# 1 Introduction

To most people, embedded systems are not recognizable as computers. Instead, they are hidden inside everyday objects that surround us and help us in our lives. Embedded systems typically do not interface with the outside world through familiar personal computer interface devices such as a mouse, keyboard and graphical user interface. Instead, they interface with the outside world through unusual interfaces such as sensors, actuators and specialized communication links.

Real-time and embedded systems operate in constrained environments in which computer memory and processing power are limited. They often need to provide their services within strict time deadlines to their users and to the surrounding world. It is these memory, speed and timing constraints that dictate the use of real-time operating systems in embedded software.

## 1.1 Motivation

The motivation for this work was the requirement of an operating system that could do all the basic functionality of a real time Operating system (OS) and is Linux flavoured for the COFFEE RISC Core ™ that was developed in the Institute of Digital and Computer Systems at Tampere University of Technology, Finland. Linux is becoming more and more powerful as there are millions of minds working on it. There are forums where Linux guys working on the related things discuss on the efficiency of the code they wrote for the Linux OS, lot of discussion is made on the particular code and efforts are made to come with the solution for the drawback it has (if any). Once the piece of code is there in the OS, as it is an open source, many will use if and find any drawback it has (if there are any), also try to figure out if the security is compromised in any sort. If anything is found it will be discussed in the forum related to the specific topic (there are many forums which are dedicated to a specific topic like device drivers only, emulation board problems etc) and the problem is solved in the very early stages before anyone else would find and use that drawback for wrong purposes. Each line of the code will be there in the OS after vigorous testing by the programmer. Obviously the loop holes left by him will be filled by guys like us who are using it.

## 1.2 Problem Formulation

The work is intended to result in the operating system that would give the same interface as the other earlier versions of Linux with least new interfaces that could make the developer of the application who would run the application on the COFFEE processor think as if he is running on

Linux kernel, i.e. to port the Linux flavoured kernel uClinux to COFFEE. This was a difficult job because of

- The job requires identifying the processor dependent and independent codes in the kernel.
- The adaptation to the new instruction set.
- Need of understanding of the processor architecture thoroughly.
- Need of good understanding of the real time operating system concepts.
- Good programming skills at least in C and ASSEMBLY languages.
- Testing OS with the tools that were also under development and probably with some bugs in them and finding where the problem is i.e. in the kernel or in the tool chain or was it because of lack of understanding between the tool chain and the kernel.

The main problems that are addressed in this thesis are

- Definition and characterizing the real time operating system in general
- Structuring and categorizing the existing scheduling and memory security methods in real-time operating systems.
- Detailed explanation of how and where processor dependent codes are mapped to the existing versions of the uClinux.
- Difference in behaviour of CUP-OS (name of the operating system for our Processor) to the uClinux.

## 1.3 Method

In order to solve these problems study and analysis of current literature in the Real-Time Operating Systems (RTOS) area was conducted thoroughly. An analysis was performed and the result structured in order to make it general and useful for the further study. Second, a detailed study of the available popular processors using same flavour of operating systems area was performed, and also observing and analyzing the way of utilizing the resources which include the register set(s), timers and other things related to OS, the results summarized and categorized. These results form a structured definition and description of concepts and services that should be ported. This also forms the basis for the rest of the study.

In order to make more efficient use of the processor that was designed in the Institute of Digital and Computer Systems of Tampere University of Technology, a very efficient and reliable OS was proposed by the team and Linux is now known to be the best and fast evolving operating system, it

was chosen that the port to be a Linux flavoured. This will fulfil the requirement of having thoroughly tested algorithms to be inserted into the algorithms that are processor dependent.

With the requirements and specifications based on many literature findings and the examples practical constraints the evolved operating system was found useful and working well to the proposed requirements with some small modifications and limitations. During this process some ideas for the future work (section 10.3) was also discovered and are suggested in the final chapter.

## 1.4 Disposition

Until now Chapter 1 presented a general picture of concepts that inspired me in working in this project with the details of the background needed and possible future works. Chapter 2 explains the real-time OS concepts briefly which will be helpful to understand how far CUP-OS is able to accomplish these tasks by reading through the chapters coming after this one. Chapter 3 gives some background needed about the COFFEE RISC Core™ so as to make the understanding of the concepts in the further chapters more clear. Processes are a fundamental abstraction offered by Linux and are introduced in Chapter 4.Here we also explain how each process runs either in an unprivileged User Mode or in a privileged Kernel Mode. The simple Memory Management of using the data memory fairly and efficiently is given very shortly in Chapter 5. Transitions between User Mode and Kernel Mode happen only through well-established hardware mechanisms called *interrupts* and *exceptions*. Process running in User Mode makes requests to the kernel that is how system calls are implanted for coffee and the compatibility for the API (Application Programmer Interface) for adding more systems calls in the future; all these features represent the *kernel entry and exit* points while running user applications which are introduced in Chapter 6. One type of interrupt is crucial for allowing Linux to take care of elapsed time; further details on such *Timing Measurement* can be found in Chapter 7.

Chapter 8 explains how Linux executes, in turn, every active process in the system so that all of them can progress toward their completion that is about *Process Scheduling*. Chapter 9 covers a very detail description of how the testing was made on the kernel and the test cases and the test environment used to test each of its modules. In Chapter 10 we see how the problems that were discussed in the first chapter were overcome and some conclusions about the work and ending with the future works that will follow this work immediately.

# 2 Real Time Operating System Concepts

An operating system is the interface between a user's program and the underlying computer hardware. It also manages the execution of user programs such that multiple programs can run simultaneously and access the same hardware [7]. Everyone who uses a computer encounters an operating system, whether it is Windows, Mac-OS, Linux, DOS, or UNIX.

This chapter describes the architecture and functionality of an operating system and then summarizes the requirements of a real-time operating system. Discussion of standards and of specific details will be limited to Unix-like operating systems, since Linux is a Unix-like OS and CUP-OS is a Linux-like OS. The chapter concludes with an overview of real time kernels.

## 2.1 Architecture of an Operating System

An operating system, or more specifically the core or kernel of the operating system, is always resident in memory and provides the interfaces between user programs and the computer hardware. The name kernel follows from the analogy of a nut, where the kernel is the very heart of the nut and, in the computing domain, the kernel is the very heart of the operating system.

COFFEE RISC Core ™ is based on Harvard architecture where data memory space is same for both user and kernel and the processor and kernel both jointly have the responsibility of making use of the memory securely. The kernel of a multi-tasking operating system can manage multiple user programs running simultaneously in user space so that each program thinks it has complete use of all of the hardware resources of the computer and, other than for intentional messages sent between programs, each program thinks that it has its own memory space and is the only program running.

Communication between user-space programs and the kernel code is achieved through system calls to the kernel code. These system calls typically are to access shared physical resources such as disk drives, serial/parallel ports, network interfaces, keyboards, mice, display screens, and audio and video devices [19]. One unifying aspect of Linux/Unix systems is that all the physical resources appear to the user programs as files and are controlled with the same system calls.

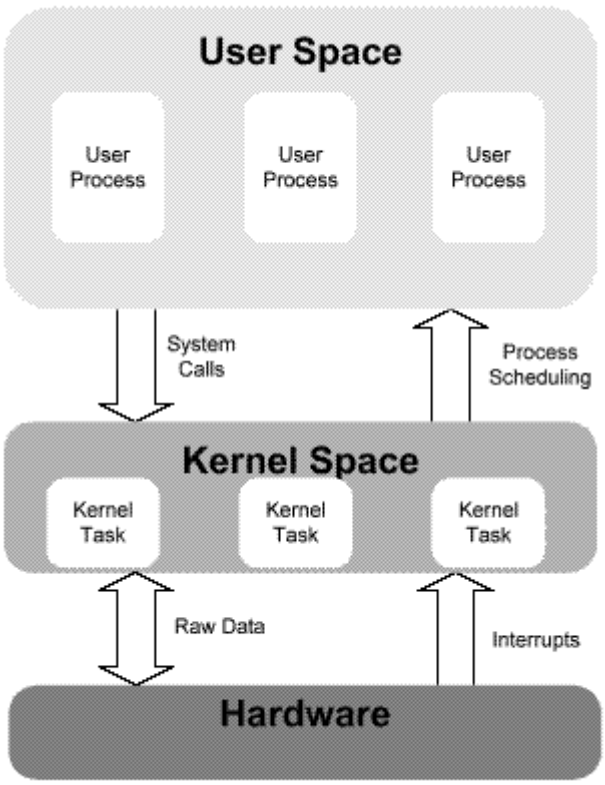


**Figure 1: Outer look of the Operating System Interface and Services. [19]**

All of the input/output activity is controlled by the kernel code so that the user-space programs do not have to concern the details of sharing common physical resources. Device-specific drivers in the kernel manage those details. An operating system is thus tailored to run on specific computer hardware and it isolates user programs from the specifics of the hardware, allowing for portability of user-space application code.

The architecture of an operating system is thus a core or kernel that remains in memory at all times, a set of processes in user-space that support the kernel, plus various modules and utility programs that remain stored in COFFEE Core ™ until needed. The kernel manages simultaneous execution of multiple user programs and isolates user programs from the details of managing the specific hardware of the computer.

## 2.2 Services of the operating system

The main services provided by the kernel of the operating system are memory management, process scheduling, interfacing to the hardware, timing services, special routines to interface to the kernel called system calls and communication with external devices and networks. They are all discussed briefly in this section.

### 2.2.1 Multitasking

A Task is a semi-independent program with a dedicated purpose. Each executing program is a **task** under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be **multitasking.** The **kernel** is the core component within an operating system. Operating systems such as Linux employ kernels that allow users access to the computer seemingly simultaneously. Multiple users can execute multiple programs apparently concurrently.

The use of a multitasking operating system can simplify the design of what would otherwise be a complex software application:

- The multitasking and inter-task communications features of the operating system allow the complex application to be partitioned into a set of smaller and more manageable tasks.
- The partitioning can result in easier software testing, work breakdown within teams, and code reuse.
- Complex timing and sequencing details can be removed from the application code and become the responsibility of the operating system.

A conventional processor can only execute a single task at a time - but by rapidly switching between tasks a multitasking operating system can make it appear as if each task is executing concurrently.

### 2.2.2 Process Management

Process management is the collection of activities of planning and monitoring the performance of a process. A process is a program or set of instructions that take some amount of processor time to complete a task. The task could be anything from just updating the memory or some complex thing like monitoring a network port. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task. The operating system is responsible for the following activities in connection with process management:

- process creation and deletion.
- process suspension and resumption.
- provision of mechanisms for:

  - process synchronization
  - process communication
  - Deadlock handling
- process sharing (scheduling),
- process synchronization mechanisms, and
- process protection and security.

### 2.2.3 Scheduling

On most multitasking systems, only one process can truly be active at a time - the system must therefore share its time between the executions of many processes. This sharing is called scheduling.

Different methods of scheduling are appropriate for different kinds of execution. A queue is one form of scheduling in which each program waits its turn and is executed serially. This is not very useful for handling multitasking, but it is necessary for scheduling devices which cannot be shared by nature. An example of the latter is the printer. Each print job has to be completed before the next one can begin; otherwise all the print jobs would be mixed up and interleaved resulting in nonsense.

There are many types of scheduling algorithms which are given below and the detailed description for them are given more precisely in ***chapter 8 Process Scheduling***

- First-In-First-Out Scheduling (FIFO)
- Last-In-First-Out (LIFO)
- Real-Time Scheduler also called RT scheduler
- Round Robin Scheduler etc ….

To choose an algorithm for scheduling tasks we have to understand what it is we are trying to achieve, i.e., what are the criteria for scheduling. The efficiency we need and what factors can be compromised for improving other factors like improving throughput with the expense of making the scheduler unfair or non real time behaviour to others. Many other factors come into picture and these factors depend on actually the purpose of the scheduler, which could be any one or more of the following

- to distribute the processor time fairly among the processes.
- to make one or more processor real-time, i.e., they have processor time when ever they needed without any competition from other processes.

**Scheduling hierarchy**

Complex scheduling algorithms distinguish between short-term and long-term scheduling. This helps to deal with tasks which fall into two kinds: those which are active continuously and must therefore be serviced regularly, and those which sleep for long periods.

For example, in UNIX the long term scheduler moves processes which have been sleeping for more than a certain time out of memory and onto disk, to make space for those which are active. Sleeping jobs are moved back into memory only when they wake up (for whatever reason). This is called swapping.

The most complex systems have several levels of scheduling and exercise different scheduling polices for processes with different priorities. Jobs can even move from level to level if the circumstances change.

### 2.2.4 Context Switching

Context switching occurs when a multitasking operating system stops running one process and starts running another. Many operating systems implement concurrency by maintaining separate environments or "contexts" for each process. The amount of separation between processes, and the amount of information in a context, depends on the operating system but generally the OS should prevent processes interfering with each other, e.g. by modifying each other's memory.

A context switch can be as simple as changing the value of the program counter and stack pointer or it might involve resetting the MMU to make a different set of memory pages available.

In order to present the user with an impression of parallel execution, and to allow processes to respond quickly to external events, many systems will context switch tens or hundreds of times per second.

## 2.3 Memory Management

Memory management is one of the most fundamental areas of computer programming. In many scripting languages, knowing the abilities and limitations of your memory manager is critical for effective programming. In most systems languages like C and C++ [17], you have to do memory management. Back in the days of assembly language programming, memory

management was not a huge concern. You basically had run of the whole system. Whatever memory the system had, so did you. You do not even have to worry about figuring out how much memory it had, since every computer had the same amount. If the requirements were pretty static, we just choose a memory range to use and used it.

However, even in such a simple computer there are still some issues, i.e. if we do not know how much memory is needed by each part of the program [6], if we have limited space and varying memory needs then we need some way to meet these requirements which can be:

- Determine if we have enough memory to process data
- Get a section of memory from the available memory
- Return a section of memory back to the pool of available memory so it can be used by other parts of the program or other programs

The libraries that implement these requirements are called allocators, because they are responsible for allocating and deallocating memory. The more dynamic a program is, the more memory management becomes an issue, and the more important your choice of memory allocator becomes.

## 2.4 System Calls

In any modern operating system, there is a basic dichotomy between code which runs in privileged mode (kernel space) and code which executes in user space. Kernel code has complete control over the machine; it can access any of the machine's resources, such as memory, network adapters, and disk drives. User space code has limited access to system resources. In order to read from a disk drive or write to the network, for example, user code has to ask the kernel to perform the work on behalf of the user code. If the user code tries to carry out an operation which it does not have permission to do, the microprocessor notifies the kernel, which normally kills the user space process.

This split between kernel and user code allows computers to juggle many independent tasks. The kernel allows a user space program to run for a while, and then stops it to let other tasks run. Additionally, the kernel can instruct the microprocessor to prevent one program from interfering with resources being used by another, thus preventing tasks from harming one another. Whenever user space programs need to access system resources they don't own, they have to ask the kernel for help. File and network access, creating and destroying other processes, and allocating additional memory are all areas where the kernel becomes involved.

By being involved in these types of operations, the kernel retains complete control over the system. One task can be refused access to a file when another is accessing it, memory allocation requests can be denied if the system is running low on resources, and users can be prevented from killing each other's processes.

System calls allow user space programs to request services from the kernel. In C, system calls look just like normal function calls, but they have a very different implementation. Rather then simply transferring control of the program, system calls switch the system to kernel mode. Once the kernel has control, it performs the requested service, returns the system to user mode, and then transfers control back to the originating process.

Every Linux program can be thought of as a very simple loop:

1. Compute something
2. Make a system call
3. Go to step 1

System calls can be just thought of ordinary function calls for an application programmer but they are very special function calls if you look under the hood, so for a newbie or for a pure application programmer it would be quite confusing to differentiate between the normal function call and a system call. There is a basic dichotomy between code which runs in privileged mode (kernel mode) and code which executes in user space. In Linux there are two categories of functions, based on how they are implemented

- A library function is an ordinary function that resides in a library external to your program. Most of the library functions are in the standard C library, libc. For example, getopt_long and mkstemp are functions provided in the C library.
- A call to a library function is just like any other function call. The arguments are placed in processor registers or onto the stack, and execution is transferred to the start of the function's code, which typically resides in a loaded shared library.
- When a program makes a system call, the arguments are packaged up and handed to the kernel, which takes over execution of the program until the call completes. A system call isn't an ordinary function call, and a special procedure is required to transfer control to the kernel. However, the GNU C library (the implementation of the standard C library

provided with GNU/Linux systems) wraps Linux system calls with functions so that you can call them easily as if you call an ordinary function within the program. Low-level I/O functions such as open and read are examples of system calls on Linux [3].

- The set of Linux system calls forms the most basic interface between programs and the Linux kernel. Each call presents a basic operation or capability.
- Some system calls are very powerful and can exert great influence on the system. For instance, some system calls enable you to shut down the Linux system or to allocate system resources and prevent other users from accessing them. These calls have the restriction that only processes running with superuser privilege (programs run by the root account) can invoke them. These calls fail if invoked by a non-superuser process [7].

# 3 COFFEE RISC Core ™

This chapter gives introduction to the RISC Core hardware specifications. It is important to know the processor before dwelling deeply into further topics. As the OS acts as the abstract layer for the hardware, from OS designer point of view this chapter is found very useful. All the information in this chapter can be found in the official web page of the processor. [1]

COFFEE RISC Core ™ is a Reduced Instruction Set Computer (RISC) core developed in the Institute of Digital and Computer Systems at Tampere University of Technology, Finland. Its features are summarised below:

- single six-stage pipeline
- Harvard architecture
- clean RISC style instruction et with 67 instructions
- two instruction encoding:16 bit and 32 bit
- conditional execution of instructions
- operating system support: privileged operating mode, runtime configurable memory address checks and internal timers.
- two register banks, 32 registers in each bank
- memory mapped configuration block for easy access by software. Can be extended and remapped anywhere in the address space.
- internal interrupt controller, 12 external interrupts supported with configurable priorities
- four coprocessors can be connected

## 3.1 Operating Modes

**16 bit mode and 32 bit decoding modes**

16 bit mode refers to length of the instruction word. When in this mode, core expects to get instruction words encoded in 16 bits. Mode can be switched on the fly using *swm* –instruction. Of course when running actual code, the encoding really has to change after *swm* –instruction (See document instruction execution cycle times).

**Limitations in 16 bit mode**

- only 8 registers per set available: registers 24...31 mapped as registers 0...7
- Conditional execution is not available

- Only one condition register(CR0) in use
- Immediate constants are shorter, see instruction specifications.
- Instructions lui, lli, exbfi and cop not available (available as pseudo –operations if supported by assembler).
- 2nd source register and destination register shared.

**Super user mode**

The core can operate in *super user* –mode or *user* –mode. In super user –mode, core can access the whole memory space and both register banks. In user –mode, access to protected memory areas (software configurable) is denied and only 1st register bank is accessible. It's possible to switch from super user -mode to user –mode but not vice versa, except using scall –instruction which transfers execution to system code. System code entry address must be configured in startup code. Interrupt service routines can be run in both modes. This can also be configured by startup code. Core boots in super user –mode, which makes it possible to do the necessary configurations before starting application in user –mode.

**Resetting the processor**

After powering up the core, rst_x pin should be pulsed low (clock has to be stable) to set the core in correct state. If boot address selection is enabled (boot_sel –pin pulled high), boot address should be driven to data bus simultaneously with rst_x –signal. If boot address selection is disabled, core will boot at address 0x00000000h. Normal operation will start two clock cycles after the rising edge of the rst_x –signal.

**Defaults after reset and boot procedure**

Core will boot in super user and 32 bit –modes. Interrupts are disabled. A typical boot procedure would be to execute assembly written boot code which sets all CCB registers to suitable values and switches to user –mode by executing *retu* –instruction. See instruction specifications.

**About configuring the core**

Several features of the core can be configured via the core configuration block (CCB) which is a memory mapped register bank. When writing a new value to a configuration register, the new value will be valid when the instruction accessing CCB is in stage 5 of the pipeline. It follows that, if some configurations affect the execution of some instructions, or some configurations should be valid, when executing certain instructions, one has to make sure that there is enough instructions

between the ones accessing CCB and dependent instructions. These can be nop – instructions or other instructions which do not depend on values of the configuration registers. Table below shows few examples of situations where it is essential to have few instructions between a CCB write and an instruction depending on the configuration made. If you're not sure about the number of 'guard' –instructions, use four.

**Table 1 Configuring the Core**

| instruction | purpose | notes | Dependency |
|---|---|---|---|
| st R1, R0, 0h | Remapping CCB to new address. | Assume R0 contains address of CCB_BASE – register and R1 contains a new address for CCB. | The 2nd st –instruction needs the value of CCB_BASE in stage 3 of the pipeline. CCB_BASE is valid when the 1st st – instruction is in stage 5 of the pipeline => There needs to be one instruction between the stores. In this case it is addi. |
| addi R0, R1, 1h | incrementing the new address of CCB. R0 should point now to CCB_END. | 'guard' instruction | |
| st R2, R0, 0h | Configuring the size of configuration block itself (internal + external blocks) | Assuming R2 contains an address to be written to CCB_END. | |
| **instruction** | **purpose** | **notes** | **Dependency** |
| st R1, R0, 0h | Set an interrupt vector. | Assume R0 contains address of EXT_INT0_VEC and R1 points to interrupt service routine. | Interrupt vector will be valid when st –instruction has proceeded to stage 5 of the pipeline. Interrupts will be enabled when ei – instruction reaches stage 2 of the pipeline. Need to fill stages 3 and 4 to be safe. |
| nop | idle instructions ('guard' instructions) | Could use some other 'useful' instructions | |
| nop | | | |
| ei | Enable interrupts | | |
| **instruction** | **purpose** | **notes** | **Dependency** |
| st R1, R0, 0h | Configure register translation for coprocessor access. | Assume R0 contains address of CREG_INDX_I and R1 valid configuration. | Configuration will be valid when st –instruction has proceeded to stage 5 of the pipeline. Configuration is needed when cop –instruction reaches stage 2 of the pipeline. Need to fill stages 3 and 4 to be safe. |
| nop | idle instructions ('guard' instructions) | Could use some other 'useful' instructions | |
| nop | | | |
| cop sqr(R2, R15) | Transfer an instruction word to coprocessor for execution | | |

## 3.2Registers

COFFEE has two different register sets. The first set (SET 1) is intended to be used by application programs. The second set of registers (SET 2) is for privileged software which could be an operating system or similar. SET 2 is protected from application program. Privileged software can

access both sets. There's a total of 32 registers in both sets including general purpose registers (GPRs) and special purpose registers (SPRs).
In addition COFFEE has eight condition registers (CRs) which are used with conditional branches or when executing instructions conditionally. These are visible to application software as well as to privileged software. Besides the register bank described here, COFFEE has another register bank, CCB (core control block), which is mapped to memory (accessed using ld and st –instructions). CCB is for controlling the processor operation and as such should be configured by boot code. CCB also contains few status registers.

The usage of general purpose registers is not restricted by hardware in any way. In any case, good programming means fixing some registers for a certain purpose.

**Table 2 Core Registers**

| SET 1 | | | SET 2 | | |
|---|---|---|---|---|---|
| R0 | GPR | 32 bits | PR0 | GPR | 32 bits |
| R1 | GPR | 32 bits | PR1 | GPR | 32 bits |
| ... | | | ... | | |
| R28 | GPR | 32 bits | PR28 | GPR | 32 bits |
| R29 | GPR | 32 bits | PR29 | PSR | 8 bits |
| R30 | GPR | 32 bits | PR30 | SPSR | 32 bits |
| R31 | GPR/LR | 32 bits | PR31 | GPR/LR | 32 bits |

## SET 1 GPRs

SET 1 has 32 identical general purpose registers R0...R31 with one exception: R31 is used as a link register(LR) with some instructions. The programmer is free to use R31 for any other purpose as long as it's special behaviour is taken into account. All general purpose registers (and the link register) are 32 bits wide.

## SET 2 GPRs

SET 2 has 30 identical general purpose registers PR0...PR28 and PR31 with one exception: PR31 is used as a link register by some instructions. The programmer is free to use PR31 for any other purpose as long as it's special behaviour is taken into account. All general purpose registers (and the link register) are 32 bits wide.

## SET 2 SPRs

There's two special purpose registers in SET 2: PSR and SPSR. PSR is eight bits wide. When reading data from PSR the 'non existent' bits are read as zeros. Writing to a read only register(PSR) is ignored.

**PSR (register index 29)**

Processor Status Register is a read only register and contains the flags explained below. Bits 7 downto 5 are reserved for future extensions.

| RESERVED | IE | IL | RSWR | RSRD | UM |
|---|---|---|---|---|---|
| 7...5 | 4 | 3 | 2 | 1 | 0 |

IE = 1: Interrupts enabled, IE = 0: Interrupts disabled.

IL = 1: Instruction word length is 32 bits, IL = 0: Instruction word length is 16 bits.

RSWR bit selects which register set to use as target:

RSWR = 1: SET2, super users set; RSWR = 0: SET1, users set.

RSRD bit selects which register set to use as source:

RSRD = 1: SET2, super users set; RSRD = 0: SET1, users set.

UM indicates which user mode the processor is in:

UM = 0: super user mode, UM = 1 : user mode.

RESERVED: Read as zeros.

**SPSR (register index 30)**

SPRS is used to save PSR flags when changing user mode by executing scall – instruction. It can be also used to set mode flags for the user: IE and IL flags are copied from SPSR to PSR when retu – instruction is executed. Note that bits 31 downto 5 are writable but only bits 7 downto 0 are saved in case of scall.

## CRs

There's eight three bit wide condition registers C0...C7 (visible both to application software and privileged software). Condition registers are used with conditional branches or when executing instructions conditionally. Each register contains three flags: Z (Zero), N (Negative) and C (Carry). When executing compare instructions or some arithmetic instructions these three flags are calculated and saved to the selected CR (arithmetic instructions always save flags to C0). When conditionally branching or executing, flags from the selected CR are compared to match a certain condition given by the programmer.

## CCB registers

Note, that 'byte' addresses (that is consecutive addresses) are used in table below. 256 consecutive addresses are reserved for core configuration block. Addresses beyond CCB_BASE + ffh can be configured to point to an external peripheral configuration block (PCB), if present.

*Registers which are shorter than 32 bits:*

- LSB of a GPR corresponds to LSB of the short register in CCB.
- Unused bits read as zeros.
- For code compatibility with future versions, you should write unused bits as you would if there were more bits (interrupt masking, for example).

**Table 3 Core control block Registers (CCB)**

| Offset | mnemonic | Width | description/usage | notes |
|---|---|---|---|---|
| 00h | CCB_BASE | 32 | Start address of this relocatable configuration block (address of the CCB_BASE itself) | Has to be aligned to 256B boundary! That is, bits 7 downto 0 must be zeros |
| 01h | CCB_END | 32 | End address of configuration register space. | |
| 02h | COP0_INT_VEC | 32 | Co-processor 0 interrupt service routine start address. | See chapter Kernel Entry and Exit - interrupts |
| 03h | COP1_INT_VEC | 32 | Co-processor 1 interrupt service routine start address. | |
| 04h | COP2_INT_VEC | 32 | Co-processor 2 interrupt service routine start address. | |
| 05h | COP3_INT_VEC | 32 | Co-processor 3 interrupt service routine start address. | |
| 06h | EXT_INT0_VEC | 32 | External interrupt 0 service routine base address. | |
| 07h | EXT_INT1_VEC | 32 | External interrupt 1 service routine base address. | |
| 08h | EXT_INT2_VEC | 32 | External interrupt 2 service routine base address. | |
| 09h | EXT_INT3_VEC | 32 | External interrupt 3 service routine base address. | |
| 0ah | EXT_INT4_VEC | 32 | External interrupt 4 service routine base address. | |
| 0bh | EXT_INT5_VEC | 32 | External interrupt 5 service routine base address. | |

| | | | | |
|---|---|---|---|---|
| 0ch | EXT_INT6_VEC | 32 | External interrupt 6 service routine base address. | See chapter Kernel Entry and Exit - interrupts |
| 0dh | EXT_INT7_VEC | 32 | External interrupt 7 service routine base address. | See chapter Kernel Entry and Exit - interrupts |
| 0eh | INT_MODE_IL | 12 | Instruction decoding mode flags for interrupt routines (PSR:IL is set accordingly when entering routine). | Bit associations: See note 3 below. |
| 0fh | INT_MODE_UM | 12 | User mode flags for interrupt routines(PSR:UM, RSRD, RSRW are set accordingly when entering routine). | See interrupts and processor status register. |
| 10h | INT_MASK | 12 | Register for masking external and cop interrupts individually. A low bit ('0') means blocking an interrupt source, a high bit enables an interrupt. | |
| 11h | INT_SERV | 12 | Interrupt service status bits. | See chapter Kernel Entry and Exit - interrupts |
| 12h | INT_PEND | 12 | Pending interrupt requests. | |
| 13h | EXT_INT_PRI | 32 | Interrupt priorites:<br>Bits 31 downto 28 : INT 7 priority<br>Bits 27 downto 24 : INT 6 priority<br>...<br>Bits 7 downto 4 : INT 1 priority<br>Bits 3 downto 0 : INT 0 priority | 0 – highest priority<br>15 – lowest priority<br>Priorities for external interrupts can only be set if external handler is not used. |
| 14h | COP_INT_PRI | 16 | Bits 15 downto 12 : COP3 priority<br>Bits 11 downto 8 : COP2 priority<br>Bits 7 downto 4 : COP1 priority<br>Bits 3 downto 0 : COP0 priority | |
| 15h | EXCEPTION_CS | 8 | Exception cause code. | See chapter Kernel Entry and Exit - exceptions. |
| 16h | EXCEPTION_PC | 32 | Address of the instruction which caused the exception. | |
| 17h | EXCEPTION_PSR | 8 | Copy of the processor status flags which were used when decoding the violating instruction. | |
| 18h | DMEM_BOUND_LO | 32 | start of protected/allowed address space for data memory | See user modes: super user. See also register MEM_PCONF. Note that bounds are included in the address space. |
| 19h | DMEM_BOUND_HI | 32 | end of protected/allowed address space for data memory | |
| 1ah | IMEM_BOUND_LO | 32 | start of protected/allowed address space for instruction memory | |
| 1bh | IMEM_BOUND_HI | 32 | end of protected/allowed address space for instruction memory | |

| | | | | |
|---|---|---|---|---|
| 1ch | MEM_PCONF | 32 | Defines whether the space between addresses set by XMEM_BOUND_LO and XMEM_BOUND_HI is protected from user or allowed for user.<br>Bit 0 controls instruction memory protection, bit 1 data memory protection. Bits 31 downto 2 are reserved.<br>Bit high => area is protected Bit low => area is allowed (and the rest is protected) | See note 4. |
| 1dh | SYSTEM_ADDR | 32 | System code entry address. (used by scall) | |
| 1eh | EXCEP_ADDR | 32 | Exception handler entry address. | See chapter Kernel Entry and Exit - exceptions |
| 1fh | WAIT_STATES | 12 | Number of wait cycles for coprocessor and memory accesses. Can be set between 0 and 15<br>bits 11 downto 8: coprocessor access wait cycles. bits 7 downto 4 : data memory and PCB access wait cycles. bits 3 downto 0: instruction<br>memory access wait cycles. | See core interface description. |
| 20h | CREG_I_INDX | 20 | **Specifying register index for coprocessor instruction word.** bits 19 downto 15: Coprocessor number 3 register index used by cop –instruction bits 14 downto 10: Coprocessor number 2 register index used by cop –instruction bits 9 downto 5: Coprocessor number 1 register index used by cop –instruction bits 4 downto 0: Coprocessor number 0 register index used by cop –instruction | |
| 21h | TMR0_CNT | 32 | Current timer value of timer 0 | See chapter about timers. |
| 22h | TMR0_MAX_CNT | 32 | Maximum value of timer 0 | |
| 23h | TMR1_CNT | 32 | Current timer value of timer 1 | |
| 24h | TMR1_MAX_CNT | 32 | Maximum value of timer 1 | |

| | | | | |
|---|---|---|---|---|
| 25h | TMR_CONF | 32 | Common configuration register for timers 0 and 1<br>bits 31 downto 16 : timer 1 configuration bits. bits 15 downto 0 : timer 0<br>configuration bits. | |
| 26h | COP_IF_MODE | 8 | Coprocessor interface configuration. | To be implemented later |
| 27...ffh | **RESERVED FOR FUTURE EXTENSIONS** | | | |

[2] Address range ([CCB_BASE] + 100h) to [CCB_END] is used to access an external configuration block directly. This makes it possible to connect peripherals directly to data cache bus instead of system bus.

[3] Bit index and interrupt source associations:

| bit | source | bit | source | bit | source |
|---|---|---|---|---|---|
| 0 | coprocessor 0 int (exception) | 4 | ext int 0 | 8 | ext int 4 |
| 1 | coprocessor 1 int (exception) | 5 | ext int 1 | 9 | ext int 5 |
| 2 | coprocessor 2 int (exception) | 6 | ext int 2 | 10 | ext int 6 |
| 3 | coprocessor 3 int (exception) | 7 | ext int 3 | 11 | ext int 7 |

[4] Memory protection can be dynamically configured which is convenient in multitasking system. Most secure way is to set the limits always when switching task and to allow one task to access only address space reserved for it(data and instruction memory). If different tasks share global data(dangerous!) address spaces can overlap. In most cases communication between tasks should follow schemes offered by operating system. In simple systems only vital part of the the memory might be protected and the rest of the memory is 'free' to everyone. In both cases it is recommended that CCB is mapped to protected area!

**Register values after reset**

PSR start value is 0000 1110b. SPSR is set to 0000 0009h Other registers in RF and CR are set to zero upon reset.

| RESERVED | IE | IL | RSWR | RSRD | UM |
|---|---|---|---|---|---|
| 7...5 | 4 | 3 | 2 | 1 | 0 |

**Table 4 CCB (internal) register values after reset**

| mnemonic | value after reset | Notes |
|---|---|---|
| CCB_BASE | 0001 0000h | 64KB offset from the 'start'. Depending on the actual memory implementation, data and instruction cache may or may not point to the same physical memory. |
| CCB_END | 0001 00ffh | Must be set if an external configuration block is present. |
| COP0_INT_VEC | 0000 0000h | |
| COP1_INT_VEC | 0000 0000h | |
| COP2_INT_VEC | 0000 0000h | |
| COP3_INT_VEC | 0000 0000h | |
| EXT_INT0_VEC | 0000 0000h | |
| EXT_INT1_VEC | 0000 0000h | |
| EXT_INT2_VEC | 0000 0000h | |
| EXT_INT3_VEC | 0000 0000h | |
| EXT_INT4_VEC | 0000 0000h | |
| EXT_INT5_VEC | 0000 0000h | |
| EXT_INT6_VEC | 0000 0000h | |
| EXT_INT7_VEC | 0000 0000h | |
| INT_MODE_IL | fffh | 32 bit mode for all routines |
| INT_MODE_UM | 000h | Super user mode for all routines |
| INT_MASK | fffh | All interrupts disabled |
| EXT_INT_PRI | 0000 0000h | |
| COP_INT_PRI | 0000h | |
| INT_SERV | 000h | |
| INT_PEND | 000h | |
| EXCEPTION_CS | 00h | |
| EXCEPTINON_PC | 0000 0000h | |
| EXCEPTION_PSR | 00h | |
| DMEM_BOUND_LO | 0000 0000h | All the address space reserved for super user. Cannot run in user mode before configuring these register appropriately. |
| DMEM_BOUND_HI | ffff ffffh | |
| IMEM_BOUND_LO | 0000 0000h | |
| IMEM_BOUND_HI | ffff ffffh | |
| MEM_PCONF | 0000 0003h | |
| SYSTEM_ADDR | 00000000h | |
| EXCEP_ADDR | 00000000h | |
| WAIT_STATES | fffh | Assuming the slowest memories possible. Sixteen clock cycles per memory and cop access. (1 basic cycle + 15 wait cycles) |
| CREG_INDX_I | 0 0000h | cop –instruction accesses register index 0 of the coprocessor. |
| TMR0_CNT | 00000000h | |
| TMR0_MAX_CNT | 00000000h | |
| TMR1_CNT | 00000000h | |
| TMR1_MAX_CNT | 00000000h | |

| | | |
|---|---|---|
| TMR_CONF | 00000000h | |
| COP_IF_MODE | 00000000h | |

## 3.3 Overview of the Pipeline

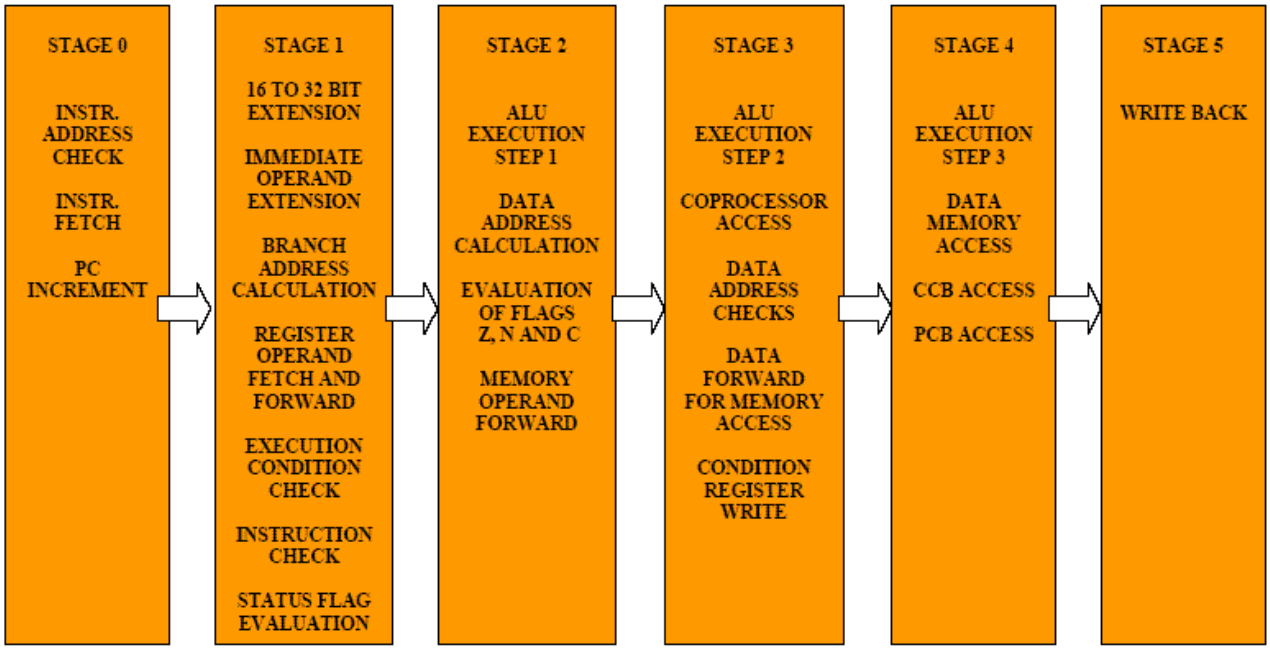


**Figure 2: Pipeline Stages inside the Core[1]**

## 3.4 Interface of the Core

[1] Figure 3 shows an example of interfacing the core. This is not the only possible way to connect to core. Optional peripherals are drawn with dashed line. External interrupt handler, boot agent, PCB (peripheral control block) and the coprocessors are optional. Also the use of bus_req and bus_ack signals is optional. bus_req and bus_ack –signals allow sharing the data bus. Boot agent can be used if the boot address has to be determined externally. PCB is a user defined block to interface peripheral devices directly. It can have, for example, configuration registers mapped to some of the memory addresses. PCB address space is defined by software. Unused inputs should be driven to a state defined in port specification.

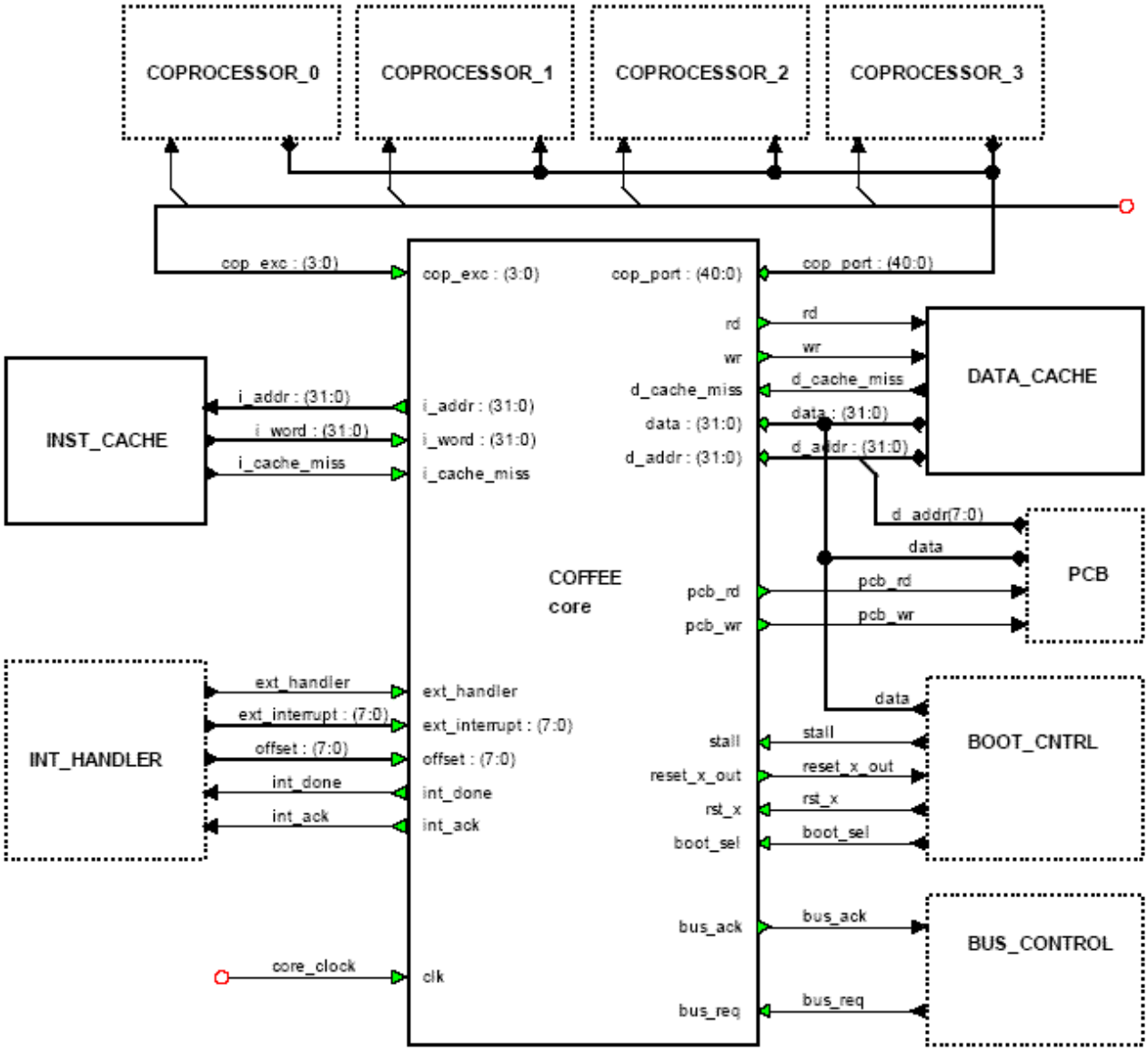


**Figure 3: Interface of the Core[1]**

# 4 Process Management

We more often use the term **process** in Linux documentations instead of **task**, both can be used interchangeably but the same convention of using the term process as in Linux is followed even here. A process is a fundamental concept to any multiprogramming operating system and a process is defined as an instance of program execution. [3] [19]

The **process manager** implements the process abstraction. It covers the following areas:

- Scheduling of processes on the CPU(s).
- Synchronization mechanisms for processes.
- Responsible for dealing with deadlocks among processes.
- Partially responsible for protection and security.

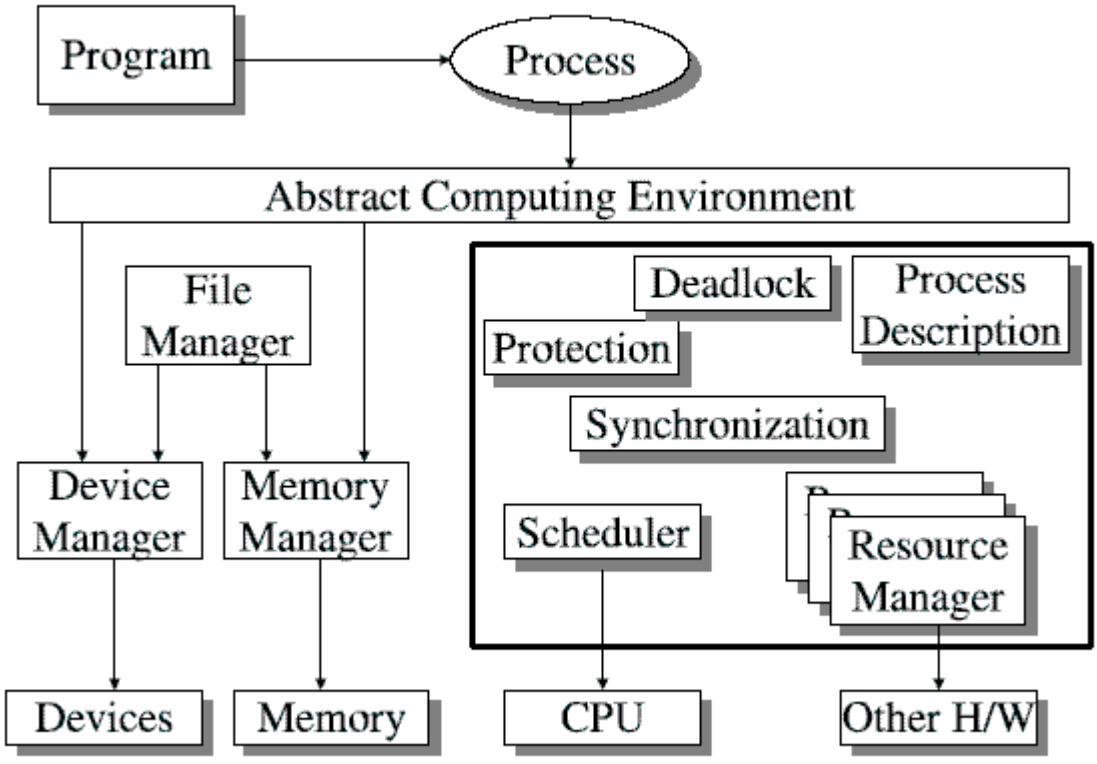


**Figure 4 Process Manager Overview [7]**

## 4.1 Process Structure

This is the point where the author started. All kernel newbie who would like to build the kernel from scratch could start from writing the process structure and adding functions to manipulate the members of the structure. *This point is very important as it took me a long time to the point where to start as there is a lot of dependency of one function on the other* [3], so this point would be suggested by the author. Note that the functions which operate on the variables like next_task, prev_task. next_run, prev_run etc inside a structure which are independent of other functions except the functions to which they are passed as parameters (pass by reference) i.e., the operation on these variables are clean and readable even to a newbie. One can start understanding code from here as it is just a matter to understand how complex algorithms are operated on these variables. For the kernel to manipulate with the processes, it should have an clear idea of what it is doing, what it did, what resources it owns, how many more resources it is requesting, the memory it is using and everything related each process. All these information is stored in a structure named **task_struct** which is the process structure or most often called **process descriptor**. It is like a report card of a student (task) which has all the information that the teacher (kernel) would need to know to take further actions on that student (task). Not only does this structure have many fields inside it but also it has some fields that are pointers that point to some similar structures that in turn points to some other structures. We can imagine this as a group of students who has a report card (say its number be 1203) and one field in this report card would point to the report card number (say 4112) of the student who got the next immediate rank. Similarly that report card could have a pointer that also points to the report card number of the student who got the previous immediate rank (should be to the number 1203 if no ranks are repeated.

Task_struct is like a report card to the kernel which sees into this report card and could figure out what each of the task was doing, how much time it was using the resources of the system and everything possible related to that task. So if the kernel just deletes a task_struct of one task then it will no further be able to know what it did, doing and going to do and hence will not be able to provide it with any resources and is equivalent to deleting the task from the kernel's knowledge.

Figure 5 describes the Linux process descriptor schematically.

The task_struct field of the port is given below:

```
struct task_struct{
        int state;
        int pid;
        int priority;
        struct task_struct *prev_task,*next_task;
        struct task_struct *prev_run, *next_run;
        long counter;
        unsigned long timeout;
        struct pt_regs *regs;
        struct mm_struct *mm;
        struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
        struct wait_queue *wait_chldexit;                /* for wait4() */
        unsigned long policy,rt_priority;
        int exit_code, exit_signal;
        unsigned long signal;
       /* signal handlers */
       struct signal_struct *sig;
       struct timer_list timer;
       int syscall_called_timer; //  If equals to 20 then delete this task
                                 //  from runqueue
};
```

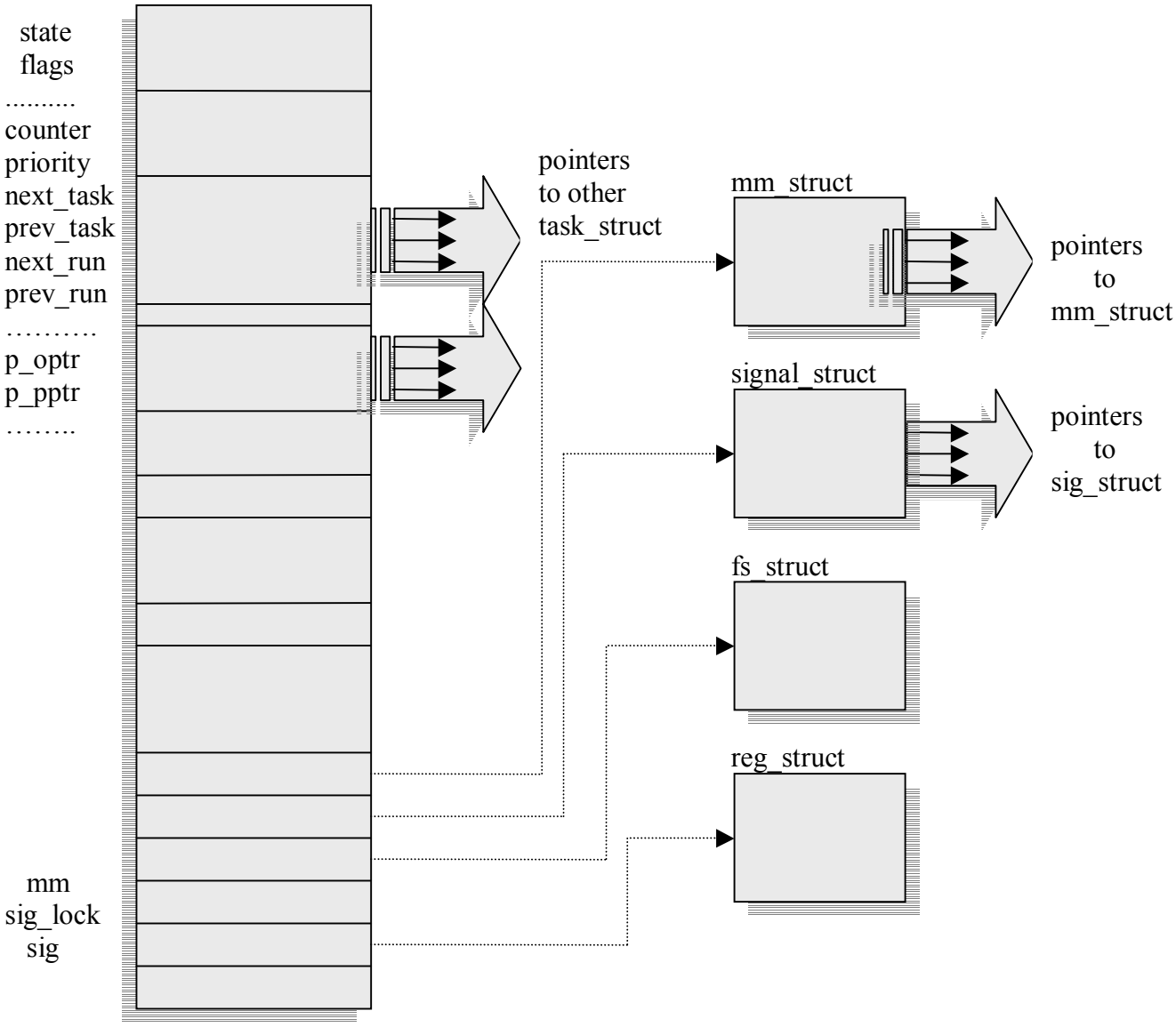


**Figure 5 Structure of the Process descriptor**

## 4.2 Process States [3]

A Process can be in any one of the six states that will be described in this section. As its name implies, the state field of the process descriptor describes what is currently happening to the process. It consists of an array of flags, each of which describes a possible process state.[3] The following are the possible process states:

**TASK_RUNNING**

The process is either executing on the CPU or waiting in some queue to get a chance to be executed.

**TASK_INTERRUPTIBLE**

The process is suspended (sleeping) until some condition becomes true. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of conditions that might wake up the process, that is, put its state back to TASK_RUNNING. This state is not yet used in this port.

**TASK_UNINTERRUPTIBLE**

Like the previous state, except that delivering a signal to the sleeping process leaves its state unchanged. This process state is seldom used. It is valuable, however, under certain specific conditions in which a process must wait until a given event occurs without being interrupted. For instance, this state may be used when a process opens a device file and the corresponding device driver starts probing for a corresponding hardware device. The device driver must not be interrupted until the probing is complete, or the hardware device could be left in an unpredictable state. This state Is not yet used in this port.

**TASK_STOPPED**

Process execution has been stopped: the process enters this state after receiving a signal that means to stop the execution of the specified task. When a process is being monitored by another (such as when a debugger executes a ptrace( ) system call to monitor a test program), any signal may put the process in the TASK_STOPPED state.

**TASK_ZOMBIE**

Process execution is terminated, but the parent process has not yet issued a wait( )- like system call (wait( ), wait3( ), wait4( ), or waitpid( )) to return information about the dead process. Before the wait( )-like call is issued, the kernel cannot discard the data contained in the dead process descriptor because the parent. This state is yet not being used int his port.

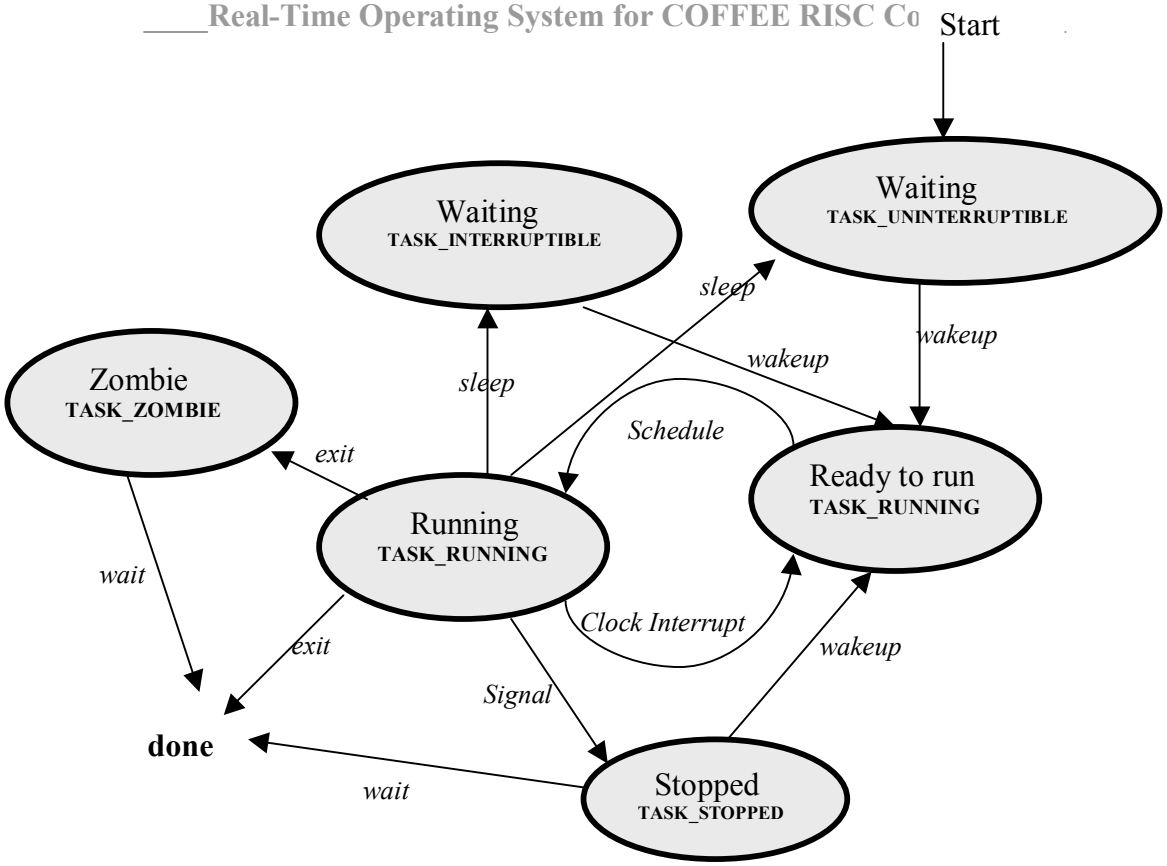


**Figure 6: State diagram showing all the possible state transitions. [6]**

## 4.3 Process Identification PID

All the Processes are given a unique Identification using PID field in their task_struct. When a process is created its PID is assigned a value of incremented value of "Global_Pid" which is an unsigned integer. It can be seen that Global_Pid would overflow after thousands of processes are forked/cloned/created and it is assumed that this will happen after a very long time at least in the embedded OS.

## 4.4 The Task Array

Processes are dynamic entities whose lifetimes in the system range from a few milliseconds to months or even some years but it is note worthy that some of the parameters in the kernel overflow after some long time like the variable JIFFIES which has a count of number of time ticks from last system boot. Thus, the kernel must be able to handle many processes at the same time. The kernel could handle NR_TASKS (defined in /kernel/sched.h file at the time of this writing) processes whose value should be kept appropriate depending on the memory usage of each task and the processor speed. At the time of writing this document the processor speed was about 50 MHz and the static RAM was 16MB, so NR_TASKS was chosen to be 10 (very optimum), depending on the application and use, developers are encouraged to change its value if they knew the effects of

changing it. The kernel reserves a global static array of size NR_TASKS called task in its own address space. The elements in the array are process descriptor pointers; a null pointer indicates that a process descriptor hasn't been associated with the array entry.

## 4.5 The current macro

Current is a pointer that holds the address of the process descriptor that is currently scheduled to run on the processor. The variable current is stored at a fixed memory inside the kernel data area and it is more often necessary for the kernel to get the address of the process descriptor stored in the current pointer, it is done using the macro “get_current (Rx)” which makes the address to be stored in the specified register Rx. The assembly instruction will be just like the following.

```
__asm__ (".macro get_current (reg)\n\t"
         "push r16\n\t"
         "ldra r16, current\n\t"
         "ld   reg,r16,0\n\t"
         "pop  r16\n\t"
         ".endm\n\t");
```

## 4.6 Linked Lists of the processes

As previously described in the teacher-student example, there are pointers in the process descriptors which point to other process descriptor based on some criteria. Two of those criteria in Linux are pointers pointing to the next/previous task on the run queue and pointers pointing to the next/previous task on (say) wait_queue. When you look at the C-language declaration of the task_struct structure, the descriptors may seem to turn in on themselves in a complicated recursive manner. However, the concept is no more complicated than any list, which is a data structure containing a pointer to the next instance of itself.

### 4.6.1. The double linked list of processes in the wait queue

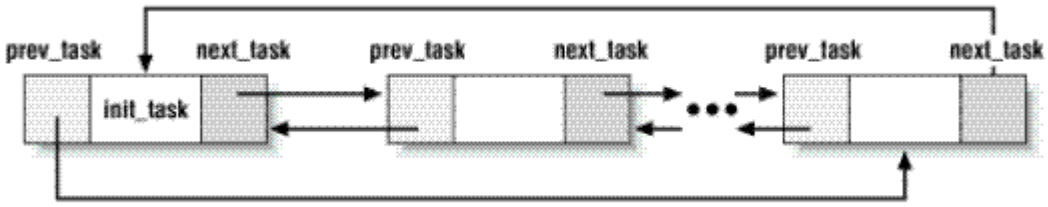


**Figure 7: Connected processes in the queue [3]**

The descriptor of the task had to be pointed by at least tasks in runqueue or waitqueue or be its parent/child/sibling or anyone so that the address of its descriptor is lost. If that happens then the kernel will be unable to update its action and eventually it will not allow this task to do anything cutting the access to the processor. So at an instance if the process has some thing left to do on the processor then its descriptor will be pointed by a pointer in at least one descriptor of some other process.

There is a macro in /kernel/sched.h which is as follows:

```
#define for_each_task(p) \
        for (p = &init_task ; (p = p->next_run) != &init_task ; )
```

This macro can be used to scan all the processes which are on the run queue; this is a bit different to that on Linux, where a similar macro is used to scan all the processes on all queues i.e. all the available processes on system. The change was made as the scan to all the available processes in the system was not used for the limited functionality the port has at the time of this writing. The future works will probably again use the macro that was used in Linux versions.

### 4.6.2. The double linked list of TASK_RUNNING processes

The double linked of the task_running processes points to the process descriptors whose STATE is TASK_RUNNING and are waiting to get a chance to be executed on the processor. The process descriptor contains the fields' next_run and prev_run exactly for these purposes. The below figure is shows the processes that are on the runqueue and the way they are pointing to each other. Note that init_task (the process descriptor of an idle task) will be the head of all queues.

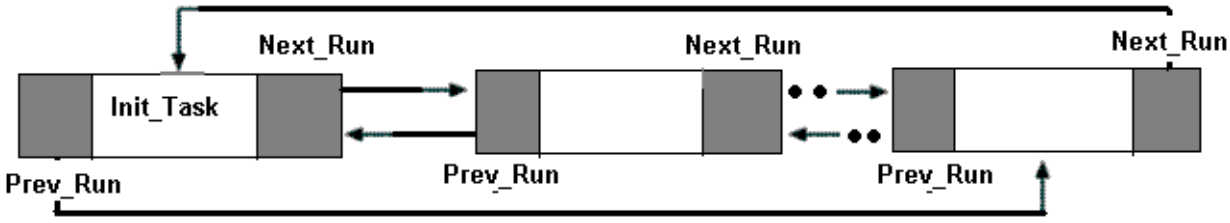


**Figure 8: Connected processes in the run queue**

The **nr_running** variable stores the total number of runnable processes. There are functions like add_to_runqueue() which adds the process descriptor on the run queue which means that the last

process pointer will point its next_run to this process descriptor and the process' prev_run will be pointing to the process which was last in the queue before this process was added. The next_run of the last process descriptor will always point to INIT_TASK and this will become a circular double linked list. Similarly del_from_runqueue would remove the process from the run queue and adjust its neighbouring process pointers such that removing the process will not break the queue flow, move_last_runqueue() and move_first_runqueue(), are provided to move the process descriptor to the last and front of the runqueue.

wake_up_process() will make a sleeping process again runnable, which in turn invokes add_to_runqueue() which adds the awaken process again to the end of the runqueue incrementing the nr_running variable. The functionality of the del_from_runqueue is exactly the reverse which decrements nr_running after removing the process descriptor from the runqueue. Note that del_from_runqueue() can delete the process descriptor from anywhere in the runqueue but the add_to_runqueue() can add the process descriptor only to the end of the queue. Again the scheduler is invoked it may change the sequence or they can be wontedly moved to the place the kernel wish using the above mentioned macros.

## 4.7 Process Switching

In order to control the execution of processes, the kernel must be able to suspend the execution of the process running on the CPU and resume the execution of some other process previously suspended. This activity is called *process switching, task switching*, or *context switching*. The following sections describe the elements of process switching in Linux:

- Hardware context
- Hardware support
- Linux code
- Saving the Stack Pointers

### 4.7.1 Hardware Context

While each process can have its own address space, all processes have to share the CPU registers. So before resuming the execution of a process, the kernel must ensure that each such register is loaded with the value it had when the process was suspended. The set of data that must be loaded into the registers before the process resumes its execution on the CPU is called the *hardware context*. The hardware context is a subset of the process execution context, which includes all information needed for the process execution.

Here in this section detailed description of how this hardware context is carried out is given.
It is evident that in Linux context switch can only occur after the system call i.e. scheduler is called only in System Call handler and not anywhere. This is because of the fact that system call is a sort of least priority interface to the kernel for the processes, which means that all the interrupts and exceptions for this process (if there were any) should have been already handled before coming back to scall(system call) handler.

To get a better view of why this is done let us consider a situation where *current* is the process that is now under execution and made a system call. There will be a switch to kernel routines to handle to request and then the requested system call routines are now executing. At this stage a timer interrupt arrives and because interrupts have more priority then *scall* in COFFEE™ RISC core the processor immediately starts executing the interrupts handler. More details on how interrupt handler is started and the hardware support for them is given in *chapter 5*. If the interrupt handler has the capability of calling the scheduler and the scheduler makes another process as *current* process to be next run on processor, then after servicing the interrupt the control will return to the system call handler which will be giving services to the new *current* process that was the result of calling scheduler in interrupt handling routines which is absurd. The below figure illustrates the above mentioned scenario.

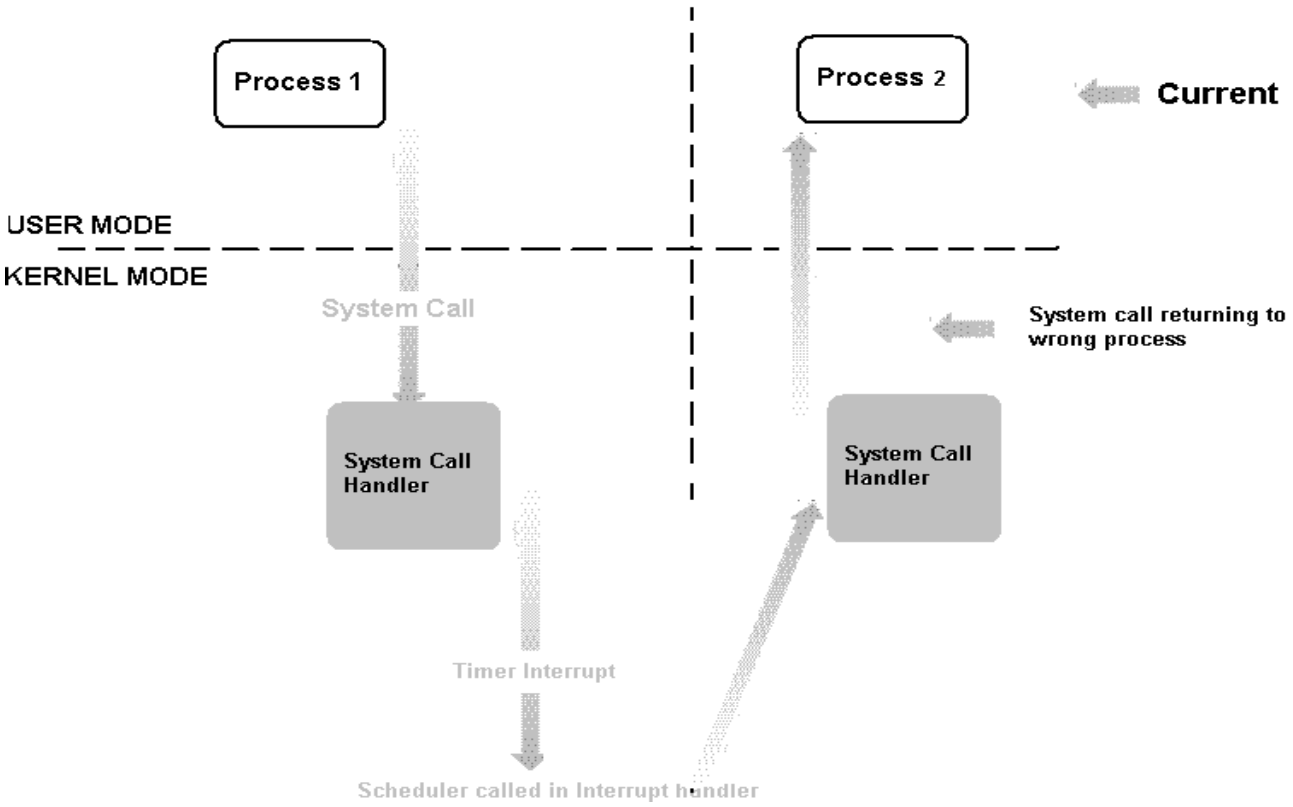


**Figure 9: Effects of making the interrupt or exception handler capable of calling scheduler**

All the contents of the User Mode registers along with some Superuser Mode registers (stack and frame pointer, PSR, SPSR and control registers) are saved only in *scall handler* which is the only situation where the process has no guarantee that it will be resumed. Unlike in the case of interrupts and exceptions, where in interrupts the execution of the process returns to the same process (Guaranteed) and after the exception handling the process may never resume, in both cases there is no need to save the user registers. The user registers are saved using the macro *SAVE_USR_PT_REGS* which is a C macro in the file */kernel/entry.h.*

In the figure illustrated below it is seen how a copy of all registers of one process is taken into the struct pt_regs, which is sufficient information for the processor to resume the process when selected by the scheduler from the same instance where it was suspended. The figure has been divided into two parts to illustrate that a time epoch has passed as we passed from one side to another. Process 1 made a system call that will make processor to run kernel routines, where the first thing to do is to save the user registers into the kernel data structures as there is no guarantee that this process be resumed, then system call validity is made and the appropriate system call is executed. After the system call is done then the scall handler checks if the process needs rescheduling make another process as *current process*. When the *scall handler* returns it resumes all the *current process'* saved registers into the User Register Bank (set 1 registers) along with some registers in the Super User Set that are necessary for the processor and the kernel to know about the stack and processor status when this *current* process was previously suspended. The prev local variable refers to the process descriptor of the process being switched out and next refers to the one being switched in to replace it. We can thus define *process switching* as the activity consisting of saving the hardware context of prev and replacing it with the hardware context of next. Since process switches occur quite often, it is important to minimize the time spent in saving and loading hardware contexts.

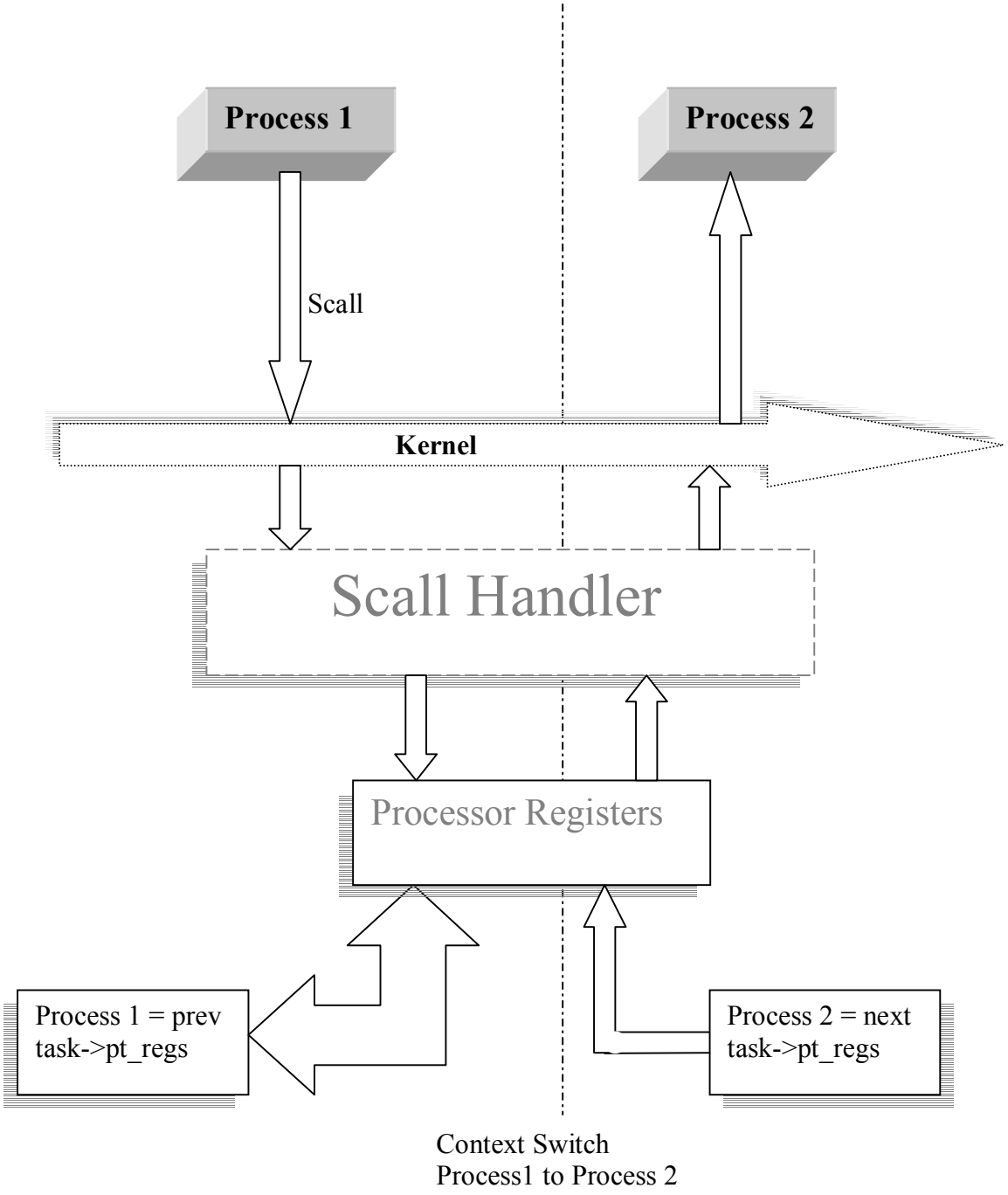


**Figure 10 Saving and Restoring User registers.**

### 4.7.2 Saving and Restoring the SP and FP

The kernel and each of the processes have their own space in the memory. When a process request a service from the kernel, the first thing is that the kernel checks for the system call validity and then servicing is done in the kernel space. This is done by taking a backup of the FP (register 28) and SP

(register 29) and replacing those registers with the kernel FP and kernel SP which are stored in the variable KERNEL_SP and KERNEL_FP. The below figure illustrates how the two registers are used as Stack Pointer and Frame Pointer by the compiler and because the kernel is partly written in C so the kernel follows the compiler conventions.

Even though the figure is a bit confusing it is quite useful to explain the operation of saving and restoring address space of kernel and processes. When the processor restarts first the kernel runs in its address space in the area darkened in the memory. Once the kernel did all initializing stuff it dumps the values of FP and SP into KERNEL_FP and KERNEL_SP and then puts the value of FP and SP of the *current* process from struct *pt_regs* and restores them in the respective registers (r28 and r29). When this process is interrupted then these two registers along with some other registers that the kernel believes will get corrupted back into the *pt_regs* struct.

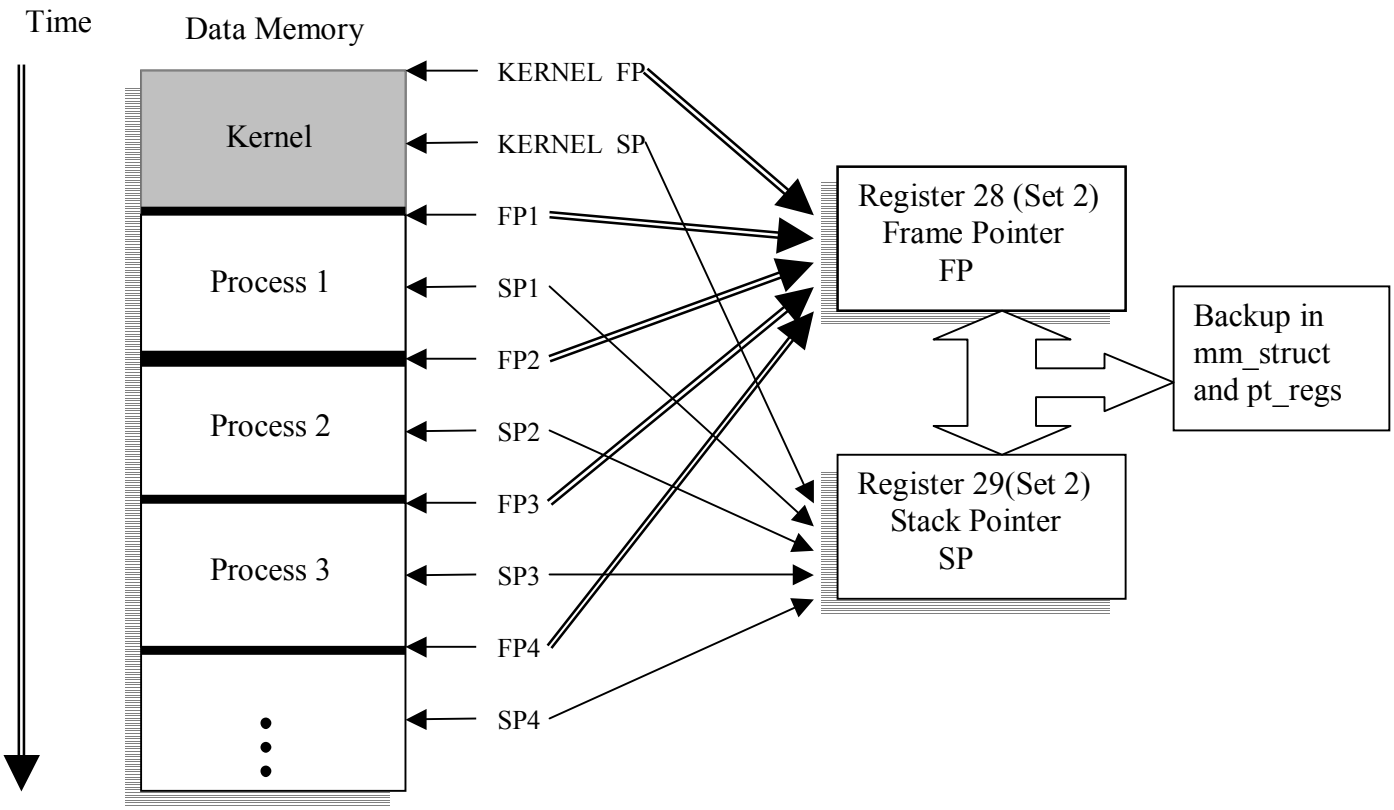


**Figure 11 Illustrating how stack space are restore for each process and kernel**

```
1. current->syscall_called_timer = 0;                    //time units since syscall called
2. servicing_syscall = 1;                                //switch from task to kernel is true
3. asm ("disable_interrupts\n\t");
4.   SAVE_USR_PT_REGS
5.   SAVE_STACK_POINTER
6.   MOVE_INCOMING_ARGUMENTS                            //incoming arugments from set1 to set2
```

The above is a part of the code taken from the system_call( ). Line 5 shows when the stack pointers are saved which is a simple *#define* written in the file */kernel/entry.h* at the time of this writing.

## 4.8 Creating Processes

Creating a process here in this version of the OS is a bit tedious. Every time a new task is to be added to the processes list, then the file */kernel/main.c* is to be changed a bit. Information of the task number and the approximate amount of stack memory it needs is given so that the kernel reserves this space for this. Because there is no concept of pages here and because of the absence of MMU in COFFEE RISC Core ™ processor dynamic memory is still not implemented. In future Core will have a MMU and the port will then be modified to allow dynamic memory allocation. So the point is that whenever a process is added the kernel should be given knowledge of the task's static memory requirements in the file */kernel/main.c*. Also all the structures related to this new process are build manually in this file as kernel is incapable of building this structures because of the only reason that it can only provide static memory in runtime and structures in the processes own static space is liable of corrupting then. Kernel depends a lot on these structures and cannot risk corrupting them. All the System Calls related to creating a process like clone( ), fork( ), and vfork( ) will be added in the future works.

**Process 0**

The ancestor of all processes, called *process 0* or, for historical reasons, the *swapper process* or *mother process* or *idle task* is in real a process that does nothing. But this process being doing nothing is here utilized in many ways. It is the head for all the processes, head for all the queues and the only process executing when no other process is on the run queue. This Idle
This ancestor process makes use of the following data structures:

- A process descriptor and a Kernel Mode stack stored in the init_task variable. The init_task and init_stack macros yield the addresses of the process descriptor and the stack, respectively. It uses the following structures
  - init_ptregs
  - init_task
  - init_mm
  - init_signals

The INIT_TASK macro uses the above mention data structures and is the process descriptor for the init_task. This process descriptor is useful to get the address of any processor descriptor as it is somehow linked to the init_task and one can propagate through all the processes starting from INIT. A brief description of the init_task is described below.

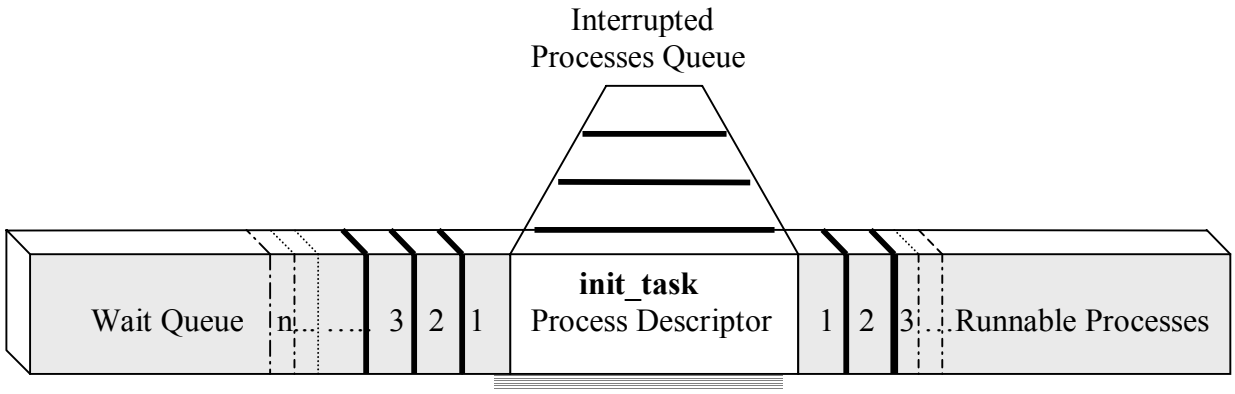


**Figure 12: 3-dimension illustration of init_task primary functionality.**

It is already mentioned that init_task is the head of all the queues and lists in the kernel. It also has a functionality of running its own idle process (does nothing but loops itself) when no other processes have anything to do in the run queue. The main problem in this was that init_task runs inside the kernel and is having super user privileges and cannot be interrupted. This problem was solved by making the init_task pointing to the function idle_brain() which calls a dummy system call sys_dummy_idle_brain() which does nothing but just enable need_resched = 1, so that at the end of

the system call, scheduler is invoked to check if there are any changes in the queues that make processes other than init_task RUNNABLE. The other way to say this is scheduler is forcibly invoked by idle_task/init_task be calling a dummy system call which enables a variable which is checked and the time of returning from the system call and a decision is made whether to call scheduler, as it is important to invoke scheduler regularly when init_task/idle_brain() is running as it does nothing productive.

## 4.9 Process Termination/Removal

This was the simplest thing that could be done to a process in this port. As there are no parent processes for a process as fork(), clone() etc are not yet ported, there will be neither be any parent nor any child for the current process which depend on the current process' execution. So just by making the state of the process as TASK_STOPPED and just deleting it from the run queue, all the information regarding this process will be lost and the kernel have no track of it and so this process can never be resumed. In the Linux, the scenario is totally different, when a process has to terminate then the kernel has to take care of snatching all the resources the process own and also all the memory it was using including the *task structures* had to be freed so that the memory used for those structures will be again useful for other processes. But here in this Core the only resources at the time of this writing were CPU (or better to say Processor) and *data memory*, and because no new processes are created at runtime and also sufficient memory is already allocated to each process before they start, no memory is recovered. But later versions would have the capability of freeing the dynamic memory. A process is forcibly removed when it does not execute a system call for some long time (adjusted to 2000 timer ticks then the process no more gets the processor time on the assumption that calling no system call for that long time means no productive work is bein done by that process or more specifically it is assumed that it got stuck in a loop) or when it creates an exception.

# 5. MEMORY MANAGEMENT

Management of memory on this port is quite simple as there are no pages, no swapping of memory and no automatic allocation of extra memory if the process needs more. First the memory model which was considered is explained in this chapter.

## 5.1 Data Memory Model

The data memory is divided into parts so that each process has its own stack space. It is known that the kernel also has its own stack space and its memory space should be protected from the user applications (unprivileged software). The memory model

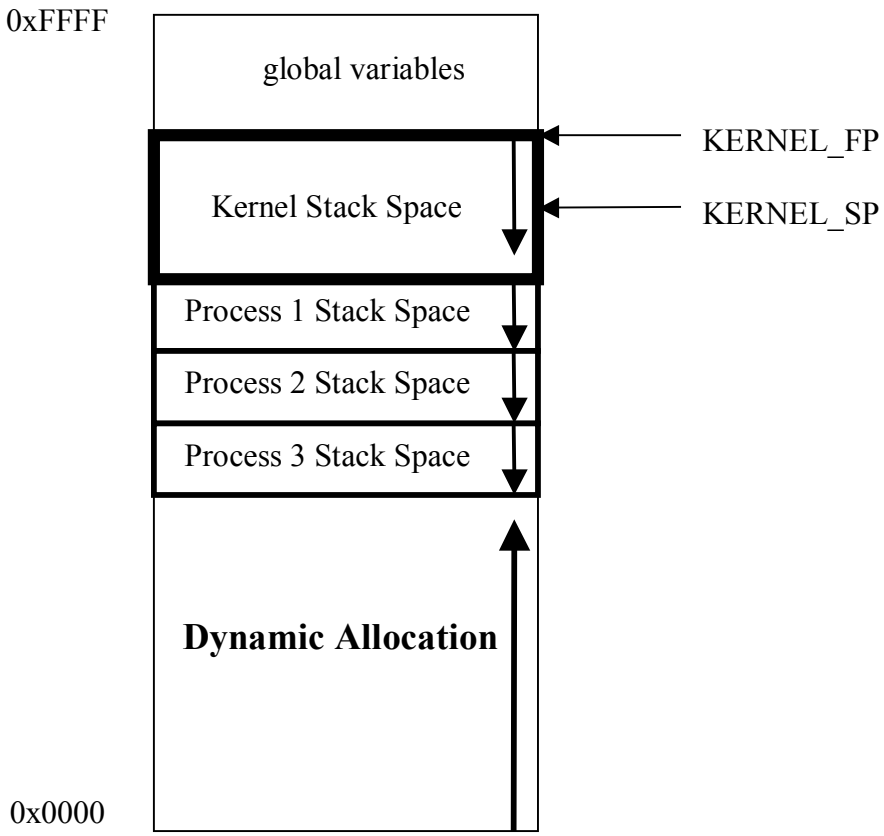


**Figure 13: Data Memory Model**

## 5.2 Stack Memory

Each process has its own stack space and it is supposed to use the memory space allocated to it. All the running processes are trusted not to point to the address space that is owned by some other process. As these processes represent embedded applications, they are trusted not to corrupt any other process data by pointing to their memory space. It is seen that the security is a bit compromised but it is not an issue in the embedded environment, as all the applications are written by knowing this limitation of the kernel. Ideas of implementing the fence registers are in mind, also this writing are based on the very first version of the kernel. Later in the future works they will be soon implemented, i.e. these security issues will be raised and the compromises will be removed.

The process structures (struct task_struct xyz) are stored in the stack where the global variables are placed by the linker. This structure will consume only *116 bytes* of memory and the structure that stores all the register values (struct pt_regs pqr) consumes *144 bytes*. Process structure has all the information related to a process except that it doesn't know the register values of the process, but a pointer to a structure which holds the latest values of the registers or the registers that are linked to the current process (like PSR, SPSR and conditional registers) hold in this structure. When the current task relinquishes or sleeps then there is a context switch and the kernel saves all the registers of the process so that it can be resumed again into this structure.

# 6 Kernel Entry and Exit

The kernel has some specific entry and exit points from where it comes into picture and has control over the execution flow of the processes. These points are mentioned and explained deeply in this chapter, along with the services provided and handled by the kernel.

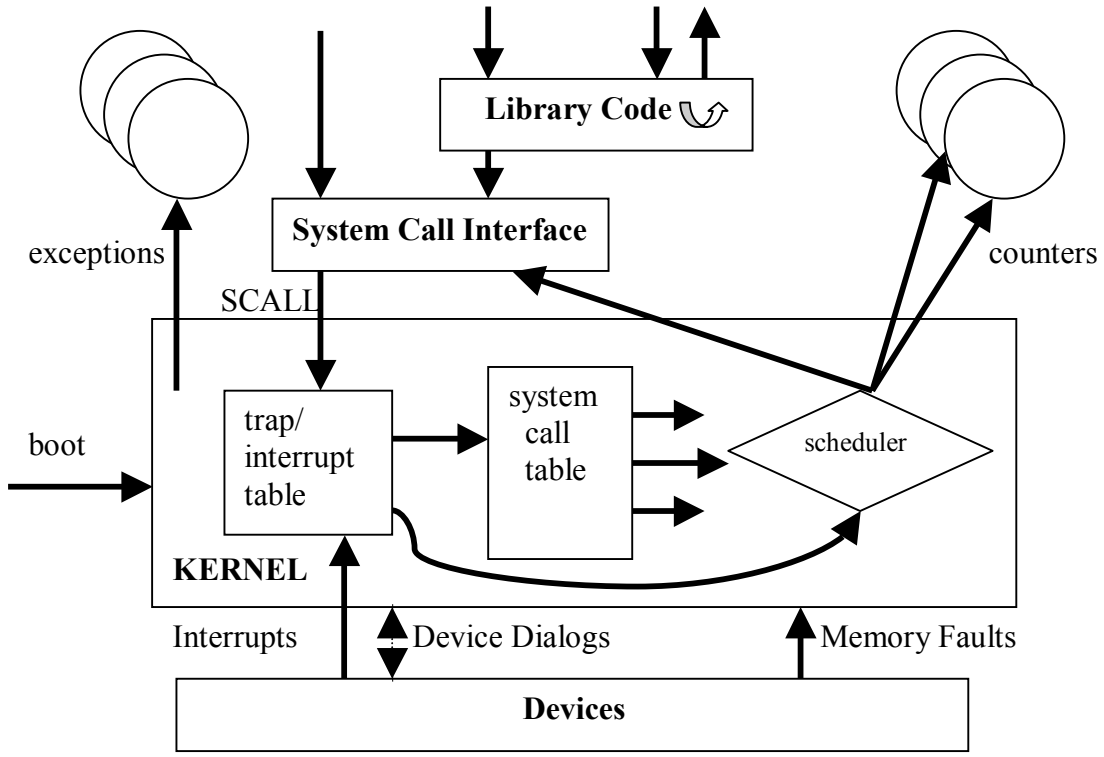


**Figure 14: illustrating Kernel Entry and Exit points. [11]**

## 6.1 Interrupts

An Interrupt is defined as an event that alters the sequence of instructions executed on the processor. Such events correspond to electrical signals generated by hardware circuits both inside and outside of the CPU chip. [1]

Interrupts are often divided into *synchronous* and *asynchronous* interrupts:

• *Synchronous* interrupts are produced by the processor control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution

of an instruction. The timer interrupt that generates the interrupt at the known times is an example for synchronous interrupts.

• *Asynchronous* interrupts are generated by other hardware devices at arbitrary times with respect to the processor clock signals.

COFFEE RISC Core™ currently supports connecting eight external interrupt sources directly. If coprocessors are not connected the four inputs reserved for coprocessor exception signalling can be used as interrupt request lines giving possibility to connect twelve sources. An external interrupt handler can be connected to expand the number of sources even further.

*If internal interrupt handler is used, the priorities between sources can be set by software, with external handler; priorities will be fixed* according to table below. Note that priorities for coprocessor exceptions/interrupts are always set by software. Internal exception handler has synchronization circuitry allowing signals to be directly connected to the core. If an external handler is used, synchronization is bypassed in order to reduce signalling latency. See interface document. Status signals are provided to give feedback about the status of the latest interrupt request. Interrupt sources can be masked individually and disabled or enabled all at once using di and ei –instructions. All interrupts are vectored. The address of an interrupt service routine can be the corresponding vector directly (see interrupt registers) or a combination of the vector and an offset given externally.

**Table 5, Interrupt priorities if external handler is used, 0 - highest.**

| Priority | Name |
|---|---|
| software controlled | coprocessor number 0 exception/interrupt |
| | coprocessor number 1 exception/interrupt |
| | coprocessor number 2 exception/interrupt |
| | coprocessor number 3 exception/interrupt |
| 15 | external interrupt 0 |
| 15 | external interrupt 1 |
| 15 | external interrupt 2 |
| 15 | external interrupt 3 |
| 15 | external interrupt 4 |
| 15 | external interrupt 5 |
| 15 | external interrupt 6 |
| 15 | external interrupt 7 |

### 6.1.1 Interrupt interface modes

Two interfacing modes are supported: *external handler* and *internal handler* The mode is selected by EXT_HANDLER –signal. A summary is given in the table below. Note that an external handler usually allows priorities between sources to be set quite freely. In this case an external handler sees a fixed priority between the lines it is driving. The user may see whatever configuration.

**Table 6, Interrupt interface modes**

| mode/ EXT_HANDLER state | request signal timing | interrupt vector calculation | priorities |
|---|---|---|---|
| internal handler / LOW | asynchronous | BASE address directly 1 | set by software (see configuration registers) |
| external handler / HIGH | synchronous | BASE(31 downto 12) & OFFSET & “0000” 2 | fixed between lines (usually configurable via external handler) |

1 BASE address is set by software. See CCB configuration registers.
2 8 bit OFFSET provided by an external handler, & means concatenation. Coprocessor exceptions/interrupts do not use OFFSET.

### 6.1.2 Signalling an interrupt

An interrupt request is signalled by driving a high pulse on one of the interrupt lines. The timing of the pulse depends on the mode: whether an external handler is used or not. The timing of the coprocessor interrupt/exception lines is fixed. Each interrupt line has a pulse detection circuitry and an interrupt request gets through when that circuitry sees a pulse, that is, after seeing a falling edge. If an external handler is used, the offset should be driven simultaneously with the request line, see interface –document.

Once detected, a request is saved in a register called INT_PEND, which is visible to the software. After this it has to go through the priority resolving and masking stage. The following conditions have to be true for a pending request to get through: - interrupts enabled: IE –bit in processor status register (PSR) must be high.

- Interrupt mask register has to have a high bit (‘1’) in the corresponding position.
- No interrupts with higher priority are pending or in service.
- No exceptions on pipeline (see document about exceptions)

Once a request gets through, the processor starts execution of an interrupt service routine as soon as possible: pipeline is executed to a point where it is safe to switch to interrupt service routine. This takes 1 – 3 cycles depending on the contents of the pipeline. When a service routine is started the corresponding bit in INT_SERV –register is set. At the same time, the processor drives a pulse to INT_ACK –output in order to signal to an external handler that the latest request got through and is now in service. *This is the earliest point where a new request from the same source can be accepted*.

### 6.1.3 Priority resolving

A priority for a particular source is set by writing a four bit value in a field reserved for that source in the EXT_INT_PRI or COP_INT_PRI –register. Priority can have any value between 0 and 15, zero being the highest priority. Whether the priority is fixed (external handler used) or set by software, priority resolving works the same way. If multiple interrupts are signalled simultaneously, the one with the highest priority (lowest number) will be served first. Note that for coprocessor exceptions/interrupts the priority can always be set by software. If multiple sources have the same priority, resolving is performed internally in the following order (COP0_INT having the highest priority): COP0_INT, COP1_INT, COP2_INT, COP3_INT, EXT_INT0, EXT_INT1, EXT_INT2, EXT_INT3, EXT_INT4, EXT_INT5, EXT_INT6, EXT_INT7. If the same interrupt that is currently in service, is signalled, the interrupt service routine is restarted as soon as it has finished (of course assuming there's no interrupt requests with higher priority pending). A request with higher priority can interrupt the current service routine if interrupts have been re-enabled with *ei* – instruction (nesting of interrupts).

### 6.1.4 Switching to an interrupt service routine

The following steps are taken when switching to an interrupt service routine:

- return address is saved to hardware stack (a special logic structure to allow fast switching)
- Processor status register (PSR) is saved to hardware stack - condition register CR0 is saved to hardware stack.
- The start address of an interrupt service routine is calculated (see table 2) and placed to program counter.
- Signal INT_ACK is pulsed (***except with coprocessor exceptions/interrupts!***).

- The bit corresponding to the interrupt source is set high in INT_SERV –register.
- The bit corresponding to the interrupt source is cleared from INT_PEND – register.
- Further interrupts are disabled by setting IE bit low in PSR
- Processor status: user mode and, instruction decoding are set according to control registers INT_MODE_IL and INT_MODE_UM. (If super user –mode is set, register set 2 is selected as default for reading and writing)
- Execution of the interrupt service routine in question is started.

### 6.1.5 Interrupt Service Routine

As the interrupt service routines has higher priority than the system call handlers and it is possible that an interrupt occurs while the kernel was servicing a system call (one can disable interrupts while in kernel but note that kernel could support even interrupts inside kernel routines. More extensive testing is needed) and corrupts the register bank 2 (or Set 2 registers or super user register) the very first thing does in the interrupt service routines is to save all this registers using the macro SAVE_ALL.

Then enquiry is made if there is nested execution of exception and/or interrupt handlers i.e., whether the kernel was servicing a system call request by some process or executing some lower priority interrupts or was executing the user process. That is to know if the execution was in kernel mode or user mode before the interrupt occurred. The following things will be done in the different situations just mentioned above

1. If the kernel was in user mode before the interrupt then there is need to change the stack pointer and the frame pointer to point to the kernel reserved static memory.
2. If it was previously in the kernel mode servicing some system call or servicing an interrupt then there is no need to change the stack pointers and it can continue using the same space for handling the interrupt, but the prev stack pointers will be saved in the variables KERNEL_FP and KERNEL_SP and the previous values of these variables will be pushed to the stack.

The addresses for this interrupts are registered into the *core control block* (CCB) registers while kernel start-up and is done in the file */kernel/boot.x*. The processor automatically starts executing the instructions starting from the address given in the CCB registers related to that interrupt. Only Timer 0 interrupt was enabled at the time of this writing along with the Coprocessor interrupts.

Timer 1 can be enabled very easily in the *boot.x* file and the same interrupt handler can be used for both the timer interrupts or a different handler can be used, in that case the handlers starting address have to be registered (stored) in the CCB appropriate registers. Timer interrupts are used for timing measurements inside the kernel and are discussed in the next chapter.

Notes:

- The interrupts are disabled using the macro *block_interrupts* and are restored to the same state which was just before blocking the interrupts. Restoring is done using the macro *restore_interrupts* both written in the file */kernel/all_macros.h*
- Block_interrupts stores the current value of INT_MASK in the CCB registers into a global memory location and then disables the interrupts. There are three different storage locations used by the block_interrupts in three different cases, i.e. in Interrupt handler, Exception handler and elsewhere in the kernel.
- Similarly restore_interrupts uses the INT_MASK value stored in the memory and restores it into CCB registers once executing in the critical region is done. These two macros are used instead of the instructions *di* and *ei*.

### 6.1.6 Returning from an interrupt service routine

An interrupt service routine has to execute a ***reti*** –instruction in order to resume program execution where it was interrupted. This causes the following things to happen:

- Processor status is restored from the hardware stack.
- CR0 is restored from the hardware stack.
- Program counter is restored from the hardware stack.
- signal INT_DONE is pulsed (***except with coprocessor exceptions/interrupts!***).
- The INT_SERV bit is cleared.
- Interrupts are enabled if they were enabled before entering the service routine. (There is a possibility that di –instruction is executed just before entering the service routine, but after a request got through in which case the interrupt is served but interrupts will be disabled on return)

### 6.1.7 Internal interrupt handler control & status registers

Bit positions and interrupt sources are associated as follows:
(INT_MODE_IL, INT_MODE_UM, INT_MASK, INT_SERV, INT_PEND)

Bit 11 – EXT_INT7,

Bit 10 – EXT_INT6,

...

Bit 4 – EXT_INT7,

Bit 3 – COP3_INT,

...

Bit 0 – COP0_INT.

**Table 7, Internal interrupt handler registers (in CCB)**

| offset | mnemonic | width | description | notes |
|---|---|---|---|---|
| 02h | COP0_INT_VEC | 32 | Co-processor 0 interrupt service routine start address. | should be properly aligned. |
| 03h | COP1_INT_VEC | 32 | Co-processor 1 interrupt service routine start address. | |
| 04h | COP2_INT_VEC | 32 | Co-processor 2 interrupt service routine start address. | |
| 05h | COP3_INT_VEC | 32 | Co-processor 3 interrupt service routine start address. | |
| 06h | EXT_INT0_VEC | 32 | External interrupt 0 service routine base address. | |
| 07h | EXT_INT1_VEC | 32 | External interrupt 1 service routine base address. | |
| 08h | EXT_INT2_VEC | 32 | External interrupt 2 service routine base address. | |
| 09h | EXT_INT3_VEC | 32 | External interrupt 3 service routine base address. | |
| 0ah | EXT_INT4_VEC | 32 | External interrupt 4 service routine base address. | |
| 0bh | EXT_INT5_VEC | 32 | External interrupt 5 service routine base address. | |
| 0ch | EXT_INT6_VEC | 32 | External interrupt 6 service routine base address. | |
| 0dh | EXT_INT7_VEC | 32 | External interrupt 7 service routine base address. | |
| 0eh | INT_MODE_IL | 12 | Instruction decoding mode flags for interrupt routines. | See registers – document: PSR |
| 0fh | INT_MODE_UM | 12 | User mode flags for interrupt routines. | |
| 10h | INT_MASK | 12 | Register for masking external and cop interrupts individually. A low bit (‘0’) means blocking an interrupt source, a high bit enables an interrupt. | |
| 11h | INT_SERV | 12 | Interrupt service status bits (active high). | Read only. See chapter Tricks. |
| 12h | INT_PEND | 12 | Pending interrupt requests(active high). | |

| | | | | |
|---|---|---|---|---|
| 13h | EXT_INT_PRI | 32 | Bits 31 downto 28 : INT 7 priority<br>Bits 27 downto 24 : INT 6 priority<br>...<br>Bits 7 downto 4 : INT 1 priority<br>Bits 3 downto 0 : INT 0 priority | 0 – highest priority<br>15 – lowest priority<br>Priorities for external interrupts can only be set if internal handler is used. |
| 14h | COP_INT_PRI | 16 | Bits 15 downto 12 : COP3 priority<br>Bits 11 downto 8 : COP2 priority<br>Bits 7 downto 4 : COP1 priority<br>Bits 3 downto 0 : COP0 priority | |

### 6.1.8 Clearing a pending interrupt without running the service routine

The ability to clear bits in the INT_PEND –register directly would lead to situations where an external interrupt handler would not know the real status of the latest interrupt request because INT_ACK -signal would never go high for these *cancelled* interrupts. This kind of inconsistency is not acceptable and that's why INT_PEND is a read only register. If there is a need to 'cancel' a request it can be done as follows (If internal CCB is mapped to protected memory area, super user mode is needed):

- Interrupts should be disabled during these operations!
- Save the current value in the interrupt vector register of the INT source in question.
- Replace the old vector with a new one which points to a dummy routine (remember OFFSET, if external handler is present) which executes reti – instruction only (and maybe some acknowledge instructions for external handler).
- Set the interrupt source to highest priority and make sure that no other source shares the same priority (of course save old values).
- Set mask bit for the interrupt source in question (save old value of INT_MASK)
- enable interrupts
- Poll the INT_PEND register; disable interrupts when the bit in question is low.
- Restore vector and priorities.
- Continue normally

**Do not do this!**

Do not change interrupt priorities while in interrupt service routine if you use nested interrupts (unless you are 100% sure that a new request from a source cannot arise before a service routine is

finished). In extreme cases this can lead to hardware stack overflow if interrupt nesting level is twelve and priorities are changed so that multiple requests from a single source can be active simultaneously. Normally an interrupt service routine cannot be interrupted by a new request from the same source because of priority resolving.

## 6.2 Exceptions

An *exception* means an event which will halt the processing of the current thread immediately and causes the core to switch to an exception handling routine. An exception is considered an error condition and has to be dealt with immediately. Note that very often in literature exception means interrupting the processor in general. Exceptions in COFFEE™ core can be thought as synchronous interrupts as they are generated by the control unit to notify the privileged software like the kernel to do something when an anomalous condition arises. The Processor when identifies an exception stops everything it is doing and just jump to the exception handler, where the task that generated the exception is made to TASK_STOPPED and deleted from the run queue. Below table gives the types and codes of the exceptions in COFFEE Core. [1]

**Table 8, Exception types and codes.**

| pri | code | name | description |
|---|---|---|---|
| 10 | 00000000 | instruction address violation 3 | While in user mode, instruction is fetched from memory address not allowed for user. |
| 6 | 00000001 | unknown opcode | Version 1.0 of COFFEE RISC does not have any unused opcodes which makes this obsolete. |
| 7 | 00000010 | Illegal instruction | While in 16 bit mode, trying to execute an instruction which is valid only in 32 bit mode or trying to execute a super user only instruction in user mode. |
| 3 | 00000011 | miss aligned jump address 4 | Calculated jump target is not aligned to word(32 bit mode) or halfword(16 bit mode) boundary. |
| 2 | 00000100 | jump address overflow | A PC relative jump below the bottom of the memory or above the top of the memory. |
| 9 | 00000101 | miss aligned instruction address 1 | Instruction address is not aligned according to mode. This can be caused by:<br>- External boot address was not aligned to word boundary -An interrupt vector is not properly aligned or interrupt mode is not correctly set<br>- Exception handler entry address is not aligned to word boundary (this will lock the core by causing an eternal loop!) |

| | | | |
|---|---|---|---|
| | | | - System entry address is not aligned to word boundary |
| 8 | 111xxxxx | trap 2 | processor encountered a trap instruction |
| 5 | 00000110 | arithmetic overflow | The result of a signed arithmetic operation exceeds $2^{31}$ -1 or falls below - $2^{31}$ |
| 0 | 00000111 | data address violation | While in user mode, a data address refers to memory address not allowed for user. |
| 1 | 00001000 | data address overflow | Trying to index data below of the bottom or above of the top of the memory |
| 4 | 00001001 | Illegal jump | Trying to jump to protected instruction memory area while in user -mode. |
| x | 00001010 ... 00011111 | | Reserved for future extensions |

### 6.2.1 Handling an exception

In case of an exception, core performs following tasks:

- Saves the address of the instruction causing the exception (or just an address, see table on previous page) to CCB register EXCEPTION_PC.
- Saves to CCB register EXCEPTION_PSR processor status flags which were used when the violating instruction was decoded.
- Saves the exception code (see table above) to CCB register EXCEPTION_CS.
- Disables interrupts.
- Switches to 32 bit decoding mode and super user mode with register set 2 as default for reading and writing.
- Starts execution from a handler routine pointed by the CCB register EXCEP_ADDR.

Following things are guaranteed by hardware:

- The violating instruction is not able to modify the state of the processor (registers, status flags, data memory).
- All instructions before the violating one (in the order of execution) are executed.
- None of the instruction following the violating one are executed (pipeline is flushed up to the violating instruction).
- If multiple instructions on pipeline cause an exception simultaneously, the one which is first in the order of execution is taken into account.
- Interrupt requests cannot get through if an exception is signalled.

- An exception handler routine will always see updated values of EXCEPTION_XX – registers immediately.

### 6.2.2 Exception Handler

- When a new exception is signalled, then the processor stops everything it was doing previously and starts executing the instructions that start from the address in the register EXCEP_ADDR in the CCB registers.
- The code of the exception is fetched from the CCB register EXCEPTION_CS whose offset is "15h" from the CCB_BASE. Depending on the code, if the exception was caused by the user application (unprivileged software) then its state is just changed to TASK_STOPPED or if it was caused by the kernel then the processor is restarted using the system call sys_reboot( ). Traps and individuals handling for all exception codes will be done in the future works.
- It is worth to note here that exception handler works in the same static space of the task/application that generated an exception. This will not create any problems and all the situations that could occur at this time are examined.

    1. When the user application caused an exception, then the kernel starts executing the exception handler (exception_handler ( )) using the same stack space that the process was using. If the exception handler is able to recover from the abnormal situation that created the exception then it returns just had to start executing the process without thinking anything about the stack pointers. If the exception handler is not able to recover from the abnormal condition caused by the user process, then the scheduler is called indirectly by calling a dummy system call dummy_idle_brain(). The scheduler will remove the process from the run queue and the stack pointers are automatically updated for the new process by the system call handler and that process selected by the scheduler is resumed at the end of the end of system call by executing the instruction *retu*. The reason of using the system call to call the scheduler indirectly is to follow the conventions of the Linux calling the scheduler only after the system call.

    2. It is required that exceptions are not generated inside the kernel, but if it happens to occur inside the kernel then the kernel should be intelligent enough to identify that before it looses control over the execution sequences. In the CUP-OS, when an exception is caused in the kernel, it is identified and the processor is made to reboot.

### 6.2.3 Returning from the Exception Handler

Depending on which process caused the exception, execution can be resumed from a different context or from the same context or it might not be resumed at all. In any case, appropriate flags are written to SPSR and the resume address written to PR31 (the link register). Then, executing retu – instruction will update the PSR with flags written to SPSR and load the program counter with the value in PR31 causing the processor to start executing instructions from the desired memory location in the desired mode.

Notes

- It is clear that at the time of this writing, the kernel deletes the process from the run queue if it is an unprivileged process else it restarts the processor.
- EXCEP_ADDR –register is initialised in boot code. Incorrect address may cause eternal loop which will lock the processor until it is reset.
- Interrupts are disabled when entering the handler routine by both hardware and also software; they are enabled after just before the handler exits.
- If the exception is caused by an interrupt service routine (see interrupts) and the routine is disabled permanently, you should pop the return address of that routine from the hardware stack to ensure correct operation of other interrupt routines.

## 6.3 System Calls

The process running in User Mode cannot access the system resources by themselves but ask the Operating System to do it instead. The Operating system accesses the system resources in a way which is believed to be secure by other users and the operating system itself. Operating systems offer processes running in User Mode a set of interfaces to interact with hardware devices such as the CPU, disks, printers, and so on. Putting an extra layer between the application and the hardware has several advantages. First, it makes programming easier, freeing users from studying low-level programming characteristics of hardware devices. Second, it greatly increases system security, since the kernel can check the correctness of the request at the interface level before attempting to satisfy it [14]. Last but not least, these interfaces make programs more portable since they can be compiled and executed correctly on any kernel that offers the same set of interfaces. UNIX systems implement most interfaces between User Mode processes and hardware devices by means of *system calls* issued to the kernel. This chapter examines in detail how system calls are implemented by the Linux kernel.

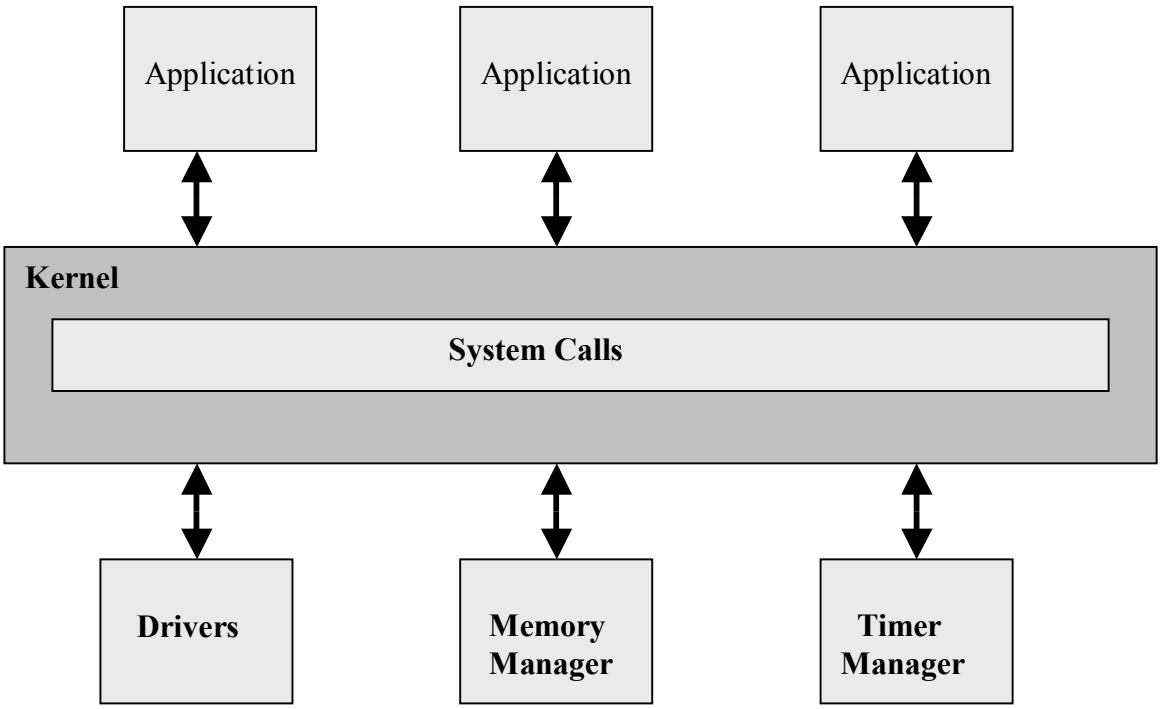


**Figure 15: System Calls**

### 6.3.1 API and System Calls

The API is a function definition that specifies how to obtain a given service and a System Call is an explicit request to the kernel made via a software interrupt using the instruction SCALL. UNIX like systems (which means mostly UNIX and all flavours of LINUX OS) include several libraries of functions that provide APIs to programmers. APIs are mostly just wrapper routines which just make a system call in an accepted style. Usually, each system call corresponds to a wrapper routine; the wrapper routine defines the API that application programs should refer to. The converse is not true; an API does not necessarily correspond to a specific system call [9]. The API could offer its services directly in User Mode but no such services are available in the API at the moment of this writing. A single API function could make several system calls. Moreover, several API functions could make the same system call but wrap extra functionality around it.

A programmer who writes programs in user mode, writes functions such as getpid() to access the file system and write to the hard drive, or the floppy. Such function is called a wrapper routine. The way it works is that the open wrapper routine will send a software interrupt with its system call number to be looked up in the sys_call_table in the /kernel/entry.h which has the same number in

the defines called __NR_getpid() in the unistd.h file. When this number is matched in the symbol table, it will have the address to execute the open function. The user mode will be switched to kernel mode, also called CPU mode and execute the proper C/assembly code to get the PID of the current running process. User makes system call as they call normal functions without needing to know the internals of the kernel. This function get converted to another function with the same name and arguments but does nothing except checking the validity of the arguments passed and then calling for the kernel service using the instruction *SCALL*. The system call numbers in the unistd.h looks like the following

```
#define __NR_setup                0
#define __NR_exit                 1
#define __NR_vfork                2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
#define __NR_close                6
#define __NR_waitpid              7
#define __NR_creat                8
#define __NR_link                 9
…
…
```

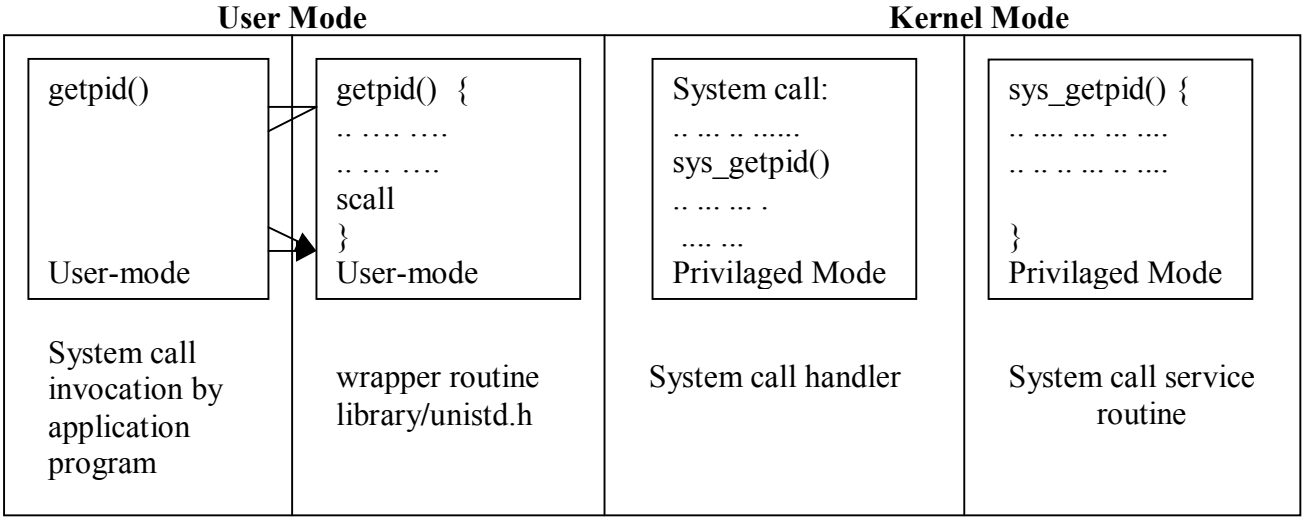


**Figure 16: System Call interface from user to kernel**

System call handler does the following [13]

- SAVE_ALL
- get current task struct
- syscall # not OK? !badsys
- dispatch specific syscall !*(sys_call_table[call_number])
- save return value
- Service system call if good system call number.
- need to reschedule? !reschedule if yes
- RESTORE_ALL.
- Return to the current task.

### 6.3.2 Available System Calls

Even though uClinux has about 200 system calls, not all have been ported at the time of this writing. Some handful of them were ported and well tested and some calls just return the an error value indicating that this system call has not yet been implemented. The following is the list of those system calls.

**Table 9, System calls available in CUP-OS**

| Name | Description |
|---|---|
| **sys_getpid** | Returns the process ID of the current process. |
| **sys_pause** | The pause library function causes the invoking process (or thread) to sleep until a signal is received that either terminates it or causes it to call a signal-catching function. |
| **sys_nice** | adds inc to the nice value for the calling pid. (A large nice value means a low priority.) |
| **sys_getppid** | Returns the process ID of the parent of the current process. |
| **sys_reboot** | Reboots the processor within one timer interrupt time. |
| **sys_getpriority** | Get the priority of the current process. |
| **sys_setpriority** | Set the priority of the current process. |
| **sys_idle** | idle is an internal system call used during bootstrap. It lowers process priority, and enters the main scheduling loop. Idle never returns. Only process 0 may call idle. Any user process, even a process with super-user permission, will receive EPERM. |
| **sys_sched_setparam** | sets the scheduling parameters associated with the scheduling policy for the process identified by pid. If pid is zero, then the parameters of the current process are set. The interpretation of the parameter p depends on the selected policy. Currently, the following three scheduling policies are supported under Linux: SCHED_FIFO, SCHED_RR, and SCHED_OTHER. |
| **sys_sched_setscheduler** | sets both the scheduling policy and the associated parameters for the process identified by pid. If pid equals zero, the scheduler of the calling process will be set. The interpretation of the parameter p depends on the selected policy. Currently, the following three scheduling policies are supported under Linux: SCHED_FIFO, SCHED_RR, and SCHED_OTHER |
| **sys_sched_getscheduler** | queries the scheduling policy currently applied to the process identified by pid. If pid equals zero, the policy of the calling process will be retrieved. |
| **sys_sched_yield** | A process can relinquish the processor voluntarily without blocking by calling sched_yield. The process will then be moved to the end of the queue for its static priority and a new process gets to run. If the current process is the only process in the highest priority list at that time, this process will continue to run after a call to sched_yield. |

| | |
|---|---|
| **sys_sched_get_priority_max** | returns the maximum priority value that can be used with the scheduling algorithm identified by policy. Supported policy values are SCHED_FIFO, SCHED_RR, and SCHED_OTHER. |
| **sys_sched_get_priority_min** | returns the minimum priority value that can be used with the scheduling algorithm identified by policy. Supported policy values are SCHED_FIFO, SCHED_RR, and SCHED_OTHER. |
| **sys_dummy_idle_brain** | Just sets a value so the scheduler is activated while returning from the system call. This system call is mostly used by the kernel itself for indirectly calling the scheduler. |

### 6.3.3 More about system calls

Several system calls have been introduced to allow processes to change their priorities and scheduling policies. In Linux, users are always allowed to lower the priorities of their processes. However, if they want to modify the priorities of processes belonging to some other user or if they want to increase the priorities of their own processes, they must have superuser privileges. CUP-OS doesn't support users, hence no Superuser concept, any process can change its priorities but it is not given privileges to change the priorities of other processes. From the very beginning the kernel is being designed by assuming that the applications to be run on the COFFEE core are trust worthy and they will not do any nasty things to corrupt or effect other processes.

**The nice( ) System Call**

The nice( ) system call allows processes to change their base priority. The integer value contained in the increment parameter is used to modify the priority field of the process descriptor. The nice Unix command, which allows users to run programs with modified scheduling priority, is based on this system call. Note that all the C code for the system calls has been used for the latest version of uClinux and it can be found here that some stuff does nothing for now but they are still in the CUP-OS for making upgrades in it easy. Most of them will be found useful in the later versions.

The sys_nice( ) service routine handles the nice( ) system call. Although the increment parameter may have any value, absolute values larger than 40 are trimmed down to 40. Traditionally, negative values correspond to requests for priority increments, while positive ones correspond to requests for priority decrements.

The function starts by copying the value of increment into the newprio local variable. sys_nice( ) changes the sign of newprio and it sets the increase local flag:

```
newprio = increment;
if (increment < 0) {
  //            if (!suser())        everyone is super user right now
  //            return -EPERM;
                newprio = -increment;
                increase = 1;
}
```

If newprio has a value larger than 40, the function trims it down to 40. At this point, the newprio local variable may have any value included from 0 to 40, inclusive. The value is then converted according to the priority scale used by the scheduling algorithm. The resulting value is copied into increment with the proper sign. Since the highest base priority allowed is 2 * DEF_PRIORITY, the new value is given below:

```
if (newprio > 40)
    newprio = 40;
newprio = (newprio * DEF_PRIORITY + 10) / 20;
increment = newprio;
if (increase)
      increment = -increment;
```

The function then sets the final value of priority by subtracting the value of increment from it. However, the final base priority of the process cannot be smaller than 1 or larger than 2*DEF_PRIORITY.

```
newprio = current->priority - increment;
if ((signed) newprio < 1)
                newprio = 1;
if (newprio > DEF_PRIORITY*2)
                newprio = DEF_PRIORITY*2;
current->priority = newprio;
return 0;
```

**The getpriority( ) and setpriority( ) System Calls**

The nice( ) system call affects only the process that invokes it. Two other system calls, denoted as getpriority( ) and setpriority( ), act on the base priorities of all processes. getpriority( ) returns 20 plus the highest base priority among all processes; setpriority( ) sets the base priority of all processes to a given value. In Linux these system calls work only on the processes of the same group but since there is no groups support yet in CUP-OS they work fine on all processes.
The kernel implements these system calls by means of the sys_getpriority( ) and sys_setpriority( ) service routines. The parameter *who* doesn't mean anything for now until the kernel supports the groups, hope it doesn't confuse the readers, they will be used in near future.

**System Calls Related to Real-Time Processes**

We now introduce a group of system calls that allow processes to change their scheduling discipline and, in particular, to become real-time processes.

**The sched_getscheduler( ) and sched_setscheduler( ) system calls**

The sched_ getscheduler( ) system call queries the scheduling policy currently applied to the process identified by the pid parameter. If pid equals 0, the policy of the calling process will be retrieved. On success, the system call returns the policy for the process: SCHED_FIFO, SCHED_RR, or SCHED_OTHER. The corresponding sys_sched_getscheduler( ) service routine invokes find_task_by_pid( ), which locates the process descriptor corresponding to the given pid and returns the value of its policy field.

The sched_setscheduler( ) system call sets both the scheduling policy and the associated parameters for the process identified by the parameter pid. If pid is equal to 0, the scheduler parameters of the calling process will be set.

The corresponding sys_sched_setscheduler( ) function checks whether the scheduling policy specified by the policy parameter and the new static priority specified by the param->sched_priority parameter are valid. If everything is OK, it executes the following statements:

```
p->policy = policy;
p->rt_priority = param->sched_priority;
if (p->next_run)
move_first_runqueue(p);
current->need_resched = 1;
```

**The sched_setparam( ) system call**

The sched_setparam( ) system call is similar to sched_setscheduler( ): it differs from the latter by not letting the caller set the policy field's value.[8] The corresponding sys_sched_setparam( ) service routine is almost identical to sys_sched_setscheduler( ), but the policy of the affected process is never changed.

**The sched_ yield( ) system call**

The sched_ yield( ) system call allows a process to relinquish the CPU voluntarily without being suspended; the process remains in a TASK_RUNNING state, but the scheduler puts it at the end of the runqueue list. In this way, other processes having the same dynamic priority will have a chance to run. The call is used mainly by SCHED_FIFO processes. The corresponding sys_sched_ yield( ) service routine executes these statements:

```
asm("block_interrupts int_mask_temp\n\t");
            move_last_runqueue(current);
            current->counter = 0;
            need_resched = 1;
asm("restore_interrupts int_mask_temp\n\t");
            return 1;
```

**The sched_get_priority_min( ) and sched_get_priority_max( )**

These system calls return, respectively, the minimum and the maximum real-time static priority value that can be used with the scheduling policy identified by the policy parameter. The sys_sched_get_priority_min( ) service routine returns 1 if current is a real-time process, 0 otherwise.

The sys_sched_get_priority_max() service routine returns 99 (the highest priority) if current is a real-time process, 0 otherwise.

**NOTES:**

1. Recall that stopped and suspended processes cannot be selected by the scheduling algorithm to run on the CPU.
2. Actually, things could be much worse than this; for example, if the time required for task switch is counted in the process quantum, all CPU time will be devoted to task switch and no process can progress toward its termination. Anyway, you got the point.
3. These conditions look like voodoo magic; perhaps, they are empirical rules that make the SMP scheduler work better.

4. The Linux kernel has been modified in several ways so it can handle a few hard real-time jobs if they remain short. Basically, hardware interrupts are trapped and kernel execution is monitored by a kind of "superkernel." These changes do not make Linux a true real-time system, though.
5. Since this system call is usually invoked to lower the priority of a process, users who invoke it for their processes are "nice" toward other users.

### 6.3.4 Adding a System Call

- link statically or implement as a kernel module
- create a “library wrapper” with _syscallN macros
    - /kernel/library/unistd.h
    - syscallN( return_type, entry, type1, arg1, type2, arg2, …)
    - allocate a system call number ( i ) to it in /kernel/library/unistd.h
- register the address of the new system call in the $i^{th}$ array location in the sys_call_table [].
- register the wrapper routine, library call related to the sys_whatever in /kernel/library/coffee.h to sys_callX where X is the number of parameters to the system call.
- Validate all parameters!
- return appropriate error codes

# 7 Timing Measurement

The main attention in this chapter is kernel timers. Kernel timers are used to dispatch the execution of a particular function (called 'timer handler') at a specific time in the future.

## 7.1 Hardware Timers

COFFEE core has two independent built-in timers. Both of then are programmable timers, 32 bit wide and both having separate 8 bit divisor. Timers can be configured as watchdog timers or timer tick generators for system. Timer registers reside inside CCB (core configuration block) and can be accessed using **ld** and **st** instructions. Table below explains the meaning and usage of timer registers.

### 7.1.1 Timer registers

**Table 10, Timer Control and Configuration Registers**

| register mnemonic | bit field mnemonic | bits | explanation |
|---|---|---|---|
| TMR0_CNT | | [31:0] | Current value of the timer0 counter. Can be set to arbitrary value. |
| TMR0_MAX_CNT | | [31:0] | The maximum value of timer0 counter. Depending on CONT –bit, the timer will stop at maximum value or restart from zero. Note that, you can set a value greater than maximum count in TMR0_CNT –register in which case the timer counter will count to 0xffffffff and start over from zero. |
| TMR1_CNT | | [31:0] | Current value of the timer1 counter. Can be set to arbitrary value. |
| TMR1_MAX_CNT | | [31:0] | The maximum value of timer1 counter. Depending on CONT –bit, the timer will stop at maximum value or restart from zero. Note that, you can set a value greater than maximum count in TMR1_CNT –register in which case the timer counter will count to 0xffffffff and start over from zero. |
| TMR_CONF | TMR1_CONF | [31:16] | Configuration bits for timer1. See table 2 for bit field definitions. |
| | TMR0_CONF | [15:0] | Configuration bits for timer0. See table 2 for bit field definitions. |

**Configuration registers TMR1_CONF and TMR0_CONF bit fields**

| | | |
|---|---|---|
| EN | 31/15 | EN = 1 enables timer. A timer can be stopped at any moment by writing EN = 0. Clearing EN bit will zero timer divider => timer will be incremented [DIV] + 1 clock cycles after enabling it. |
| CONT | 30/14 | CONT = 1: Continuous mode. Timer counter will start from zero after reaching maximum count defined in TMRx_MAX_CNT – register.<br>CONT = 0: Timer counter will stop at maximum count. |
| GINT | 29/13 | GINT = 1: Generate an interrupt when maximum count is reached. GINT = 0: Do not generate interrupts. |
| WDOG | 28/12 | WDOG = 1: Enable watchdog function. If the timer reaches maximum count defined in TMRx_MAX_CNT the core will be reset. |
| - | 27/11 | Reserved, 0 or 1 can be written. |
| INTN | [26:24]/[10:8] | Bit field defining which interrupt to associate the timer with: “000” => EXT_INT0 ... “111” => EXT_INT7 |
| DIV | [23:16]/ [7:0] | Divider value which defines how many clock cycles corresponds to one timer cycle: A timer counter will be incremented every [DIV] + 1 cycles, that is a zero value in DIV field sets the timer frequency to be the same as clock frequency of the core. |

The role of this timer is similar to the alarm clock of a microwave oven: to make the user aware that the cooking time interval has elapsed. Instead of ringing a bell, this device issues a special interrupt called *timer interrupt*, which notifies the kernel that one more time interval has elapsed. There is a difference between this alarm and the Timer that is it can be set if the timer issues this interrupt only once the time elapsed or it issues interrupts continuously every time the time specified in TMRx_MAX_CNT elapsed. This is done using the bit CONT inside the register TMR_CONF.

### 7.1.2 Configuration of Timer registers at start up

Continuous interrupting mode for timer 0, timer 1 disabled

EN 31/15 EN = 1 enables timer.

CONT 30/14 CONT = 1: Continuous mode.

GINT 29/13 GINT = 1: Generate an interrupt when maximum count is reached.

WDOG 28/12 WDOG = 0: Disable watchdog function

- 27/11 Reserved, 0 or 1 can be written.

INTN [26:24]/[10:8] = 0b111 Bit field defining which interrupt to associate the timer with

DIV [23:16]/ [7:0] = 0xFF Divider value

```
// Timer0 generates an interrupt every 10ms assuming 50MHz processor speed
// Writes to instruction memory are mapped to
// 0xffffc000...0xffffffff
```

**Timer Tick Handler**

[6] At the time of this writing only one timer interrupt (TIMER0) was enabled to the interrupt line *external interrupt 7* and also the coprocessors interrupts were enabled. Timer 0 is used as a timer tick for the kernel which is used for all timing management purposes. When this interrupt occurs the kernel updates its timing count in the variable JIFFIES and also checks if any process has run out of its time quantum and sets the value of the *need_resched* = 1 if the scheduler needs to be called be the near future to make a process switch. Also it is checked if the process has made any system call by checking the incremented value of the variable *syscall_called_timer* in the current process descriptor to the value 2000 (one can change this value to appropriate one), if no system call is made from the past 2000 ticks then it is assumed that the current process is stuck or doing nothing other than just looping doing no productive work and eating up the processor time, and hence its state is made to TASK_STOPPED and the scheduler is activated by calling a dummy system call which will make the scheduler remove this task from the run queue. Then some timer queues are checked if there are any processes with expired timers.

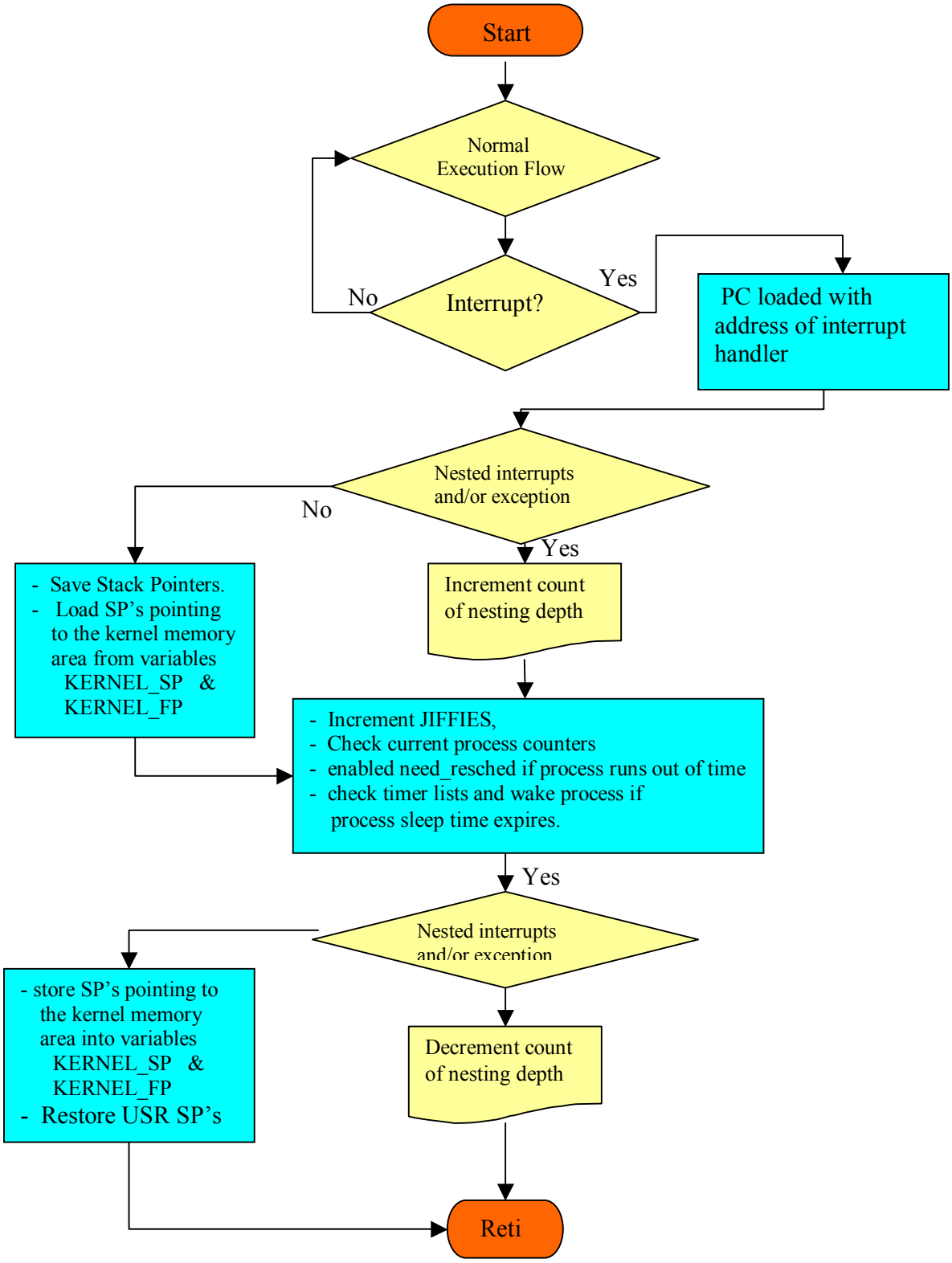


**Figure 17: Flowchart of interrupt handler**

## 7.2 Software Timers

The ultimate resources for time keeping in the kernel are the timers. Timers are used to schedule execution of a function (a timer handler) at a particular time in the future. They thus work differently from task queues and tasklets in that you can specify when in the future your function will be called, whereas you can't tell exactly when a queued task will be executed. On the other hand, kernel timers are similar to task queues in that a function registered in a kernel timer is executed only once -- timers aren't cyclic.

There are times when you need to execute operations detached from any process's context, like turning off the floppy motor or finishing another lengthy shutdown operation. In that case, delaying the return from close wouldn't be fair to the application program. Using a task queue would be wasteful, because a queued task must continually reregister itself until the requisite time has passed.

A timer is much easier to use. You register your function once, and the kernel calls it once when the timer expires. Such functionality is used often within the kernel proper, but it is sometimes needed by the drivers as well, as in the example of the floppy motor.

The kernel timers are organized in a doubly linked list. This means that you can create as many timers as you want. A timer is characterized by its timeout value (in jiffies) and the function to be called when the timer expires. The timer handler receives an argument, which is stored in the data structure, together with a pointer to the handler itself.

The data structure of a timer looks like the following, which is extracted from “/kernel/sched.h”):

```
struct timer_list {
    struct timer_list *next;        /* never touch this */
    struct timer_list *prev;        /* never touch this */
    unsigned long expires;           /* the timeout, in jiffies */
    unsigned long data;             /* argument to the handler */
    void (*function)(unsigned long);  /* handler of the timeout */
    };
```

The timeout of a timer is a value in jiffies. Thus, timer->function will run when jiffies is equal to or greater than timer->expires. The timeout is an absolute value; it is usually generated by taking the current value of jiffies and adding the amount of the desired delay.

Once a timer_list structure is initialized, add_timer inserts it into a sorted list, which is then polled more or less 100 times per second.

These are the functions used to act on timers:

**void init_timer(struct timer_list *timer);**

This inline function is used to initialize the timer structure. Currently, it zeros the prev and next pointers. Programmers are strongly urged to use this function to initialize a timer and to never explicitly touch the pointers in the structure, in order to be forward compatible.

```
timer->next = NULL;
timer->prev = NULL;
```

**void add_timer(struct timer_list *timer);**

This function inserts a timer into the global list of active timers.

```
unsigned long flags;
asm("block_interrupts int_mask_temp\n\t");
internal_add_timer(timer);  //actual calculations of time
asm("restore_interrupts int_mask_temp\n\t");
```

**int del_timer(struct timer_list *timer);**

If a timer needs to be removed from the list before it expires, del_timer should be called. When a timer expires, on the other hand, it is automatically removed from the list.

```
int ret;
unsigned long flags;
asm("block_interrupts int_mask_temp\n\t");
ret = detach_timer(timer);
timer->next = timer->prev = 0;
asm("restore_interrupts int_mask_temp\n\t");
return ret;
```

# 8 Process Scheduling

Scheduler selects the most deserving process to run out of all of the runnable processes in the system. A process cannot run on the processor continuously for a very long time if there are other processes waiting for their chance to get the CPU time. The scheduler is the one that decides which process to be allowed to run and for how long time depending on some policies. This chapter dwells into the topic of process scheduling.

## 8.1 Scheduling Policies

The scheduler is the kernel part that decides which runnable process will be executed by the CPU next. The Linux scheduler offers three different scheduling policies, one for normal processes and two for real-time applications. A static priority value sched_priority is assigned to each process and this value can be changed only via system calls. Conceptually, the scheduler maintains a list of runnable processes for each possible sched_priority value, and sched_priority can have a value in the range 0 to 99. In order to determine the process that runs next, the Linux scheduler looks for the non-empty list with the highest static priority and takes the process at the head of this list. The scheduling policy determines for each process, where it will be inserted into the list of processes with equal static priority and how it will move inside this list.

SCHED_OTHER

This is the default universal time-sharing scheduler policy used by most processes, SCHED_FIFO and SCHED_RR are intended for special time-critical applications that need precise control over the way in which runnable processes are selected for execution. Processes scheduled with SCHED_OTHER must be assigned the static priority 0, processes scheduled under SCHED_FIFO or SCHED_RR can have a static priority in the range 1 to 99. Only processes with superuser privileges can get a static priority higher than 0 and can therefore be scheduled under SCHED_FIFO or SCHED_RR. The system calls sched_get_priority_min and sched_get_priority_max can be used to find out the valid priority range for a scheduling policy in a portable way on all POSIX.1b conforming systems.

All scheduling is preemptive: If a process with a higher static priority gets ready to run, the current process will be preempted and returned into its wait list. The scheduling policy only determines the ordering within the list of runnable processes with equal static priority.

SCHED_FIFO: First In-First Out scheduling

SCHED_FIFO can only be used with static priorities higher than 0, which means that when a SCHED_FIFO processes becomes runnable, it will always preempt immediately any currently running normal SCHED_OTHER process. SCHED_FIFO is a simple scheduling algorithm without time slicing. For processes scheduled under the SCHED_FIFO policy, the following rules are applied: A SCHED_FIFO process that has been preempted by another process of higher priority will stay at the head of the list for its priority and will resume execution as soon as all processes of higher priority are blocked again. When a SCHED_FIFO process becomes runnable, it will be inserted at the end of the list for its priority. A call to sched_setscheduler or sched_setparam will put the SCHED_FIFO (or SCHED_RR) process identified by pid at the start of the list if it was runnable. As a consequence, it may preempt the currently running process if it has the same priority. (POSIX 1003.1 specifies that the process should go to the end of the list.) A process calling sched_yield will be put at the end of the list. No other events will move a process scheduled under the SCHED_FIFO policy in the wait list of runnable processes with equal static priority. A SCHED_FIFO process runs until it is blocked by an I/O request, it is preempted by a higher priority process, or it calls sched_yield.

SCHED_RR: Round Robin scheduling

SCHED_RR is a simple enhancement of SCHED_FIFO. Everything described above for SCHED_FIFO also applies to SCHED_RR, except that each process is only allowed to run for a maximum time quantum. If a SCHED_RR process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A SCHED_RR process that has been preempted by a higher priority process and subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum. The length of the time quantum can be retrieved by sched_rr_get_interval.

SCHED_OTHER: Default Linux time-sharing scheduling

SCHED_OTHER can only be used at static priority 0. SCHED_OTHER is the standard Linux time-sharing scheduler that is intended for all processes that do not require special static priority real-time mechanisms. The process to run is chosen from the static priority 0 list based on a dynamic priority that is determined only inside this list. The dynamic priority is based on the nice level (set by the nice or setpriority system call) and increased for each time quantum the process is ready to run, but denied to run by the scheduler. This ensures fair progress among all SCHED_OTHER processes.

## 8.2 Priorities of processes:

- Static: assigned by the processes and are not changed by the scheduler. Not used here in CUP-OS
- Dynamic: Priority is adjusted to non real-time processes by the scheduler, i.e. the priority of the process that did not have the access of the CPU for a very long time is boosted so that it gets more chances to be selected next by the scheduler.

## 8.3 Process Pre-emption

All the Versions of Linux *kernel* (privileged user) are non pre-emptive! And the same with the CUP-OS. Processes running in kernel mode cannot be interrupted after any instruction. Linux *processes* are pre-emptive! Processes running in user mode can be interrupted after any instruction. Pre-emption is done when either a process exits voluntarily i.e. it has nothing more to do on the processor or when the time quantum assigned to that process in that epoch has expired. Time quantum is an important parameter that decides the multitasking capability of the kernel. It is sometimes also called as Context Switch.

Context switches can occur only in kernel mode. Kernel mode is a privileged mode of the CPU in which only the kernel runs and which provides access to all memory locations and all other system resources. Other programs, including applications, initially operate in user mode, but they can run portions of the kernel code via system calls. A system call is a request in a Unix-like operating system by an active process (i.e., a process currently progressing in the CPU) for a service performed by the kernel, such as input/output (I/O) or process creation (i.e., creation of a new process). I/O can be defined as any movement of information to or from the combination of the CPU and main memory (i.e. RAM), that is, communication between this combination and the computer's users (e.g., via the keyboard or mouse), its storage devices (e.g., disk or tape drives) or other computers.

The existence of these two modes in Unix-like operating systems means that a similar, but simpler, operation is necessary when a system call causes the CPU to shift to kernel mode. This is referred to as a mode switch rather than a context switch, because it does not change the current process.

Context switching is an essential feature of multitasking operating systems. A multitasking operating system is one in which multiple processes execute on a single CPU seemingly simultaneously and without interfering with each other. This illusion of concurrency is achieved by

means of context switches that are occurring in rapid succession (tens or hundreds of times per second). These context switches occur as a result of processes voluntarily relinquishing their time in the CPU or as a result of the scheduler making the switch when a process has used up its CPU time slice.

A context switch can also occur as a result of a hardware interrupt, i.e., a signal from a hardware device (such as a keyboard, mouse, modem or system clock) to the kernel that an event (e.g., a key press, mouse movement or arrival of modem data) has occurred.

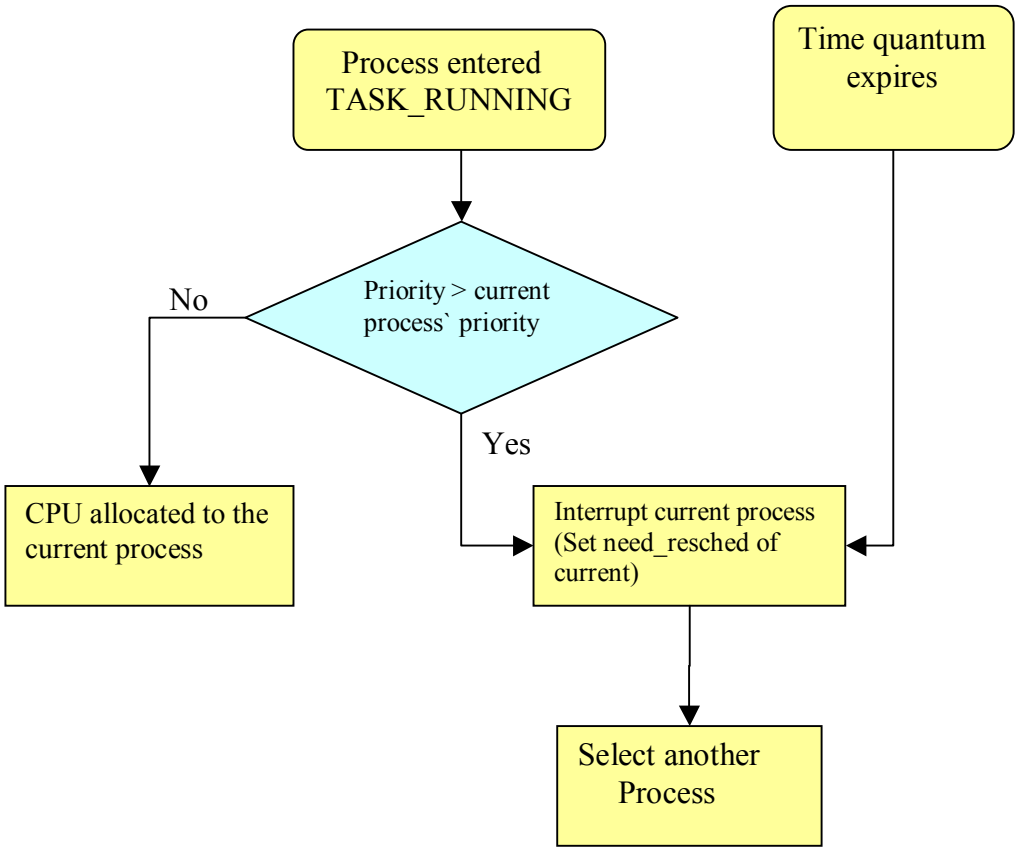


**Figure 18: Flowchart illustrating the scheduler functionality.**

## 8.4 Duration of the Time Quantum

**Too Short:**

If the quantum is too short then the real productive time for a process is very small compared to the overhead needed for the kernel to make a switch and manage things. Quantum expires quite often and the time it takes for the kernel to come into picture and understand the current scenario

and make another process as current one is more than the time the process actually runs on the CPU.

**Too Long:**

If the quantum is too long then each process productive time on the CPU is good enough but the process no longer seems to be executing concurrently. That is the system does not appear concurrent and one can humanly differentiate that each process gets the CPU in chances which is not tolerable to many of the users.

Within the kernel, processes that are in memory and are ready to run or are running are in state **TASK_RUNNING**. The scheduler selects a process ready to run and allocate the CPU to it. The scheduler is implemented by the function schedule( ) (found in the file /kernel/sched.c). The state field in task_struct can take one of the following values:

```
#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_ZOMBIE 4
#define TASK_STOPPED 8
#define TASK_SWAPPING 16
```

A process is in the TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE state while the OS performs an I/O on its behalf or is sleeping. In the TASK_INTERRUPTIBLE state, a process can be reactivated by a signal whereas it cannot in the TASK_UNINTERRUPTIBLE state. In state TASK_ZOMBIE, a process has completed its execution but its parent has not done yet the system call wait( ).

Within the kernel, statements that change the state of a process are of the form:
current->state = TASK_XXXX;
For example:

- The function sleep_on() assigns TASK_INTERRUPTIBLE before a process is put in the wait queue.
- The function wake_up_process() assigns TASK_RUNNING after a process is removed from the wait queue.

## 8.5 Implementation of the scheduler

The Linux scheduling algorithm works by dividing the CPU time into epochs. In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins. In general, different processes have different time quantum durations. The time quantum value is the maximum CPU time portion assigned to the process in that epoch. When a process has exhausted its time quantum, it is pre-empted and replaced by another runnable process. Of course, a process can be selected several times from the scheduler in the same epoch, as long as its quantum has not been exhausted--for instance, if it suspends itself to wait for I/O, it preserves some of its time quantum and can be selected again during the same epoch. The epoch ends when all runnable processes have exhausted their quantum; in this case, the scheduler algorithm recomputes the time-quantum durations of all processes and a new epoch begins.

Each process has a base time quantum: it is the time-quantum value assigned by the scheduler to the process if it has exhausted its quantum in the previous epoch. The users can change the base time quantum of their processes by using the nice( ) and setpriority( ) system calls. A new process always inherits the base time quantum of its parent.

In order to select a process to run, the Linux scheduler must consider the priority of each process. Actually, there are two kinds of priority:

### 8.5.1 Static Priority

This kind is assigned by the users to real-time processes and ranges from 1 to 99. It is never changed by the scheduler.

### 8.5.2 Dynamic Priority

This kind applies only to conventional processes; it is essentially the sum of the base time quantum (which is therefore also called the base priority of the process) and of the number of ticks of CPU time left to the process before its quantum expires in the current epoch.

Of course, the static priority of a real-time process is always higher than the dynamic priority of a conventional one: the scheduler will start running conventional processes only when there is no real-time process in a TASK_RUNNING state.

### 8.5.3 Counter

The number of ticks of CPU time left to the process before its quantum expires; when a new epoch begins, this field contains the time-quantum duration of the process. The update_process_times( ) function decrements the counter field of the current process by 1 at every tick.

Notice that the priority and counter fields play different roles for the various kinds of processes. For conventional processes, they are used both to implement time-sharing and to compute the process dynamic priority. For SCHED_RR real-time processes, they are used only to implement time-sharing. Finally, for SCHED_FIFO real-time processes, they are not used at all, because the scheduling algorithm regards the quantum duration as unlimited.

### 8.5.4 The schedule ( ) Function

schedule ( ) implements the scheduler. Its objective is to find a process in the runqueue list and then assign the CPU to it. It is invoked, directly or in a lazy way, by several kernel routines. [3]

**Direct invocation**

The scheduler is invoked directly when the current process must be blocked right away because the resource it needs is not available. In this case, the kernel routine that wants to block it proceeds as follows:

1. Inserts current in the proper wait queue.
2. Changes the state of the current to either TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE.
3. Invokes schedule ( ).
4. Checks if the resource is available; if not, goes to step 2.
5. Once the resource is available, removes current from the wait queue.

As can be seen, the kernel routine checks repeatedly whether the resource needed by the process is available; if not, it yields the CPU to some other process by invoking schedule( ). Later, when the scheduler once again grants the CPU to the process, the availability of the resource is again checked.

The scheduler is also directly invoked by many device drivers that execute long iterative tasks. At each iteration cycle, the driver checks the value of the need_resched field and, if necessary, invokes schedule( ) to voluntarily relinquish the CPU.

**Lazy invocation**

The scheduler can also be invoked in a lazy way by setting the need_resched field of current to 1. Since a check on the value of this field is always made before resuming the execution of a User Mode process, schedule( ) will definitely be invoked at some close future time.

Lazy invocation of the scheduler is performed in the following cases:

- When current has used up its quantum of CPU time; this is done by the update_process_times( ) function.
- When a process is woken up and its priority is higher than that of the current process; this task is performed by the reschedule_idle( ) function, which is invoked by the wake_up_process( ) function.
- if (goodness(current, p) > goodness(current, current)) current->need_resched = 1.
- When a sched_setscheduler( ) or sched_ yield( ) system call is issued.

### 8.5.5 Actions performed by schedule( )

Before actually scheduling a process, the schedule( ) function starts by running the functions left by other kernel control paths in various queues. The function invokes run_task_queue( ) on the tq _scheduler task queue. Linux puts a function in that task queue when it must defer its execution until the next schedule( ) invocation:

Now comes the actual scheduling, and therefore a potential process switch. The value of current is saved in the prev local variable and the need_resched field of prev is set to 0. The key outcome of the function is to set another local variable called next so that it points to the descriptor of the process selected to replace prev.

First, a check is made to determine whether prev is a Round Robin real-time process (policy field set to SCHED_RR) that has exhausted its quantum. If so, schedule( ) assigns a new quantum to prev and puts it at the bottom of the runqueue list:

```
if (!prev->counter && prev->policy == SCHED_RR)
{
prev->counter = prev->priority;
move_last_runqueue(prev);
```

```
}
prev->state = TASK_RUNNING;
```

If prev is not in the TASK_RUNNING state, schedule( ) was directly invoked by the process itself because it had to wait on some external resource; therefore, prev must be removed from the runqueue list:

```
if (prev->state != TASK_RUNNING)
del_from_runqueue(prev);
```

Next, schedule( ) must select the process to be executed in the next time quantum. To that end, the function scans the runqueue list. It starts from the process referenced by the next_run field of init_task, which is the descriptor of process 0 (swapper). The objective is to store in next the process descriptor pointer of the highest priority process. In order to do this, next is initialized to the first runnable process to be checked, and c is initialized to its "goodness".

```
c = -1000;
next = idle_task;
while (p != idle_task) {
        int weight = goodness(p, prev);
        if (weight > c)
                c = weight, next = p;
        p = p->next_run;
```

Now schedule( ) repeatedly invokes the goodness( ) function on the runnable processes to determine the best candidate:

```
p = init_task.next_run;
 while (p != &init_task) {
  weight = goodness(prev, p);
  if (weight > c)
      c = weight;   next = p;
  p = p->next_run;
   }
```

The while loop selects the first process in the runqueue having maximum weight. If the previous process is runnable, it is preferred with respect to other runnable processes having the same weight.

Notice that if the runqueue list is empty (no runnable process exists except for swapper), the cycle is not entered and next points to init_task. Moreover, if all processes in the runqueue list have a priority lesser than or equal to the priority of prev, no process switch will take place and the old process will continue to be executed.

A further check must be made at the exit of the loop to determine whether c is 0. This occurs only when all the processes in the runqueue list have exhausted their quantum, that is, all of them have a zero counter field. When this happens, a new epoch begins, therefore schedule( ) assigns to all existing processes (not only to the TASK_RUNNING ones) a fresh quantum, whose duration is the sum of the priority value plus half the counter value:

```
if (c = 0) {
        for_each_task(p)
                p->counter = (p->counter >> 1) +  p->priority;
}
```

In this way, suspended or stopped processes have their dynamic priorities periodically increased. As stated earlier, the rationale for increasing the counter value of suspended or stopped processes is to give preference to I/O-bound processes. However, even after an infinite number of increases, the value of counter can never become larger than twice[3] the priority value.

Now comes the concluding part of schedule( ): if a process other than prev has been selected, a process switch must take place. Before performing it, however, if the new selected process is not the same as the current process then the software timer is added to the current process and it is made to sleep for some period of time so that it if it doesn't interfere the newly selected process at least for the time it is made to sleep. This is done to make the new switch a bit more productive.

```
if (prev != next) {
struct timer_list *timer_pointer;

timer_pointer = &prev->timer;
if (timeout) {
        init_timer(timer_pointer);
        prev->timer.expires = timeout+jiffies;
        prev->timer.data = (unsigned long) prev;
        prev->timer.function = process_waittimeout;
```

```
                    add_timer(timer_pointer);
                    }
current = next;
                    }
return;
```

**Note :System Call Return Value:**

A very important note which at least I couldn't find in other documentations on the kernel on web is that, when the process is out of its quantum then the timer interrupt handler will know this fact and enable the need_resched value to "1" so that when the this process makes a system call then the scheduler is called. Here in the scheduler there is one tricky place, In the above code find that the most eligible process is made as current (current = next) before the system call's return value is given to the process that made a system call. So as there is no guarantee that the process that made the system call will get the return value (Super User register R0) as it may be replaced by some other process, the return value is saved into the current process structure and then the scheduler is called. There are two possibilities now

1. The scheduler has done nothing that is it selected the prev process as current process; as there is no other process more eligible than this one. Then it again restores the return value into the register R0 and when the kernel returns to the user then user reads this R0.
2. The scheduler selects some other process as current process. Then when the scheduler is done then again kernel restores the return value into register R0, but this time from the newly selected process structure which has the return value related only to that process, which is again correct.

### 8.5.6 Goodness of a Runnable Process

The heart of the scheduling algorithm includes identifying the best candidate among all processes in the runqueue list. This is what the goodness ( ) function does. It receives as input parameters prev (the descriptor pointer of the previously running process) and p (the descriptor pointer of the process to evaluate). The integer value c returned by goodness( ) measures the "goodness" of p and has the following meanings:

**c = -1000**

p must never be selected; this value is returned when the runqueue list contains only init_task.

**c = 0**

p has exhausted its quantum. Unless p is the first process in the runqueue list and all runnable processes have also exhausted their quantum, it will not be selected for execution.

**0 < c < 1000**

p is a conventional process that has not exhausted its quantum; a higher value of c denotes a higher level of goodness.

**c >= 1000**

p is a real-time process; a higher value of c denotes a higher level of goodness.

The goodness ( ) function is equivalent to:

```
if (p->policy != SCHED_OTHER)
return 1000 + p->rt_priority;
if (p->counter == 0)
return 0;
if (p->mm == prev->mm)
return p->counter + p->priority + 1;
return p->counter + p->priority;
```

If the process is real-time, its goodness is set to at least 1000. If it is a conventional process that has exhausted its quantum, its goodness is set to 0; otherwise, it is set to p->counter + p->priority.

A small bonus is given to p if it shares the address space with prev (i.e., if their process descriptors' mm fields point to the same memory descriptor). The rationale for this bonus is that if p runs right after prev, it will use the same page tables, hence the same memory; some of the valuable data may still be in the hardware cache.

# 9 TESTING ENVIROMENT AND TEST CASES

In this chapter effort are made to describe how the kernel image is actually build and tested for its correct functionality. Also given the information regarding how to execute a program and the things that should be done to add a new program. The resources that were available while development and testing are mentioned and it will seen how these are used to testing/hunting and debugging.

## 9.1 Building the Kernel Image

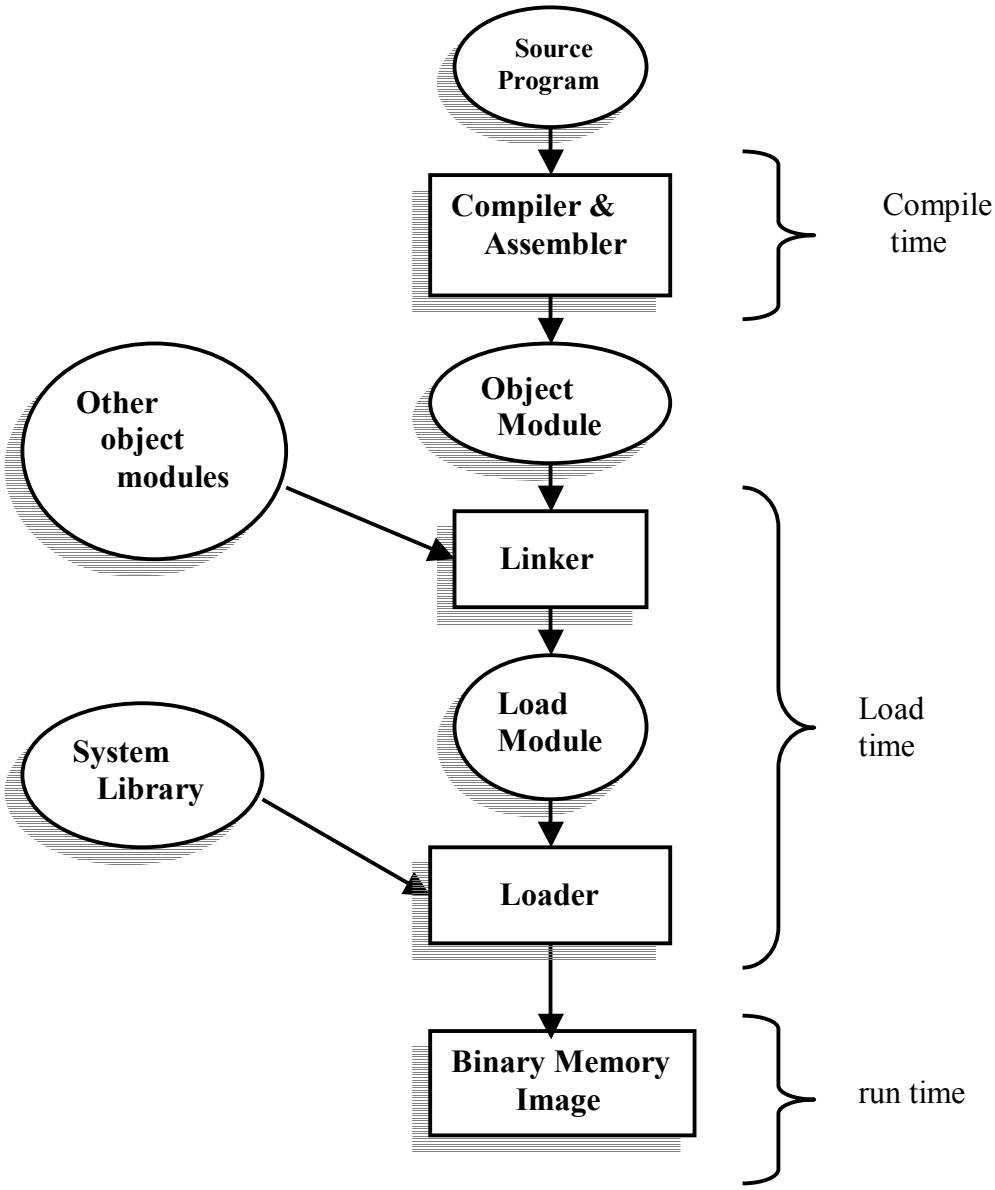


**Figure19 Multi-step Processing of a User Program [6]**

The above figure shows all the steps that are required to get the binary output of the kernel which is written in many different files. Even though some of the modules in the kernel are written in assembly code, it is written in a compiler compatible way using the macro "ASM" which allows adding inline assembly codes inserted to C codes. All the user applications are embedded into the image but difference is made by assigning the values in CCB registers to know the boundary between the kernel and user programs, so that access rights are changed, i.e., kernel given super user privileges and the rest are not.

## 9.2 Start-up Routine

In the startup routine in the file /kernel/boo.x ( .x is used as the Makefile's automatic cleaning will remove all the .s/.S files, that is the reason something other than .S was used). The core when restarts starts to execute the instruction residing at address location 0x0000H, this does mean that the program that starts from this location has the super user privileges and can make the core to do anything. Kernel is supposed to be a good guy, so it starts configuring all CCB (Core Control Block) registers, so as to make sure the core restarts in the way the kernel thinks to be better. So the things that happened here are

- The base address is used (0x10000H) and the registers which store the address of the all the sort of handlers are assigned appropriate values of the respective handlers (Scall, interrupt and exception handlers).
- All the interrupt modes, masks, external interrupts and co-processor interrupt registers initialized.
- Memory bounds set for both Instruction and Data Memory are set and the access privileges set (that is which part of the bounds can a user access and which they cannot). These is done in the register MEM_PCONF
- Timer registers which hold the maximum number that each register can count.
- Timer configuration register used to enable and disable the hardware timers. The same registers are configured so as to make Hardware Timer0 active and disable the Timer1; detailed description is given in the chapter "Timing Measurement" chapter 7.
- Stack pointers and Frame pointers (register 27 (R27) and register 28 (R28) are initialized).
- Jump to main program where the rest of the initialization about process structure initialization is done in C language and can be found in the file /kernel/main.c

## 9.3 Program Execution

The kernel has to know much about the tasks that are need to be run on the core. All the structures are to be build about the task so that these structures are used by the kernel to update the information regarding what each task is doing. This building of task's structures (Process Descriptors) is done manually in the file */kernel/main.c*. A glance into this file will show that here some structures are created for each task and the programmer's initial knowledge about the tasks is given to the kernel. Once this knowledge transferred, then the kernel take cares of them by updating the structures with the relevant information.

### 9.3.1 Memory Requirement Limitations

It is told that all the initial information regarding all the tasks is passed to the kernel by the programmer (section 9.3); here the data memory requirement information is also given to the kernel. The kernel assumes that this process takes no more memory than it knew it would use. So it reserves the specified memory, the immediate free space (unreserved to any process) would be given to the next process asking for memory. It is important to know that the kernel doesn't maintain any linked list of these assigned chunks of memory neither does it offer any protection to these boundaries (for example using some fence registers etc …). All the embedded programs that are to be run under the CUP-OS would be written within the team with the knowledge of these limitations and hence the operating system trusts them. Any invalid access to other processes memory space would corrupt its data and in the worst case if the corrupted data has some Frame Pointer or Stack Pointer value then it would make possibly make the system crash.

### 9.3.2 Starting Kernel and Tasks

Kernel Start point is the function *maintain( )*, even though it is a function with return type *int* it never returns, i.e., the *return* instruction in it is never reached. All the initialisation of all structures is done here and the kernel at this time already knows all regarding the processes. All the start addresses of the tasks are hand calculated and even this information is given to the kernel through the process descriptors. *task1* (first task) is selected as current process and before the kernel returns to the user all the user registers are restored into register set 1 (user registers) along with some registers in the super user set which are PSR, SPSR and the *link register* R31 (set2) along with the control register CR0 is restored. So when *RETU* instruction is executed by the kernel after restoring the registers then the PC (Program counter) jumps to the value stored in link register (appropriate values are stored in R31 before *retu* instruction).

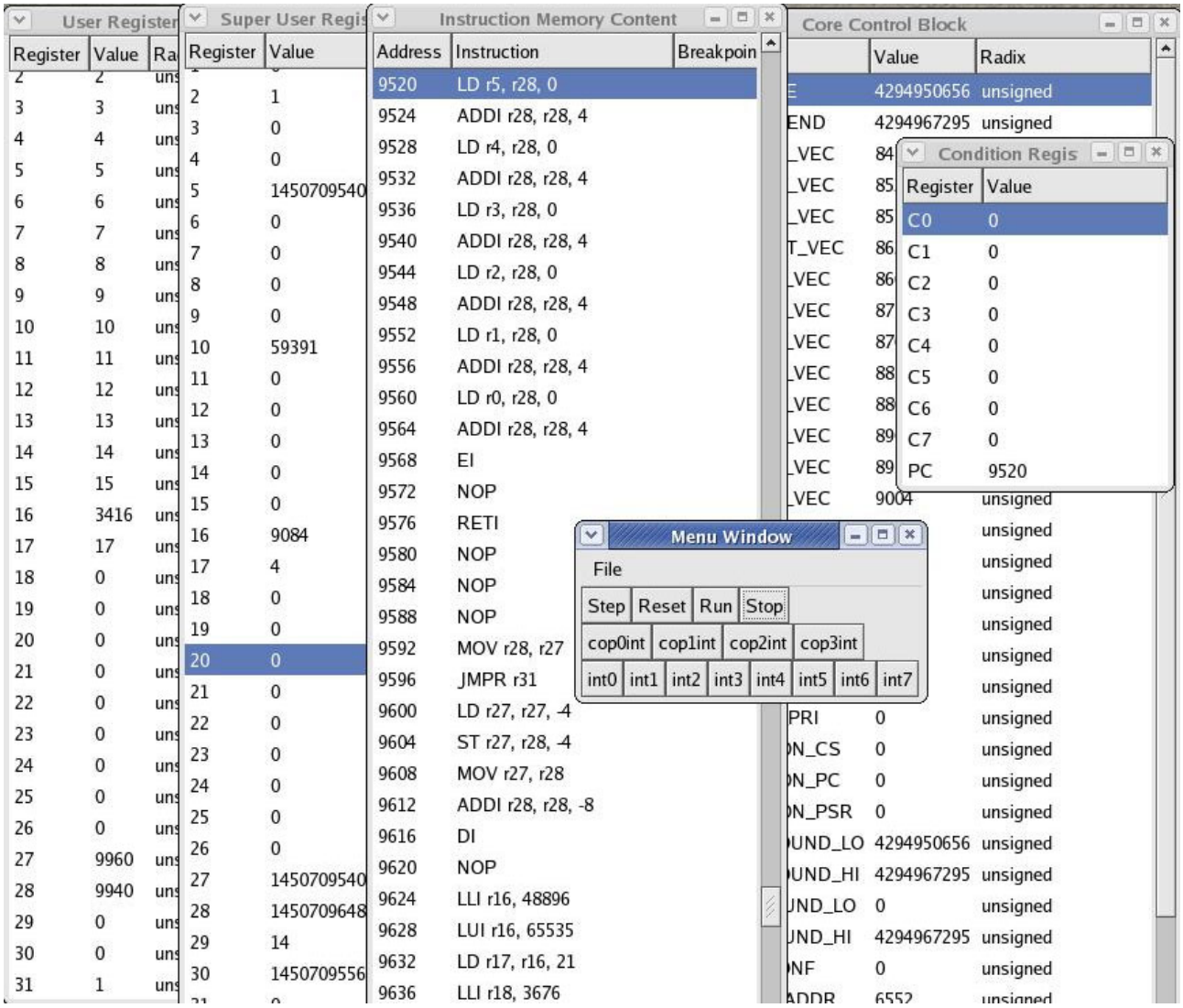


**Figure 19: Screenshot of the ISS (Instruction Set simulator for COFFEE RISC Processor)**

# 10 Conclusions

The problems faced while starting this work which were stated at the beginning of this document chapter 1, are checked against the different chapters and solutions suggested throughout this document. Next, these results are summarized and final conclusions are drawn. The question whether the suggestions in this document can be applied to other networks is briefly discussed and finally, future development of the proposed system and the area in general, is considered.

## 10.1 Problem Formulation Revisited

- Definition and characterizing the real time operating system in general
    - The whole of chapter 2 is dedicated for giving the real time general operating system concepts. Also the later chapters describe the same concepts from the implementation point of view.
- Structuring and categorizing the existing scheduling and memory security methods in real-time operating systems.
    - The algorithm used for the scheduler has been placed into categories in the section 8.5. It is important to note that there are many scheduling algorithms used in different versions of Linux and one can find that some other algorithm could have been efficient than the one used here, but efforts have been made to keep the codes look very close to the uClinux even though at many situations functionality was changed and limitations added. These limitations will be removed in the future works.
- Detail explanation of how and where processor dependent codes are mapped to the existing versions of the uClinux.
    - The processor dependent codes come into picture only when there is switch from user to kernel or vice versa. Device drivers ask a lot of services from the kernel and most of them are processor and device dependent, but as there are no devices as the moment the only place where these switches occur are at kernel entry and exit points and the detailed explanation for this goes in chapter 6 as a whole.
- Difference in behaviour of CUP-OS to the uClinux.

- From the application programmer point of view, one should not find much difference in the kernel's behaviour except most of the system calls (kernel services) are yet to be implemented. But from the kernel developers point of view one finds the difference in the
  - way how kernel starts the tasks, which is not automatic.
  - Memory Management

  This topic has been raised and explained a bit further in the chapter 9 and more specifically in the *section 9.2* where the knowledge of the kernel starts is being given before talking more about how it is being tested. Many efforts have been made to keep the kernel codes in the way so as to make future versions compatible with the first version and to make it look like uClinux with less changes and adding more functions is much easier from this point.

## 10.2 Results

Although the area of the Linux Kernel Development is very huge and varying, it is possible to structure and categorize it to simplify discussions and further studies of the area. By using all the categorization shown in chapter 2, it can be clearly understood the minimum services expected by the kernel.

All this has aimed at fully understanding what the layer of kernel is, how to begin porting it and the problems which were raised meanwhile are pointed and the solutions to that problems for a kernel newbie like me, this document should be very useful in making understanding them.

The main target of this work was to have an OS layer which could do all the general purpose OS services and should be a Linux flavoured. By Linux flavoured it is meant that the kernel to be monolithic and an application programmer for Linux should find no differences while using this kernel. The target has been achieved and also well tested with some programs. This is a unique document as there is no thesis or work that mentions the starting point needed to start this work and the problems faced during the development. Many documents were glanced, but it was difficult to get a point to start in the big ocean of functions, it is pointed in the beginning of chapter 3, the good starting point for this work and it would save a lot of time for the developers who would like to do similar porting but do not know where to start.

Many simple and different approaches were thought and described, also the implementation steps and the problem facing while implementation were clearly explained.

The conclusion is that, as Linux is becoming very popular for its open source nature and millions of minds working on it to make it better, all the processors available, wants to have it working on it. This thesis aimed to have the kernel able to do all the minimum expected services from the kernel and at the time of this writing it was successful.

## 10.3 Future Works

It is already mentioned that even thought the application programmer finds no big difference while working on CUP-OS from the uClinux; there are dissimilarities inside it from the kernel developer point of view. The main difference is that the kernel calling system calls. Other small difference are negligible and the future works will include the efforts to remove these dissimilarities and make as many system calls working as possible. Also efforts will be made to make this port accepted in the official uClinux website and make it as one of the official ports of uClinux. This is quite a lot of work but after having reaching this stage, it feels very encouraging to see the efforts fruitful and make the work more efficient.

# References

[1] COFFEE RISC Core ™ official website, http://coffee.tut.fi/documents.html

[2]'what is Linux?' Linux Online, Inc., http://www.linux.org/info/index.html, March 29, 2005

[3] Oreilly *Understanding the Linux Kernel,* 2nd Edition, December 2002 by Daniel Bovet, Marco Cesati

[4] S.R.Ball, *Embedded Microprocessor Systems*, Second edition, Butterworth-Heinemann, 2000.

[5] H.Gomaa, *Software Design Methods for Concurrent and Real-time Systems*, First edition, Addison-Wesley, 1993.

[6] P.A.Laplante, *Real-Time Systems Design and Analysis: An Engineer's Handbook*, Second edition, IEEE Press, 1997.

[7] W.Stallings, *Operating Systems: Internals and Design Principles*, Third edition, Prentice-Hall, 1997.

[8] Chapter 11 of *Embedded Linux* by Craig Hollabaugh, published by Addison-Wesley.

[9] *Linux Device Drivers*, 2nd Edition by Alessandro Rubini, Jonathan Corbet

[10] Con Kolivas, Linux Kernel CPU Scheduler Contributor, IRC conversations, no transcript. December 2004.

[11] Andrew S. Woodhull, Andrew S. Tanenbaum. *Operating Systems Design and Implementation*, 2nd Edition. Prentice-Hall, 1997.

[12] Andrew S. Tanenbaum. *Modern Operating Systems*, Second Edition. Prentice Hall, 2001.

[13] *Linux Kernel Programming*, Third Edition by Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Dirk Verworner, Claus Schroter

[14] *Building Embedded Linux Systems*, First Edition by Karim Yaghmour, April 2003

[15] *Linux for Embedded and Real-Time Applications* by Doug Abbott

[16] *Beginning Linux Programming,* First Edition by Neil Matthew

[17] *Programming Embedded Systems* in C and C ++ by Michael Barr,1999

[18] *Building Embedded Linux Systems* (concepts, techniques, tricks and traps) by Karim Yaghmour

[19] *Real-Time Concepts For Embedded Systems*, Qing Li, Caroline Yao, June 2003

[20] *Embedded Systems Design*: An Introduction to Processes, Tools and Techniques by Arnold Berger

[21] Embedded C by Micheal J Pont, Feb 2003

[22] Real-Time Linux on the CRIS Architecture thesis, http://www.efd.lth.se/~d98mad/thesis_article.pdf