Run/Wait Queues

W4118 Operating Systems I

columbia-os.github.io

Credits to Jae

Logistics

1. HW2 due Monday 2/22

2. For Part 3, the IP and SP values will differ from the reference image. Why?

Process States

```
/* Used in tsk->state: */
#define TASK_RUNNING                0x0000
#define TASK_INTERRUPTIBLE          0x0001
#define TASK_UNINTERRUPTIBLE        0x0002
```

TASK_RUNNING: the task is runnable – either currently running or on a run queue waiting to run

TASK_INTERRUPTIBLE: the task is sleeping waiting for some condition to exist - can be awakened prematurely if it receives a signal

TASK_UNINTERRUPTIBLE: the task is sleeping waiting for some condition to exist - cannot be awakened prematurely if it receives a signal

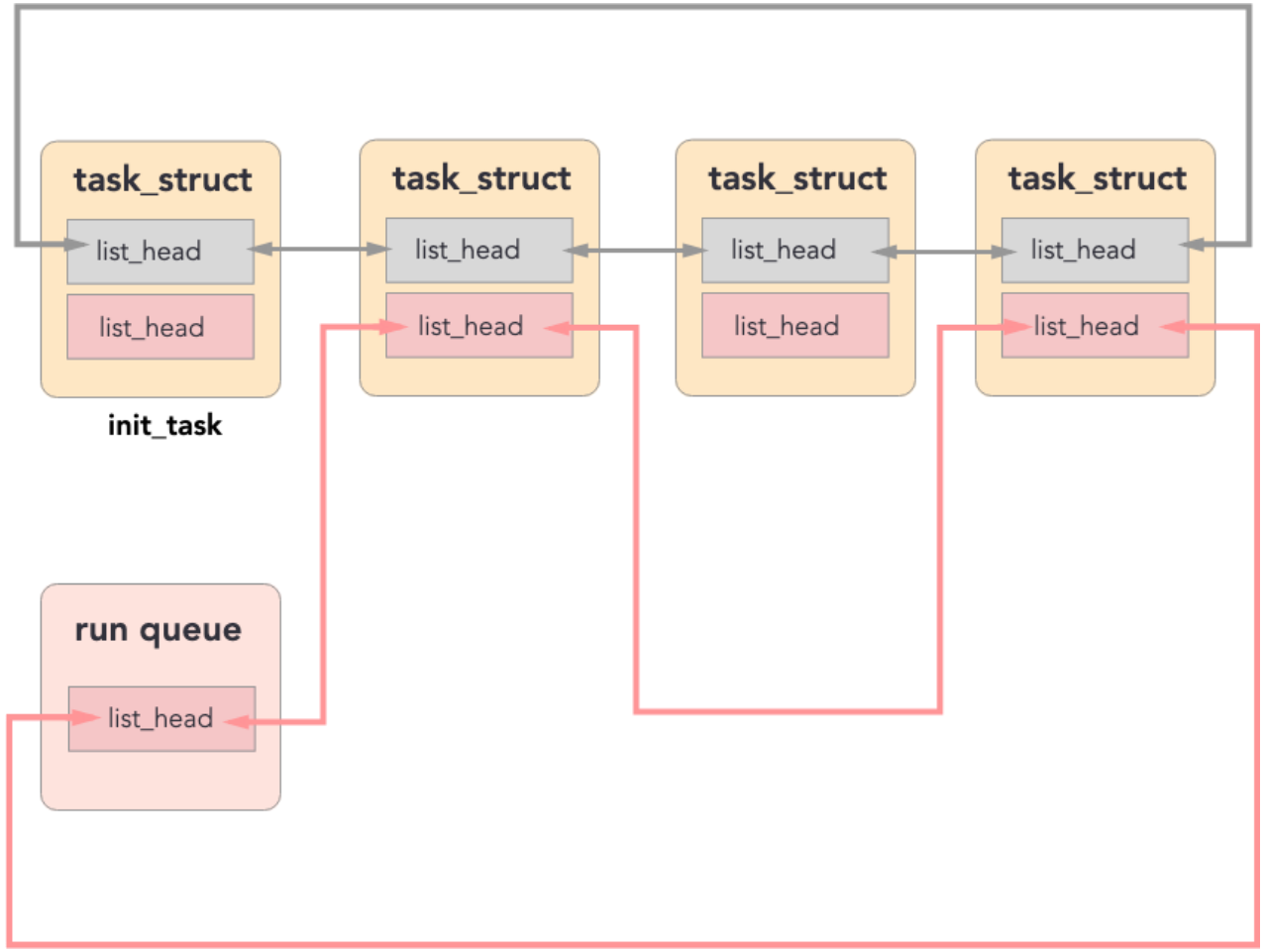
Run Queue

`task_structs` are linked via `children/sibling` list_heads

Per-CPU `run_queue` links tasks with state `TASK_RUNNING`

Why need a separate `list_head` for the run queue?

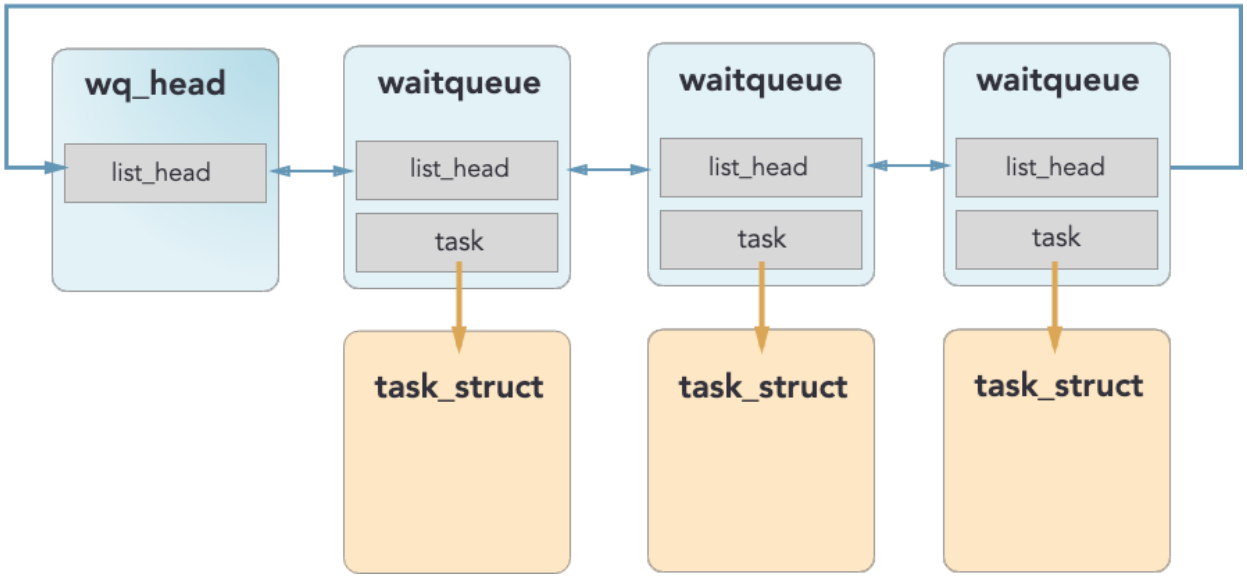
`include/linux/sched.h`



Wait Queue

Per-event `wait_queue`

Wait queue entry is NOT embedded in `task_struct`



Wait Queue Data Structures

***pseudocode

```
struct wait_queue_head {
        spin_lock_t lock;
        struct list_head task_list;
};

struct waitqueue {
        struct task_struct *task;
        wait_queue_func_t func; // callback function, e.g. try_to_wake_up()
        struct list_head entry;
};
```

How to wait

include/linux/wait.h – (kernel 3.12.74 for simplicity)

1. **`prepare_to_wait()`:** add yourself to wait queue, change state to **`TASK_INTERRUPTIBLE`**
2. **`signal_pending()`:** check for “spurious wakeup”, i.e. signal interrupted sleep before condition was met
   - break out of loop instead of sleeping
3. **`schedule()`**: put yourself to sleep
4. **`finish_wait()`**: change state to **`TASK_RUNNING`**, remove yourself from the wait queue

Perform 1-3 in a loop to handle spurious wakeups

Notes:

1. LKD page 59 is outdated and incorrect, use **`wait_event_interruptible()`**
2. **`wait_event_interruptible()`** is a generic macro, probably not appropriate to use directly
   a. Doesn’t account for synchronization
   b. You may want to handle `signal_pending()` differently

Scheduling Basics

`kernel/sched/core.c`

1. `pick_next_task()`: choose a new task to run from the run queue
2. `context_switch()`: put current task to sleep, start running new task

Wait Queue Walkthrough

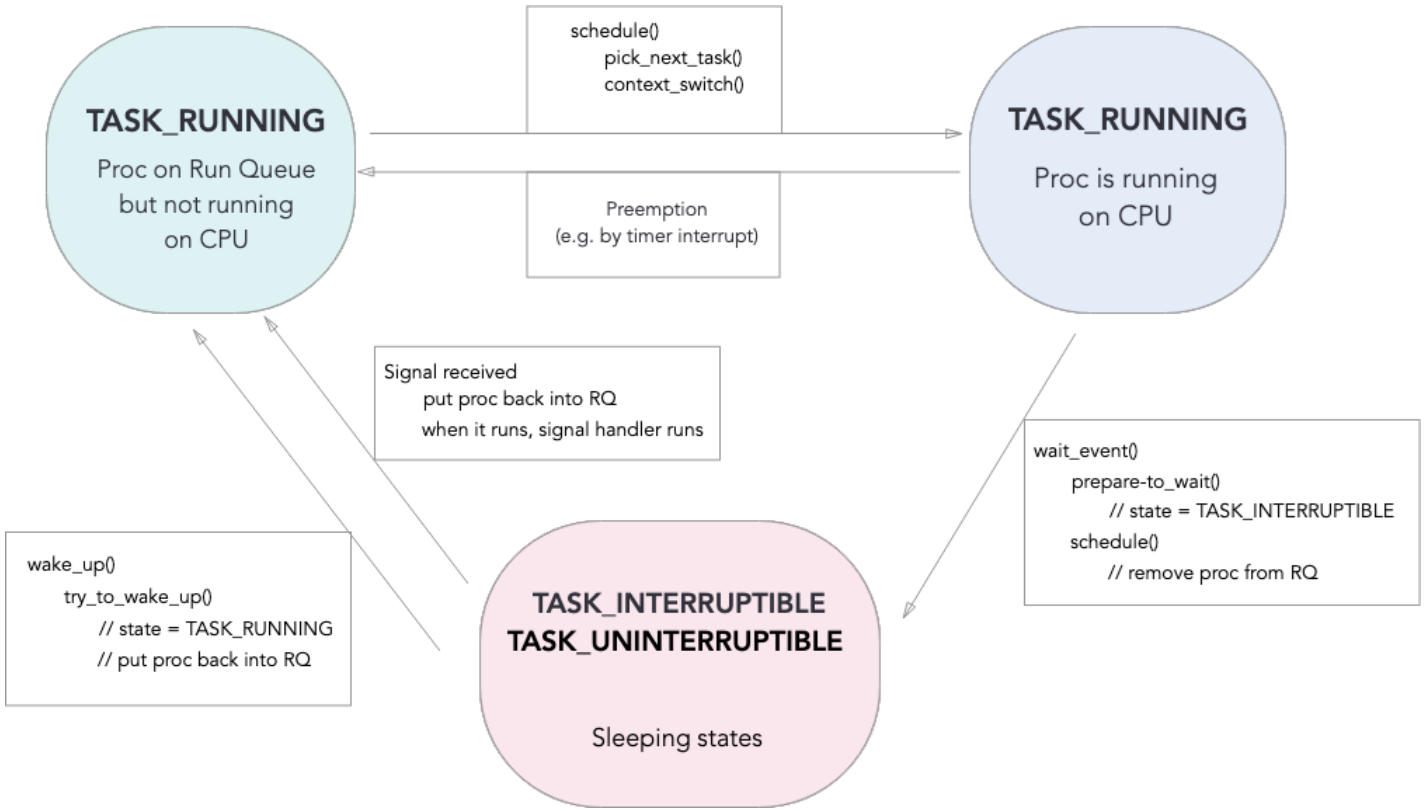
Sleeping:

1. `wait_event()`
2. Enqueued on wait queue
3. Remove from run queue
4. `schedule()`
    - `pick_next_task()`
    - `context_switch`
5. Other task runs

Waking up:

1. Task signals event: `wake_up()`
2. Call `try_to_wake_up()` on each task
3. Enqueue each task on run queue
4. Eventually other tasks calls `schedule()` and previously sleeping task gets chosen*
5. Previously sleeping task checks condition
    - If true, `finish_wake()`
    - Else, repeat 2-5 from “sleeping”

Process State Transition



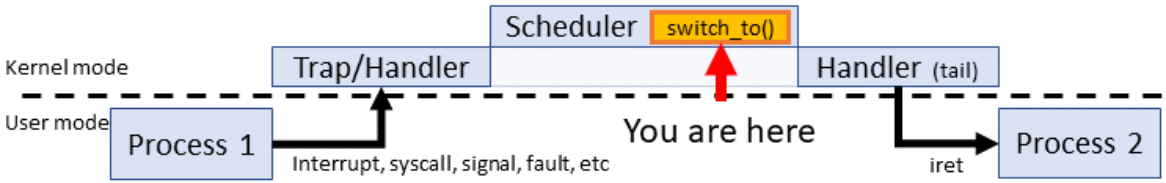
Example: `read()`

1. Trap into kernel
   - save registers into per-proc kernel stack
2. Device driver issues an I/O request to the device
3. Put the calling process to sleep
   - `wait_event()` → `schedule()` → `pick_next_task()` → `context_switch()`
4. Another process starts running
5. The device completes the I/O request and raised a hardware interrupt
6. Trap into kernel and jump to the interrupt handler:
   - `wake_up():` enqueue blocked tasks back on run queue
   - Current task eventually calls `schedule()` → `pick_next_task()` → `context_switch()`
7. Another process starts running
   - This process may or may not be the one that called `read()`

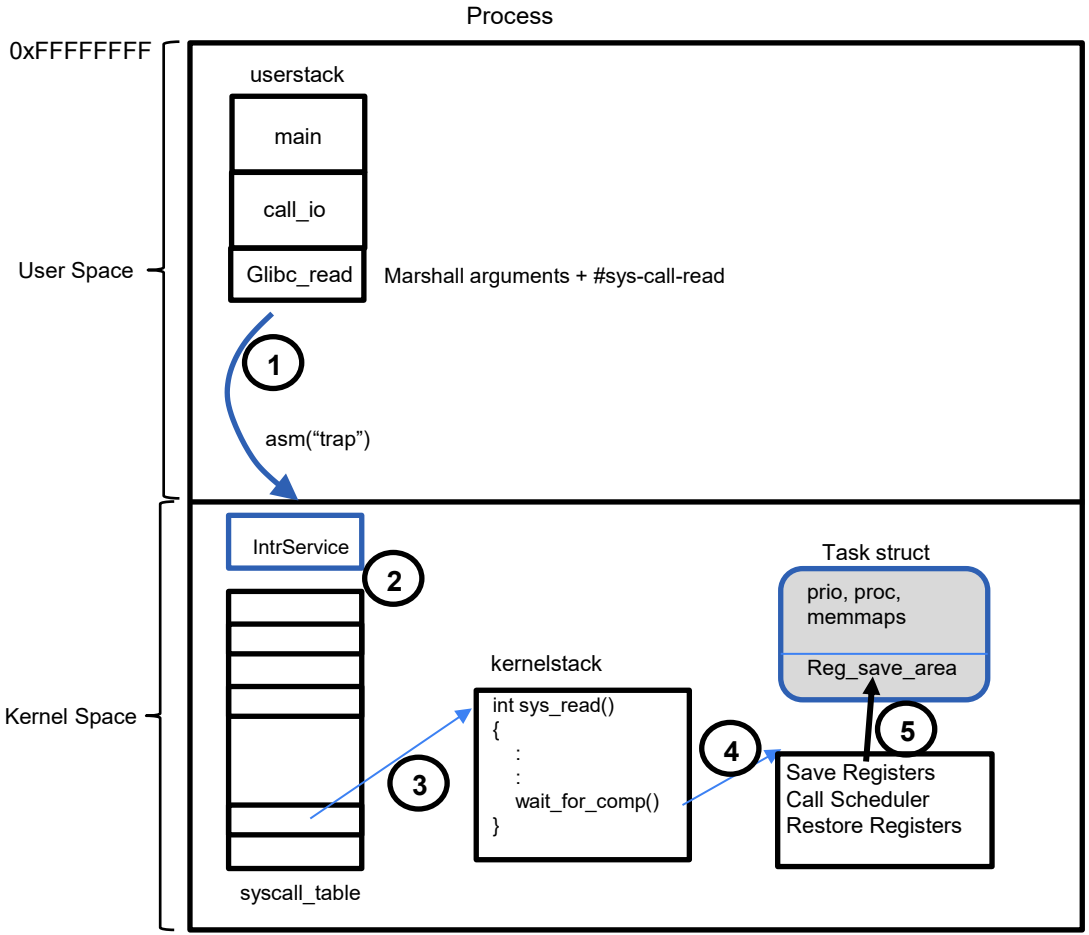
Context Switch (a slightly deeper dive)

- Scenarios:
  1) Current process (or thread) blocks OR
  2) Preemption via interrupt

- Operation(s) to be done
  - Must release CPU resources (registers)
  - Requires storing “all” non privileged registers to the PCB or TCB save area
  - Tricky as you need registers to do this
  - Typically an architecture has a few privileged registers so the kernel can accomplish this
  - All written in assembler



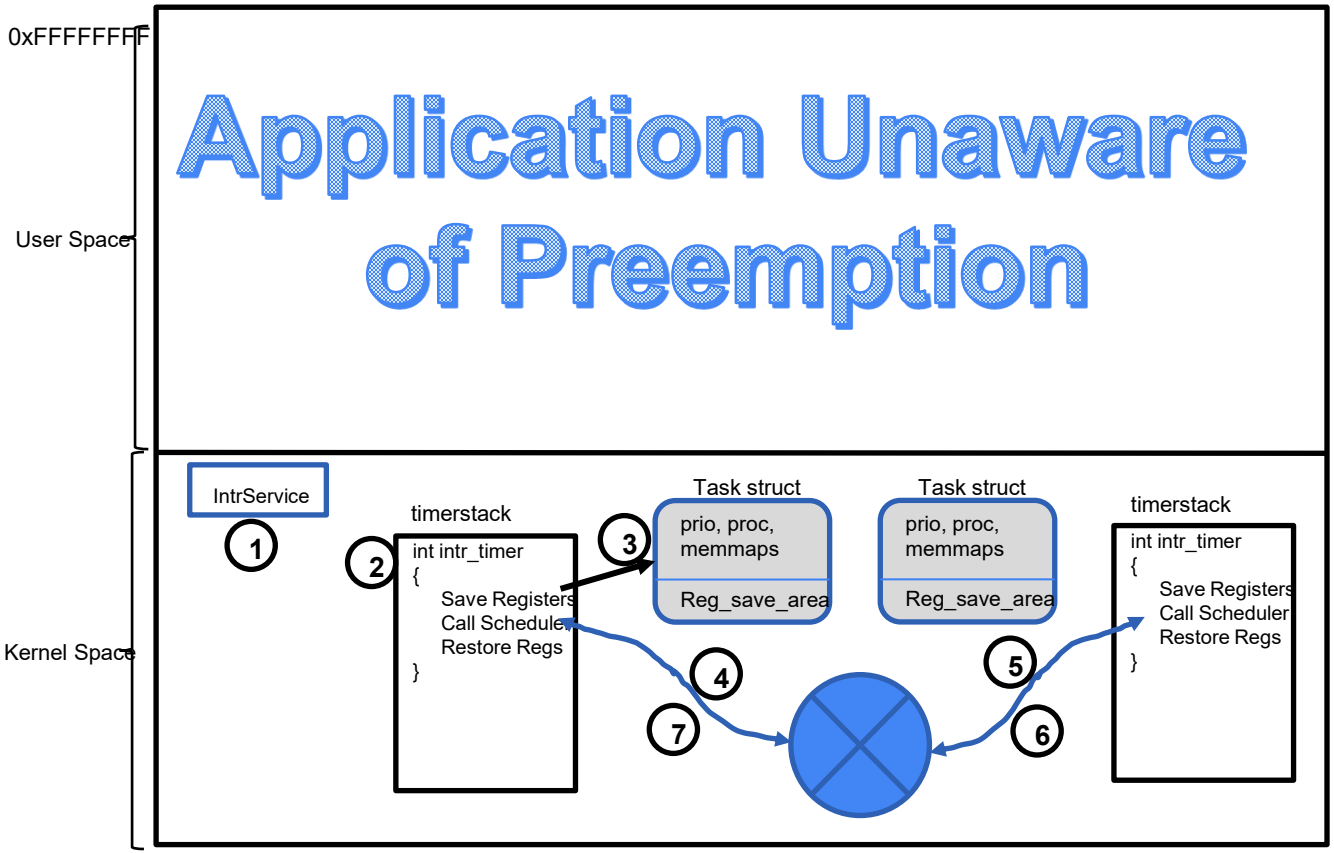
CtxSwitch: Process Blocks




```
int call_io()
{
   char buffer[128];
   return read(0,buffer,128);
}

main() {
   call_io();
}
```

CtxSwitch: Preemption



Examples of low-level context switch routine for x86 (ugly, ugly, ugly)

Details not important, appreciate the complexity of the ASM code

```
/** include/asm-x86_64/system.h */

#define SAVE_CONTEXT    "pushfq ; pushq %%rbp ; movq %%rsi,%%rbp\n\t"
#define RESTORE_CONTEXT "movq %%rbp,%%rsi ; popq %%rbp ; popfq\n\t"
#define __EXTRA_CLOBBER
    ,"rcx","rbx","rdx","r8","r9","r10","r11","r12","r13","r14","r15"

#define switch_to(prev,next,last)
    asm volatile(SAVE_CONTEXT
         "movq %%rsp,%P[threadrsp](%[prev])\n\t" /* save RSP */
         "movq %P[threadrsp](%[next]),%%rsp\n\t" /* restore RSP */
         "call __switch_to\n\t"
         ".globl thread_return\n"
         "thread_return:\n\t"
         "movq %%gs:%P[pda_pcurrent],%%rsi\n\t"
         "movq %P[thread_info](%%rsi),%%r8\n\t"
         "btr  %[tif_fork],%P[ti_flags](%%r8)\n\t"
         "movq %%rax,%%rdi\n\t"
         "jc   ret_from_fork\n\t"
         RESTORE_CONTEXT
         : "=a" (last)
         : [next] "S" (next), [prev] "D" (prev),
           [threadrsp] "i" (offsetof(struct task_struct, thread.rsp)),
           [ti_flags] "i" (offsetof(struct thread_info, flags)),
           [tif_fork] "i" (TIF_FORK),
           [thread_info] "i" (offsetof(struct task_struct, thread_info)),
           [pda_pcurrent] "i" (offsetof(struct x8664_pda, pcurrent))
         : "memory", "cc" __EXTRA_CLOBBER)
```

```
/** arch/x86_64/kernel/process.c */

struct task_struct *__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
    struct thread_struct *prev = &prev_p->thread,
                *next = &next_p->thread;
    int cpu = smp_processor_id();
    struct tss_struct *tss = init_tss + cpu;

    unlazy_fpu(prev_p);

    tss->rsp0 = next->rsp0;

    asm volatile("movl %%es,%0" : "=m" (prev->es));
    if (unlikely(next->es | prev->es))
        loadsegment(es, next->es);

    asm volatile ("movl %%ds,%0" : "=m" (prev->ds));
    if (unlikely(next->ds | prev->ds))
        loadsegment(ds, next->ds);

    load_TLS(next, cpu);

    /* Switch FS and GS. */
    {
        unsigned fsindex;
        asm volatile("movl %%fs,%0" : "=g" (fsindex));

        if (unlikely(fsindex | next->fsindex | prev->fs)) {
            loadsegment(fs, next->fsindex);
            if (fsindex)
                prev->fs = 0;
        }
        /* when next process has a 64bit base use it */
        if (next->fs)
            wrmsrl(MSR_FS_BASE, next->fs);
        prev->fsindex = fsindex;
    }
    {
        unsigned gsindex;
        asm volatile("movl %%gs,%0" : "=g" (gsindex));
        if (unlikely(gsindex | next->gsindex | prev->gs)) {
            load_gs_index(next->gsindex);
            if (gsindex)
                prev->gs = 0;
        }
        if (next->gs)
            wrmsrl(MSR_KERNEL_GS_BASE, next->gs);
        prev->gsindex = gsindex;
    }

    /* Switch the PDA context. */
    prev->userrsp = read_pda(oldrsp);
    write_pda(oldrsp, next->userrsp);
    write_pda(pcurrent, next_p);
    write_pda(kernelstack, (unsigned long)next_p->thread_info + THREAD_SIZE - PDA_STACKOFFSET);

    /* Now maybe reload the debug registers */
    if (unlikely(next->debugreg7)) {
        loaddebug(next, 0);
        loaddebug(next, 1);
        loaddebug(next, 2);
        loaddebug(next, 3);
        /* no 4 and 5 */
        loaddebug(next, 6);
        loaddebug(next, 7);
    }

    /* Handle the IO bitmap */
    if (unlikely(prev->io_bitmap_ptr || next->io_bitmap_ptr)) {
        if (next->io_bitmap_ptr) {
            memcpy(tss->io_bitmap, next->io_bitmap_ptr, IO_BITMAP_BYTES);
            tss->io_bitmap_base = IO_BITMAP_OFFSET;
        } else {
            tss->io_bitmap_base = INVALID_IO_BITMAP_OFFSET;
        }
    }

    return prev_p;
}
```

Task switching in ARM

```
extern struct task_struct *__switch_to(struct task_struct *, struct thread_info *, struct thread_info *);

#define switch_to(prev,next,last)                                       \
do {                                                                    \
        __complete_pending_tlbi();                                      \
        if (IS_ENABLED(CONFIG_CURRENT_POINTER_IN_TPIDRURO))             \
                __this_cpu_write(__entry_task, next);                   \
        last = __switch_to(prev,task_thread_info(prev), task_thread_info(next));        \
} while (0)

#endif /* __ASM_ARM_SWITCH_TO_H */
```

```
ENTRY(__switch_to)
        .fnstart
        .cantunwind
        add     ip, r1, #TI_CPU_SAVE
        stmia   ip!, {r4 - r11}         @ Store most regs on stack
        str     sp, [ip], #4
        str     lr, [ip], #4
        mov     r5, r0
        add     r4, r2, #TI_CPU_SAVE
        ldr     r0, =thread_notify_head
        mov     r1, #THREAD_NOTIFY_SWITCH
        bl      atomic_notifier_call_chain
        mov     ip, r4
        mov     r0, r5
        ldmia   ip!, {r4 - r11}         @ Load all regs saved previously
        ldr     sp, [ip]
        ldr     pc, [ip, #4]!
        .fnend
ENDPROC(__switch_to)
```