        // Allocate and initialize a user stack
        int frameID = alloc_frame();
        int free_frame = find_free_frame(processPT);
        set_ss_pag(processPT, free_frame, frameID);
        long *user_stack = (long*)(free_frame << 12);
        user_stack[1023] = param;
        user_stack[1022] = 0; // return address
…
}
```



Thread-local storage

- The thread-local storage is a memory section, usually a memory page, inside the process memory but only accessible for one thread
- Its implementation is very dependent on the operating system and architecture
- it is very important since there the variable errno is stored

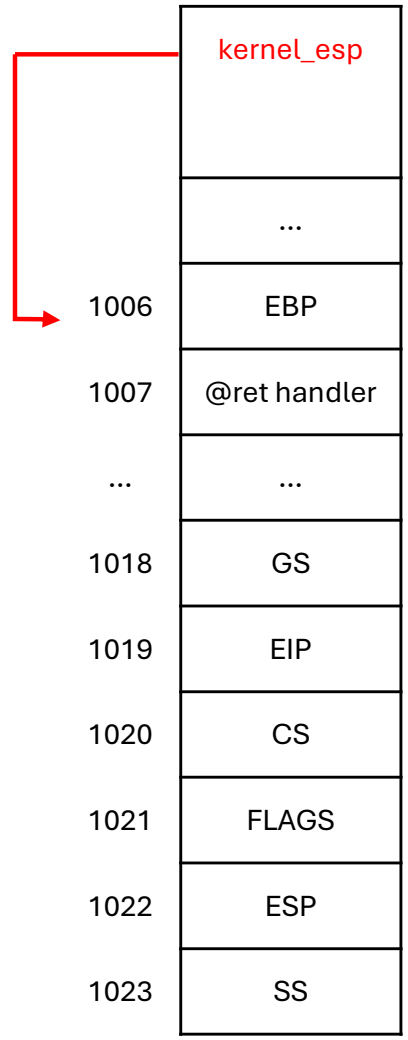
Preparing the thread's system stack

- As in sys_fork, the new thread already has a valid system stack since it has been copied from the current thread
- Besides, there are three differences:
- Pthread_create does not return any value to the new thread (EAX in software context is not modified)
- There isn't any function similar to ret_from_fork for threads
- When returning to user mode, the first instruction to execute is the one of the thread's function (passed as parameter) using the previously allocated user stack

Preparing the system stack

- First, kernel_esp must point to the ebp of function sys_pthread_create

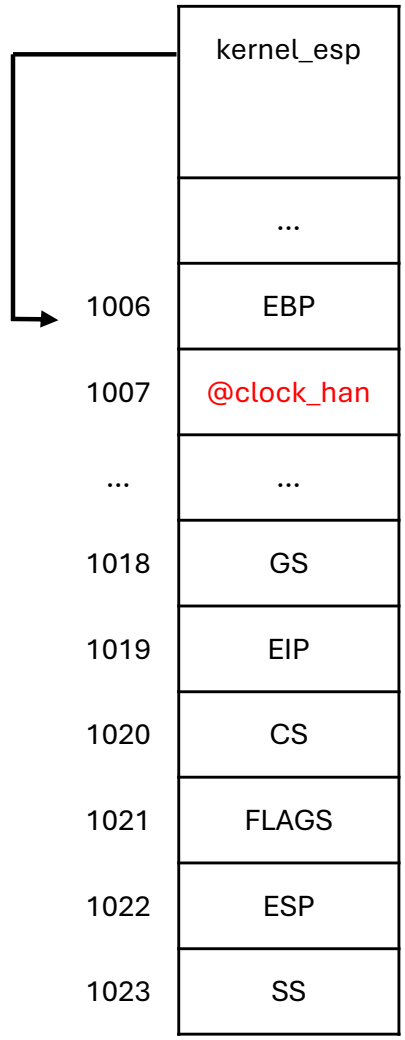
```
int sys_pthread_create(pthread_t *id, pthread_attr_t *attr,
            void* (*function)(void*), void *param)
{
…
            new->task.kernel_esp = &new->stack[1006];
…
}
```



Preparing the system stack

- Modify the return address of the handler for that in the clock_handler

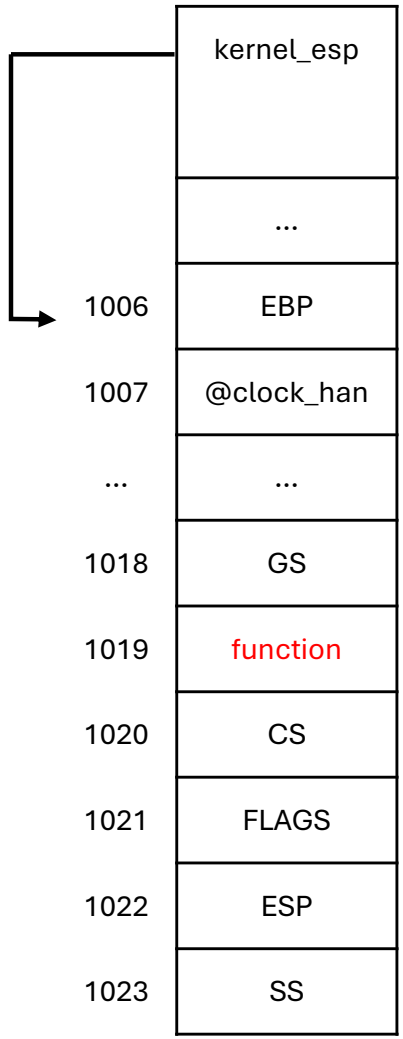
```
int sys_pthread_create(pthread_t *id, pthread_attr_t *attr,
            void* (*function)(void*), void *param)
{
…
            new->task.kernel_esp = &new->stack[1006];
            new->stack[1007] = clock_handler_end;
…
}
```



Preparing the system stack

- Modify the return address to user code with the thread function

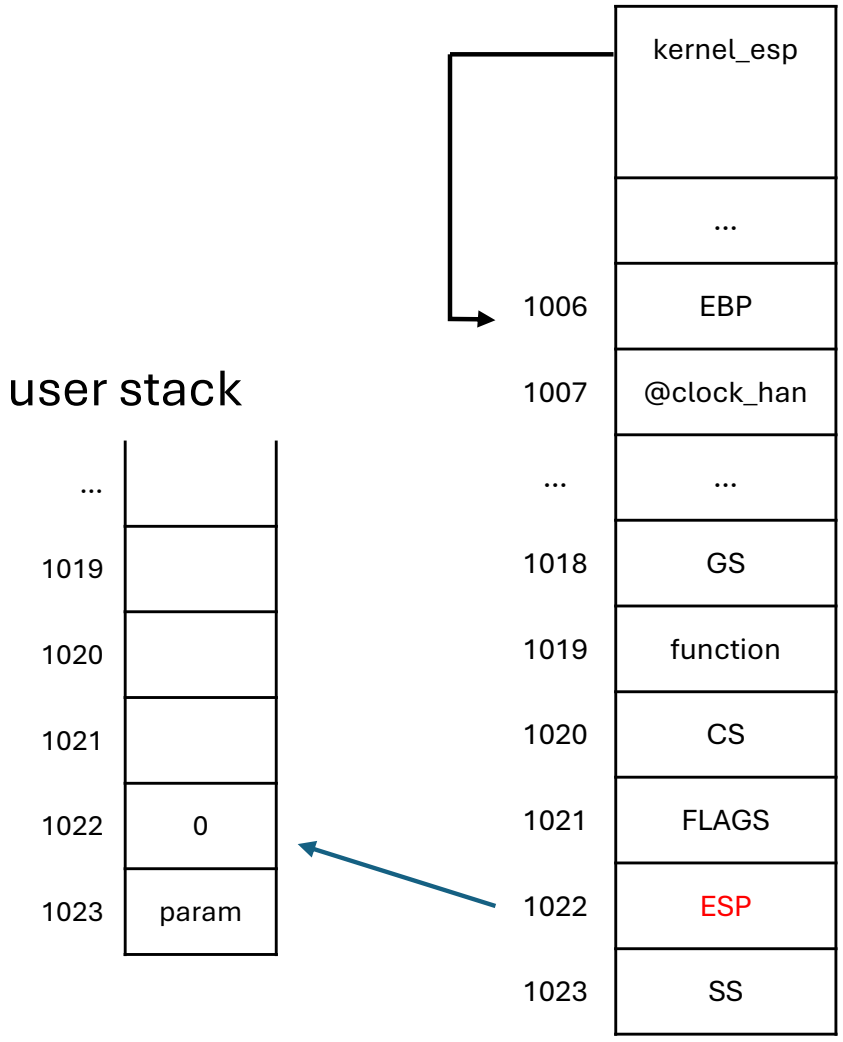
```
int sys_pthread_create(pthread_t *id, pthread_attr_t *attr,
            void* (*function)(void*), void *param)
{
…
            new->task.kernel_esp = &new->stack[1006];
            new->stack[1007] = clock_handler_end;
            new->stack[1019] = function;
…
}
```



Preparing the system stack

- And modify the ESP in the system stack to point to the allocated user stack

```
int sys_pthread_create(pthread_t *id, pthread_attr_t *attr,
            void* (*function)(void*), void *param)
{
…
            new->task.kernel_esp = &new->stack[1006];
            new->stack[1007] = clock_handler_end;
            new->stack[1019] = function;
            new->stack[1022] = &user_stack[1022];
…
}
```

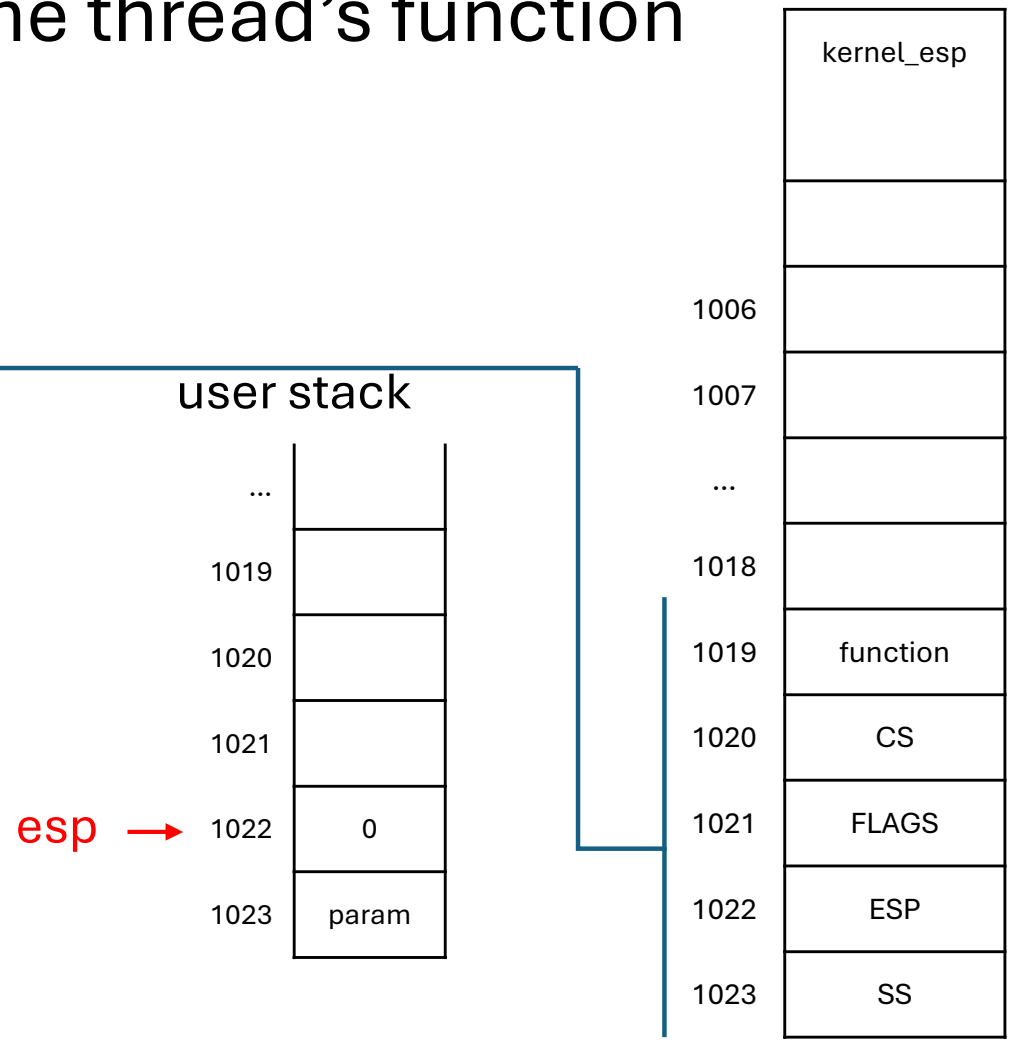


Returning to OS mode

- When the IRET instruction of the clock_handler is executed, it will return to the thread's function

```
clock_handler:
            SAVE_ALL
            call clock_routine
Clock_handler_end:
            EOI
            RESTORE_ALL
            iret
```

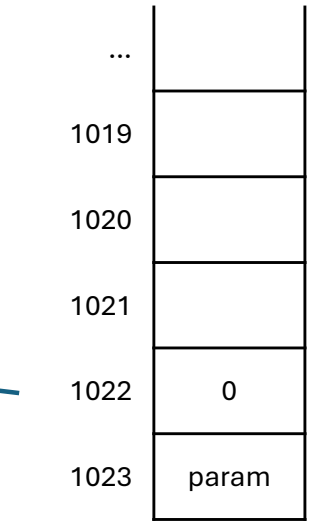
```
void *function(void *param)
{
…
}
```



Problem with the user stack

- Whenever a thread must exit, it must call pthread_exit
- But the user does not have to write it or simply it can be forgotten
- Even, the thread's function can use return to finish, what will cause a segmentation fault since the return address of the thread's function is 0

```
void *function(void *param)
{
…
            movl %ebp, %esp
            popl %ebp
            ret
}
```



Ret will take 0 as the return address, what will raise a #PF fault

Ensuring that pthread_exit will be called

- To ensure that pthread_exit will always be called when finalizing a thread, instead of directly calling the thread's function, call the following function

```
void thread_wrapper(void * (*function)(void*), void *param)
{
        void * result = function(param);
        pthread_exit(result);
}
```

- This function calls the thread's function and always execute pthread_exit in case the user does not do it
- This function is available in the OS libraries (lib.so, ntdll.dll)

Modifications to call thread_wrapper

- Since this function is in user memory, the OS does not know its memory address. The wrapper of sys_pthread_create must be modified to pass the address of thread_wrapper as fifth parameter of the syscall

```
int pthread_create(pthread_t *id, pthread_attr_t *attr, void * (*function)(void*), void *param)
{
            pushl %ebp
            movl %esp, %ebp
            pushl %ebx
            pushl %esi
            pushl %edi
            movl 8(%ebp), %ebx
            movl 12(%ebp), %ecx
            movl 16(%ebp), %edx
            movl 20(%ebp), %esi
            movl $thread_wrapper, %edi
            movl %eax, $15
            int $0x80
            …
}
```

Notice that the wrapper of the syscall has 4 parameters but the syscall will have 5

Modifications to call thread_wrapper

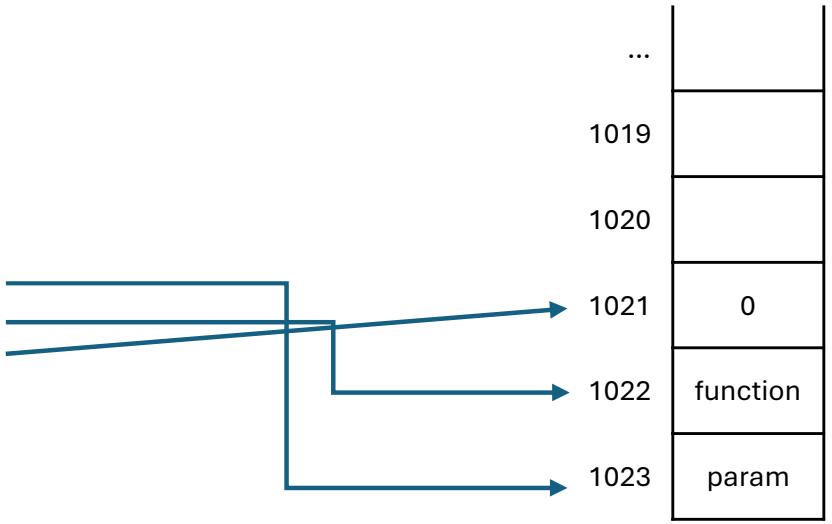
- Modify the header of sys_pthread_create to accept a fifth parameter

```
int sys_pthread_create(pthread_t *id,
        pthread_attr_t *attr,
        void * (*function)(void*),
        void *param,
        void * (*wrapper)(void*)
        )
{
…
}
```

Modifications to call thread_wrapper

- Modify the creation of the user stack. Now it is calling the thread_wrapper, so it has 2 parameters

```
int sys_pthread_create(pthread_t *id,
           pthread_attr_t *attr,
           void * (*function)(void*),
           void *param,
           void * (*wrapper)(void*)
           )
{
…
           user_stack[1023] = param;
           user_stack[1022] = function;
           user_stack[1021] = 0;
…
}
```

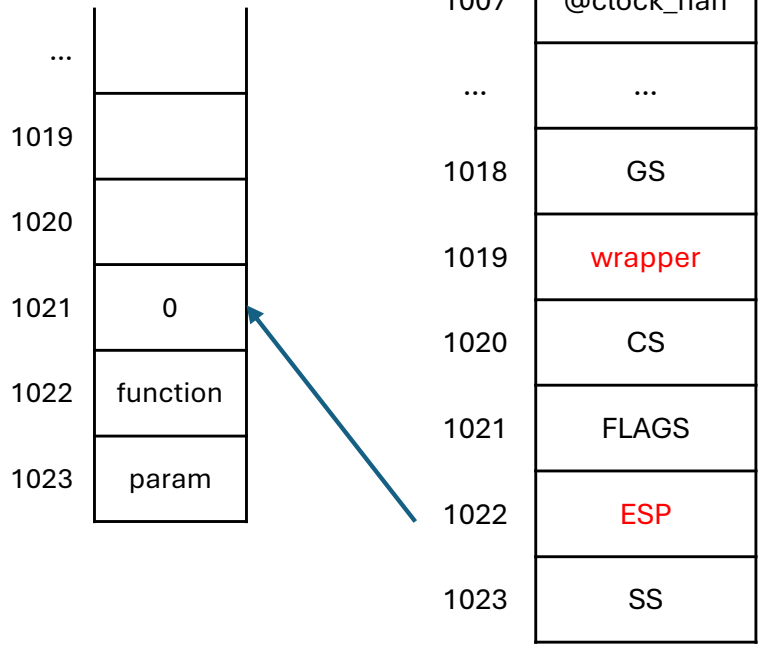


Keep on pushing 0 as return address since it will never be used

Modifications to call thread_wrapper

- And modify the hardware context accordingly to execute the wrapper with the new shape of the user stack

```
int sys_pthread_create(pthread_t *id,
            pthread_attr_t *attr,
            void * (*function)(void*),
            void *param,
            void * (*wrapper)(void*)
            )
{
…
            new->stack[1019] = wrapper;
            new->stack[1022] = &user_stack[1021];
…
}
```



Resource sharing

- Threads share the resources allocated to the process
- This could improve performance in parallel task since threads can communicate/synchronize using memory
- No OS execution required
- But resource sharing is not straightforward
- Remember that the OS has a scheduler which state is not known

Resource sharing example

- Imagine threads T1 and T2 cooperate to initialize array v1. For that, there is an integer pos which points to the next position of v1 to initialize

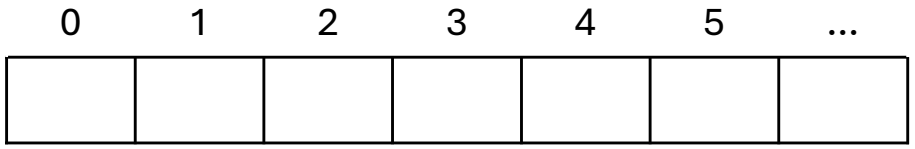
int v1[N];
int pos = 0;

Thread T1

```
while (pos < N)
{
    v1[pos] = D1;
    pos++;
}
```

Thread T2

```
while (pos < N)
{
    v1[pos] = D2;
    pos++;
}
```



pos = 0

Resource sharing example

- The execution starts with T1 who executes a full iteration

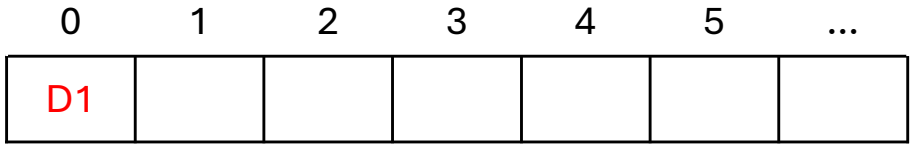
int v1[N];
int pos = 0;

Thread T1

```
while (pos < N)
{
    v1[pos] = D1;
    pos++;
}
```

Thread T2

```
while (pos < N)
{
    v1[pos] = D2;
    pos++;
}
```



pos = 1

Resource sharing example

- If now the scheduler selects T2 and this thread executes one iteration completely, it modifies the second position correctly

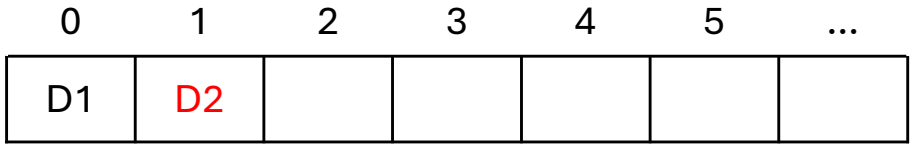
```
int v1[N];
int pos = 0;
```

Thread T1

```
while (pos < N)
{
    v1[pos] = D1;
    pos++;
}
```

Thread T2

```
while (pos < N)
{
    v1[pos] = D2;
    pos++;
}
```



pos = 2

Resource sharing example

- And threads continue its execution until pos == N. The array is correctly initialized

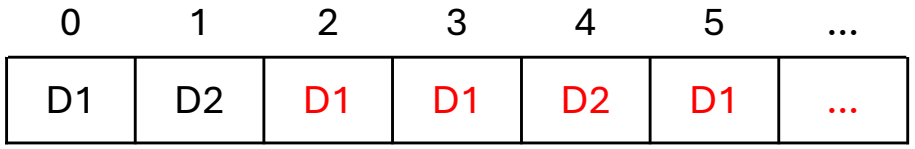
```
int v1[N];
int pos = 0;
```

Thread T1

```
while (pos < N)
{
    v1[pos] = D1;
    pos++;
}
```

Thread T2

```
while (pos < N)
{
    v1[pos] = D2;
    pos++;
}
```



pos = N

Resource sharing example

- Let's start again. Now T1 starts execution but it is preempted between the modification of the array and the increment of pos

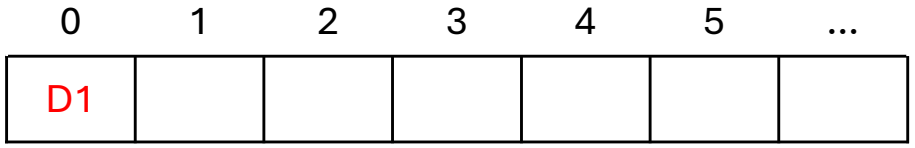
```
int v1[N];
int pos = 0;
```

Thread T1

```
while (pos < N)
{
    v1[pos] = D1;
    pos++;
}
```

Thread T2

```
while (pos < N)
{
    v1[pos] = D2;
    pos++;
}
```



pos = 0

Notice that pos has not been update by T1

Scheduler preempts T1 and selects T2

Resource sharing example

- The scheduler selects T2 who executes a complete iteration

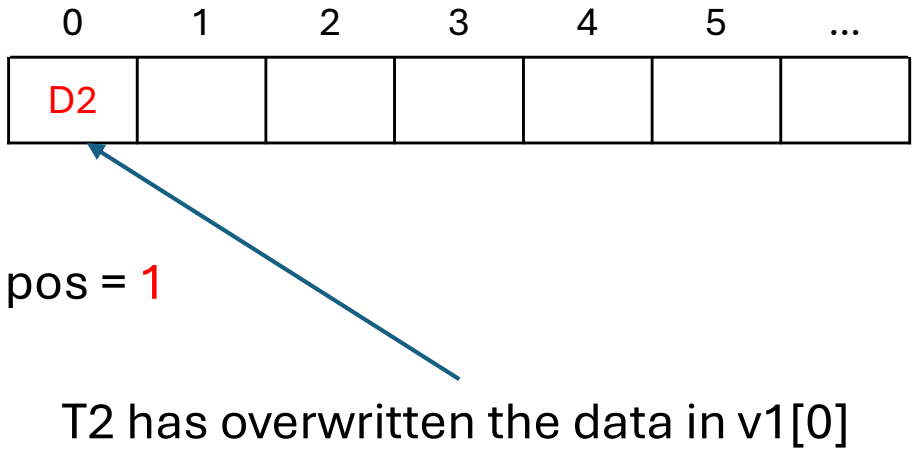
int v1[N];
int pos = 0;

Thread T1

```
while (pos < N)
{
    v1[pos] = D1;
    pos++;
}
```

Thread T2

```
while (pos < N)
{
    v1[pos] = D2;
    pos++;
}
```



Resource sharing example

- After that, the scheduler selects T1 to continue executing. Remember that T1 was preempted before executing pos++;

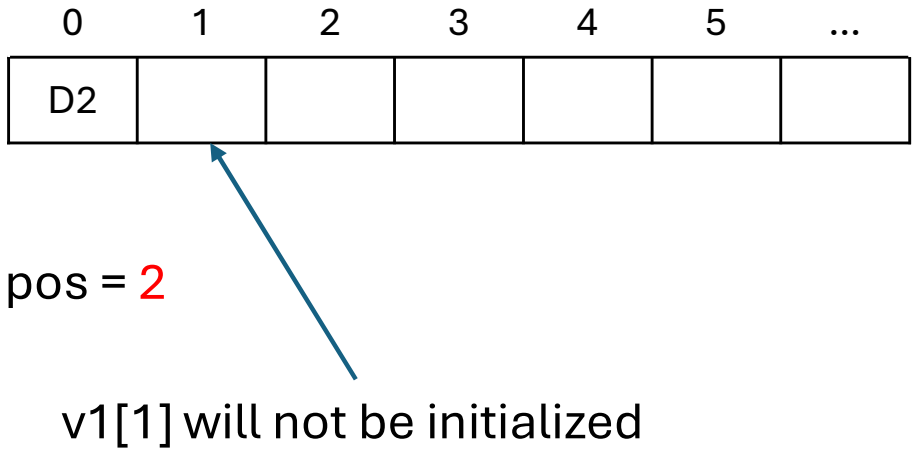
```
int v1[N];
int pos = 0;
```

Thread T1

```
while (pos < N)
{
    v1[pos] = D1;
    pos++;
}
```

Thread T2

```
while (pos < N)
{
    v1[pos] = D2;
    pos++;
}
```



Resource sharing example

- In this case v1 has not been correctly initialized. There is a race condition

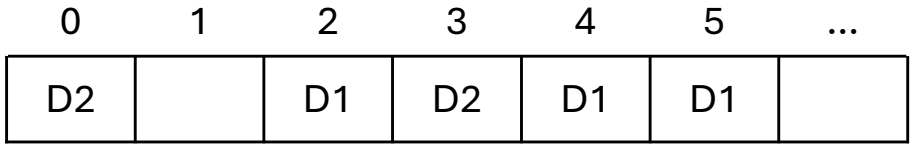
```
int v1[N];
int pos = 0;
```

Thread T1

```
while (pos < N)
{
    v1[pos] = D1;
    pos++;
}
```

Thread T2

```
while (pos < N)
{
    v1[pos] = D2;
    pos++;
}
```



pos = N

Race condition

- A race condition happens when the final result of executing more than one thread depends on the order of execution of the instructions of these threads
- Basically, because these threads are updating the value of global variables
- A race condition happens because the state of the scheduler cannot be known beforehand
- Solution: mutual exclusion

Mutual exclusion

- Critical region: set of consecutive lines of code that can present race conditions
  - Because they are updating global variables
- Mutual exclusion: if one thread is executing a critical region, avoid other threads to execute critical regions with the same global variables
- So, threads exclude each other when executing critical regions
- This does not mean that the scheduler cannot preempt a thread that is executing a critical region

Critical region

```
int v1[N];
int pos = 0;
```

Thread T1

```
while (pos < N)
{
    v1[pos] = D1;
    pos++;
}
```

Thread T2

```
while (pos < N)
{
    v1[pos] = D2;
    pos++;
}
```

- The user must detect those critical regions.
- The user must mark the start and end of a critical region with OS services
- The OS offers system calls to mark critical regions
  - Start: Let a thread enter into a critical region if no other thread is inside the critical region. If there is one, pause the thread that wants to enter
  - End: a thread that leaves a critical region notify one of the threads that are waiting to enter the critical region to continue execution

Marking critical regions

- The OS can provide more than one mechanism to mark critical regions.
- The user must decide which one to use
- All these mechanisms use a global variable to determine the access of the critical region
  - Usually, that global variable is used to protect the same set of global variables sensible of having race conditions
- Critical regions must be as short as possible. Long critical regions reduce parallelism
- OS mechanisms: spinlocks and semaphores

Spinlocks

- A spinlock is a synchronization lock mechanism used in concurrent programming to protect a shared resource (a critical section) from being accessed simultaneously by multiple threads or CPU cores.
- A thread waiting on a spinlock "spins" in a continuous loop ("busy-waiting") repeatedly checking whether the lock has become available
- The key is that the CPU provides an atomic instruction to return the value of a memory position while changing its value: test and set
- The memory coherence protocol and the invalidation of caches ensure the correctness of the atomic instruction

Test and set

- The typical implementation of a test and set in x86 is:

```
test_and_set:
    push %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ecx
    movl $1, %eax
    lock xchgl %eax, (%ecx)
    popl %ebp
    ret
```

Atomic instruction: exchanges the value in %eax with the value pointed by ecx

```
int test_and_set(int *a)
{
    int temp = *a;
    *a = 1;
    return temp;
}
```

Equivalent in high level (it does not work because it is not atomic)

Using spinlocks

- To use a spinlock, first, declare a global variable
- To mark the beginning of the critical region, use:

```
int a = 0;
while (test_and_set(&a));
```

- A thread will not enter in the critical region if the value of the global variable is 1. If the thread enters, it will put the value to 1
- To mark the end of the critical region, set the value of the spinlock variable to 0
- If there is any thread spinning testing the value of the spinlock variable, it will enter the critical region

Spinlock example

- If T1 is executing the critical region, lock will be 1. If T2 tries to enter the critical region it will spin in the loop until T1 sets lock to 0.

```
int v1[N];
int pos = 0;
int lock = 0;
```

Thread T1

```
while (pos < N)
{
    while (test_and_set(&lock));
    v1[pos] = D1;
    pos++;
    lock = 0;
}
```

Thread T2

```
while (pos < N)
{
    while (test_and_set(&lock));
    v1[pos] = D2;
    pos++;
    lock = 0;
}
```

Spinlocks pros and cons

- Pros: Extremely fast when the lock is only held for very short periods. It eliminates the overhead of a thread context switch or OS kernel scheduler interaction. It can be used in both user and OS privileged levels.
- Cons: Consumes high CPU usage during the wait time because the CPU is actively executing a loop. Overwhelms the memory bus.
- Best Use Case: Multi-core systems where critical sections are very small (e.g., updating a pointer, incrementing a counter, or operating system kernel routines).

Semaphores

- Idea: instead of spinning and using the CPU to enter a critical region, if a thread cannot enter, it will be blocked
- A semaphore is a structure with a variable that dictates how many units of a given resource are available and a queue to block threads

```
struct sem_t
{
        int count;
        struct list_head *blocked;
};
```

- Remember that blocking a thread means that the thread is neither running nor ready, but it is alive

Initialization of semaphores

- Before using a semaphore, it must be initialized
- The system call to initialize a semaphore is

```
int sem_init(sem_t *s, int count)
{
        s->count = count;
        INIT_LIST_HEAD(&s->blocked);
}
```

- The initial value for a semaphore will be discussed later

Marking the beginning of a critical region with semaphores

- After been initialized, a semaphore can be used
- The system call to mark the beginning of a critical region is:

```
int sem_wait(sem_t *s)
{
        s->count--;
        if (s->count < 0)
        {
                list_add(&current()->anchor, &s->blocked);
                sched_next();
        }
}
```

- If a thread cannot enter the critical region (s->count < 0) it is blocked in the queue of the semaphore and the scheduler is called to select another thread

Marking the end of a critical region with semaphores

- The system call to mark the end of a critical region is:

```
int sem_post(sem_t *s)
{
        s->count++;
        if (s->count <= 0)
        {
                struct list_head *e = list_first(&s->blocked);
                list_del(e);
                list_add(e, &ready_queue);
        }
}
```

- If the counter is <= 0 it means that some thread is blocked in the semaphore. Take the first one and enqueue it in the ready queue to continue its execution.

Semaphore example

- When T1 enters the critical region, sets the counter of the semaphore s to 0. If T1 is preempted by T2, when trying to enter the critical region, T2 will be blocked in the semaphore's queue. It will remain in the queue until T1 abandons the critical region (sem_post)

```
int v1[N];
int pos = 0;
sem_t s;
sem_init(&s, 1);
```

Thread T1

```
while (pos < N)
{
    sem_wait(&s)
    v1[pos] = D1;
    pos++;
    sem_post(&s)
}
```

Thread T2

```
while (pos < N)
{
    sem_wait(&s);
    v1[pos] = D2;
    pos++;
    sem_post(&s);
}
```

Initializing the semaphore to 0

- When a semaphore is initialized with a value of 1, it means that the semaphore will be used to delimitate critical regions. It is a mutual exclusion semaphore.
- When the semaphore is initialized with a value of 0, it will be used to synchronize two threads. It is a synchronization semaphore.
- A synchronization semaphore is used when the order of execution of the instructions of two threads wants to be enforced.

Example of a synchronization semaphore

- Imagine that instructions T1A and T1B must be always executed before T2A and T2B.

Thread T1	Thread T2
T1A	T2A
T1B	T2B
T1C	T2C
T1D	T2D

Example of a synchronization semaphore

- First, declare and initialize a synchronization semaphore

```
sem_t s;
sem_init(&s, 0);
```

Thread T1	Thread T2
T1A	T2A
T1B	T2B
T1C	T2C
T1D	T2D

Example of a synchronization semaphore

- T2A cannot be executed before T1B. If it is, it must wait until T1B has been executed

```
sem_t s;
sem_init(&s, 0);
```

Thread T1	Thread T2
	sem_wait(&s);
T1A	T2A
T1B	T2B
T1C	T2C
T1D	T2D

Example of a synchronization semaphore

- T2A will be notified when T1B is executed

```
sem_t s;
sem_init(&s, 0);
```

Thread T1	Thread T2
	sem_wait(&s);
T1A	T2A
T1B	T2B
sem_post(&s)	
T1C	T2C
T1D	T2D

Example of a synchronization semaphore

- So, if due to the scheduler, T2A is going to be executed before T1B, the thread T2 will be blocked in the queue of the semaphore and will remain there until T1B (sem_post) is executed

```
sem_t s;
sem_init(&s, 0);
```

Thread T1	Thread T2
	sem_wait(&s);
T1A	T2A
T1B	T2B
sem_post(&s)	
T1C	T2C
T1D	T2D

Example of a synchronization semaphore

- Finally, a semaphore can be initialized with a value > 1
- In that case, it is a resource semaphore
- It is used to protect a resource that can be used by several threads at the same time, or to protect a pool of resources
- Imagine that there are N resources of the same kind:

```
sem_t s;
sem_init (&s, N);

sem_wait(&s);
use_resource();
sem_post(&s);
```

Summary of initialization values

Initial value	Use
0	Synchronization of two threads
1	Mutual exclusion
> 1	Protect a pool of resources

Last note on semaphores

- Semaphores solve the busy-waiting problem of spinlocks
- The use of semaphores is advised when it is expected that waiting to enter a critical region will take long
- Semaphores only work if sem_init, sem_wait and sem_post are system calls. If not, they will present race conditions when accessing the semaphore