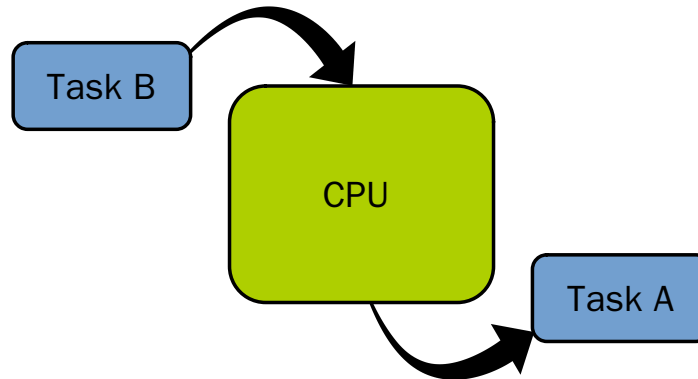


COMP26020 Programming Languages and Paradigms Part 1: C Programming

Case Study: Operating System Kernel

System Call and Context Switch

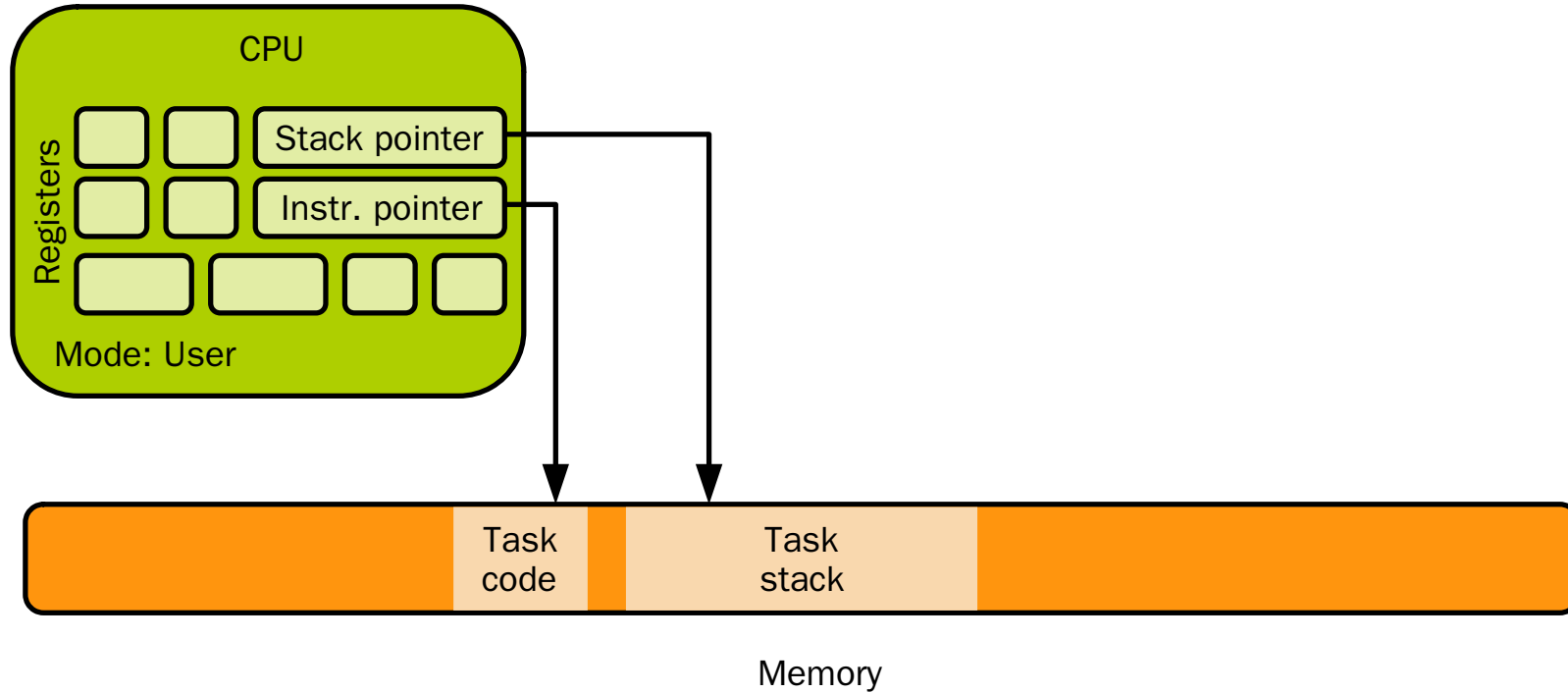
- C is the default language to write OS kernels
- Let's study the implementation of:
 - **System calls handling**
 - **Context switch** (replacement of a task running on the CPU by another one)



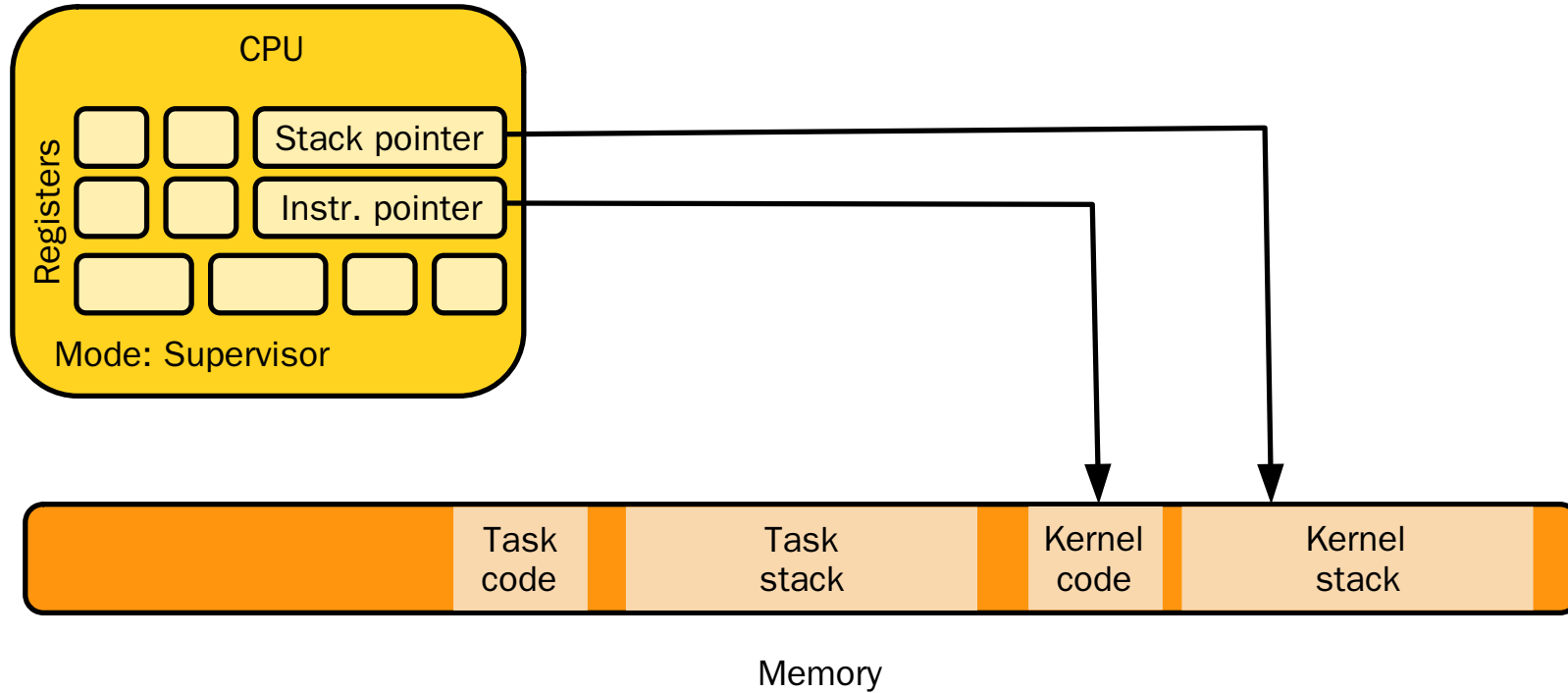
- Theory first, then implementation in the Linux kernel

System Calls in a Nutshell

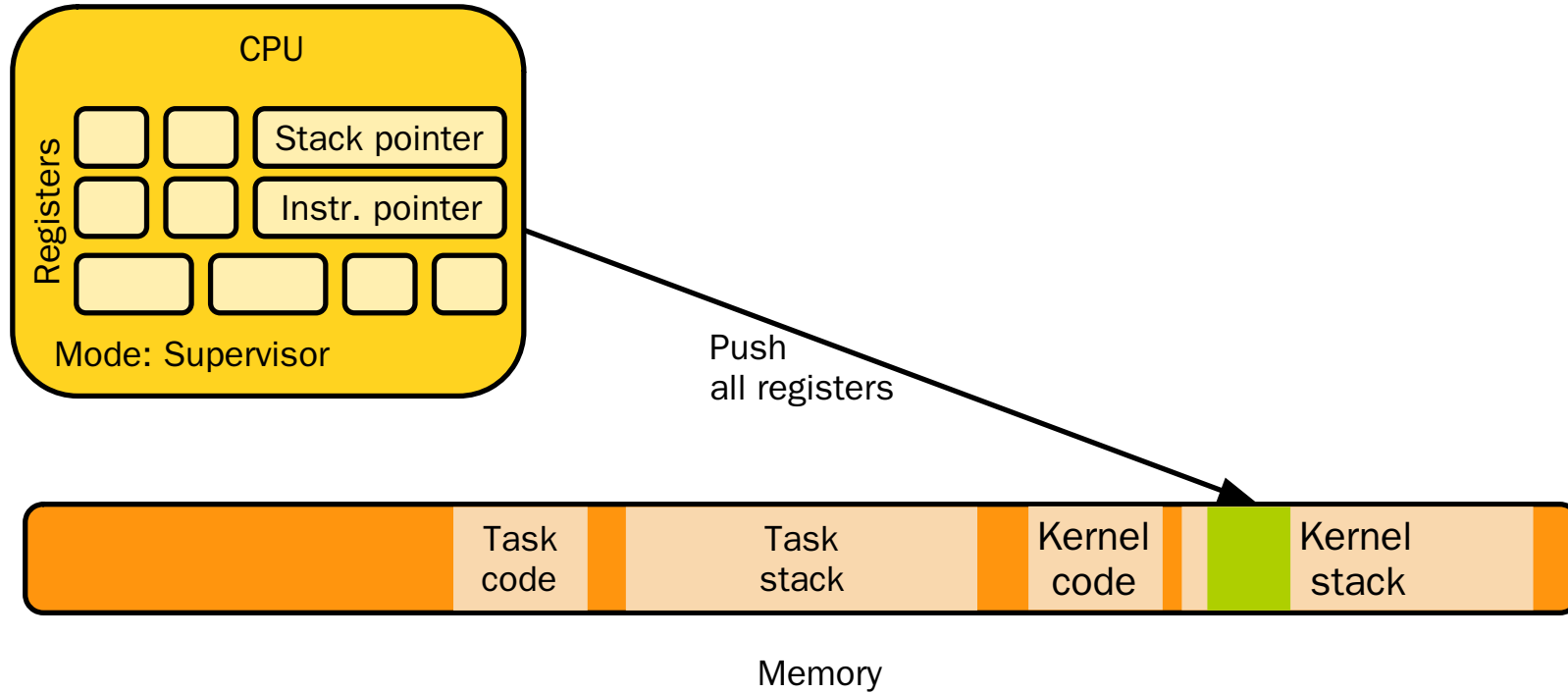
System Call in a Nutshell



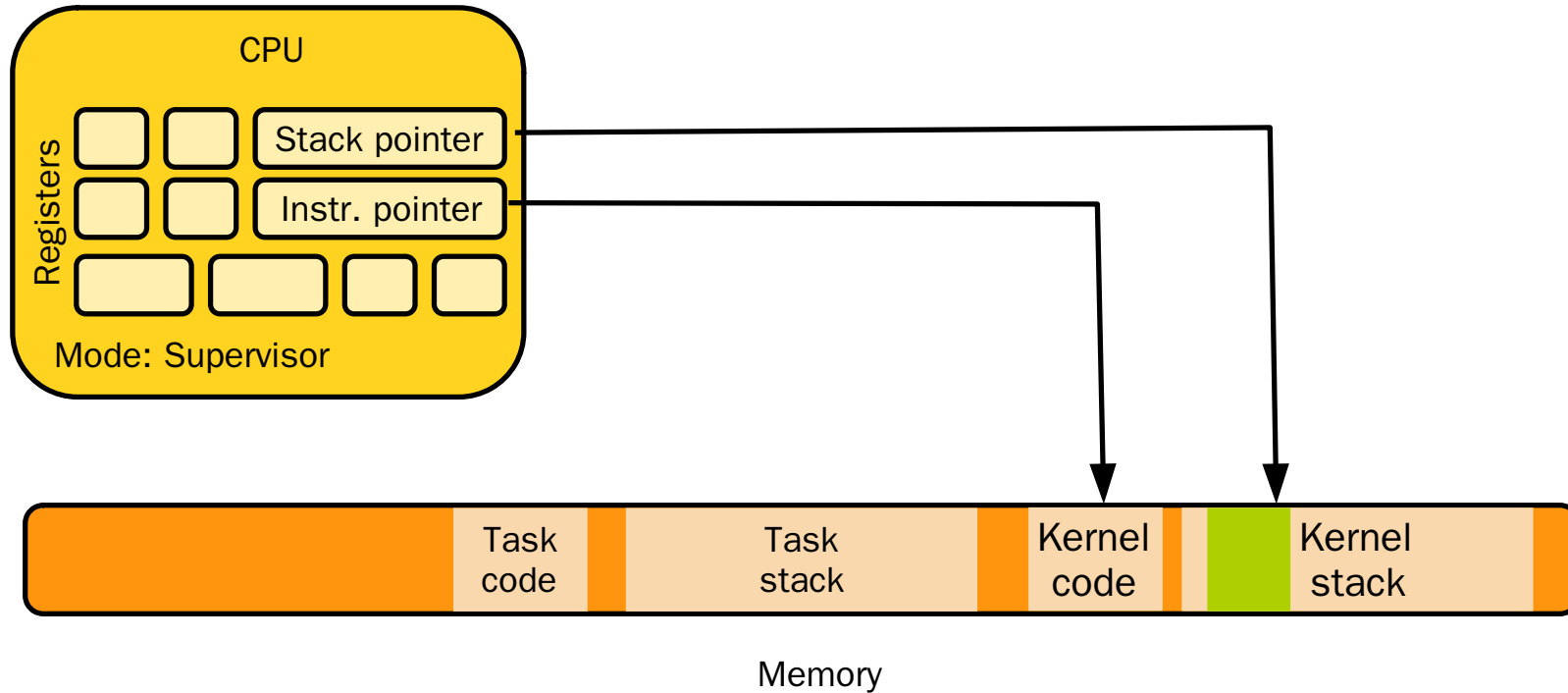
System Call in a Nutshell



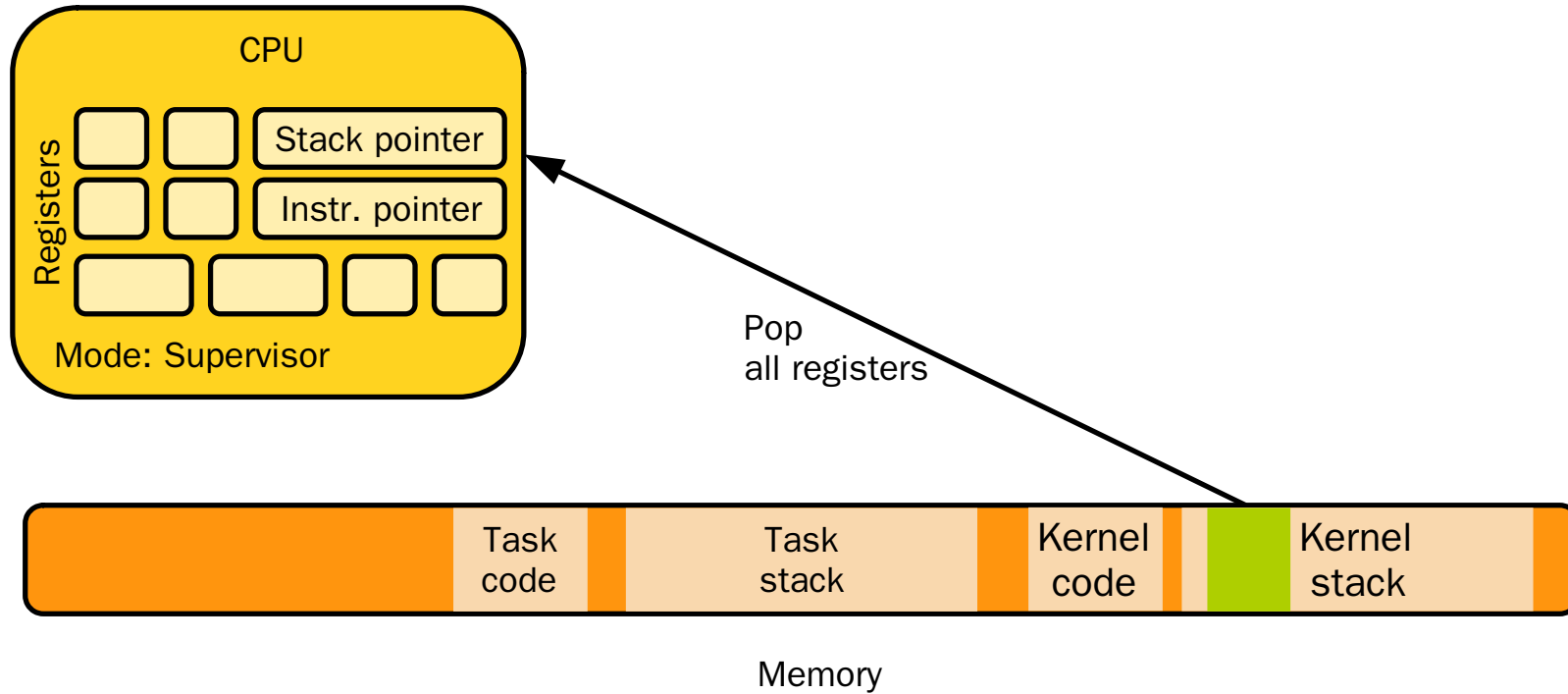
System Call in a Nutshell



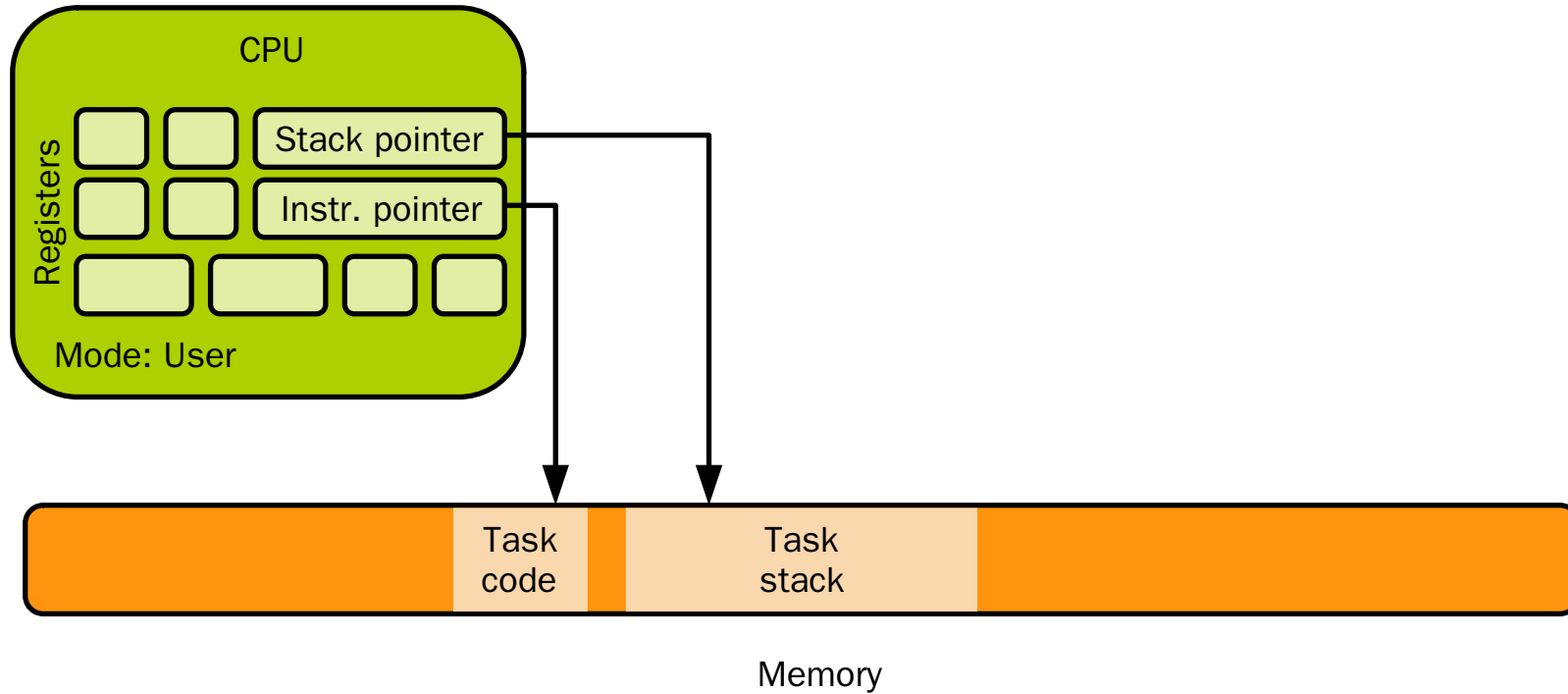
System Call in a Nutshell



System Call in a Nutshell

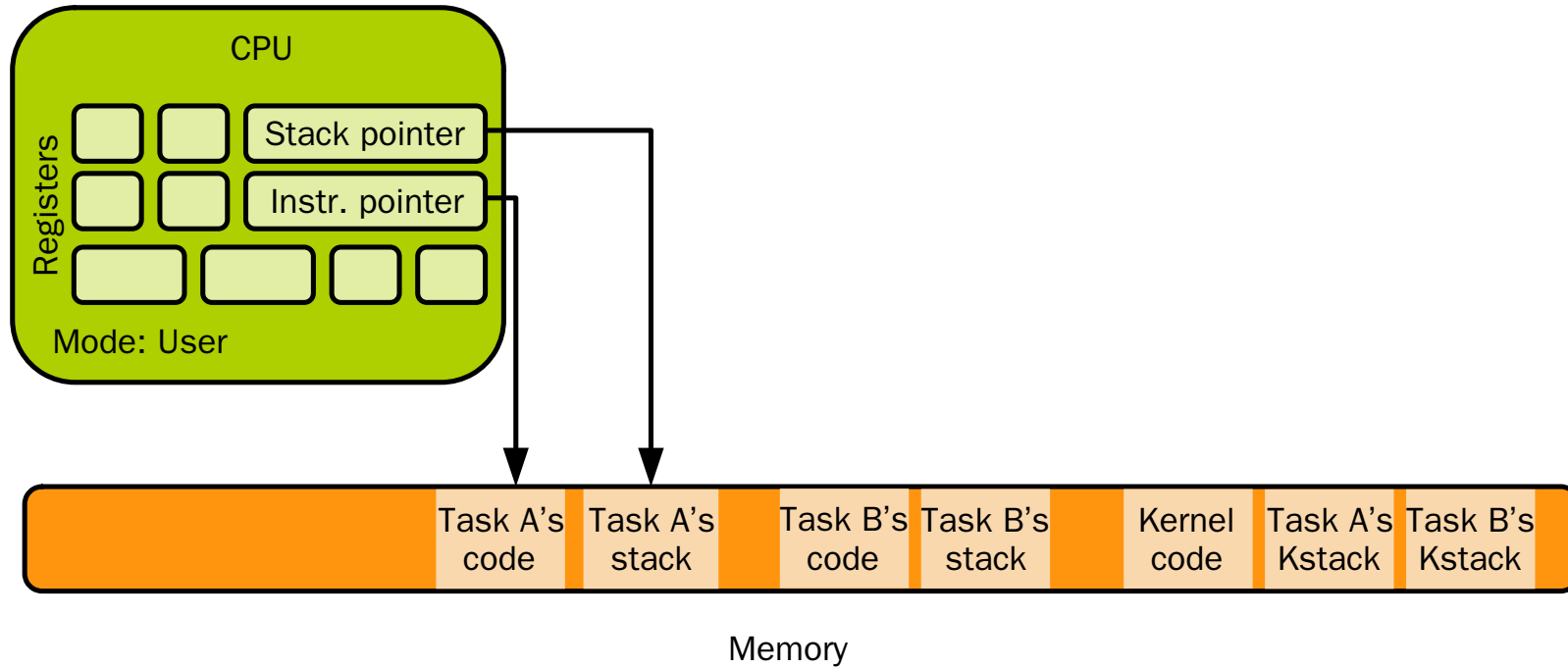


System Call in a Nutshell

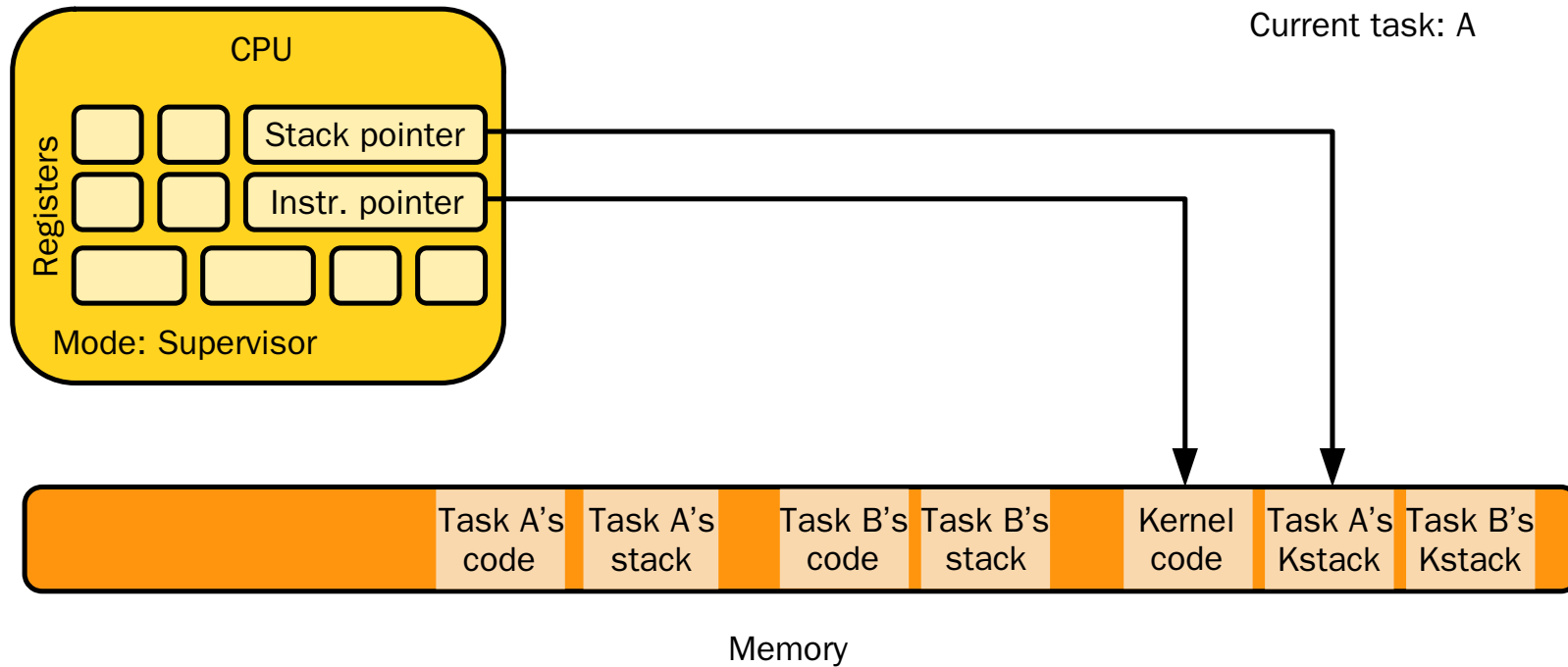


Context Switches in a Nutshell

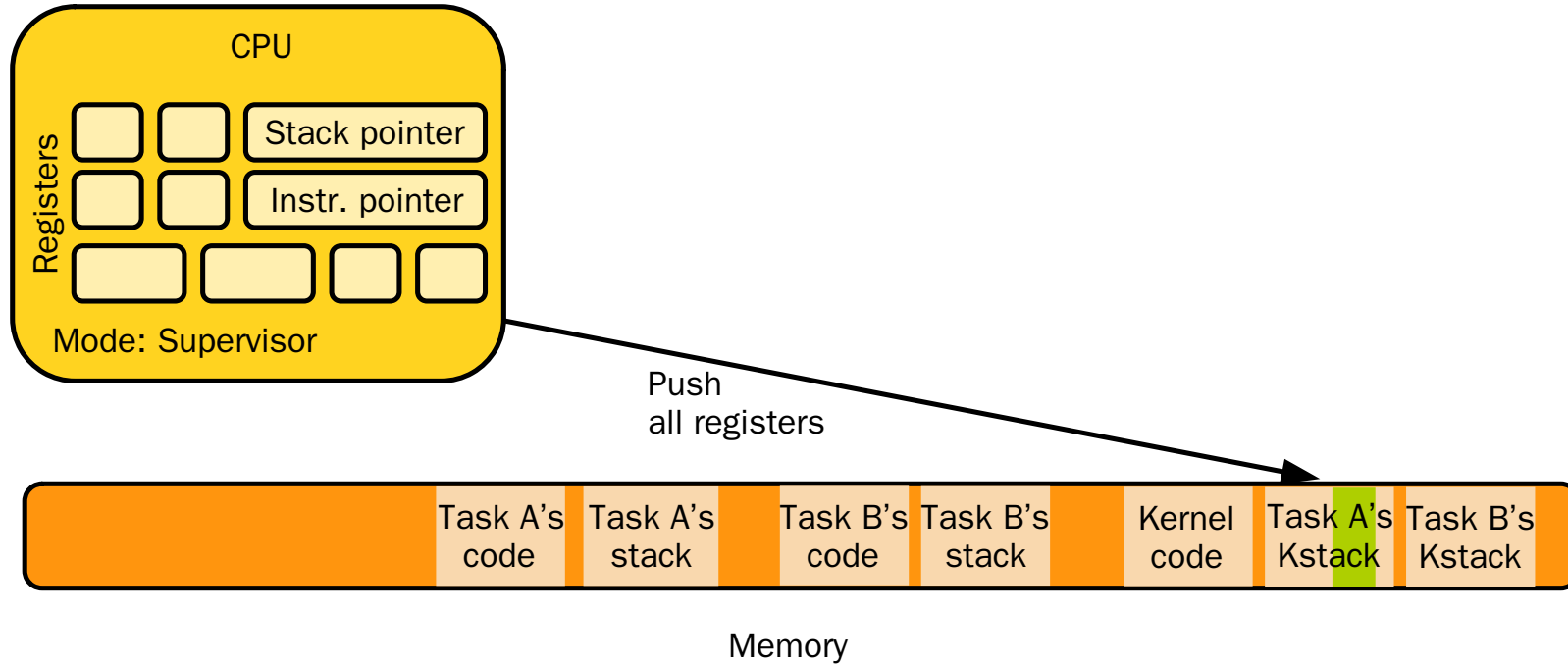
Context Switch in a Nutshell



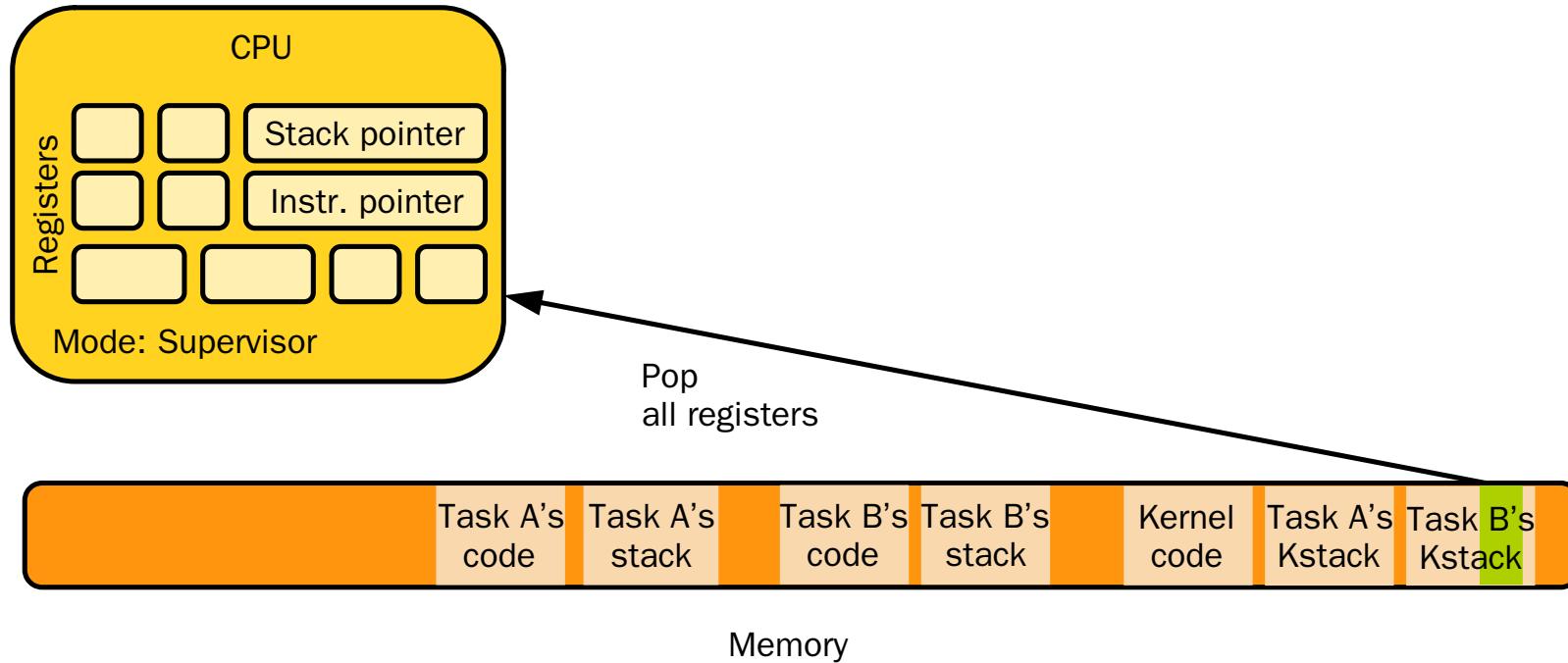
Context Switch in a Nutshell



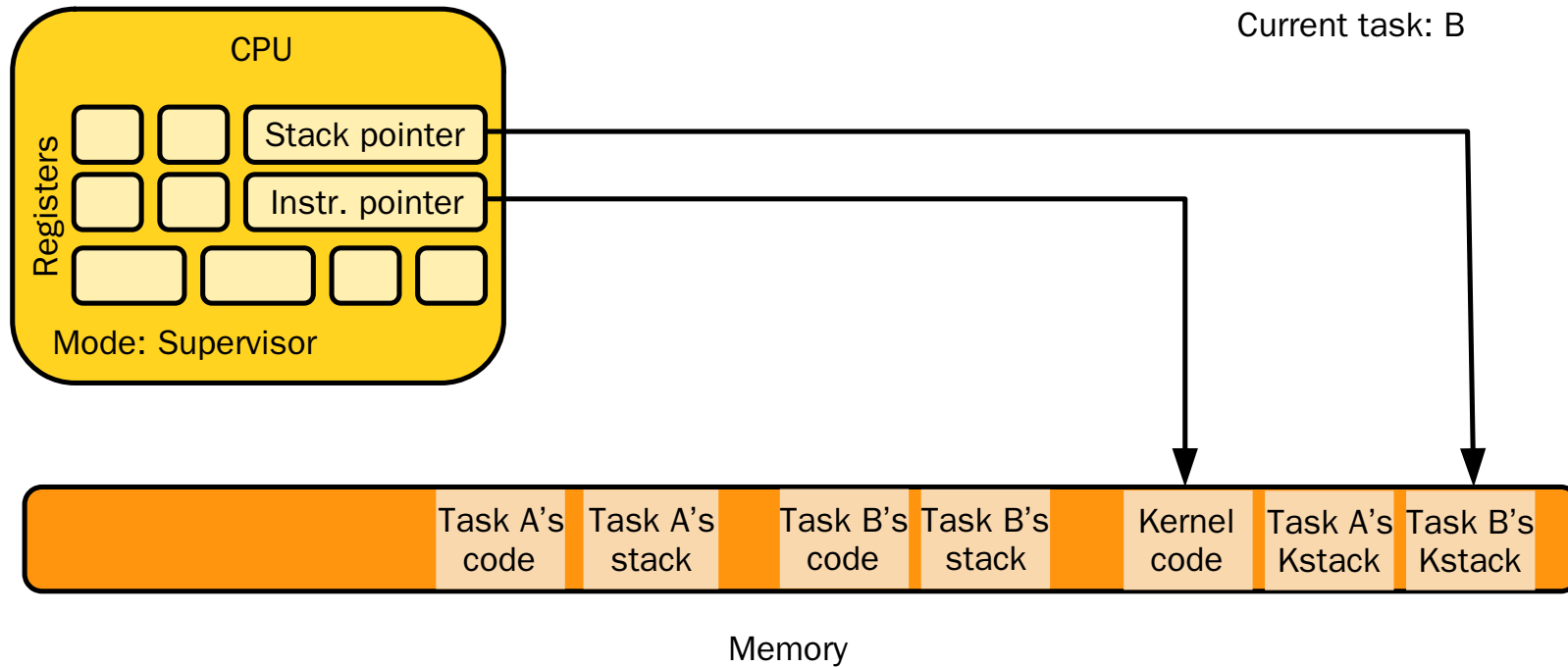
Context Switch in a Nutshell



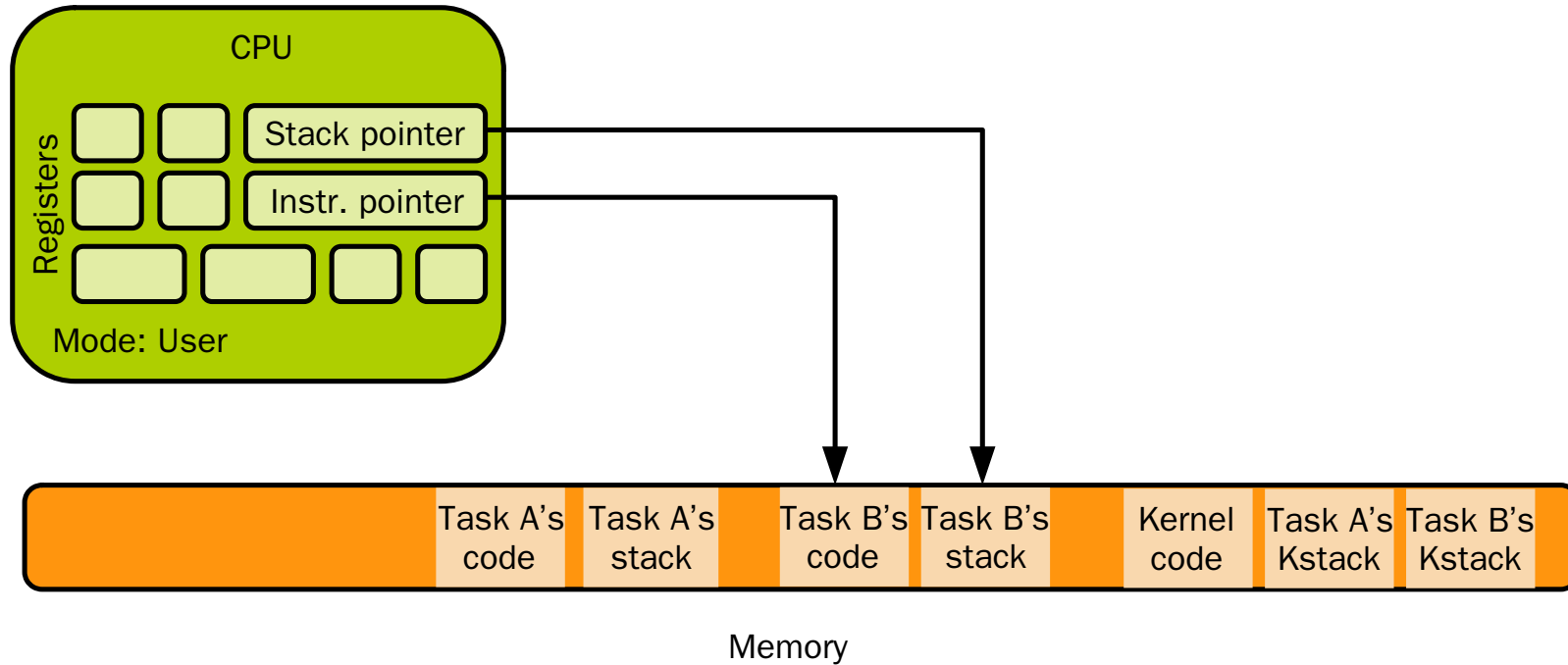
Context Switch in a Nutshell



Context Switch in a Nutshell



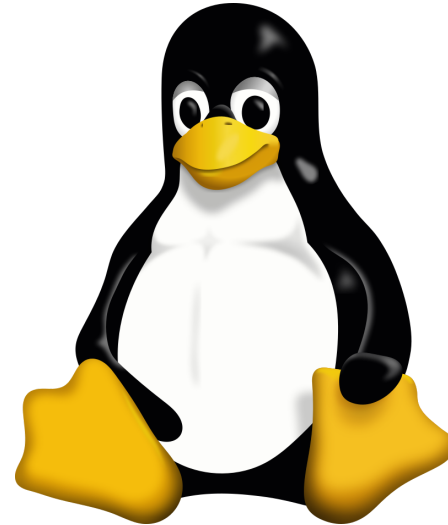
Context Switch in a Nutshell



System Call & Context Switch Implementations in a Real OS

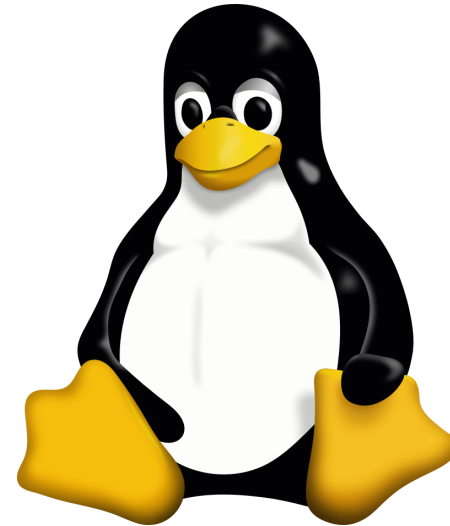
The Linux Kernel

- **Most widespread OS kernel:**
 - Dominates the mobile (Android), server, supercomputers markets
 - Widespread in embedded systems



The Linux Kernel

- **Most widespread OS kernel:**
 - Dominates the mobile (Android), server, supercomputers markets
 - Widespread in embedded systems
- **Open Source**
 - Majority of contributors are from the industry
 - Will browse Linux's code using Elixir:
<https://elixir.bootlin.com/>



Most active 6.6 employers

By changesets			By lines changed		
Linaro	1333	9.5%	Red Hat	56102	9.5%
Intel	1221	8.7%	Linaro	48883	8.3%
Huawei Technologies	962	6.8%	Intel	47457	8.0%
Red Hat	940	6.7%	NVIDIA	38849	6.6%
Google	937	6.7%	Google	37066	6.3%

Linux v6.6 top contributors, source:

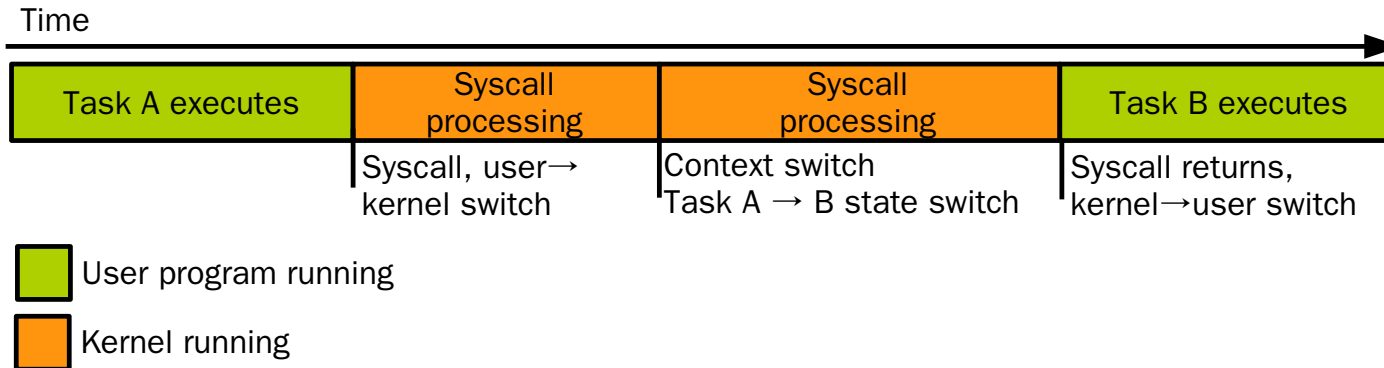
<https://lwn.net/Articles/948970/>

Context Switch & System Call in Linux

- Context switch & system call handling can only be done by the kernel
- Example with `nanosleep` system call (system call + context switch)
 - System call invoked by the standard C library when the program calls `sleep`

Context Switch & System Call in Linux

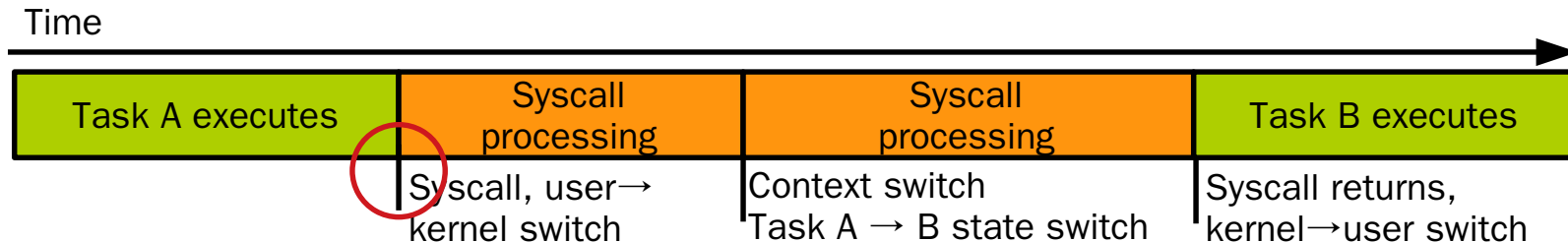
- Context switch & system call handling can only be done by the kernel
- Example with `nanosleep` system call (system call + context switch)
 - System call invoked by the standard C library when the program calls `sleep`



System Call Handling in Linux

- System call entry point is in [entry_SYSCALL_64](#) in `arch/x86/entry/entry_64.S`:

```
SYM_CODE_START(entry_SYSCALL_64)
// ...
movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)      // save user stack pointer
// ...
movq    PER_CPU_VAR(pcpu_hot + X86_top_of_stack), %rsp // load kernel stack pointer
// ...
PUSH_AND_CLEAR_REGS rax=$-ENOSYS                    // save user registers
movq    %rsp, %rdi                                    // prepare do_syscall_64 arguments
movslq   %eax, %rsi
// ...
call    do_syscall_64                                // switch to c Code
```



System Call Handling in Linux

[do_syscall_64](#) gets register state and system call ID as parameters:

```
__visible noinstr bool do_syscall_64(struct pt_regs *regs, int nr);
```

System Call Handling in Linux

[do_syscall_64](#) gets register state and system call ID as parameters:

```
__visible noinstr bool do_syscall_64(struct pt_regs *regs, int nr);
```

Then calls [do_syscall_x64](#) which itself calls [x64_sys_call](#):

```
long x64_sys_call(const struct pt_regs *regs, unsigned int nr)
{
    switch (nr) {
        #include <asm/syscalls_64.h>
        default: return __x64_sys_ni_syscall(regs);
    }
};
```


System Call Handling in Linux

[do_syscall_64](#) gets register state and system call ID as parameters:

```
__visible noinstr bool do_syscall_64(struct pt_regs *regs, int nr);
```

Then calls [do_syscall_x64](#) which itself calls [x64_sys_call](#):

```
long x64_sys_call(const struct pt_regs *regs, unsigned int nr)
{
    switch (nr) {
        #include <asm/syscalls_64.h>
        default: return __x64_sys_ni_syscall(regs);
    }
};
```

Syscall jump table included is generated from the [system call table](#), e.g. for `nanosleep` the entry is:

```
35      common      nanosleep      sys_nanosleep
```

System Call Handling in Linux

[sys_nanosleep](#) calls [hrtimer_nanosleep](#), itself calling [do_nanosleep](#) that contains the main logic for `nanosleep`:

```
static int __sched do_nanosleep(struct hrtimer_sleeper *t, enum hrtimer_mode mode)
{
    struct restart_block *restart;

    do {
        set_current_state(TASK_INTERRUPTIBLE|TASK_FREEZABLE);
        hrtimer_sleeper_start_expires(t, mode);

        if (likely(t->task))
            schedule();

        hrtimer_cancel(&t->timer);
        mode = HRTIMER_MODE_ABS;
    } while (t->task && !signal_pending(current));

    /* ... */
}
```

System Call Handling in Linux

[schedule](#) calls [__schedule_loop](#), which itself calls the main scheduler function: [__schedule](#):

```
static void __sched notrace __schedule(unsigned int sched_mode) {
    /* ... */
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;           // get the task_struct for the current task
    /* ... */
    local_irq_disable();      // disable interrupts
    /* ... */
    next = pick_next_task(rq, prev, &rf); // get the task_struct of the next task to run
    /* ... */
    if (likely(prev != next)) {
        /* ... */
        rq = context_switch(rq, prev, next, &rf); // perform context switch from prev to next
    }
    /* ... */
}
```

System Call Handling in Linux

[schedule](#) calls [__schedule_loop](#), which itself calls the main scheduler function: [__schedule](#):

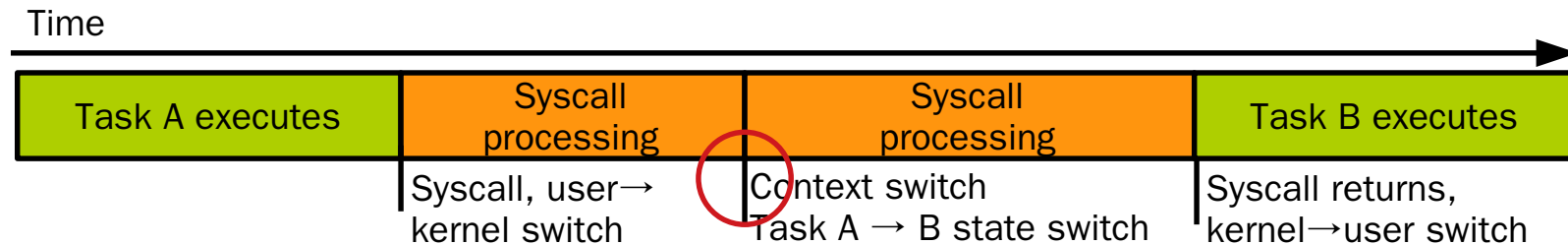
```
static void __sched notrace __schedule(unsigned int sched_mode) {
    /* ... */
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;          // get the task_struct for the current task
    /* ... */
    local_irq_disable();      // disable interrupts
    /* ... */
    next = pick_next_task(rq, prev, &rf); // get the task_struct of the next task to run
    /* ... */
    if (likely(prev != next)) {
        /* ... */
        rq = context_switch(rq, prev, next, &rf); // perform context switch from prev to next
    }
    /* ... */
}
```

[local_irq_disable](#) ends up calling [native_irq_disable](#):

```
static __always_inline void native_irq_disable(void) { asm volatile("cli": : : "memory"); }
```

Context Switch in Linux

- A context switch swaps the CPU state from the execution context of the **previous task** to that of the **next task**
- State includes, among others:
 - General purpose registers
 - Stack pointer (each task has its own kernel stack)
 - Control register holding the root of the page table, (each task has its own address space)



Context Switch in Linux

Done by the [context_switch](#) function:

```
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next, struct rq_flags *rf) {

    if (!next->mm) {
        /* switching to a kernel thread, no need to switch page tables ... */
    } else {
        /* ... */
        switch_mm_irqs_off(prev->active_mm, next->mm, next); // switch page tables
        /* ... */
    }
    /* ... */
    switch_to(prev, next, prev); // switch registers including the stack pointer
    /* ... */
}
```

Context Switch in Linux

Done by the [context_switch](#) function:

```
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next, struct rq_flags *rf) {

    if (!next->mm) {
        /* switching to a kernel thread, no need to switch page tables ... */
    } else {
        /* ... */
        switch_mm_irqs_off(prev->active_mm, next->mm, next); // switch page tables
        /* ... */
    }
    /* ... */
    switch_to(prev, next, prev); // switch registers including the stack pointer
    /* ... */
}
```

[switch_mm_irqs_off](#) ends up calling [native_write_cr3](#):

```
static inline void native_write_cr3(unsigned long val) {
    asm volatile("mov %0,%%cr3" : : "r" (val) : "memory");
}
```

Context Switch in Linux

[switch_to](#) calls [switch_to_asm](#) that performs the register switch:

```
SYM_FUNC_START(__switch_to_asm)
    pushq    %rbp                // save callee-saved registers representing the kernel
    pushq    %rbx                // execution state of the current task (i.e. the task being
    pushq    %r12                // scheduled out) by pushing them to the stack
    pushq    %r13
    pushq    %r14
    pushq    %r15
    movq     %rsp, TASK_threadsp(%rdi) // save current task's stack pointer

    movq     TASK_threadsp(%rsi), %rsp // load the next task's stack pointer
    /* ... */
    popq     %r15                // restore the callee-saved registers representing the kernel
    popq     %r14                // execution state of the task being scheduled in
    popq     %r13
    popq     %r12
    popq     %rbx
    popq     %rbp

    jmp     __switch_to
```


Context Switch in Linux

[switch_to](#) calls [switch_to_asm](#) that performs the register switch:

```
SYM_FUNC_START(__switch_to_asm)
    pushq    %rbp                // save callee-saved registers representing the kernel
    pushq    %rbx                // execution state of the current task (i.e. the task being
    pushq    %r12                // scheduled out) by pushing them to the stack
    pushq    %r13
    pushq    %r14
    pushq    %r15
    movq     %rsp, TASK_threadsp(%rdi) // save current task's stack pointer

    movq     TASK_threadsp(%rsi), %rsp // load the next task's stack pointer
    /* ... */
    popq     %r15                // restore the callee-saved registers representing the kernel
    popq     %r14                // execution state of the task being scheduled in
    popq     %r13
    popq     %r12
    popq     %rbx
    popq     %rbp

    jmp     __switch_to
```

- [__switch_to](#) restores more state (e.g. floating point registers)
- Then the newly scheduled task takes the return path

Context Switch in Linux

- Assume the last time the current task was scheduled out it was following a `nanosleep` system call

Context Switch in Linux

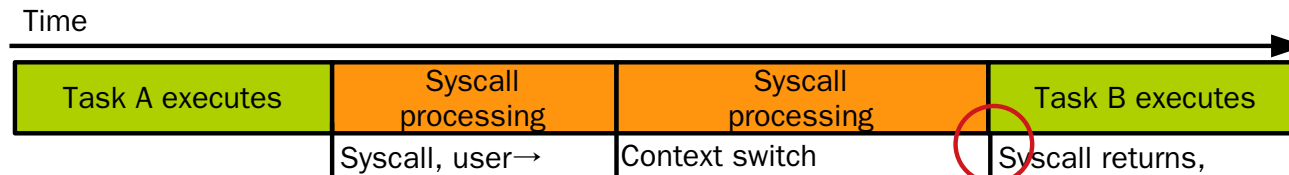
- Assume the last time the current task was scheduled out it was following a `nanosleep` system call
- Execution takes the return path up to the system call handler `entry_SYSCALL_64`, [right after the call to `do_syscall_64`](#):

```
SYM_CODE_START(entry_SYSCALL_64)
    /* ... */
    call    do_syscall_64
    /* ... */
    POP_REGS pop_rdi=0          // Restore user space state for general purpose registers
    movq    %rsp, %rdi
    movq    PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp
    pushq    RSP-RDI(%rdi)      /* RSP */
    /* ... */
    popq    %rsp                // Restore user space stack pointer from scratch space
    sysretq                    // return back to user space
```

Context Switch in Linux

- Assume the last time the current task was scheduled out it was following a `nanosleep` system call
- Execution takes the return path up to the system call handler `entry_SYSCALL_64`, [right after the call to `do_syscall_64`](#):

```
SYM_CODE_START(entry_SYSCALL_64)
/* ... */
call    do_syscall_64
/* ... */
POP_REGS pop_rdi=0      // Restore user space state for general purpose registers
movq    %rsp, %rdi
movq    PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp
pushq   RSP-RDI(%rdi)   /* RSP */
/* ... */
popq    %rsp            // Restore user space stack pointer from scratch space
sysretq                // return back to user space
```



Summary

- We have seen how syscall handling and context switching are implemented
 - It's made with a combination of C and assembly
 - Assembly is necessary for low level operations such as saving CPU state or switching tasks
 - It integrates very well with C
-

Feedback form: <https://bit.ly/2VD4kfx>

