Morbo will now introduce tonight's candidates…
PUNY HUMAN NUMBER ONE
PUNY HUMAN NUMBER TWO
and Morbo's good friend

Linux

ENGN8537

Embedded Systems

Overview

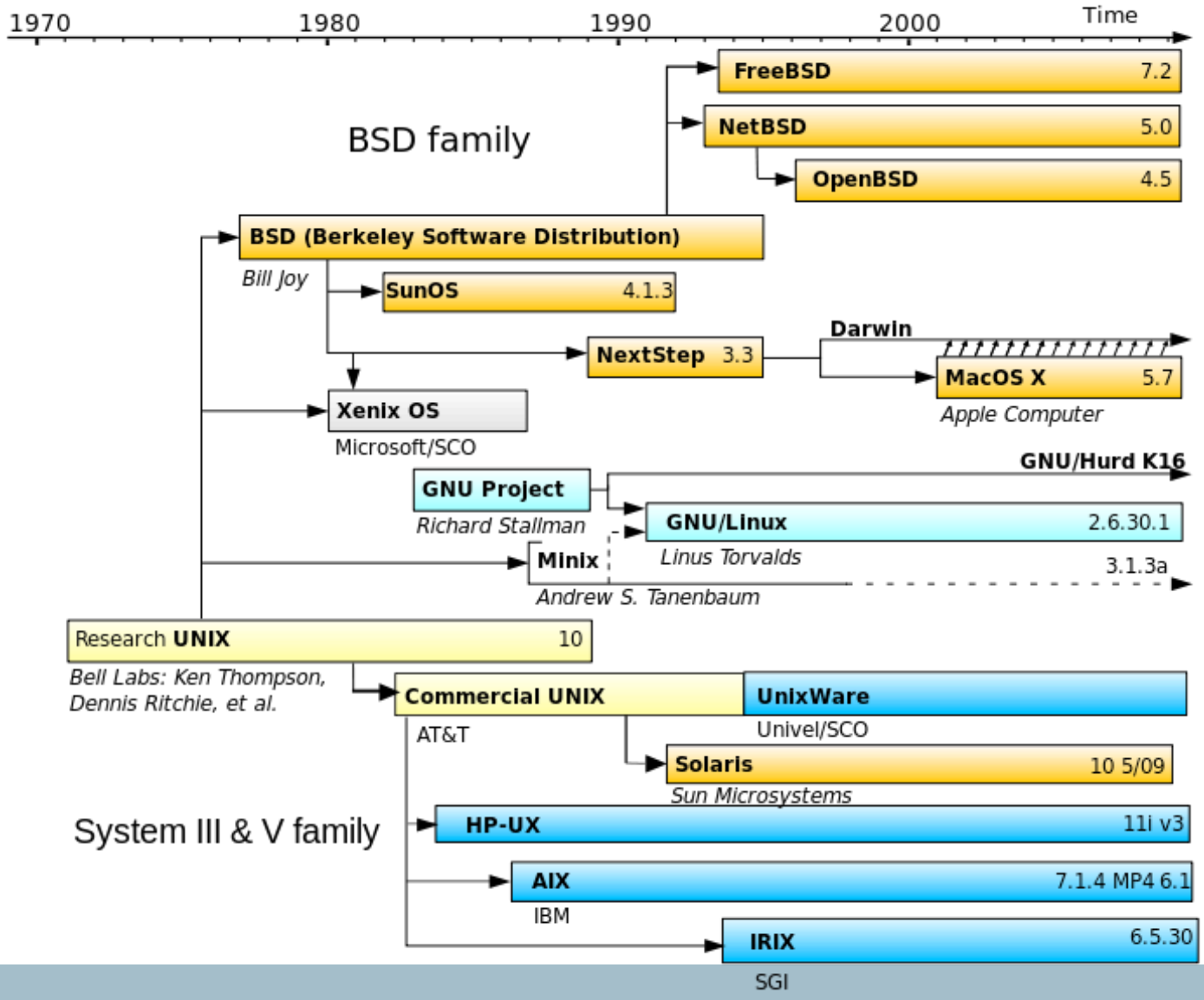
- Intro to Linux
- Device Model
- Scheduler

References:

Robert Love, *Linux Kernel Development (2nd Edition)*, Pearson Education 2005

Linux Kernel in-tree documentation (“Documentation/”)

Introduction to Linux



Introduction to Linux

Linux was first released in 1991. It is an Operating System Kernel, not a whole operating system. The whole system is typically referred to as a Linux Distribution or, if packaged with a specific set of userspace programs, GNU/Linux.

The Linux Kernel is also the core of the Android Operating System. Android is the only mainstream use of the Linux Kernel in anything other than a “traditional” GNU/Linux style operating system.

The Kernel is “Monolithic” in style (regardless of the use of Kernel modules) in that is has very clearly defined kernel and userspace portions and no low level access at all is permitted from the userspace portion.

Introduction to Linux

Linux is commonly used on embedded devices for a number of reasons:

- Cheap (no per-unit costs)
- Configurable
- Runs on more processors than any other kernel ever has (26 official at last count, many more unofficial)
- Has a huge range of libraries and tools available
- Provides a familiar programming environment for those coming from non-embedded backgrounds
- Fast (compared to other big operating systems)
- …

Introduction to Linux

It was never designed for real time use, so applications in embedded systems must be approached with care.

There are a number of efforts that have gone in to making Linux more suitable for hard real time tasks. One approach is to run Linux inside another real time executive (RTLinux, RTAI).

The less radical approach is found in the “PREEMPT_RT” work that is sponsored by many commercial embedded Linux vendors. Later in the lecture we will examine some examples of where this works and where it doesn’t, but for now a quote to set the mood from someone who ought to know:

Introduction to Linux

"Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with your using PREEMPT_RT."

-- Linus Torvalds

Linux Device Model

When getting Linux to boot on a new board, the following steps must be accomplished:

- Support for the processor architecture used
- Support for the processor type used
- Support for the particular board used
- Support for the devices on the board

Linux Device Model

We will assume that the architecture work has been done and that the devices on the board have their drivers written. We will examine what might have to be done to write a Board Support Package (BSP).

A BSP is a selection of code and other binary or plain text files that describe the operation of a particular board to the kernel.

The BSP should also include board documentation, build environment and userspace code required to operate the board such as the graphical environment.

That stuff is boring*, we will examine only the kernel portion of the BSP

* but necessary

Linux Device Model

The fundamental questions that have to be answered by a board support package are:

1. What devices are present
2. Where they are, both in terms of bus and address
3. Which drivers are required to talk to them
    - Some devices also require that the kernel load firmware in to them before they can operate. We will lump firmware configuration in with the drivers here.

We will ignore devices that can be automatically detected (USB devices for example) and focus only on describing parts that are connected in ways that can't be automatically detected.

Linux Device Model

The concept of a BSP is unusual to people who are used to building their own x86-based computers. In those machines, you can connect anything where ever it fits and at the next boot, the hardware should be automatically detected.

This is true for two reasons:

1. The BIOS on the motherboard is preloaded with information regarding the board itself. Stock-standard code can be used in the kernel to read and act on this (through commands called ACPI commands)
2. All the busses within an x86 computer (SATA, PCIe, USB etc.) have functionality to probe locations and/or to auto-detect device arrival and removal.

Linux Device Model

It hasn't always been true.

Anyone ever fit an ISA card?

The address and interrupt number of each card had to be manually selected by means of jumpers on the boards. If it was incorrectly configured, the system would crash.

Modern busses such as PCI either have dedicated resources for each physical slot, or some sort of arbitration and control algorithm to assign unique resources to each device at boot time.

Linux Device Model

A device is a physical chip (or part of one) which is connected in some way to the board, for example:

- a Network Card
- a Programmable Interrupt Controller
- a Memory Bank
- even information regarding the CPU and its features

It may have a set of parameters unique to it, such as the number of bits on a parallel port, the capacity of the memory etc.

Linux Device Model

Each device is connected to a particular bus (eg USB, PCIe).

The bus is conceptually the part of the system that makes the link between the purely software driver and the physical device.

The bus will be responsible for

- Discovering the new device (if it has an autodetect feature)
- Pairing the device with its driver
- Managing data transfers between driver and device

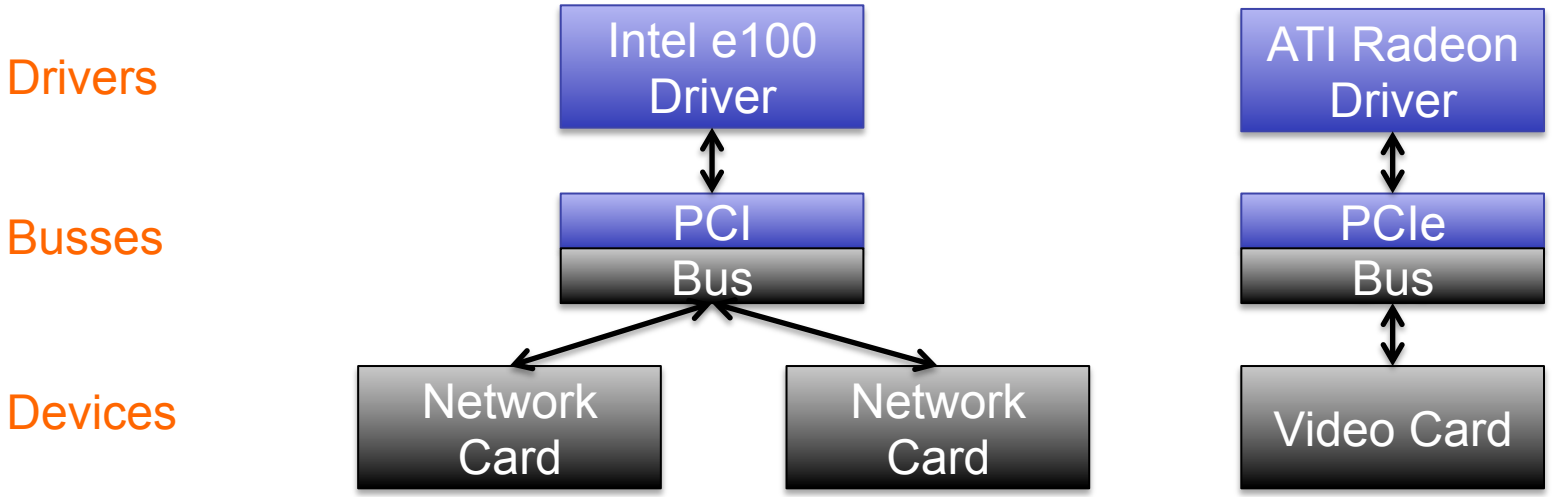
If a device doesn't have a formal bus then it is connected to the "platform bus", a fake bus that provides skeletal services for on-chip peripherals.

Linux Device Model

A driver is the piece of software that has been written to interface with a device.

If all the devices on the computer are operational, they each have associated with them exactly one driver.

Each driver may have several devices attached.



Linux Device Model

We've already seen how to create devices on a soft core processor.

On the Nios, all the devices were on the same bus so there was no need to specify it explicitly

Device

Memory Address

Interrupt Number

Use	C...	Name	Description	Clock	Base	End	IRQ	Tags
☑		cpu	Nios II Processor	clk	0x10000800	0x10000fff		
☑		sdram	SDRAM Controller	clk	0x00000000	0x07ffffff		
☑		Pushbuttons	Parallel Port	clk	0x10000050	0x1000005f		
☑		Slider_Switches	Parallel Port	clk	0x10000040	0x1000004f		
☑		Green_LEDs	Parallel Port	clk	0x10000010	0x1000001f		
☑		Red_LEDs	Parallel Port	clk	0x10000000	0x1000000f		
☑		HEX3_HEX0	Parallel Port	clk	0x10000020	0x1000002f		
☑		HEX7_HEX4	Parallel Port	clk	0x10000030	0x1000003f		
☑		sysid	System ID Peripheral	clk	0x10002020	0x10002027		
☑		Interval_Timer	Interval Timer	clk	0x10002000	0x1000201f	0	
☑		JTAG_UART	JTAG UART	clk	0x10001000	0x10001007	1	

Linux Device Model

As discussed, x86 uses a BIOS and ACPI to describe where things like the PCI controllers are then auto-detectable busses take care of everything from there.

The only other scheme that has wide acceptance is the Flattened Device Tree concept.

This was developed by IBM (in Canberra) for use on PowerPC computers

Linux Device Model

A portion of a device tree is shown here representing a CPU, some memory, the interrupt controller and the PCI controller.

For each device, you have a type, a location and properties specific to that device.

All these devices are directly connected to the processor and therefore on the “Platform Bus”. If there were an Ethernet card or similar on the list, it would have a bus descriptor as well.

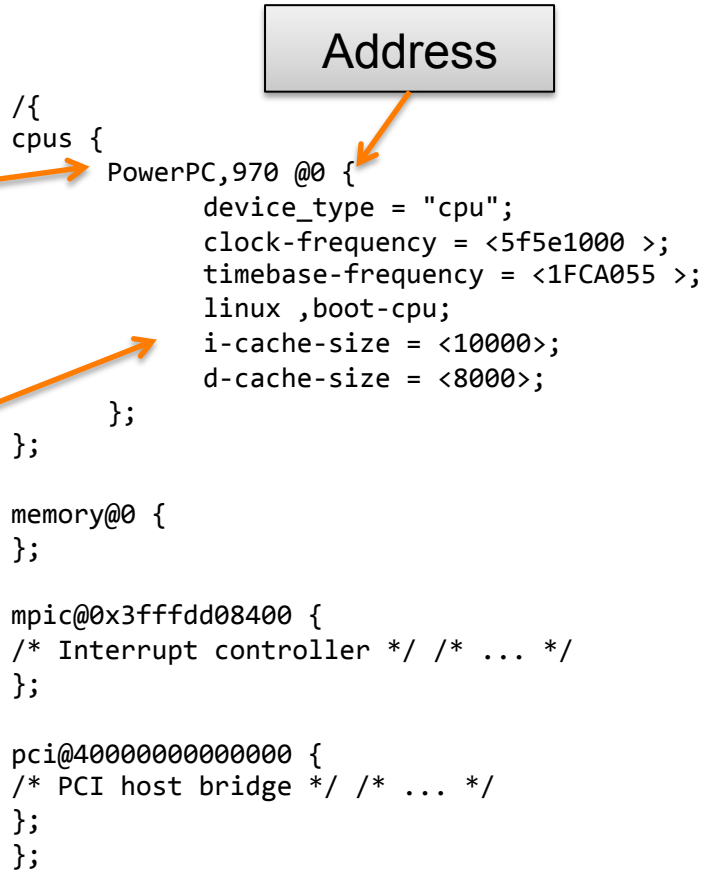
```
/{
cpus {
      PowerPC,970 @0 {
            device_type = "cpu";
            clock-frequency = <5f5e1000 >;
            timebase-frequency = <1FCA055 >;
            linux,boot-cpu;
            i-cache-size = <10000>;
            d-cache-size = <8000>;
      };
};

memory@0 {
      device_type = "memory";
};

mpic@0x3fffdd08400 {
/* Interrupt controller */ /* ... */
};

pci@40000000000000 {
/* PCI host bridge */ /* ... */
};
};
```

Linux Device Model

A portion of a device tree is shown here representing a CPU, some memory, the interrupt controller and the PCI controller.

For each device, you have the location and properties specific to that device.

All these devices are directly connected to the processor and therefore on the “Platform Bus”. If there were an Ethernet card or similar on the list, it would have a bus descriptor as well.



Linux Device Model

The third option for describing a particular board is to write C code that exactly describes that board and compile it in to the kernel.

This is the way that most embedded systems have traditionally been described as it's simple and efficient.

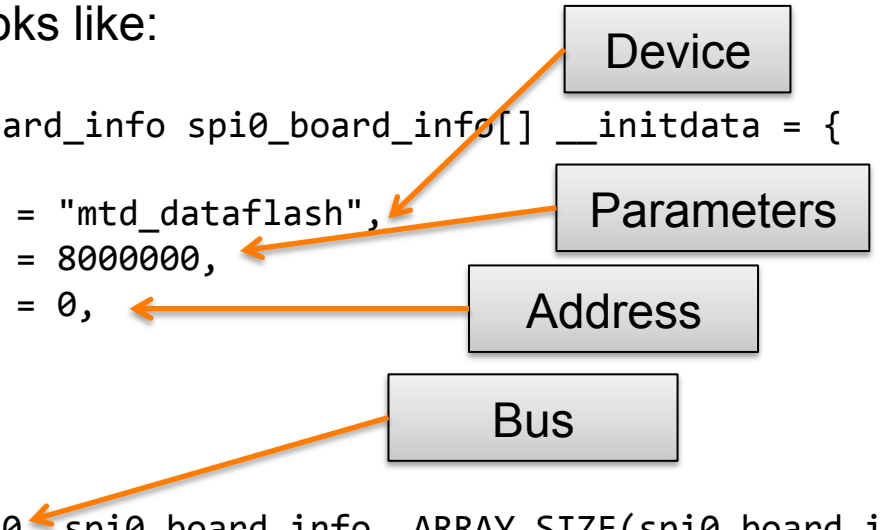
It has the significant down side that you have to completely recompile the kernel each time you add a new device to your board.

It also leads to significant bloat of the source tree as you have to keep code describing every variation of every board that has ever been supported by the kernel.

Linux Device Model

Let's see what it looks like:

```
static struct spi_board_info spi0_board_info[] __initdata = {
{
     .modalias       = "mtd_dataflash",
     .max_speed_hz   = 8000000,
     .chip_select    = 0,
},
};


at32_add_device_spi(0, spi0_board_info, ARRAY_SIZE(spi0_board_info));
```



You need one of these for every device on the board! At least, every device that can't be automatically detected.

Linux Device Model

In general, Linux Kernels for embedded devices certainly aren't plug and play.

This is why Linux distributions (such as Ubuntu) have trouble supporting architectures other than x86 and, sometimes, PowerPC: The kernel built is specific to the exact hardware set up of the target board.

Surely there's a better way?

Linux Device Model

Surely there's a better way?

Many ARM platforms are moving to FDT just as PowerPC has done. This will allow standardised kernels to boot on different machines, just as your favourite desktop Linux distribution does now.

Vendors that provide Embedded Linux services might also offer build services. This means that they will provide a simple interface that presents all the possible hardware choices, you enter some details about how it's connected and click OK. Their beefy computers build you a kernel with the right set up and make it available to you.

This is great so long as the device you are trying to use has a driver in their automated system!

Linux Device Model

Device Model Summary

- Devices are physically on the board
- Drivers are the software that is written to communicate with (a class of) device(s)
- Busses have both hardware and software components that wire the two together
- If your devices are not on an auto-detectable bus then you must manually tell the kernel what the device is, and where it is via either
  - BIOS (x86)
  - FDT (PowerPC, some ARM)
  - “Board Code”
- Just because a device has a Linux Driver doesn’t mean it’s going to automatically work when connected. It will quite likely have to be “plumbed in”

Scheduling in Linux

The Linux Kernel doesn't differentiate between Tasks and Threads in the same way that we have been up until now.

Recall:
A thread is a schedulable entity with implicitly shared resources (e.g. memory). Threads may communicate with simple shared memory constructs.

A task is a schedulable entity completely isolated from other entities in terms of resource usage. Inter-task communication requires a formal mechanism such as pipes, sockets or explicitly shared memory regions.

Scheduling in Linux

Linux has a single concept: Processes.

When a process is created, the programmer must elect which resources are to be shared from the parent (the code making the new process) and the child (the new process itself).

Such resources might include:

- Memory, as discussed previously
- File descriptors, the numbers used to identify open files and pipes
- File system roots, what each process thinks of as the ‘base’ of the filesystem hierarchy
- etc.

Scheduling in Linux

Processes are created on Linux using the `clone()` system call. More usually, the programmer will call `fork()` which uses clone internally. Fork creates what we would think of as Tasks, in that parent and child will not share memory once they're running.

These functions are interesting: You call them once, they return twice. From the point of view of the parent, fork returns the process identifier of its new child. From the point of view of the child, fork returns zero.

```
if (fork()) {
    /* This code is run by the parent */
} else {
    /* This code is run by the child */
}

/* Anything after the above if statement
 * will be run by both parent and child */
```

Note: Doesn't do error handling, don't copy!

Scheduling in Linux

It's rare that you'd want any single piece of code to be executed by both parent and child, so the usual practice is to use the `exit()` system call to terminate the child before it leaves the if statement.

```
if (fork()) {
    be_parental();
} else {
    be_childish();
    exit();
}

wait_for_child_to_die();
clean_up();
exit();
```

Note: Doesn't do error handling, don't copy!

Scheduling in Linux

Note the order of exit, the parent should wait for the child to die before exiting itself.

Why?

- It's assumed the parent has started the child for a reason, it is usual for it to be around to see the results
- If the child doesn't execute correctly, the parent presumably needs to know about it

In Linux, if a parent doesn't wait for a child to die, the child might become a 'zombie' and never completely disappear. It waits around, using system resources indefinitely.

If the parent dies before the child, the orphan tasks are 're-parented' to the init task in order to maintain a (semi-) sane process hierarchy.

Scheduling in Linux

A note on task creation: What happens to existing memory?

```
int i = 7;

if (fork()) {
    i = 42; /* Parent code */
} else {
    print(i); /* Child code */
    exit();
}
```

Will the child print ‘7’, ‘42’, race to produce some random combination of both of them or will it not have any concept of the variable ‘i’ any more?

Scheduling in Linux

Forking doesn't share memory from parent to child, so it can't print 42, nor can 'race' with the parent thread to print some hybrid of 42 and 7.

It might make sense for the child to start with no variables, but in practice that's ungainly and makes it difficult to perform tasks like setting up communication mechanisms between parent and child.

The child starts off with a value of `i = 7`.

Now what happens when i is modified?

Scheduling in Linux

If the child starts with `i = 7` and that view of the variable can't be changed by the parent, then the child must have its own private copy of i.

By extension, if it has its own copy of i, it must have its own copy of every variable the parent ever created.

Every time you start a child task, you have to completely duplicate all of the parent's memory!

Scheduling in Linux

Actually not really, not if you're smart. If you're smart then the parent's memory is COW.

Key observation: "The parent and child must have their own copy of memory" is too strong; the parent and child must have their own copy of any memory that differs between parent and child.

When a parent spawns a new child, all of the parent's memory is marked "copy on write". Any future writes to memory by the parent or child will first copy the memory, then perform the required update.

Scheduling in Linux

Once processes are created in Linux, they must of course be scheduled to run on the CPU.

Linux has three scheduling classes:
SCHED_RR, SCHED_FIFO, SCHED_OTHER.

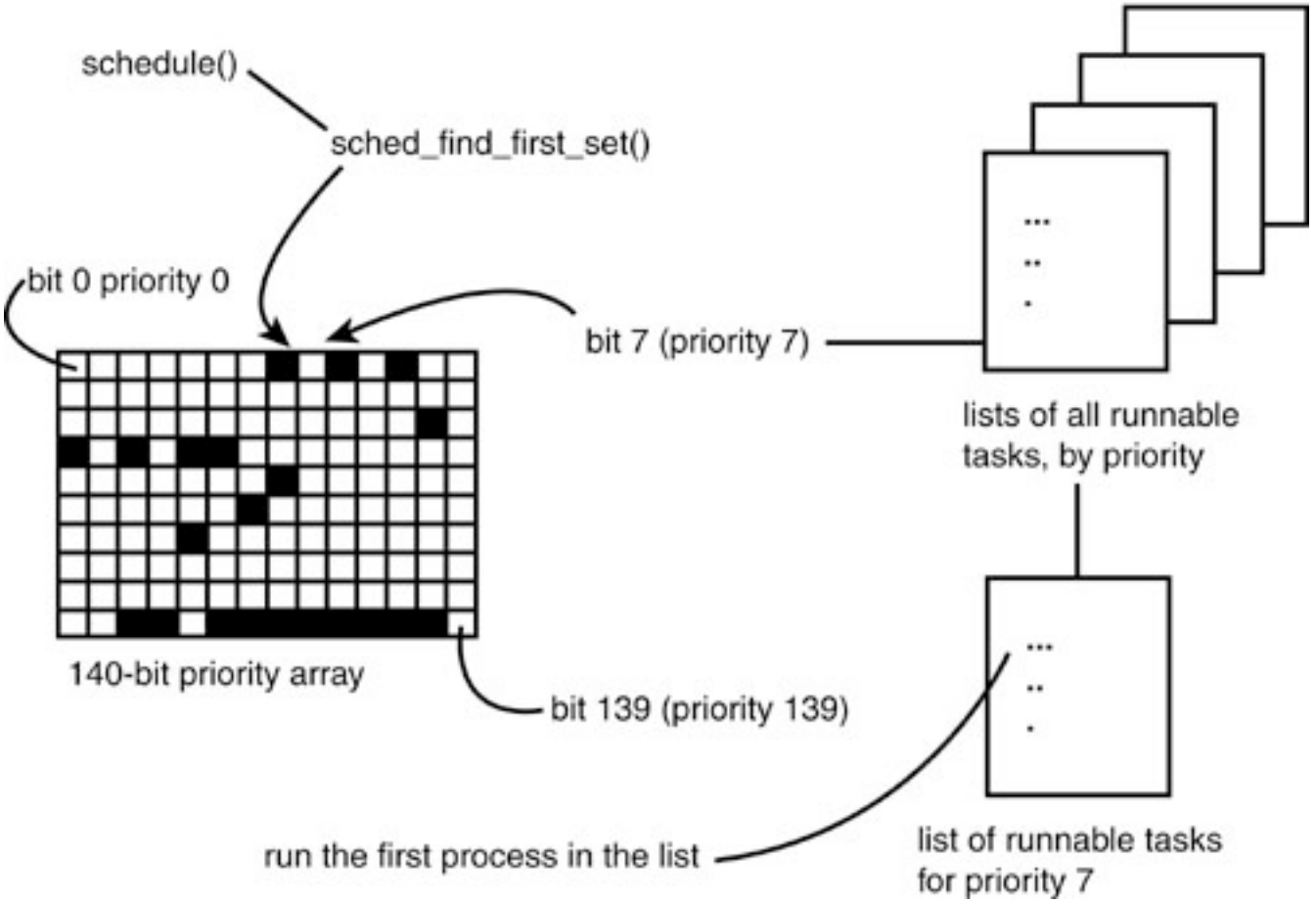
The first two are ‘real time’ by some definition, the last is the common class for every normal thread. The real time threads are scheduled first, then left over time is used for SCHED_OTHER.

Scheduling in Linux

The O(1) scheduler was introduced in the development of the 2.6.0 Linux kernel which was released in 2003. Its responsibility was to schedule the SCHED_OTHER class and its selling points were:

- Able to run in constant time, regardless of the number of processes on the system
- Worked well with multiple CPUs, both keeping them all busy but also keeping tasks on the same CPU if possible
- Good interactivity (compared to the previous scheduler)

Scheduling in Linux



Scheduling in Linux

At the time the O(1) scheduler was developed, time in the kernel was tracked in units of milliseconds or tens of milliseconds, depending on the way the kernel was configured.

In the O(1) scheduler, each process is allocated a timeslice, an amount of time the process is allowed to run until it was preempted. The timeslice varies from 5ms to 800ms depending on a processes priority and behavior.

New processes are picked to run based on their priority and whether they have any timeslice left. Once a process runs out of timeslice, it cannot be scheduled again until every other task has run out of timeslice.

Scheduling in Linux

An example:

Thread 1: Priority 1, Timeslice 10ms
Thread 2: Priority 1, Timeslice 20ms
Thread 3: Priority 2, Timeslice 30ms

Thread 3 is allowed to run until either its 30ms is up or it blocks (for example, waiting for input).

One of Thread 1 or 2 gets chosen at random to run until its timeslice expires, it blocks or Thread 3 unblocks. If Thread 2 gets run, doesn't block or get preempted, then Thread 1 will not get any CPU time at all for 20ms. This is a long time in terms of processes!

If Thread 1 and 2 both exhaust their timeslices before Thread 3 does (because Thread 3 is blocking), then the CPU may sit idle rather than refreshing the timeslices of the first two threads!

Scheduling in Linux

In order to overcome this, the O(1) scheduler implemented various heuristics to try and detect ‘interactive’ or ‘I/O-bound’ tasks as compared to ‘processor bound’ tasks.

Interactive tasks are something like a text editor that may wait for a long time before the user presses a key, but must respond quickly.

Processor bound tasks are things like video processors that will quickly use up their timeslice if allowed to, but don’t require quick responses.

Processor bound tasks require high priorities to ensure they get enough of the CPU to be productive. Interactive tasks require high priorities in order to be able to wake quickly. Obviously the concept of ‘priority’ isn’t good enough to describe this process!

Scheduling in Linux

The O(1) scheduler's heuristics modify process priority dynamically depending whether they're behaving in a processor-bound or I/O-bound manner.

On the whole, the heuristics did a good job of ensuring that the CPU was properly utilized, however there were still several workloads, both natural and malicious, that could trick the scheduler in to being unfair.

Hence, in 2007, O(1) was replaced by CFS.

Scheduling in Linux

The Complete Fair Scheduler completely rethought the scheduler for the Linux Kernel. Since the O(1) scheduler was written, the kernel had added support for High Resolution Timers, allowing actual task run time to be accounted for with nanosecond resolution, opening up new options for scheduling.

The operation of CFS can be summed up in a single sentence: CFS models the ideal multithreading processor set on real hardware.

Ideal multithreading hardware doesn't exist, so it's emulated on a real CPU quite simply: If two tasks of the same priority want to run, each should get 50% of the CPU.

Conceptually, the CPU should flick back and forwards between those two tasks as fast as possible so that neither is particularly favored over the other. Of course, in the real world, there must be some granularity here. (Why?)

Scheduling in Linux

From this premise, it's very easy to incorporate concepts of priority by splitting the CPU time in different fractions. A high priority task might have a 'virtual' run time of 0.5ms for every 1ms on it actually runs on the CPU while a lower priority thread might be 1:1. In that case, balancing virtual run times would share the processor 2:1 in favour of the high priority task without either task being able to starve the other.

It's also easy to incorporate hierarchical scheduling concepts. For example, what if you don't want two equal priority threads to have half the CPU each? What about two equal priority users?

With CFS, it's easy to group processes such that each group gets the same runtime as each other group. This allows a system administrator to isolate users and allocate CPU usage fairly, for any definition of 'fair'.

Scheduling in Linux

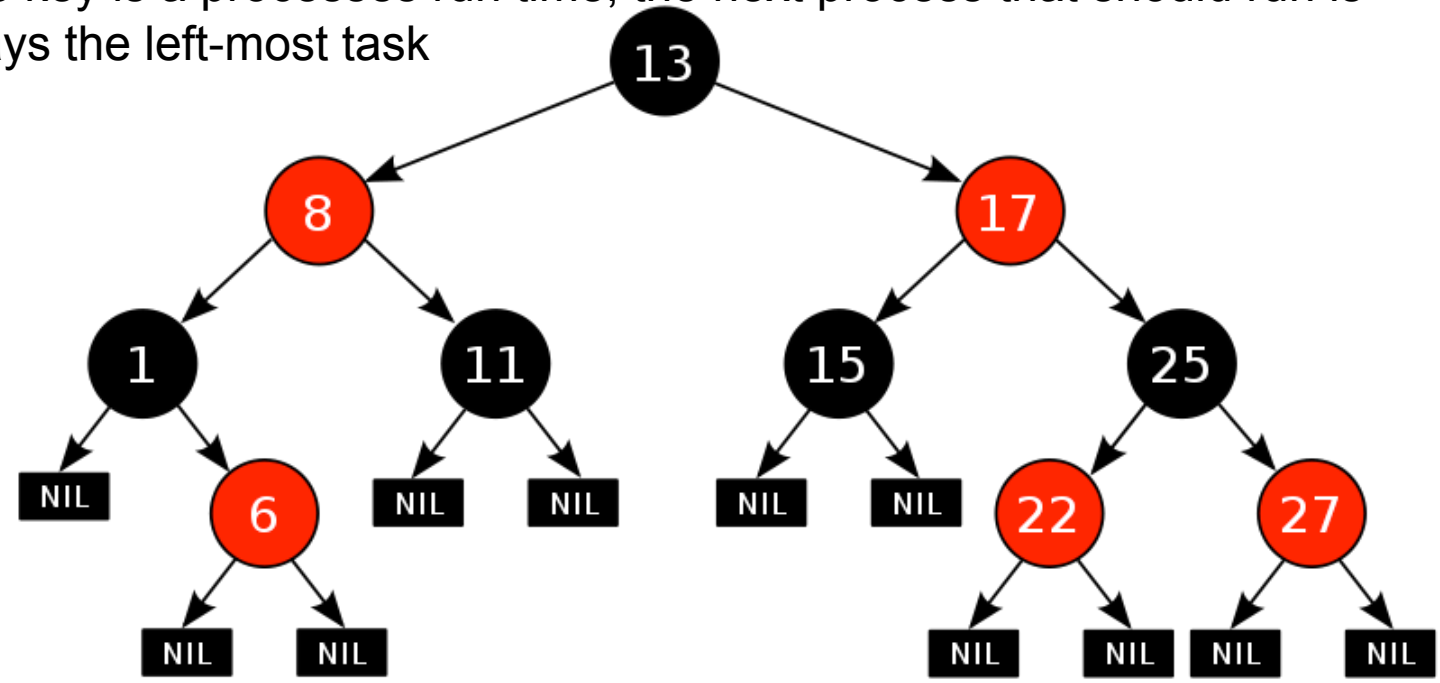
Despite CFS being ‘fair’, it isn’t real time. This is due to its response time behaviour: When a thread signals it wants to run (e.g. in response to an external event), the amount of time taken before the thread actually gets the CPU is non-deterministic.

It depends on every other thread on the system, the relative balance of their priorities and the granularity of the scheduling.

Scheduling in Linux

CFS is implemented using a time-ordered Red/Black Tree. This type of structure is a semi-balanced binary tree, with a few different rules for the insertion of nodes. Most important for us, every node has only lower values on its left, and only higher values on its right.

If the key is a processes run time, the next process that should run is always the left-most task



Scheduling in Linux

Discussion of O(1) and CFS has focused on the SCHED_OTHER class of process. There are two 'real time' classes specified by POSIX: SCHED_RR and SCHED_FIFO.

These haven't been talked about in detail because they're very simple: The highest priority task runs as long as it likes. In the case of multiple processes of the same priority, those at that level are scheduled in a Round Robin (RR) or First-in, First-out (FIFO) manner.

There is much talk about introducing deadline based scheduling for real time tasks, but at the time of writing, no such solution has been merged.

Scheduling in Linux

The RT scheduling classes are all well and good, however there are still plenty of things that can keep RT tasks off the CPU, ruin their latency and determinism:

- The Kernel itself has many internal tasks, some of which have RT priority
- Kernel interrupt handlers are relatively large and not able to be preempted, even by RT processes
- Aggressive power management means that tasks are moved around available CPUs in order to best manage power. Migration to another CPU will almost certainly change the cache behaviour
- The kernel is full of complex interdependencies and locking that can't be preempted
- Large portions of kernel code must be run with interrupts disabled, compounding the issues above.

Scheduling in Linux

Many of these points are largely mitigated by the PREEMPT_RT patch set which is maintained in order to improve the Linux Kernel real time behaviour. While there are still problems in terms of predictability, the PREEMPT_RT patchset does things like making locks more likely to be preemptable, running interrupts within special preemptable threads where possible etc.

This work is not actually driven by embedded systems, though it is useful for it.

It's largely driven by financial institutions, to whom the latency between the decision to make a trade and actually making it can translate to real money.

Scheduling in Linux

Scheduling Summary

- Linux has no explicit concept of Tasks vs. Threads, it just knows about Processes, each of which may opt to share different resources with other Processes
- There are three scheduling classes: SCHED_RR, SCHED_FIFO, SCHED_OTHER
- Most processes are SCHED_OTHER and their run order is determined by the Complete Fair Scheduler
- The “real time” levels have predictable behaviour with respect to each other, but the core of the Linux Kernel behaves badly from a real time point of view.
- Linux, particularly with the PREEMPT_RT patchset, is usable for real time and low latency requirements within specific but complicated bounds. Be careful.