**RPI 3:**

The RPI3 is a quad core 64 bit ARM cortex A53 clocked at 1.2 Ghz.

The cortex -A53 processor is a high efficiency processor that implements the Armv8-A architecture.

Cortex -A53 has one to 4 cores, each with L1 memory system and a single shared L2 cache.

It supports TrustZone security technology.

VFPv4 floating point unit

The Cortex-A53 processor delivers significantly more performance than its predecessors at a higher level of power efficiency. This takes the performance of the core above that of the Cortex-A7 processor, which defines many popular mainstream and entry-level mobile platforms. The performance graph to the right shows the performance improvements of the Cortex-A53 processor against the Cortex-A7 processor.  
  
The Armv8-A architecture brings a number of new features. These include 64-bit data processing, extended virtual addressing and a 64-bit general purpose registers. The Cortex-A53 processor is Arm’s first Armv8-A processor aimed at providing power-efficient 64-bit processing. It features an in-order, 8-stage, dual-issue pipeline, and improved integer, NEON, Floating-Point Unit (FPU) and memory performance.

The Cortex-A53 can be implemented in two execution states: AArch32 and AArch64. The AArch64 state gives the Cortex-A53 its ability to execute 64-bit applications, while the AArch32 state allows the processor to execute existing Armv7-A applications.

////////////////////////////////////////////////////

TL;DR: The Raspberry Pi 3 Model B is out now. This latest model includes 802.11n WiFi, Bluetooth 4.0, and a quad-core 64-bit ARM Cortex A53 running at 1.2 GHz. It’s a usable desktop computer. Available now at the usual Pi retailers for $35.

News of the latest Raspberry Pi swept around the Internet like wildfire this last weekend, thanks to a published FCC docs showing a Pi with on-board WiFi and Bluetooth. While we thank the dozens of Hackaday readers that wrote in to tell us about the leaked FCC documents, our lips have been sealed until now. We’ve been doing a few hands-on tests with the Pi 3 for about two weeks now, and the reality of the Pi 3 is much cooler than a few leaked FCC docs will tell you.

The Raspberry Pi 3 Model B features a quad-core 64-bit ARM Cortex A53 clocked at 1.2 GHz. This puts the Pi 3 roughly 50% faster than the Pi 2. Compared to the Pi 2, the RAM remains the same – 1GB of LPDDR2-900 SDRAM, and the graphics capabilities, provided by the VideoCore IV GPU, are the same as they ever were. As the leaked FCC docs will tell you, the Pi 3 now includes on-board 802.11n WiFi and Bluetooth 4.0. WiFi, wireless keyboards, and wireless mice now work out of the box.

## Specs

The headlining feature of the Pi 3 is the built-in WiFi and Bluetooth, but it doesn’t stop there. Here’s the complete specs for the Pi 3:

* SoC: Broadcom BCM2837 (roughly 50% faster than the Pi 2)
* CPU: 1.2 GHZ quad-core ARM Cortex A53 (ARMv8 Instruction Set)
* GPU: Broadcom VideoCore IV @ 400 MHz
* Memory: 1 GB LPDDR2-900 SDRAM
* USB ports: 4
* Network: 10/100 MBPS Ethernet, 802.11n Wireless LAN, Bluetooth 4.0

# Operating System Scheduling algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

* First-Come, First-Served (FCFS) Scheduling
* Shortest-Job-Next (SJN) or **shortest job first** Scheduling
* Priority Scheduling
* Shortest Remaining Time
* Round Robin(RR) Scheduling
* Multiple-Level Queues Scheduling

These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

## First Come First Serve (FCFS)

* Jobs are executed on first come, first serve basis.
* It is a non-preemptive, pre-emptive scheduling algorithm.
* Easy to understand and implement.
* Its implementation is based on FIFO queue.
* Poor in performance as average wait time is high.

## Shortest Job Next (SJN)

* This is also known as **shortest job first**, or SJF
* This is a non-preemptive, pre-emptive scheduling algorithm.
* Best approach to minimize waiting time.
* Easy to implement in Batch systems where required CPU time is known in advance.
* Impossible to implement in interactive systems where required CPU time is not known.
* The processer should know in advance how much time process will take.

## Priority Based Scheduling

* Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
* Each process is assigned a priority. Process with highest priority is to be executed first and so on.
* Processes with same priority are executed on first come first served basis.
* Priority can be decided based on memory requirements, time requirements or any other resource requirement.

## Shortest Remaining Time

* Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
* The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
* Impossible to implement in interactive systems where required CPU time is not known.
* It is often used in batch environments where short jobs need to give preference.

## Round Robin Scheduling

* Round Robin is the preemptive process scheduling algorithm.
* Each process is provided a fix time to execute, it is called a **quantum**.
* Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
* Context switching is used to save states of preempted processes.

**Difference between Process and Thread**

|  |  |  |
| --- | --- | --- |
| S.N. | Process | Thread |
| 1 | Process is heavy weight or resource intensive. | Thread is light weight, taking lesser resources than a process. |
| 2 | Process switching needs interaction with operating system. | Thread switching does not need to interact with operating system. |
| 3 | In multiple processing environments, each process executes the same code but has its own memory and file resources. | All threads can share same set of open files, child processes. |
| 4 | If one process is blocked, then no other process can execute until the first process is unblocked. | While one thread is blocked and waiting, a second thread in the same task can run. |
| 5 | Multiple processes without using threads use more resources. | Multiple threaded processes use fewer resources. |
| 6 | In multiple processes each process operates independently of the others. | One thread can read, write or change another thread's data. |

## Advantages of Thread

* Threads minimize the context switching time.
* Use of threads provides concurrency within a process.
* Efficient communication.
* It is more economical to create and context switch threads.
* Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

## Types of Thread

Threads are implemented in following two ways −

* **User Level Threads** − User managed threads.
* **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

## User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

### Advantages

* Thread switching does not require Kernel mode privileges.
* User level thread can run on any operating system.
* Scheduling can be application specific in the user level thread.
* User level threads are fast to create and manage.

### Disadvantages

* In a typical operating system, most system calls are blocking.
* Multithreaded application cannot take advantage of multiprocessing.

## Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

### Advantages

* Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
* If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
* Kernel routines themselves can be multithreaded.

### Disadvantages

* Kernel threads are generally slower to create and manage than the user threads.
* Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

## Difference between User-Level & Kernel-Level Thread

|  |  |  |
| --- | --- | --- |
| S.N. | User-Level Threads | Kernel-Level Thread |
| 1 | User-level threads are faster to create and manage. | Kernel-level threads are slower to create and manage. |
| 2 | Implementation is by a thread library at the user level. | Operating system supports creation of Kernel threads. |
| 3 | User-level thread is generic and can run on any operating system. | Kernel-level thread is specific to the operating system. |
| 4 | Multi-threaded applications cannot take advantage of multiprocessing. | Kernel routines themselves can be multithreaded. |

# Device Driver Basics

Where

Device drivers live inside the operating system. They are not indepedent programs.They do not run all the time.

What

They are instead just a set of functions that are part of a larger program, and when that larger program needs them, they are called. If the functions crash, the whole program crashes. If the functions loop forever, the OS locks up. Sometimes it just locks the device, other times it locks the whole OS.

How do they get into the OS?

There are two ways. They can either be compiled into the OS, which means they are already available, or they can be loaded as modules, which means they only use RAM when needed.   
If they are modules, there is a small bit of extra code needed.

How Many

These functions can be called multiple times at the same time. It should be possible to call the function twice at the same time and everything is OK. You can do this by having each instance have it's own variable (use variables from the stack) or by using locking.

Can I print from a device driver?

There is a 'printk' function. It's mostly like 'printf', and makes it easy to print. However, the messages don't go to the screen. They go to /var/log/messages if the system is set up normally.

Can I Write Device Drivers Over the Network?

Generally, not. If you make an error, and need to reboot the computer, then you must be in the lab. However, if you are mistake-free, it is theoretically possible.

What Language Do I Write In?

The Linux kernel is written in C, and all the header files are written in C. It's therefore easiest to write in C.

What's this Top/Bottom Half thing?

Device drivers typically have a top and bottom half. The top half gets data and commands from the kernel, and places them into a data structure shared with the bottom half. The bottom half gets called when there is an interrupt, and places answers in the common data structure. This common data structure is protected from simultainious access via locking.

How Do Device Drivers Get Called

An application makes a system call   
The operating system decides it needs to use the device   
The operating system calls the top half of the device driver   
---OR---   
An interrupt goes off   
The operating system decides that the interrupt was associated with your device   
It calls the bottom half of the device driver.

Note that every system call does NOT automatically translate into a device call. Maybe the OS has the answer cached. Maybe the call had an error that could be detected before the device driver. Maybe something else.

What functions must I implement?

These   
struct file\_operations {   
       struct module \*owner;   
       loff\_t (\*llseek) (struct file \*, loff\_t, int);   
       ssize\_t (\*read) (struct file \*, char \*, size\_t, loff\_t \*);   
       ssize\_t (\*write) (struct file \*, const char \*, size\_t, loff\_t \*);   
       int (\*readdir) (struct file \*, void \*, filldir\_t);   
       unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);   
       int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);   
       int (\*mmap) (struct file \*, struct vm\_area\_struct \*);   
       int (\*open) (struct inode \*, struct file \*);   
       int (\*flush) (struct file \*);   
       int (\*release) (struct inode \*, struct file \*);   
       int (\*fsync) (struct file \*, struct dentry \*, int datasync);   
       int (\*fasync) (int, struct file \*, int);   
       int (\*lock) (struct file \*, int, struct file\_lock \*);   
       ssize\_t (\*readv) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);   
      ssize\_t (\*writev) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);

};

**Major Number and Minor Number:** The kernel uses the major number at open time to dispatch execution to the appropriate driver. The major number tells you which driver is used to access the hardware. Each driver is assigned a unique major number. The minor number is used by the driver to distinguish between the various hardware it controls.

# stack overflow

A stack overflow is an undesirable condition in which a particular computer [program](https://searchsoftwarequality.techtarget.com/definition/program) tries to use more [memory](https://searchstorage.techtarget.com/definition/memory-card) space than the call stack has available. In programming, the call stack is a [buffer](https://whatis.techtarget.com/definition/buffer) that stores requests that need to be handled.

The size of a call stack depends on various factors. It is usually defined at the start of a program. Its size can depend on the [architecture](https://whatis.techtarget.com/definition/architecture) of the computer on which the program runs, the language in which the program is written, and the total amount of available memory in the system. When a stack overflow occurs as a result of a program's excessive demand for memory space, that program (and sometimes the entire computer) may [crash](https://whatis.techtarget.com/definition/crash).