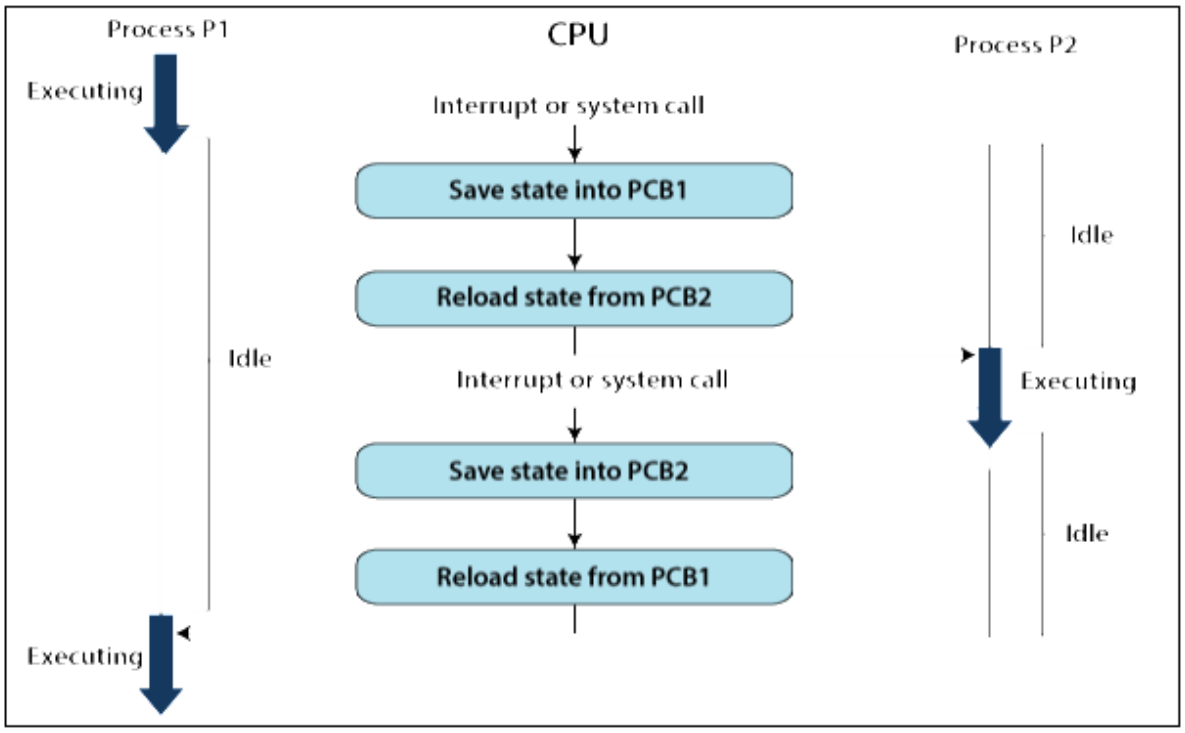
What is Context Switching

Context Switching is the saving and restoring of computational state when switching between different threads or processes, known as tasks. This is an essential feature of a multitasking operating system so that a task can be restored and resume execution at a later point. This allows multiple processes to share a single central processing unit (CPU).

There are multiple cases where context switching may occur:

- In a multitasking context, it refers to the action of storing the system state for one task, so that one task can be paused and another task resumed.
- A context switch can also occur as the result of an interrupt, such as when a task needs to access disk storage, freeing up CPU time for other tasks.
- Some operating systems also require a context switch to move between user mode and kernel mode tasks.

How does Context Switching happen



In the figure above, you can see that initially process P1 is in running state and process P2 is in ready state. Now, when an interruption occurs and calls for a context switch, you should switch process P1 from running state to ready state after saving the context, and then switch process P2 from ready state to running state. The following steps will be performed:

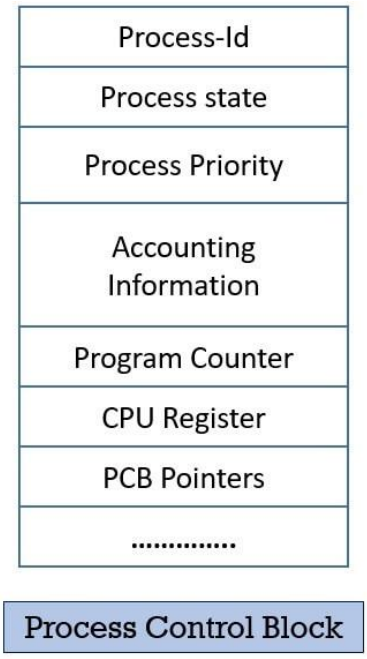
1. First, the context of process P1, that is, the process present in the execution state, will be saved in the Process Control Block of process P1, that is, PCB1.
2. Now, you must move PCB1 to the relevant queue, i.e., ready queue, I/O queue, waiting queue, etc.

3. In the ready state, select the new process that should be executed, i.e. process, P2.
4. Now update the Process Control Block of process P2, ie PCB2, setting the process state to run. If process P2 was previously executed by the CPU, you can get the position of the last instruction executed so that it can resume execution of P2.
5. Likewise, if you want to run process P1 again, you must follow the same steps mentioned above (from step 1 to 4).

Context switching is used to achieve multitasking, that is, time-sharing multiprogramming. Multitasking gives users the illusion that more than one process is running at the same time. But, in reality, only one task is being performed at any given time by a processor. Here, context switching is so fast that the user feels that the CPU is performing more than one task at the same time.

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. When a process is created (started or installed), the operating system creates a corresponding process control block.



Process Control Block

The figure above shows the main information that a PCB may include in its structure. While the details of these structures are system dependent, the common elements fall into three main categories:

- Process identification
- Process state
- Process control

Process identification data includes a unique identifier for the process (almost invariably an integer) and, in a multi-role-multitasking system, data such as parent process identifier, user identifier, user group identifier, etc. The process ID is particularly relevant as it is often used to cross-reference the definitions defined above, for example to show which process is using which I/O devices or memory areas.

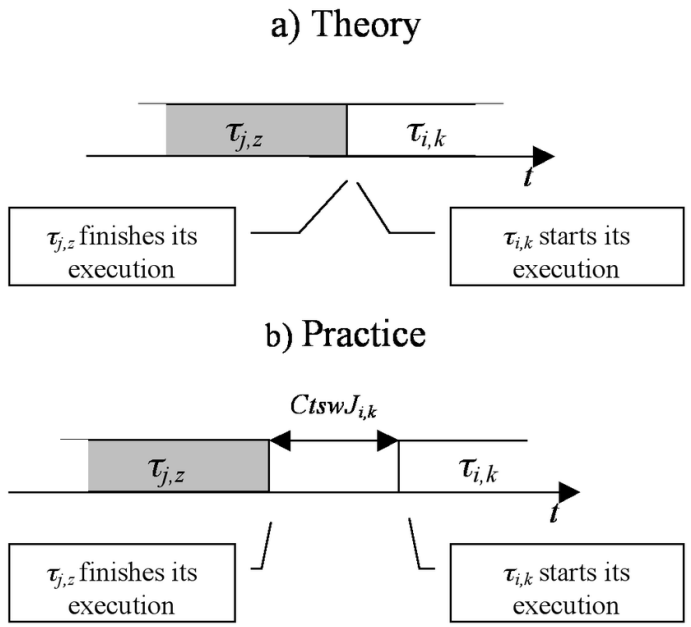
The defined process state data or status of a process when it is suspended, allowing the operating system to restart later. This always includes the contents of general purpose CPU registers, a CPU process status word, stack and frame pointers, and so on. During a context switch, the running process is stopped and another process is completed. The kernel must stop an execution of the running process, copy the values from the hardware registers to its PCB, and update the hardware registers with the values from the PCB of the new process.

Process control information is used by the operating system to manage the process itself. That includes:

- Process scheduling state - The state of the process in terms of "ready", "suspended", etc., and also other scheduling information such as priority value, the amount of time elapsed since the process gained control of the CPU or since it was suspended. Also, in case of a suspended process, event identification data must be recorded for the event that the process is waiting for.
- Process structuring information - the child ids of the process, or the ids of other processes related to the current in some functional way, which can be represented as a queue, ring, or other data structures
- Inter-process communication information - flags, signals and messages associated with communication between independent processes
- Process privileges - allowed / not allowed access to system resources
- Process number (PID) - unique identification number for each process (also known as process ID)
- Program counter (PC) - A pointer to the address of the next instruction to be executed for this process
- CPU registers - set of registers where the process needs to be stored for execution to the execution state
- CPU Schedule Information - CPU Time Schedule Information
- Memory Management Information - Page Table, Memory Limits, Segment Table
- Accounting information - amount of CPU used for the execution process, time limits, execution ID, etc.
- I/O status information - list of I/O devices allocated to the process.

Performance considerations

Context switching involves costs that may affect the system's overall performance. These direct costs arise mainly from the fact that it takes time to save the context of a process that is running and then restore the context of another process that is about to run. 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

Translating a virtual address to a physical address is expensive. The processor must access the pages table structures, which usually have 3-4 levels. Thus, a single memory access actually requires 4-5 memory accesses.

To mitigate this issue, most modern processors cache virtual-to-physical translations in a translation lookaside buffer (TLB). The TLB is part of the MMU and can be understood as a cache for the MMU.

When virtual memory is updated - for instance, when one process's address space is replaced with another's during a software context switch - the TLB suddenly contains "stale" translations that are no longer valid. These translations must be flushed for correct behavior. This is less than ideal, as the next few operations must wait for the slow virtual-to-physical translations.

Recent Intel and AMD processors sport a tagged TLB, which allows you to tag a given translation with a certain address space configuration. In this scheme TLB entries never get "stale", and thus there is no need to flush the TLB.

Address Space Identifier

On ARM systems, this TLB tagging mechanism is implemented as follows: a value called address space identifier (ASID) is assigned by the OS to each task, so the MMU can distinguish between memory pages which share the same virtual address. For ARMv7 systems in particular, the ASID is an eight-bit value. For ARMv8, it can be 8 or 16 bits in length. The presence of the ASID in the TLB allows it to identify for each entry which Address Space it belongs to. When it comes to context switching, one of the necessary steps in the switching of task context is making sure the translation process using TLB, won't translate to a physical address of another address space. One of the solutions is to use and

update the current ASID value, identifying if an entry in cache should or not be used. In some system implementations, ASID values might be ignored altogether, in this case, the solution to this problem is to just invalidate the whole TLB cache, always resulting in page-faults and fetching the translation data from the correct process page table.

When using the short-descriptor translation table, the ASID value is stored in the CONTEXTIDR register. In case of the long-descriptor, TTBR0 register is a 64 bits register and it also stores the current ASID value.

ASID on Context Switch

Below we present two implementations for updating the process page table address and the ASID value.

```
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
```

In the first example, the address of the translation table (page table) is changed to a translation table that only global-pages could be accessed or translated, ensuring that no non-global pages can be fetched, because and it is uncertain if the old or new address space would be used for translating virtual addresses.

```
Change ASID to 0

ISB

Change Translation Table Base Register

ISB

Change ASID to new value
```

In the second example, the ASID value is set to zero, which is a value normally not used for any operations and there should not exist any entries on TLB with such ASID. In this situation, we also ensure that translation would occur correctly, since translation will have to access the translation table.

Limitation of ASID

In some cases, the ASID value is represented by 8 bit, therefore, there can only be 256 different address spaces, since we only have 256 different identifiers. Because, most likely, processes do not share address spaces with each other, as a result, we are also bound to have up to 256 different tasks running at once in the system. The long-descriptor is a solution to this problem, because it offers not 8 but 16 bits for address space identifiers.

Linux 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 (unable to run) to get one.

Switching Context in ARMv7/Raspberry Pi3

ARM Processor Mode

The ARM processor has many execution modes, this is important for task context switching because some of the indispensable register read and write 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.

IRQ

IRQ handlers or *interrupt request handlers* 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 hardware interrupts are used to handle events such as receiving data from a modem or network card, key presses, or mouse movements.

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

1. Save the address of the next instruction in the appropriate Link Register LR.
2. Copy CPSR to the SPSR of new mode.
3. Change the mode by modifying bits in CPSR.
4. Fetch next instruction from the vector table.

And to exit it:

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

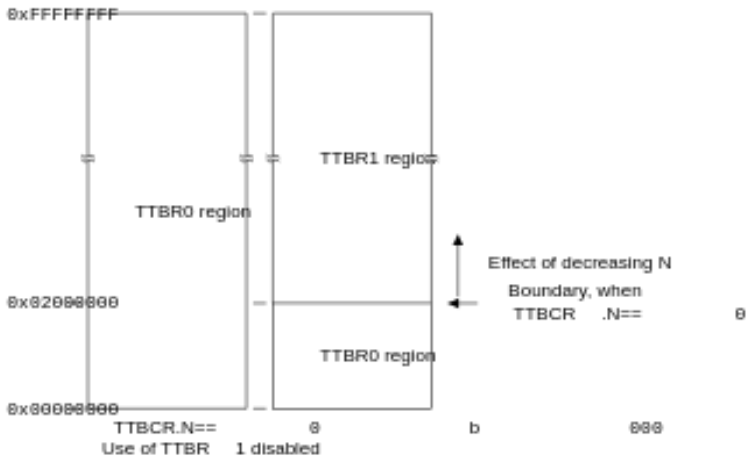
Managing Address Spaces

One of the necessary operations when switching to the current task context is managing the address spaces and references to the process page table. In the Armv7 architecture, we have a set of translation table support registers composed of TTBR0, TTBR1 and TTBCR.

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. Such division allows the entries translating virtual addresses allocated by the kernel (on the system page table) into physical addresses without duplicating these entries on multiple tasks page tables. Therefore, when it comes to context switching, it is only required to update the information contained on TTBR0.

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. The least significant two bits of the TTBCR represents an unsigned integer N, where, if the most significant N bits of the virtual address is zero, then the translation should occur on TTBR0, otherwise, translation should occur on TTBR1. Although, there is a special case, in which the value of N is zero, in this case, TTBR1 should be completely ignored and TTBR0 will be the only translation table used on the machine.

The TTBR0 register, bits 31:14 is used to store the base address of the translation table, and should be accessed by the MMU to translate virtual addresses.



In the presented image, the TTBCR.N (4 least significant bits of TTBCR) decides which of the translation tables is going to be used. Therefore, in the example on the left, where TTBCR.N is equal to 0x0, that means that every virtual address will useTTBR0. In the example on the right, TTBCR.N could be something similar to 0x1, so addresses in the format 0xdXXXXXXX will use TTBR0 as the translation table.

Accessing translation table support registers

It should first be noted that to access any of the translation table support register the ARM processor should be running on **privileged mode** at the moment of the access. The access of each register requires the use of the “MCR” and “MRC” assembly instructions. We show below the access of each register.

Assembly	Register
`MRC p15, 0, <Rt>, c2, c0, 0`	Read Translation Table Base Register 0
`MCR p15, 0, <Rt>, c2, c0, 0`	Write on Translation Table Base Register 0
`MRC p15, 0, <Rt>, c2, c0, 1`	Read Translation Table Base Register 1
`MCR p15, 0, <Rt>, c2, c0, 1`	Write on Translation Table Base Register 1
`MRC p15, 0, <Rt>, c2, c0, 2`	Read Translation Table Base Constro Register
`MCR p15, 0, <Rt>, c2, c0, 2`	Write on Translation Table Base Constro Register

References

- https://en.wikipedia.org/wiki/Process_control_block
- https://www.tutorialandexample.com/what-is-context-switching/
- https://afteracademy.com/blog/what-is-context-switching-in-operating-system
- https://wiki.osdev.org/Context_Switching
- https://developer.arm.com/documentation/den0024/a/The-Memory-Management-Uni-/Context-switching
- https://github.com/sokoide/rpi-baremetal
- https://github.com/bztsrc/raspi3-tutorial
- https://developer.arm.com/documentation/ddi0406/c/System-Level-Architecture/The-System-Level-Programmers--Model/ARM-processor-modes-and-ARM-core-registers/ARM-processor-modes?lang=en#CIHGHDGI