Task Context Switching

Gustavo de Castro Biage
Gilson Trombetta Magro
Matheus de Oliveira Saldanha

What is a Context?

- ➔ In computer science, a context usually means the computational state of a process or thread.

- ➔ The possibility to switch between processes or threads and restore its context allows these structures to share a single CPU, giving the impression of simultaneous execution.

- ➔ This pseudo-parallelism is achieved by preemptive systems and distribution of time slices between processes.

Differences between Process and Thread Context

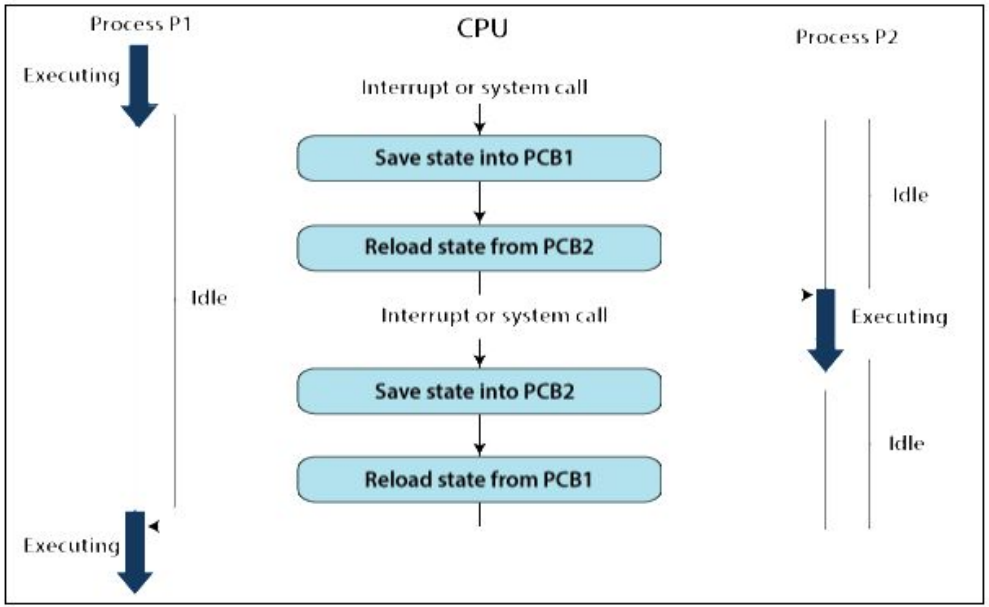
- The context of a thread usually contains:
  - a set of general purpose registers
  - a program counter (PC)
  - a stack pointer (SP)

- A process can be defined as a running program. It is loaded into memory and contains information about instructions and data.

- During a thread context switch the virtual memory space remains the same, since all threads of the same process share the same virtual address space.
  - This is not true during a process context switch.

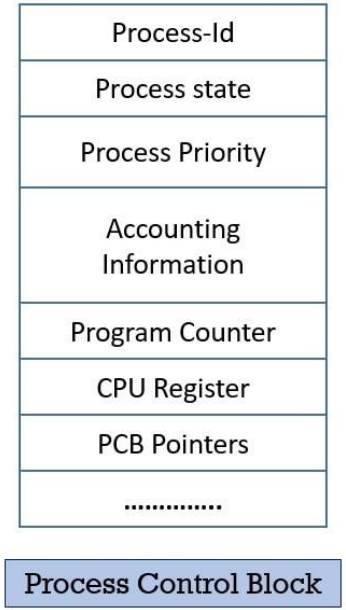
What is Context Switching?

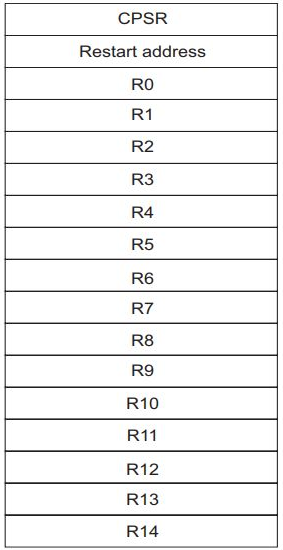
Context Switching is the saving and restoring of computational state when switching between different threads or processes, known as tasks.



What is a PCB?

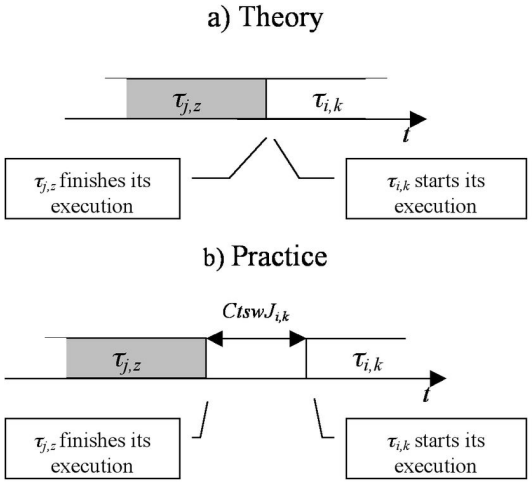
➔ A process control block (PCB) is a data structure used by computer operating systems to store all information about a process. It is also known as a process descriptor.





Some performance considerations

- ➔ It takes time to save the context of a process and restore the context of another process.
- ➔ During this time, there is no useful work done by the CPU from the user's perspective.
- ➔ Therefore, context switching is sheer overhead in this condition.



Translation Lookaside Buffer

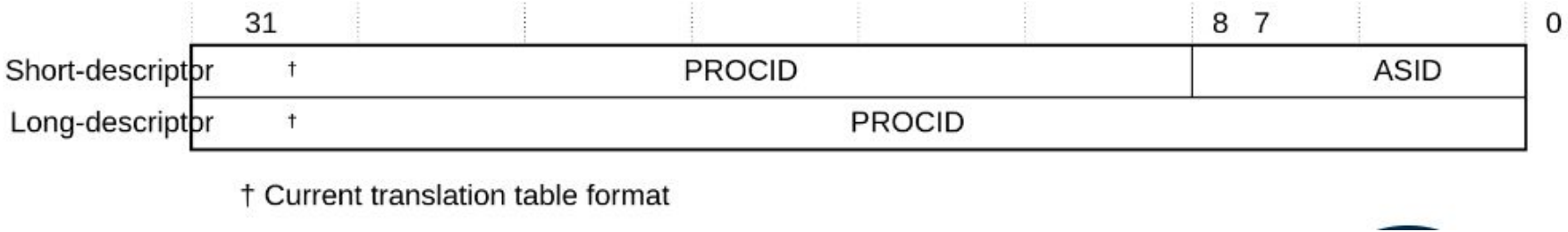
- ➔ The TLB is part of the MMU and can be understood as a cache for the MMU, translating virtual to physical memory addresses.
- ➔ TLBs must be flushed after a context switch as it contains invalid cached address translations.
- ➔ Recent Intel and AMD processors sport a tagged TLB, which allows you to tag a given translation with a certain address space configuration.
- ➔ Address Space Identifiers (ASIDs) can come in handy.

Address Space Identifier (ASID)

- ➔ Identifies each address space individually.
- ➔ It is used to determine which address space a TLB entry belongs.
- ➔ It has two different representations, on when using short-description translation table and another when using long-description translation table.
- ➔ Long and short descriptions are mentioned in ARMv7-A and ARMv7-M architecture manual, however, registers descriptions shows that ARMv7 uses short-description.

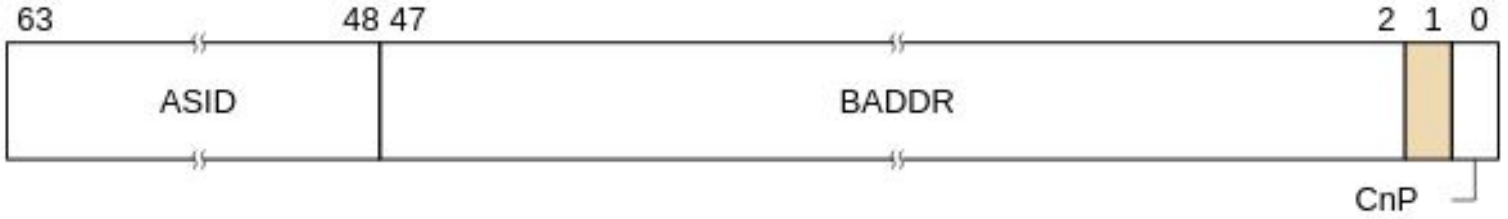
Address Space Identifier (ASID)

➔ Short-description
- ◆ ASID value is stored in the CONTEXTIDR register.
- ◆ ASID value is 8 bits long.



Address Space Identifier (ASID)

➔ Long-description
- ASID value is stored in the TTRB0 register.
- ASID value is 16 bits long.
- This image is not from ARMv7.



Address Space Identifier (ASID)

- ➔ Example 1 - ASID substitution.
- ➔ ISB - whenever instruction fetches need to explicitly take place after a certain point in the program, for example after memory map updates or after writing code to be executed. (In practice, this means "throw away any prefetched instructions at this point".)

```
Change Translation Table Base Register to the global-only mappings

ISB

Change ASID to new value

ISB

Change Translation Table Base Register to new value
```

Address Space Identifier (ASID)

➔ Example 2 - ASID substitution

```
Change ASID to 0
```

```
ISB
```

```
Change Translation Table Base Register
```

```
ISB
```

```
Change ASID to new value
```

Address Space Identifier (ASID)

- ➔ If ASID has only 8 bits, we can have only 256 different address spaces.
- ➔ If all tasks must have different address spaces, we are also limited up to 256 tasks.
- ➔ Linux (running on ARM) uses a rollover mechanism for ASID, where once the ASID options run out, ASID values are invalidated from the branch predictor, caches and TLBs, and should be allocated again for each process, offering a chance for processes without an ASID to get one.
- ➔ Linux also uses bitmap to manage used ASIDs.

Processor modes

- ➔ The ARM processor has many execution modes, this is important for task context switching because some of the indispensable register reads and writes requires it to be running on a privileged mode. Also, privileged modes offer banked registers that allow easier stack manipulation.

- ➔ A process running on user mode will have to enter a privileged mode by an interrupt before switching context. IRQ timer interrupt will bring the processor to IRQ mode, this is an example of an interrupt that can be used to achieve a privileged reschedule.

- ➔ Also, system mode has no banked register, this mode allows to update stack pointer registers, among others, for the next user process while in a privileged mode.

Interrupt Request (IRQ)

➔ IRQ or *interrupt request* is a hardware signal sent to the processor that temporarily stops a running program and allows a special program, an interrupt handler, to run instead.

➔ In the general case to enter a exception handler, we first must:

a. Save the address of the next instruction in the appropriate Link Register LR.
b. Copy CPSR to the SPSR of new mode.
c. Change the mode by modifying bits in CPSR.
d. Fetch next instruction from the vector table.

➔ And to exit it:

a. Move the Link Register LR (minus an offset) to the PC.
b. Copy SPSR back to CPSR, this will automatically changes the mode back to the previous one.
c. Clear the interrupt disable flags (if they were set).

Managing Address Spaces

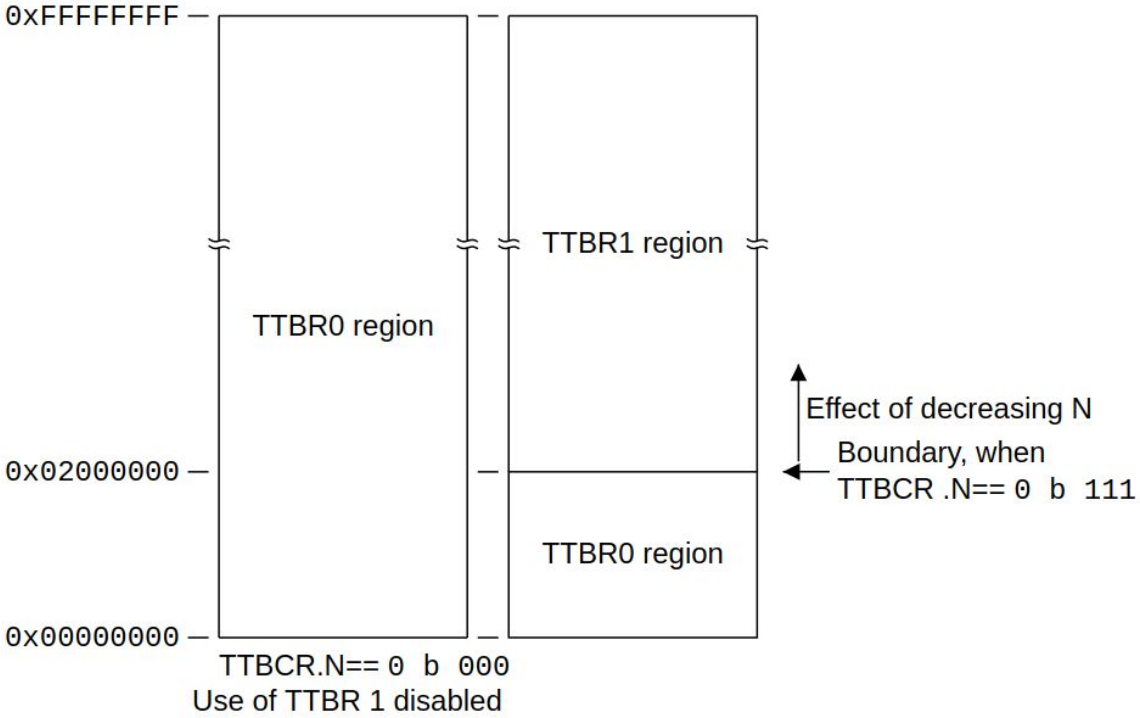
- ➔ The Translation Table Base Register 0 (TTBR0) holds information regarding the **process page table base address** and the memory it occupies.
- ➔ The Translation Table Base Register 1 (TTBR1) holds information regarding the **system page table base address** and the memory it occupies.
- ➔ The Translation Table Base Control Register (TTBCR) determines which of the Translation Table Base Registers, TTBR0 or TTBR1, should be used to translate a virtual address when it is not found on the TLB cache.
- ➔ Switching the current address space in the context switch resumes to updating the TTBR0 register used on the translating memory address.
- ➔ When TTBR0 is update, on systems when ASID value is not used, it is required to invalidate TLB cache to make sure translations will use the new translation table.

Managing Address Spaces

- If TTBCR.N == 0, then use TTBR0.
- if TTBCR.N > 0:
  - if bits[31:32-N] of the input VA are all zero then use TTBR0
  - otherwise use TTBR1.

- For an example, in a situation where TTBCR.N is equal to 0x0, that means that every virtual address will use TTBR0.
- For another example, TTBCR.N is equal to 0x4, so addresses in the format 0x0XXXXXXX will use TTBR0 as the translation table.

Managing Address Spaces



Context Switching Code

- ➔ Here we will demonstrate a simple bare metal ARMv7 context switching implementation.
- ➔ The code will create 3 tasks (main and another 2).
- ➔ A timer is set to raise an IRQ interrupt every 1 second.
- ➔ These tasks will switch context between each other in a Round-Robin fashion.
- ➔ A context switch will take place at every IRQ interrupt.
- ➔ Each task will print its own identifier (0, 1 or 2) and the base address for its page table.

Context Switching Code

```
10  typedef struct stack {
11      volatile unsigned int *stack;
12      unsigned int stack_base[MAX_STACK];
13  } task_t;
14
15  typedef struct {
16      unsigned int length;
17      volatile unsigned int current_id;
18      task_t task[MAX_TASKS];
19  } scheduler_t;
20
21  scheduler_t scheduler;
```

Context Switching Code

```
void create_task(void *task_entry) {
    unsigned int id = scheduler.length;
    scheduler.task[id].stack = scheduler.task[id].stack_base + (MAX_STACK - 1);
    *scheduler.task[id].stack-- = (unsigned int) task_entry;          // r15: pc
    *scheduler.task[id].stack-- = 0x60000110;                         // cpsr
    *scheduler.task[id].stack-- = build_page_table();                 // ttbr0
    *scheduler.task[id].stack-- = 0;                                  // r14: lr
    *scheduler.task[id].stack-- = 0;                                  // r12
    *scheduler.task[id].stack-- = 0;                                  // r11
    *scheduler.task[id].stack-- = 0;                                  // r10
    *scheduler.task[id].stack-- = 0;                                  // r9
    *scheduler.task[id].stack-- = 0;                                  // r8
    *scheduler.task[id].stack-- = 0;                                  // r7
    *scheduler.task[id].stack-- = 0;                                  // r6
    *scheduler.task[id].stack-- = 0;                                  // r5
    *scheduler.task[id].stack-- = 0;                                  // r4
    *scheduler.task[id].stack-- = 0;                                  // r3
    *scheduler.task[id].stack-- = 0;                                  // r2
    *scheduler.task[id].stack-- = 0;                                  // r1
    *scheduler.task[id].stack = 0;                                    // r0
    scheduler.length++;
}
```

Context Switching Code

```c
int main() {
    create_task(task1);
    create_task(task2);
    while (1) {
        hexstring(0);
        io_halt();
    }
    return 0;
}
```

```c
int task1() {
    while (1) {
        hexstring(1);
        io_halt();
    }
    return 0;
}
```

```c
int task2() {
    while (1) {
        hexstring(2);
        io_halt();
    }
    return 0;
}
```

```
.globl io_halt
io_halt:
    wfi
    bx lr
```

Context Switching Code

```
// set vector table handlers
ldr r0, =_vectors
mcr P15, 0, r0, c12, c0, 0

// initialize MMU
bl mmu_init

// initialize IRQ stack
msr cpsr_c, #MODE_IRQ | IRQ_BIT | FIQ_BIT
bl init_irq_stack

// initialize SVC stack
msr cpsr_c, #MODE_SVC | IRQ_BIT | FIQ_BIT
bl init_svc_stack

bl uart_init
bl init_timer

// initialize User stack
msr cpsr_c, #MODE_USR
bl init_task

ldr r3, =main
blx r3
bl hang
```

```
void init_irq_stack() {
    irq_stack.stack = irq_stack.stack_base + (MAX_STACK - 1);
    __asm__("mov sp, %0" : : "r"(irq_stack.stack): );
}

void init_svc_stack() {
    svc_stack.stack = svc_stack.stack_base + (MAX_STACK - 1);
    __asm__("mov sp, %0" : : "r"(svc_stack.stack): );
}
```

```
void init_task() {
    const int main_id = 0;
    scheduler.length = 1;
    scheduler.current_id = main_id;
    scheduler.task[main_id].stack = scheduler.task[main_id].stack_base + (MAX_STACK - 1);
    __asm__("mov sp, %0" : : "r"(scheduler.task[main_id].stack): );
}
```

Context Switching Code

```
void mmu_init() {
    invalidate_caches();
    clear_branch_prediction_array();
    invalidate_tlb();
    enable_dside_prefetch();
    set_domain_access();
    dsb();
    isb();

    page_tables = (unsigned int*) PAGE_TABLES;
    page_tables_setup(page_tables);
    page_tables -= 8 << 12;

    enable_mmu();
    dsb();
    isb();
    branch_prediction_enable();
    dsb();
    clear_bss();
}
```

```
inline void page_tables_setup(unsigned int* page_tables) {
    unsigned int aux = 0x0;
    for (int curr_page = 1006; curr_page >= 0; curr_page--) {
        aux = TTB_MEMORY_DESCRIPTOR | (curr_page << 20);
        ((unsigned int*) page_tables)[curr_page] = aux;
    }
    aux = TTB_DEVICE_DESCRIPTOR | (1007 << 20);
    ((unsigned int*) page_tables)[1007] = aux;
    for (int curr_page = 4095; curr_page > 1007; curr_page--) {
        aux = TTB_PERIPHERAL_DESCRIPTOR | (curr_page << 20);
        ((unsigned int*) page_tables)[curr_page] = aux;
    }
}

unsigned int build_page_table() {
    unsigned int base = (unsigned int) page_tables;
    page_tables_setup(page_tables);
    page_tables -= 8 << 12;
    return base;
}
```

```
enum {
    PAGE_TABLES                 = 0x3eef0000,
```

Context Switching Code

```
.global _vectors
_vectors:
    b reset_handler                 // Reset
    b undefined_handler             // Undefined instruction
    b svc_handler                   // SVC Handler
    b prefetch_abort_handler        // Prefetch abort
    b data_abort_handler            // Data abort
    nop                             // Reserved vector
    b irq_handler                   // IRQ Handler
    b fiq_handler                   // FIQ Handler
```

Context Switching Code

```
irq_handler:
    push {r0-r12}                   // save r0-r12
    push {lr}                       // save lr
    bl exc_handler                  // branch to exc_handler
    pop {lr}                        // restore lr
    pop {r0-r12}                    // restore r0-r12
    subs pc, lr, #4                 // ajust pc and return
```

```
void exc_handler(void)
{
    if (read_core0timer_pending() & 0x08) {
        write_cntv_tval(cntfrq);                // clear CNTV interrupt
        __asm__("b _before_context_switch");    // go to context_switch
    }
}
```

Context Switching Code

```
.global _before_context_switch
_before_context_switch:
    pop {r0}                    // pop lr onto r0
    sub r0, r0, #4              // adjust lr to pc
    mrs r1, spsr                // save cpsr_usr (spsr_irq) to r1
    msr cpsr_c, #0x1F           // move to System Mode
    mrc p15, 0, r2, c2, c0, 0   // save ttbr0 to r2
    push {r0}                   // save lr
    push {r1}                   // save spsr_irq
    push {r2}                   // save ttbr0
    msr cpsr_c, #0x12           // move to IRQ Mode
    pop {r0-r12}                // pop r0-r12
    msr cpsr_c, #0x1F           // move to System Mode
    push {r0-r12,lr}            // save r0-r12
    b schedule                  // go to "schedule"
```

Context Switching Code

```
void schedule() {
    int current_id = scheduler.current_id;
    int next_id = current_id + 1;
    if (next_id >= scheduler.length) {
        next_id = 0;
    }
    // get current sp
    scheduler.task[current_id].stack = _get_stack_pointer();
    // print ttbr0 of next task
    int ttbr0 = *(scheduler.task[next_id].stack + 14);
    hexstring(ttbr0);
    // do context switch
    scheduler.current_id = next_id;
    _after_context_switch(&scheduler.task[current_id].stack,
                          &scheduler.task[next_id].stack);
}
```

Context Switching Code

```
.global _after_context_switch
_after_context_switch:
    str sp, [r0]                    // store sp into PCB of old process
    ldr sp, [r1]                    // load  sp from PCB of new process
    ldr r2, [sp, #56]               // load ttbr0 that was saved previously
    ldr r1, [sp, #60]               // load spsr_irq that was saved previously
    mcr p15, 0, r2, c2, c0, 0       // write ttbr0
    msr cpsr, r1                    // move to User Mode
    pop {r0-r12,lr}                 // pop r0-r12 and lr
    add sp, sp, #8                  // adjust stack pointer
    pop {pc}                        // pop pc
```

Context Switching Code Output