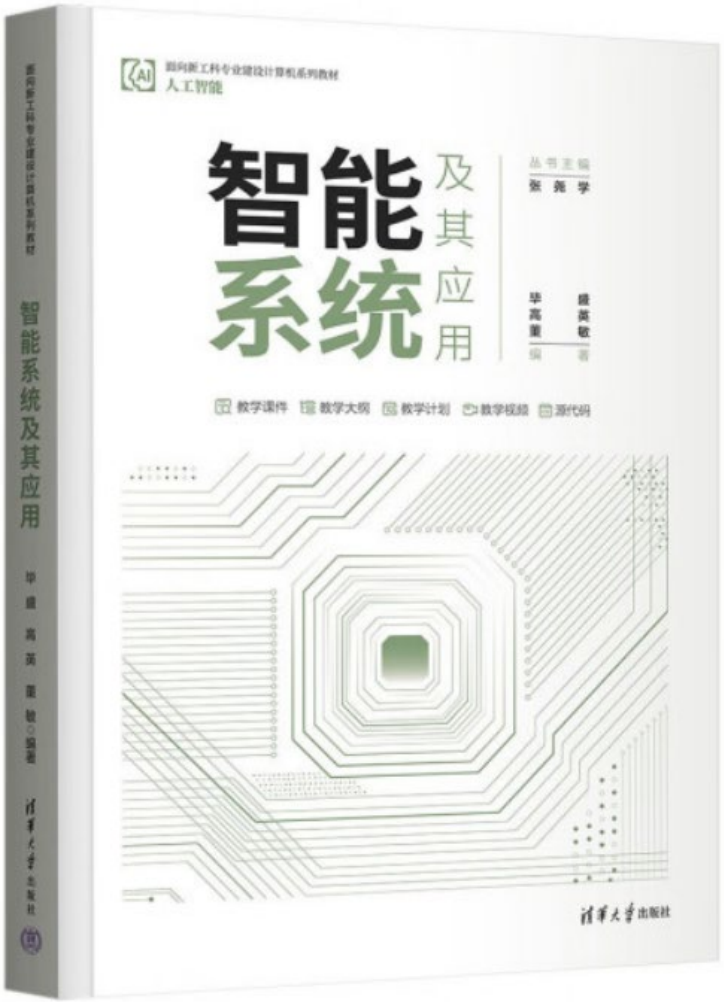
第4章
操作系统

智能系统及其应用
配套PPT

书名	书号	作者	出版社
智能系统及其应用	ISBN 978-7-302-60969-8	毕盛 高英 董敏	清华大学出版社



第4章 操作系统

4.1 操作系统概述

4.2 操作系统基础

4.3 Linux操作系统

4.4 LiteOS操作系统

4.1 操作系统概述

操作系统（operating system，简称OS）是管理计算机硬件与软件资源的计算机程序。操作系统需要处理如管理与配置内存、决定系统资源供需的优先次序、控制输入设备与输出设备、操作网络与管理文件系统等基本事务。

一种是针对具有内存管理单元（MMU），主要在微处理器平台上（X86,ARM Cotex-A系列和RISC-V64）运行的操作系统，常见的有Windows，MacOS，UNIX和Linux操作系统等。

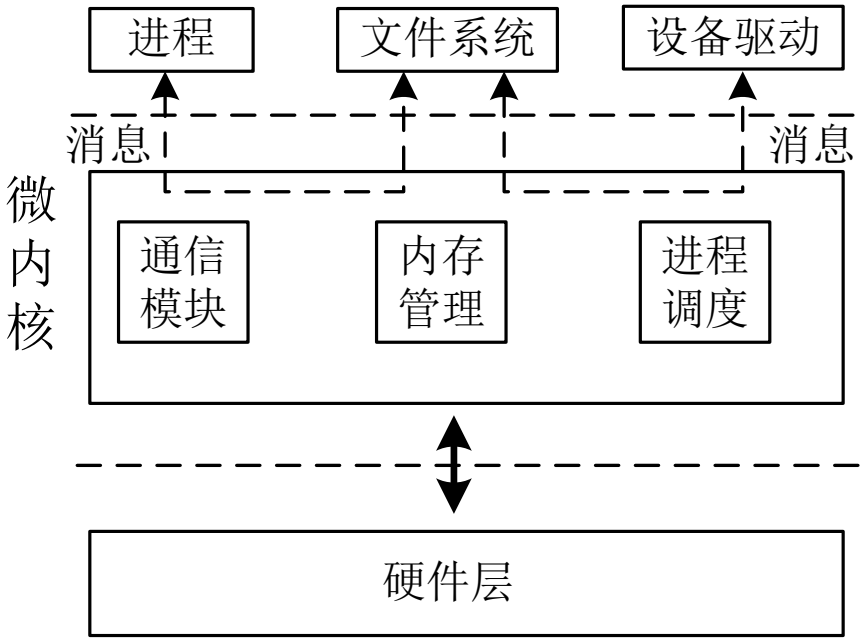
一种是可以运行在没有MMU的微控制器平台上，例如LiteOS，uCOS-II，FreeRTOS和RT-thread等实时操作系统。

4.2 操作系统基础

4.2.1操作系统内核架构

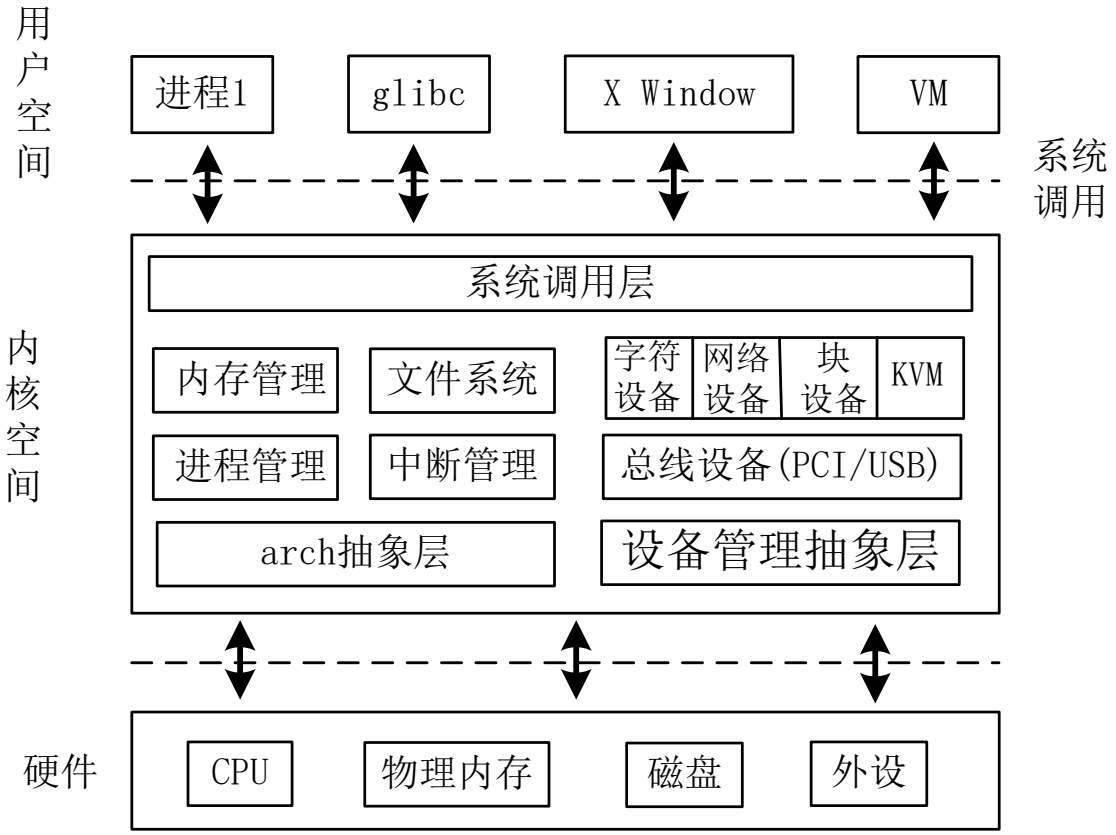
1.微内核架构

内核仅仅保留为这些服务提供通信等基础能力，使其能够互相协作以完成操作系统所必需的功能，这种架构称为微内核。但是微内核架构最大的问题就是高度模块化带来的交互的冗余和效率的损耗。基于微内核的操作系统有Mash, MINIX3, seL4和Fuchsia等。一个微内核架构示意图如下图所示。



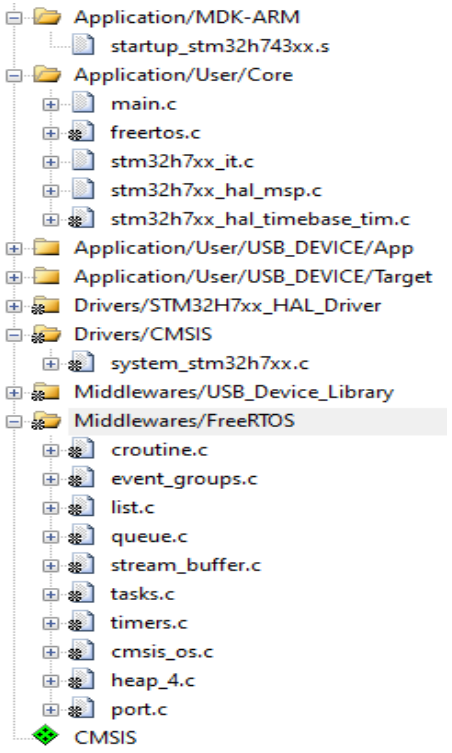
2.宏内核架构

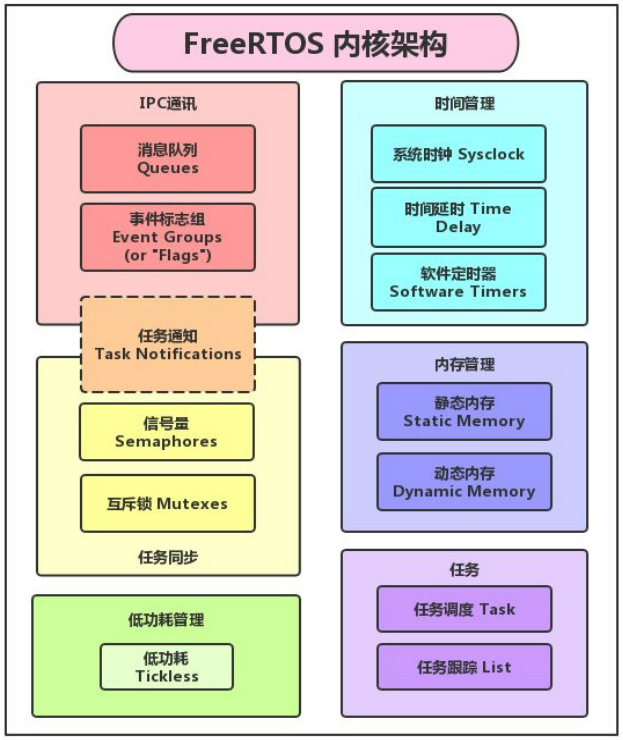
宏内核又称为单内核，其所有模块包括进程调度、内存管理、文件系统和设备驱动等，均运行在内核态，具备直接操作硬件的能力，例如操作系统 Linux，UNIX和FreeBSD都属于这类操作系统，会有类似arch/arm/的目录，用于封装与体系结构相关的功能实现。



3. 简要结构内核

当前小型嵌入式操作系统如uCOS-II和FreeRTOS虽然很小，但是不具备现代意义上的操作系统功能，包括虚拟内存、用户态和内核态分类等，因此它们并不是微内核架构，可以归类为简要结构。此结构没有分离开用户态和内核态，内核作为一个API接口文件，通过主程序调用相应内核接口函数实现操作系统的管理。此结构的优势在于，应用程序对操作系统服务的调用无须切换地址空间和权限层级，因此更加高效，但是缺点是任何一个操作系统模块或应用出现了问题，均可能使整个系统崩溃。一些面向微控制器的小型的嵌入式操作系统，完成的任务较为单一，整个系统架构较为简单，因此常常采用这种结构。





4.2.2 操作系统调用POSIX标准

在通用的操作系统中（除简要结构内核外），内核空间和用户空间多了一个中间层，这一层次称为系统调用层。这就把用户态和内核态做了较好的隔离，有如下好处：（1）应用程序开发者可以从硬件设备底层解脱出来，不用关心硬件结构是什么，只要调用接口函数即可。（2）内核可以通过系统调用层对应用程序访问进行约束，从而避免应用程序不正确的访问内核。（3）可以一套应用程序在不修改代码的情况下，在不同的操作系统或者拥有不同硬件架构的系统中重新编译并且运行，因此具有很好的移植性。

在UNIX/Linux生态中，最通用的系统调用层接口是POSIX(Portable Operating System Interface of UNIX)标准，POSIX标准定义了操作系统应该为应用程序提供的接口标准。Linux基本上逐步实现了POSIX兼容，Linux操作系统的API是以C标准库的方式提供的，其中C标准库提供了POSIX的绝大部分API的实现。

兼容POSIX标准的优势，可以让各种应用软件更加容易在本操作系统上进行移植，从而可以扩大操作系统的开发生态圈。因此那些不兼容POSIX标准的简要结构操作系统内核，有的已开始兼容POSIX标准的开发工作。

4.2.3进程管理

操作系统需要同时运行多个程序，为了管理这些程序的运行，提出了进程（process）概念，每个进程对应一个运行的程序。

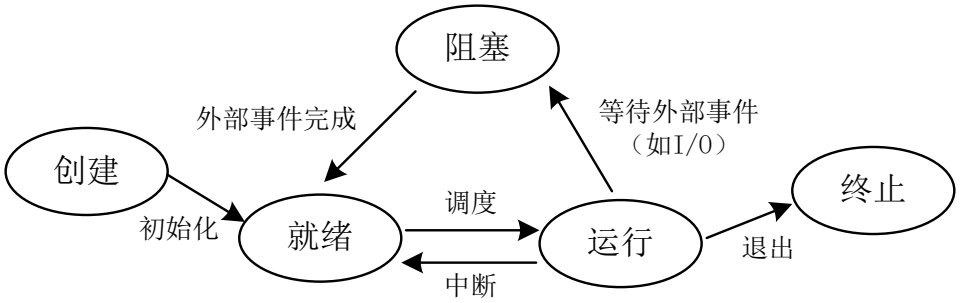
（1）创建状态：刚创建一个新进程，还未完成初始化，并不能被调度执行。

（2）就绪状态：新进程在经过初始化后，进程进入就绪状态。由于CPU数量可能小于进程数量，在某时刻只有部分进程能被调度到CPU上执行，此状态表示正等待CPU调用，但是还没有被调用。

（3）运行状态：该状态表示进程正在CPU上运行。调度器可以选择中断它的执行把它迁移到就绪状态；当进程运行结束时，它会被迁移至终止状态。

（4）阻塞状态：该状态表示进程因为等待某个外部事件（例如某个设备请求的完成），暂时无法被调度。当进程等待的外部事件完成后，会迁移到就绪状态。

（5）终止状态：该状态表示进程已经完成了执行，不会再被调度。



4.2.4内存管理

1.固定内存和动态分区管理

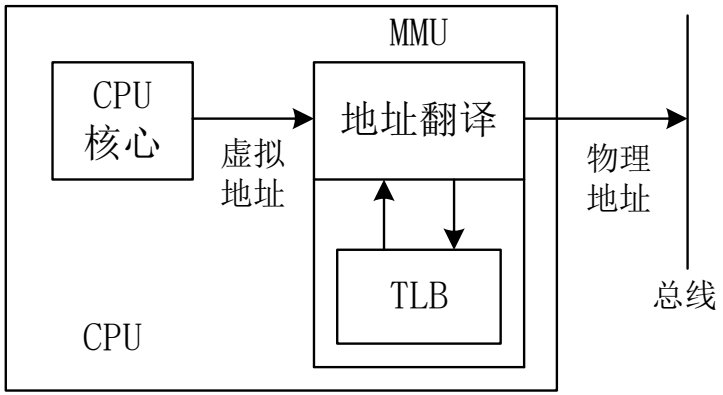
针对硬件没有内存管理单元（MMU）简要结构嵌入式操作系统来说常采用固定分区和动态分区的思路来进行内存分配。

（1）固定分区就是在系统编译阶段主存被划分成许多静态分区，进程可以装入大于或等于自身大小的分区。固定分区的缺点是程序大小和分区的大小必须匹配；活动进程的数目比较固定；地址空间无法增长。

（2）动态分区就是在一整块内存中首先划出一块内存给操作系统本身使用，剩下的内存给用户进程使用。当第一个进程A运行时，先从空闲内存中划出一块与进程A大小一样的内存给进程A使用。当第二个进程B准备运行时，可以从剩下的空间内存中继续划出一块和进程B大小相等的内存给进程B使用。以此类推，进程A和进程B以及后面进来的进程就可以实现动态分区了。但动态分区会随着时间的推移产生很多内存空洞，从而使内存的利用率下降，即常说的内存碎片。

2.虚拟内存管理

Linux操作系统可以在具有内存管理单元（MMU）微处理器上运行，因此可以通过把真实运行的物理内存转化为虚拟内存（virtual memory）的技术实现内存管理。如下图所示，其中TLB指转址旁路缓存（Translation Lookaside Buffer），它属于MMU内部的单元，用于加速地址转换的过程。



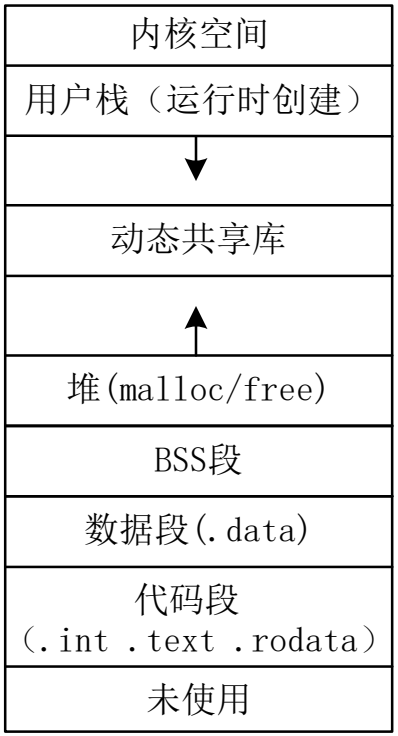
Linux程序在CPU上运行时，对于进程来说不用关心分配的内存在哪个地址，它只管分配使用，最终由处理器的MMU来处理进程对内存的请求，中间进行转换，把进程请求的虚拟地址转换成物理地址。虚拟内存机制可以使每个进程都能感觉自己拥有整个地址空间，可以随意访问，然后由处理器转换到实际的物理地址。

MMU将虚拟地址翻译为物理地址的主要机制有两种：分段机制和分页机制。

3.内存堆栈管理

程序的函数、全局变量和静态变量经过编译后，分别以section的形式存储在可执行文件的代码段、数据段和BSS段中。当程序运行中，可执行文件首先被加载到内存中，各个section分别加载到内存中对应的代码段、数据段和BSS段中。需要动态链接的动态库也被加载到内存中，完成代码的链接和重定位操作，以保证程序的正常运行。一个可执行文件加载到内存中运行时，它在内存空间的分布如下图。

在一个进程的地址空间中，代码段、数据段、BSS段在程序加载运行后，地址在整个程序运行期间不再发生变化，这部分内存称为静态内存。而在程序中使用malloc申请的内存、函数调用过程中的栈在程序运行期间是不断变化的，这部分内存称为动态内存。其中用户使用malloc申请的内存一般被称为堆内存(heap)，函数过程中使用的内存一般称为栈内存(stack)。



4.2.5操作系统调度

操作系统调度的目的是在有限的资源下，通过对多个程序执行过程的管理，尽可能满足系统和应用的指标。其中常用的调度指标包括：与性能相关的吞吐量、相应时间、周转时间；某些任务特有的需求，例如任务的实时性；一些非性能指标，例如公平性、资源利用率。

操作系统调度的策略由先到先得（First Come First Serve, FCFS）、最短任务优先（Shortest Job First, SJF）、最短完成时间任务优先（Shortest Time-to-Completion First, STCF）、时间片轮转（Round Robin, RR）策略等。

4.2.6进程间通信

1.消息队列

不同的进程间可以通过消息队列来发送消息和接收消息，发送和接收的接口是内核提供的，消息队列支持同时存在多个发送者和多个接收者。消息队列在内核中的表示是队列的数据结构，当创建新的消息队列时，内核将从系统内存中分配一个队列数据结构，作为消息队列的具体内容。

2.信号量

和消息队列“传递消息”的方案不同，信号量主要用作进程间的“同步”。信号量的主要操作是两个原语：P和V。P表示减少，在信号量中是将一个计数器减1；V表示增加，在信号量是将一个计数器加1，但是最终的计数不能超过1。该设计足够支持简单的进程同步。

创建2个任务，1个任务向串口分别发送“A”和”B”，另一个任务向串口发送”C”和“D”。这两个任务分别不用信号量同步和使用信号量同步。

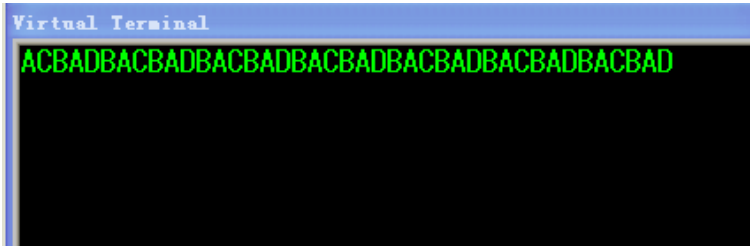
```
#define TASK_STK_SIZE 80    ///<定义任务栈大小
#define USE_SEM   1   ///<0为不使用信号量，否则使用信号量

OS_STK   Stk1[TASK_STK_SIZE]; ///<定义Task1堆栈
OS_STK   Stk2[TASK_STK_SIZE]; ///<定义Task1堆栈
OS_EVENT *psem;          ///<定义信号量指针
```

```
/**
@brief  向串口发送'A'和'B'
@param  pdata 创建任务时传入的数据
@retval None
@note 这是一个私有函数;根据USE_SEM
的值对应生成使用信号量和不使用信号量的代码
*/
static void task_1(void *pdata){
#if USE_SEM
  INT8U err;
  psem=OSSemCreate(1);  //创建信号量,初始化为1
  while(1){
    OSSemPend(psem,0,&err);
    usart_send_char('A');
    OSTimeDly(10);
    usart_send_char('B');
    OSTimeDly(10);
    OSSemPost(psem);
  }
#else
  while(1){
    usart_send_char('A');
    OSTimeDly(1);
    usart_send_char('B');
    OSTimeDly(1);
  }
#endif
}
```

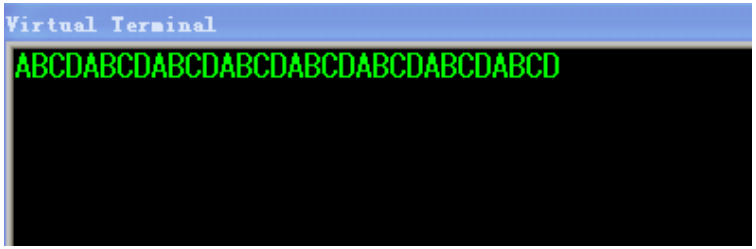
```
/**
@brief  先串口发送'C'和'D'
@param  pdata 创建任务时传入的数据
@retval None
@note 这是一个私有函数;根据
USE_SEM的值对应生成使用信号量和不使用信号量的代码
*/
static void task_2(void *pdata){
#if USE_SEM
  INT8U err;
  while(1){
    OSSemPend(psem,0,&err);
    usart_send_char('C');
    OSTimeDly(2);
    usart_send_char('D');
    OSTimeDly(2);
    OSSemPost(psem);
  }
#else
  while(1){
    usart_send_char('C');
    OSTimeDly(2);
    usart_send_char('D');
    OSTimeDly(2);
  }
#endif
}
```

```c
/**
@brief  创建task_1和task_2
@param  None
@retval 如果程序正常结束返回0
*/
int main(void){
  systick_config();
  usart_config();

  OSInit();
  OSTaskCreate(task_1,(void*)0,Stk1+(TASK_STK_SIZE-1),6); //创建task_1
  OSTaskCreate(task_2,(void*)0,Stk2+(TASK_STK_SIZE-1),8); //创建task_2
  OSStart();

  return 0;
}
```

没有使用信号量实现同步



用信号量实现同步



3.共享内存

共享内存的思路是内核为需要通信的进程建立共享区域，通信多方既可以直接使用共享区域上的数据，也可以将共享区域当成消息缓冲。操作系统将不同进程之间共享内存安排为同一段物理内存，进程可以将共享内存连接到它们自己的地址空间中，如果某个进程修改了共享内存中的数据，其它的进程读到的数据也将会改变。需要注意的是，共享内存并未提供锁机制，也就是说，在某一个进程对共享内存的进行读写的时候，不会阻止其它的进程对它的读写。

对于简要结构内核的嵌入式操作系统例如μC/OS-II可以直接通过定义一个全局的数组或者指针地址区域作为共享内存区域，实现数据的交互。Linux中提供了一组函数用于操作共享内存，例如：shmget函数用来获取或创建共享内存。

4. 管道进程间通信

管道是2个进程间的一条通道，一端负责投递，另一端负责接收。管道是Linux/Unix系统提供的一种通信机制。管道又分为无名管道和有名管道，无名管道只能用于有亲缘关系的进程间通信，有名管道则可以用于非亲缘进程间的通信。Linux系统Shell命令中因为我们通常通过符号“|"来使用管道，它通常是用来把一个进程的输出通过管道连接到另一个进程的输入。

5. 套接字进程间通信

套接字(socket)是一种即可用于本地，又可跨网络使用的通信机制。在进程通信中，可以使用基于IP地址和端口的组合的地址，通信双方通信使用本地回环地址(127.0.0.1)，然后各自绑定在不同的端口上。操作系统网络协议栈会识别回环地址，将通信消息转发到目标端口对应的进程。

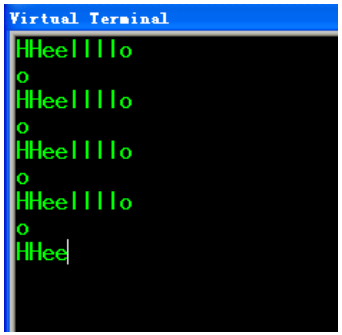
4.2.7进程间同步

编写代码常常需要对共享资源的保护，例如针对一个进程在访问临界区时，不容许同时被另一个进程被访问，防止共享资源被并发访问。

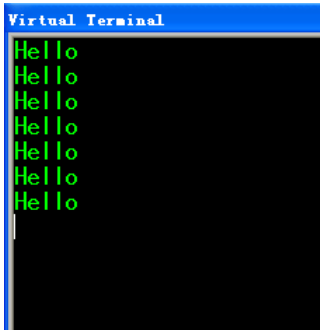
操作系统中常常采用互斥锁和信号量来解决进程间同步问题，实现对资源的独占式访问。

1.互斥锁同步

通过μC/OS-II操作系统实例来说此过程，有两个任务Task1和Task2，它们都调用SendBuf()函数--向串口发送出”hello/r/n”（注/r/n表示换行符）。在这个过程中需要用到互斥信号量。



没用互斥量的结果



用互斥量的结果

```
//任务1
static void task_1(void *pdata){
    INT8U err;
    pmutex=OSMutexCreate(MUTEX_PRIO,&err);          //创建互斥信号量
    while(1){
        send_buf();
        OSTimeDly(1);
    }
}
```

```
    //任务2
    static void task_2(void *pdata){
    while(1){
            send_buf();
            OSTimeDly(1);
    }
}
```

```
//临界共享区send_buf(void)函数
static void send_buf(void){
INT8U err;
OSMutexPend(pmutex,0,&err);  //等待互斥信号量
usart_send_char('H');    //发送字符 H
OSTimeDly(10);    //因为延时导致系统切换另一个任务
usart_send_char('e');
OSTimeDly(10);
usart_send_char('l');
OSTimeDly(10);
usart_send_char('l');
OSTimeDly(10);
usart_send_char('o');
OSTimeDly(10);
usart_send_char('\r');
usart_send_char('\n');
OSTimeDly(1);
OSMutexPost(pmutex);    //发送互斥信号量
}
```

```
//主程序
int main(void){
.....
OSInit();
OSTaskCreate(task_1,(void*)0,Stk1+(TASK_STK_SIZE-1),6);        //创建 task_1
OSTaskCreate(task_2,(void*)0,Stk2+(TASK_STK_SIZE-1),8);        //创建 task_2
OSStart();
        return 0;
}
```

2.信号量同步

给共享区通过信号量操作加个锁，只有拿到钥匙（当信号量大于等于0）时才能进入临界区。利用上面同样的例子结合信号量来说明如何实现同步，如下：

```
//创建任务 1
static void task_1(void *pdata){
INT8U err;
psem=OSSemCreate(1); //创建信号量，初始化为 1
while(1){
send_buf();
OSTimeDly(1);
}
}
```

```
//创建任务 2
static void task_2(void *pdata){
while(1){
send_buf();
OSTimeDly(1);
}
}
```

```
static void send_buf(void){
INT8U err;
OSSemPend(psem, 0, &err); //信号量减 1 操作
.....          //发送”hello/r/n”和前面同步实例一致。
OSSemPost(psem); //信号量加 1 操作
}
```

3.同步带来的问题

（1）死锁

当有多个（两个及以上）线程为有限的资源竞争时，有的线程就会因为在某一时刻没有空闲的资源而选入等待，当这一组中的每一个线程都在等待组内其他线程释放资源从而造成的无限等待，称为死锁。

（2）活锁

出现活锁时，锁的竞争者很长一段时间都无法获取锁进入临界区，出现每个线程都无法进入临界区的情况。

（3）优先级反转

优先级反转时由于同步导致线程执行顺序违反预设优先级的问题，即先执行了低优先级任务而没有执行高优先级的任务。

4.2.8 中断管理

中断最初用来替换I/O检测操作的轮询处理方式，以提供I/O口处理的效率。中断使得CPU可以在事件发生时才予以处理，而不必让微处理器连续不断地查询是否有事件发生。中断在操作系统管理框架下具有硬中断和软中断。

硬中断就是外部接口及内核产生的真实中断，对这部分中断的管理：（1）针对运行微控制器芯片上的简要结构内核来说，直接调用芯片的中断处理库函数，实现对中断的设置和相应。（2）Linux操作系统通过建立自身定义的中断号和真实硬件中断号映射表，从而在针对不同的硬件时只需更改此映射表，从而对不同硬件有很好的兼容性。

软中断通过处理器的软件指令来产生，产生中断的时机时预知的，可根据需要在程序中进行设定。软中断的处理程序以同步的方式进行执行，是一种非常重要的机制。

4.2.9 时钟管理

在实时系统中，时钟具有非常重要的作用，时钟管理一般具有以下功能：维持日历时间，任务有限等待的计时，软定时器的定时管理和维持系统时间片轮转调度。所以操作系统层提供时钟管理函数，这些时钟管理函数一般是基于CPU芯片的Systick滴答定时器或通用定时器来实现的，从而为操作系统管理提供时间函数。

4.2.10 文件系统

文件是操作系统在进行存储时使用最多的手段之一，每个文件实质上是一个有名字的字符序列。序列的内容为文件数据(file data)，而序列长度、序列修改时间等描述文件数据的属性、支撑文件功能的其他信息称为文件元数据(file metadata)。文件系统是操作系统中文件的管理者，文件系统将文件保存在存储设备中，操作系统将这些存储设备抽象为块设备(block device)，以方便文件系统使用统一的接口访问。块设备上的存储空间在逻辑上被划成固定大小的块(block)，块的大小一般是512字节或4KB，是块设备读写的最小单元。常见的有FAT文件系统、NTFS文件系统和 EXT3文件系统等。

4.2.11 设备管理

设备管理是操作系统的重要职责，一个真实硬件开发板都有真实的硬件接口电路，如LED灯、按键、USB接口和以太网口等，CPU会收到来自这些物理设备的信号，并通过驱动程序对这些信号进行处理，实现与这些设备进行交互，实现对设备的控制。

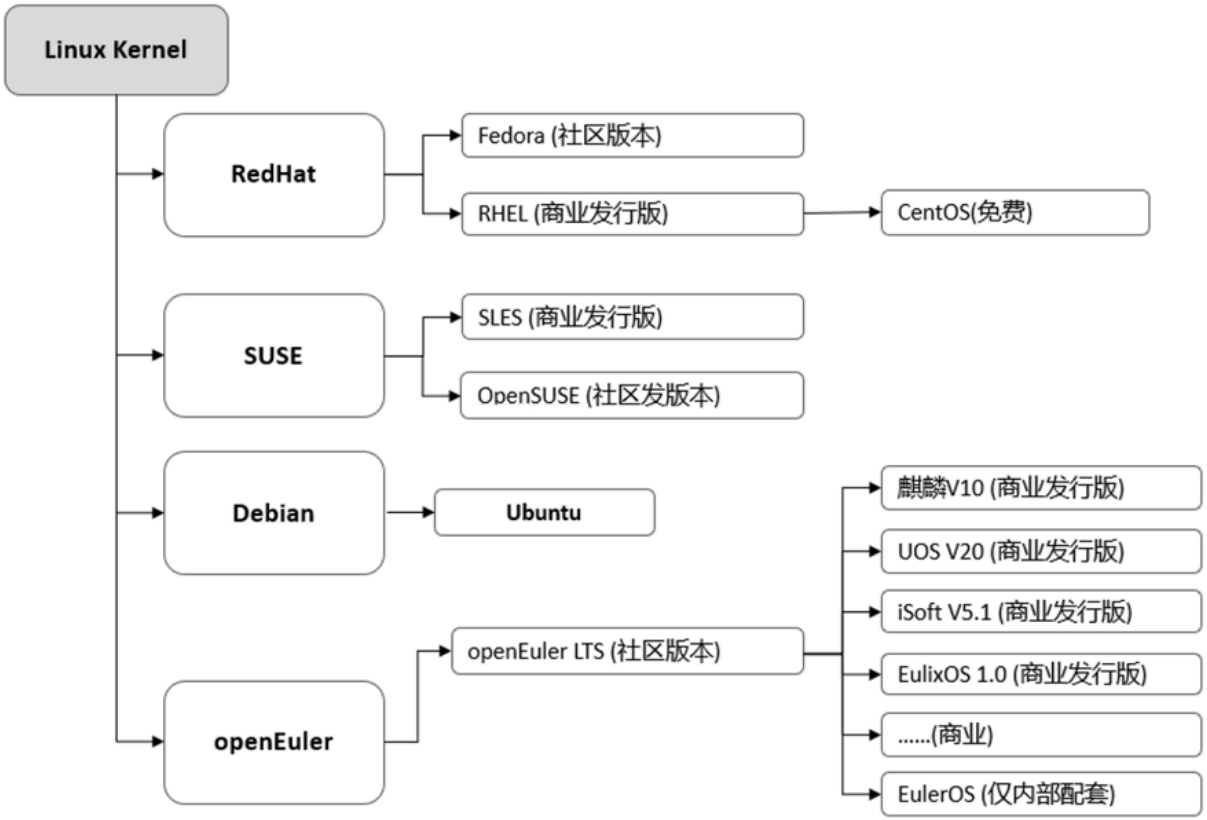
针对运行微控制器芯片上的简要结构内核来说，直接调用芯片的外部接口库函数，实现对中断的设置和相应。

针对例如Linux操作系统，CPU调用设备主要有如下方式：（1）总线，例如AMBA总线（ARM核总线）；PCI总线等。（2）可编程I/O口。（3）直接内存访问（DMA）。（4）输入输出内存管理单元（Input-Output Memory Management Unit, IOMMU）负责将总线地址翻译成物理地址。同时操作系统需要一定机制可以识别当前已经连接的设备，常见的设备识别机制有设备树和高级配置与电源接口(Advanced Configuration and Power Interface)，简称ACPI。

4.3 Linux操作系统

4.3.1 常见Linux发行版

Linux系统诞生于1991，Linus Torvalds动手实现了一个新的操作系统，然后在comp.os.minix新闻组上发布了第一个版本的Linux内核；1993年有大约100名程序员参与了Linux内核代码的编写，Linux0.99的代码已经有大约10万行，初成规模。



4.3.2 openEular操作系统

- openEuler是面向数字基础设施的开源操作系统，支持服务器、云计算、边缘计算、嵌入式等应用场景，支持多样性计算，致力于提供安全、稳定、易用的操作系统。
- openEuler通过为应用提供确定性保障能力，支持OT领域应用及OT与ICT的融合。
- 同时，openEuler又是一个开源、免费的Linux发行版平台，其致力于打造中国原生开源、可自主演进操作系统根社区。

openEuler发展历程

- 从EulerOS到openEuler：

EulerOS是一款基于Linux内核的服务器操作系统，在近10年的发展中成功支持了华为各种产品解决方案。随着云计算的兴起和鲲鹏芯片的发展，EulerOS成为与鲲鹏芯片配套最合适的软件基础设施。为推动鲲鹏生态的发展，繁荣国内和全球的计算产业，2019.09宣布开源，2019.12代码开源上线，命名为openEuler。

- 从基础版本到全场景支持

2020.03，首个LTS版本发布（基础版本）；2020.09，创新版本（多样性算力释放）；2021.03：创新版本（内核创新）；2021.09：创新版本（服务器、云计算、边缘计算、嵌入式全场景支持）；2022.03：LTS版本（全场景融合）

openEuler开源生态繁荣发展

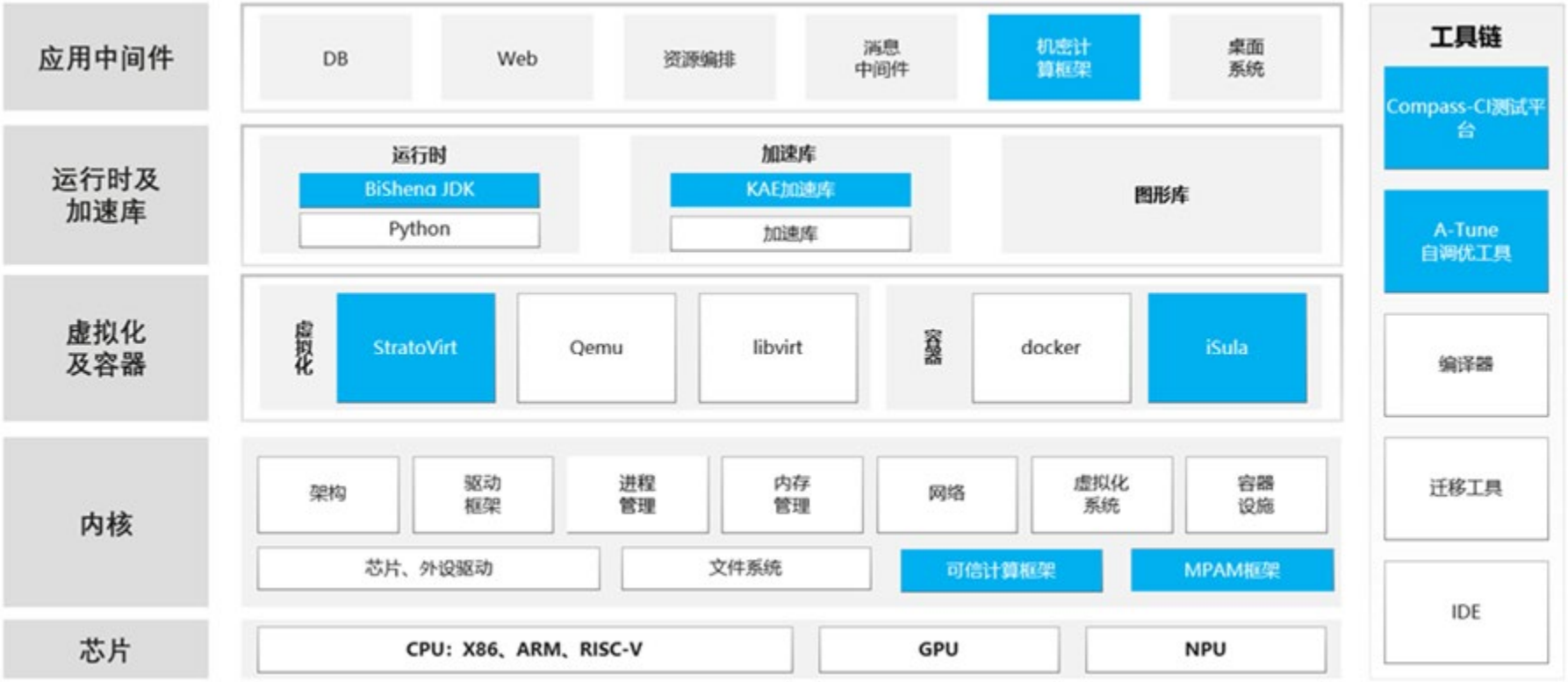
- 社区理事会

- 目前发布openEuler商业发行版的厂商

- 多行业规模商用
  - 金融：建设银行信用卡核心系统 | 运营商：中国移动打造容器云
  - 政府：电子公文80%以上市场份额 | 电力：智能电网控制系统

openEuler平台框架

- openEuler（21.03版本）架构如下图所示：



openEuler在具有通用的Linux系统架构，包括内存管理子系统、进程管理子系统、进程调度子系统、进程间通讯（IPC）、文件系统、网络子系统、设备管理子系统和虚拟化与容器子系统等。同时，openEuler又不同于其他通用操作系统，openEuler从OS内核、可靠性、安全性和生态使能等方面做了特性增强（上页图中蓝色背景，关键特性如下页表所示）

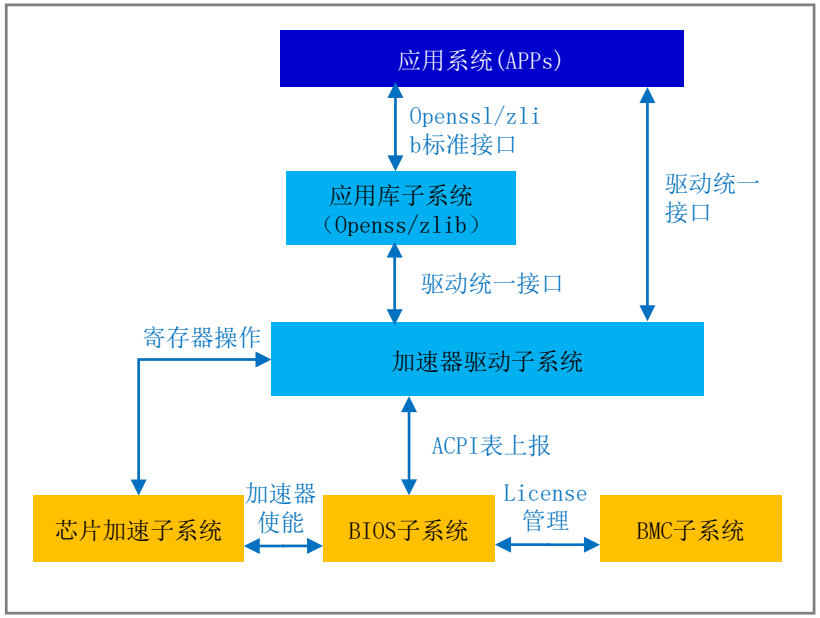
openEuler关键特性

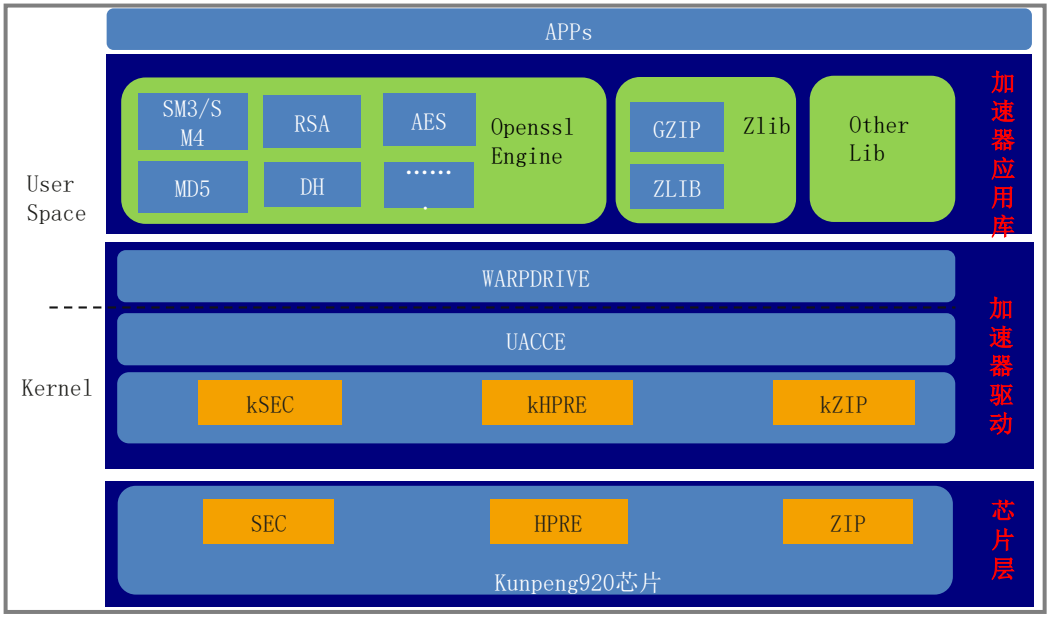
序号	名称	特性说明	网页链接
1	StratoVirt	轻量级虚拟机引擎	https://docs.openeuler.org/zh/docs/21.03/docs/StratoVirt/StratoVirtGuide.html
2	iSula	轻量级容器引擎	https://docs.openeuler.org/zh/docs/21.03/docs/Container/iSula%E5%AE%B9%E5%99%A8%E5%BC%95%E6%93%8E.html
3	A-Tune	AI智能调优引擎	https://docs.openeuler.org/zh/docs/21.03/docs/A-Tune/A-Tune.html
4	secGear	跨平台机密计算框架	https://docs.openeuler.org/zh/docs/21.03/docs/secGear/secGear.html
5	可信计算	安全可信计算	https://docs.openeuler.org/zh/docs/21.03/docs/Administration/可信计算.html
6	KAE	鲲鹏加速引擎（Kunpeng Accelerator Engine）	https://docs.openeuler.org/zh/docs/21.03/docs/Administration/使用KAE加速引擎.html
7	MPAM	Memory System Resource Partitioning and Monitoring	https://mp.weixin.qq.com/s/0TgrFjFtobmk-h1HwJskqg
8	毕昇JDK	Huawei开源JDK	https://gitee.com/openeuler/bishengjdk-8/wikis/Home

CPU调度和NUMA-aware Qspinlock

- 硬件架构的演变驱动操作系统随之改变
  - arm64 / Kunpeng
    - 多核/众核： 为避免多个CPU共用一个调度队列带来的资源竞争，采用多队列调度策略
    - NUMA：NUMA-aware Qspinlock（为避免线程间同步所依赖的共享变量在不同NUMA节点间传递所带来的开销，引入NUMA感知队列自旋锁 。）
- 场景的不同驱动操作系统与之适应
  - openEuler应用于服务器场景
    - 大部分进程是普通进程：使用CFS调度算法
- 综合以上两个因素，CFS等调度算法与多队列调度策略相融合成为openEuler中CPU调度的核心

软硬件协同：KAE逻辑架构图





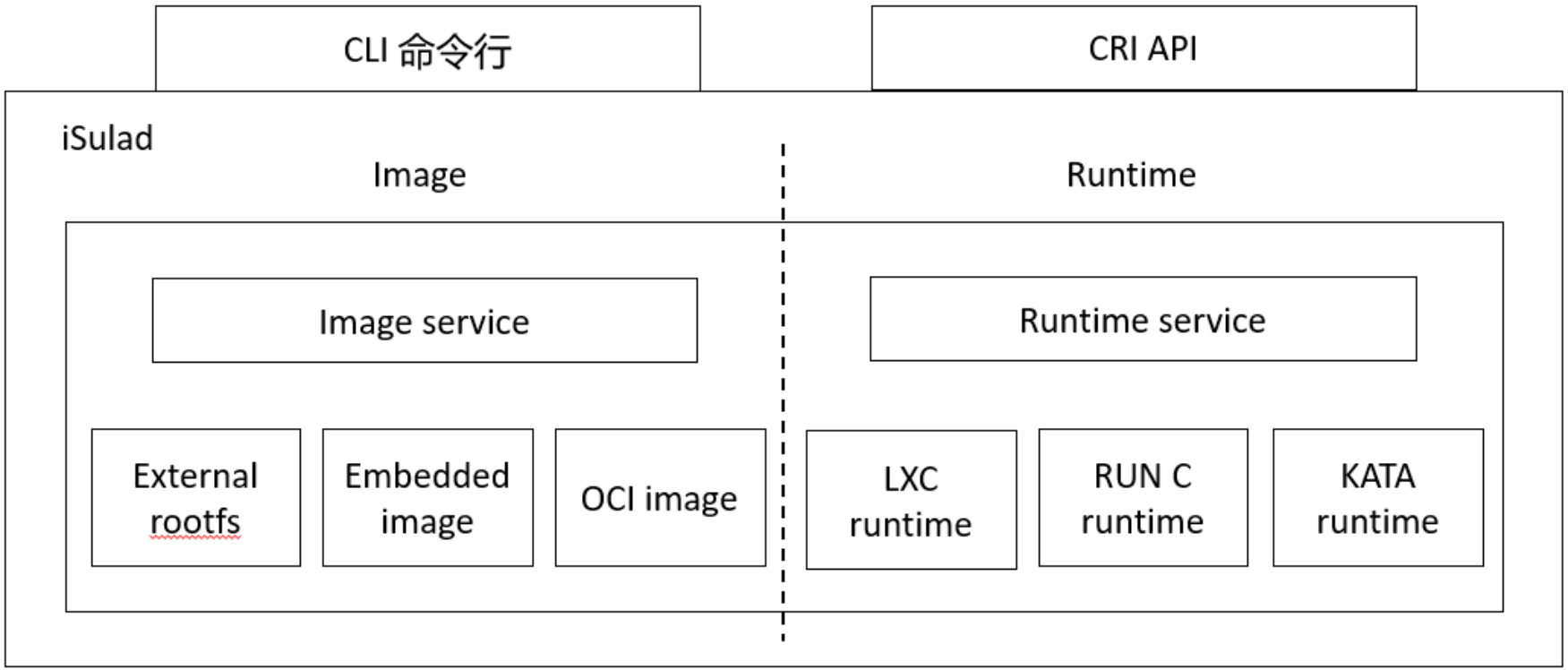
HPRE：High Performance RSA Engine，高性能RSA加速引擎
ZIP：高性能zlib/gzip压缩引擎
SEC：Security Engine，硬件安全加速引擎

鲲鹏920加速系统逻辑架构图

目前已经完成的加速库


典型应用
GKL（基因测序）AVX2Neon
Web中间件Nginx
秘钥协商RSA计算
分布式存储Ceph
压缩&MD5
大数据Hadoop
数据加解密&压缩
视频转码
X265
OS集成基础库
Glibc&gzip&OpenSSL
...
系统基础库
Glibc
指令加速
HyperScan
指令加速
AVX2Neon
异构生态迁移
压缩
Gzip
指令加速
Zlib
压缩硬加速
ZSTD
指令加速
Snappy
指令加速
加解密
OpenSSL
加解密硬加速
媒体
HMPP
媒体信号库
X265
h.265视频转码加速
已有的ARM基础库
ISA-L
存储加速库
SPDK
SSD用户态驱动加速
DPDK
用户态网络驱动加速
X264
h.264视频编解码加速
...
鲲鹏指令加速
ZIP（硬加速-压缩）
HPRE（硬加速-非对称加解密）
SEC（硬加速-对称加解密）
软加速库：基于鲲鹏指令的软加速库
硬件加速：基于鲲鹏加速引擎的加速库
已有的ARM软加速库

轻量级容器：iSulad架构图



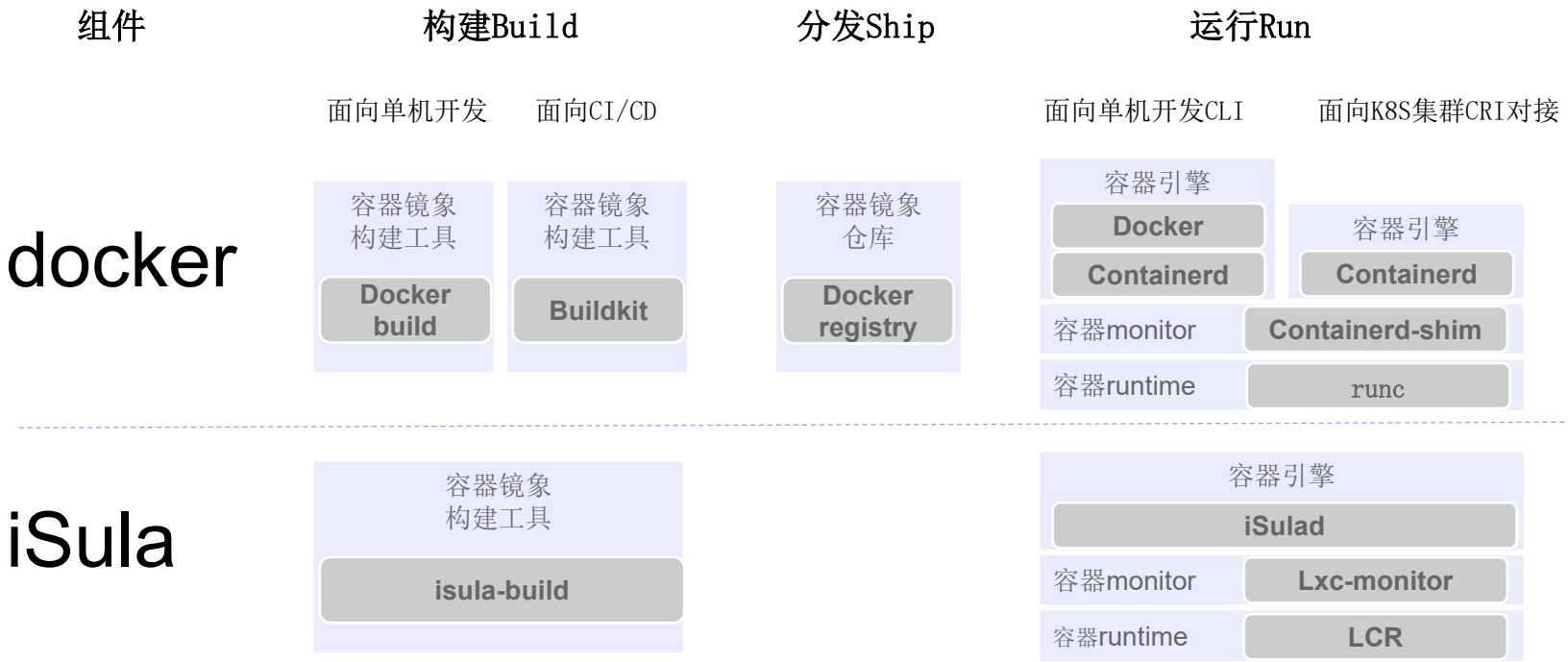
容器统一架构图

iSulad架构简介

- iSulad是一个轻量级容器引擎，目前采用C/C++语言实现，相比其它容器引擎，它的内存开销更小，并发性能更高；
- iSulad容器引擎主要包括以下几个模块：
  - 通信模块：支持gRPC/RESTFUL两种通信方式，提供对外通信的能力；
  - 镜像模块：支持OCI标准镜像，提供content/metadata、rootfs及snapshot管理能力；
  - 运行时模块：支持轻量级Runtime（lcr）和OCI标准的Runtime（runc，kata等）。
- 从接口划分：
  - 北向接口：提供CLI（Command Line Interface）接口和CRI接口；
  - 南向接口：提供统一的Runtime管理接口Plugin，支持lcr和OCI两种类型的Runtime。

isula-build和iSulad

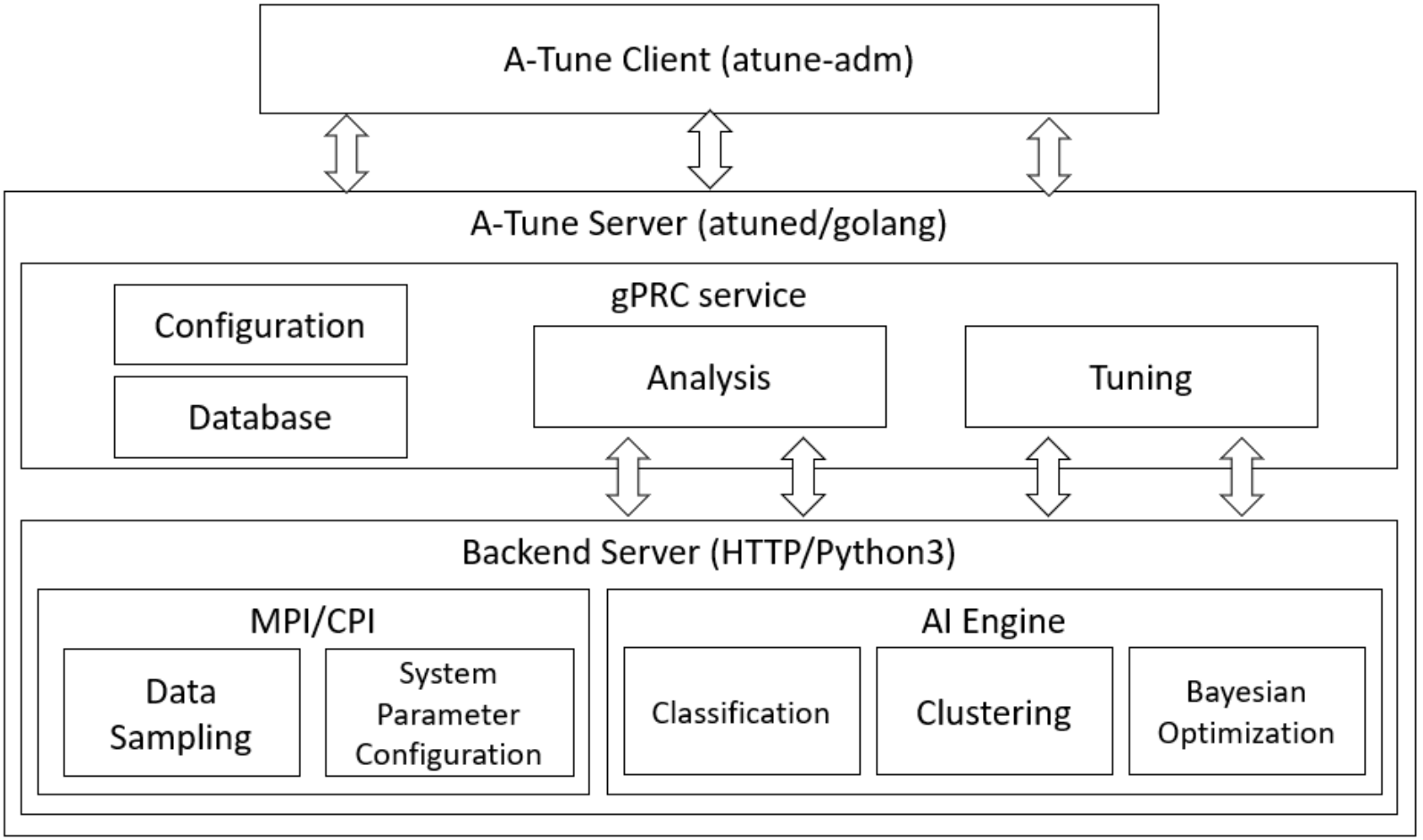
- 构建侧（isula-build）与运行侧（iSulad）分离



系统参数智能调优 — A-Tune

- 当Linux或Windows这种通用操作系统的某个功能机制无法保证对所有场景均有益时，设计者就会在系统中提供一个可配置参数，并确保该参数的默认配置对大部分通用场景有益，而使用者通过更改参数配置来满足特定的使用场景需求；
- openEuler正是基于Linux内核的，举例来说，其sysctl命令的参数超过1000个，而完整的IT系统从最底层的CPU、加速器、网卡，到编译器、操作系统、中间件框架，再到上层应用，可调节对象超过7000个；
- 自调优工具A-Tune旨在让操作系统能够满足不同应用场景的性能诉求，降低性能调优过程中反复调参的人工成本，提升性能调优效率。

智能调优：A-Tune的整体架构图



A-Tune的整体架构图

openEuler开源社区汇总

- openEuler官网：https://openeuler.org
- openEuler源码仓：https://gitee.com/openeuler
- openEuler软件包：https://gitee.com/src-openeuler
- openEuler创新实践课：https://gitee.com/openeuler-practice-courses
- openEuler相关赛事：https://gitee.com/openeuler-competition
- 为老师准备的课程SIG组：https://gitee.com/openeuler/sig-OSCourse

4.3.3 嵌入式Linux系统

嵌入式Linux系统主要包括BootLoader、内核和文件系统三部分内容。从开发过程中，有以上4个层次：

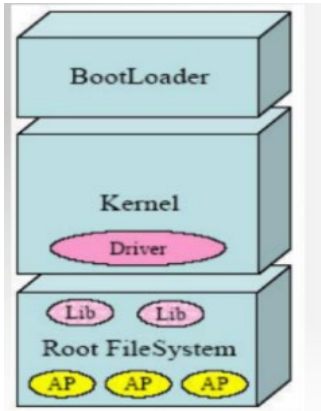
（1）引导加载程序，包括固化在芯片中的启动（boot）代码和BootLoader代码。

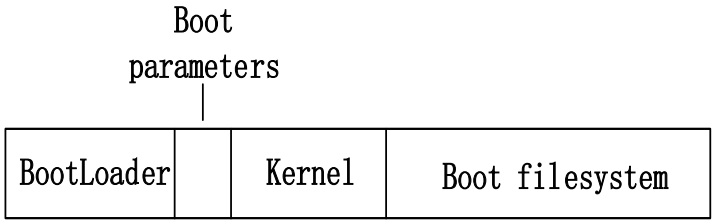
（2）嵌入式Linux内核，特定于嵌入式板子的定制内核以及内核启动参数，内核的启动参数可以是默认的也可以是BootLoader传递给它。内核中可以装载外设相应的驱动模块。

（3）文件系统，根文件系统和建立于Flash内存设备之上的文件系统，其中包含了Linux系统能够运行所必需的应用程序、库等，例如动态连接的程序运行时需要的glibc库等。

（4）用户应用程序，用户开发的应用程序，例如各种智能算法，有时还会包括一个嵌入式图形用户界面，常用的嵌入式GUI有：Qtopia等。

在BootLoader中有个“Boot parameters”子分区存放一些可设置参数，比如IP地址、串口波特率、要传递给内核的命令行参数等。





在系统运行中，BootLoader首先运行，然后它将内核复制到内存中，并且在内存某个固定的地址设置好要传递给内核的参数，最后运行内核。内核启动之后，他会挂载根文件系统，启动文件系统中的应用程序。

4.3.4 基于Bootloader 方式的Linux系统启动

BootLoader启动大多都是两阶段启动过程，第一阶段使用汇编来实现，它完成一些依赖于CPU体系结构的初始化，并调用第二阶段的代码；第二阶段实现更为复杂的功能，采用C语言来实现从而使代码有较好的可读性和可移植性。

（1）BootLoader第一阶段的功能。主要包括：硬件设备初始化，包括关闭WATCHDOG、关中断、设置CPU的速度和时钟频率、RAM初始化等；为加载BootLoader第二阶段代码准备RAM空间并把第二阶段代码复制到RAM空间中（对于Nor Flash等静态ROM类型的存储设备可以不用复制而直接在其上运行）；设置好堆栈跳转到第二阶段代码的C入口点。

（2）BootLoader第二阶段的功能。主要包括初始化本阶段要用到的硬件设备；检测系统内存映射；将内核映象和根文件系统映像从Flash上读到RAM空间中；为内核设置启动参数；调用内核。

BootLoader与内核的交互

BootLoader与内核交互过程是BootLoader把参数放在某个约定的位置后启动内核，内核启动时自动从这个位置获取参数。除了约定好参数存放的位置外，还要规定参数的结构，Linux2.4以后的内核主要以标记列表（tagged list）的形式来传递启动参数，这些参数主要包括系统的根设备标志，页面大小，内存的起始地址和大小，RAMDISK 的起始地址和大小，压缩的RAMDISK 根文件系统的起始地址和大小，内核命令参数等，标记列表以标记ATAG_CORE开始，以标记ATAG_NONE结束。这里的ATAG_CORE，ATAG_NONE 是各个参数的标记，本身是一个32位值，其他的参数标记还包括： ATAG_MEM32 ，ATAG_INITRD ， ATAG_RAMDISK ，ATAG_COMDLINE 等。

U-Boot—开源的Bootloader

U-Boot工程的特性：

（1）开放源码；

（2）支持多种嵌入式操作系统内核，如Linux、NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS, android；

（3）高度灵活的功能设置，适合U-Boot调试、操作系统不同引导要求、产品发布等，尤其对Linux支持最为强劲；

（4）支持多个处理器系列，如PowePC、Arm、AVR32、MIPS、x86、68k、Nios与MicroBlaze等；

（5）高度灵活的功能设置，适合U-Boot调试、操作系统不同引导要求、产品发布等；

（6）丰富的设备驱动源码，如串口、以太网、SDRAM、FLASH、LCD、NVRAM、EEPROM、RTC、键盘等；

（7）较为丰富的开发调试文档与强大的网络技术支持；

（8）支持NFS挂载，支持RAMDISK（压缩或非压缩）形式的根文件系统，支持从Flash中引导压缩或非压缩系统内核；

（9）支持目标板环境变量多种存储方式，如Flash、NVRAM、EEPROM；

（10）CRC32校验，可校验Flash中内核、RAMDISK镜像文件是否完好；

（11）上电自检功能：SDRAM、Flash大小自动检测，SDRAM故障检测，CPU型号。

4.3.5 Linux内核

例如Linux的版本号：4.4.126。

VERSION = 4

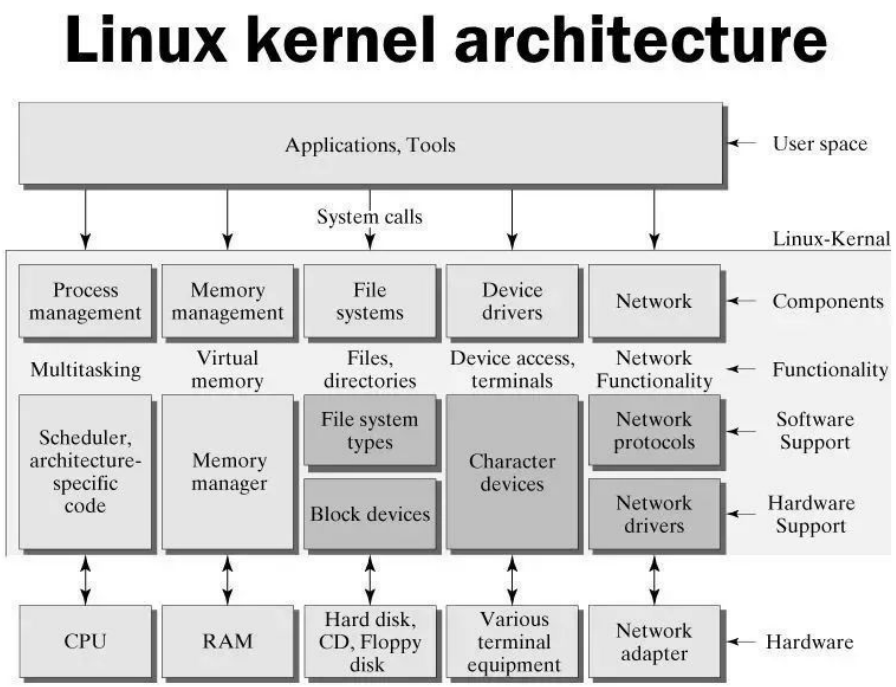
PATCHLEVEL = 4

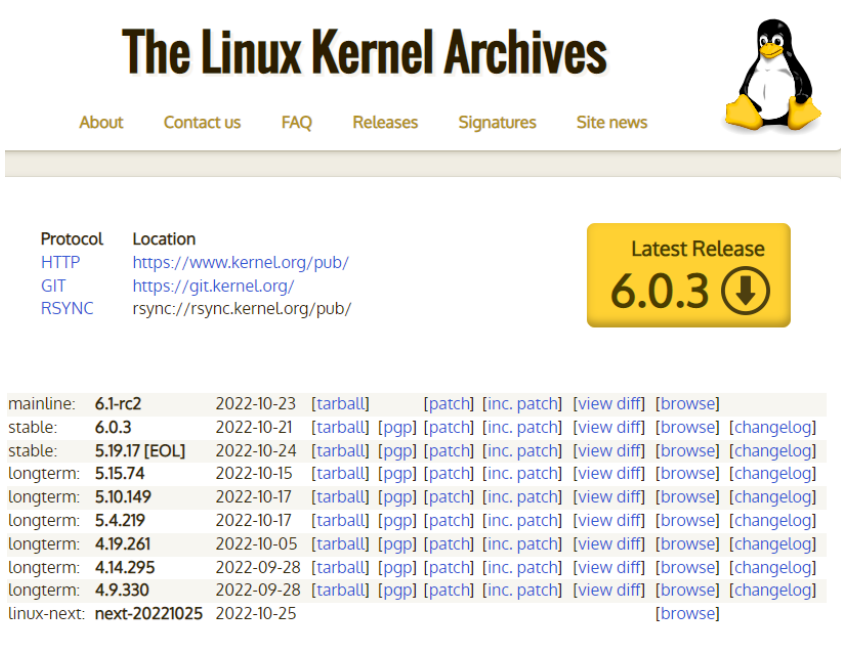
SUBLEVEL = 126

从3.x.x开始，VERSION没有特殊含义。PATCHLEVEL为主要版本，当内核增加新特性及功能变更时，该编号增加。SUBLEVEL 为次要版本，当内核功能不变，仅进行缺陷修复时该编号增加。

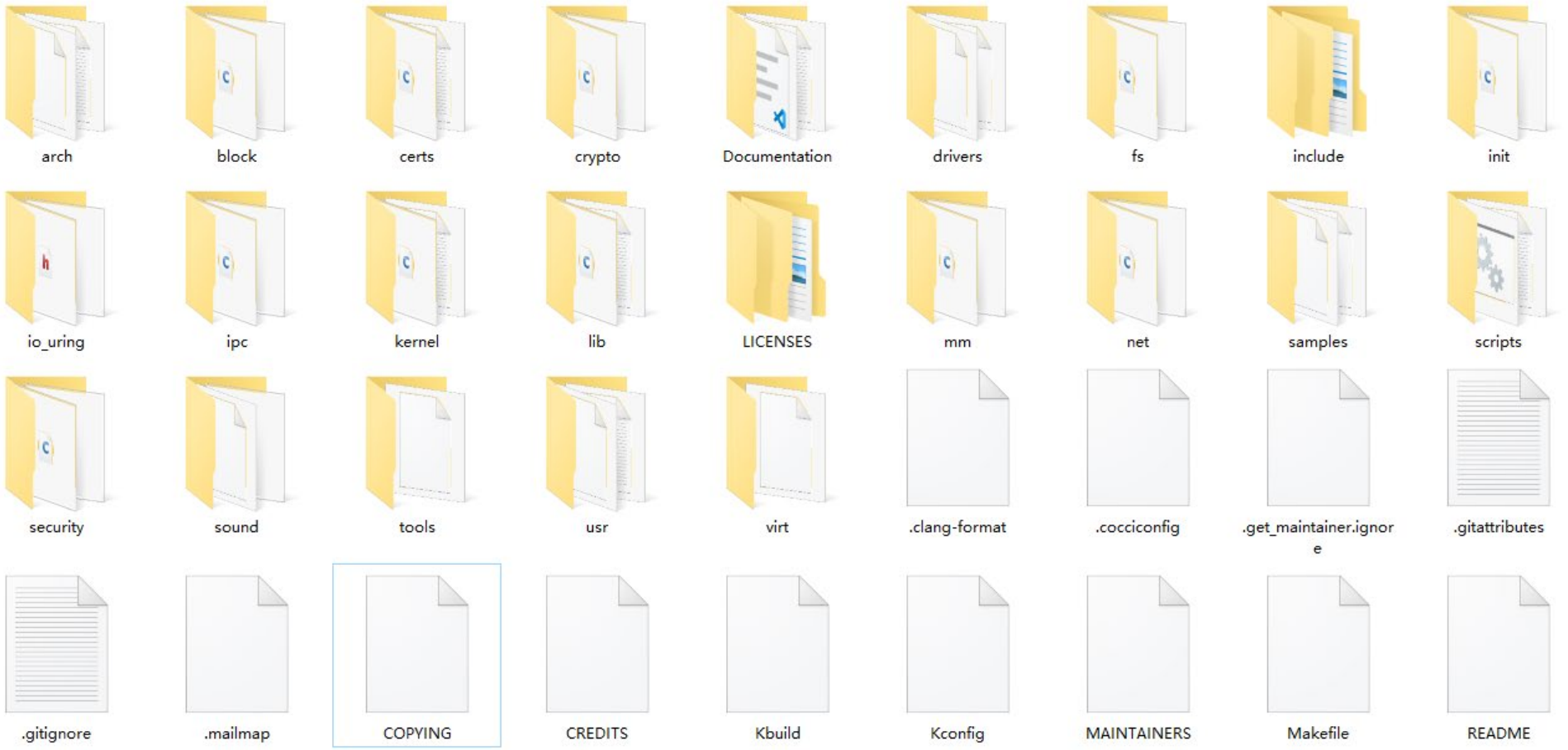
通常，主线会每隔几个月推出一个主要版本成为稳定版（stable）。稳定版将会不断发布修补版本，直至下一个稳定版推出后宣告结束维护（End of life）。但是，Linux内核社区会定期选择一个主要版本成为长期维护版本（Longteam Maintenance）。长期支持版本将在将来数年的时间里继续移植主线的缺陷修复补丁，以供其他下游软件、硬件厂商使用。

Linux内核源码结构





https://www.kernel.org/


arch
block
certs
crypto
Documentation
drivers
fs
include
init
io_uring
ipc
kernel
lib
LICENSES
mm
net
samples
scripts
security
sound
tools
usr
virt
.clang-format
.cocciconfig
.get_maintainer.ignore
.gitattributes
.gitignore
.mailmap
COPYING
CREDITS
Kbuild
Kconfig
MAINTAINERS
Makefile
README

内核配置及编译

在编译Linux内核时，需要配置内核，可以使用如下命令的一个：

make config (基于文本的配置界面)
make menuconfig (基于文本菜单的配置界面)
make xconfig(需要QT安装)
make gconfig(需要GTK+安装)

其中值得推荐的是make menuconfig，它使用ncurses库，可以在终端呈现图形化的配置菜单，，且配置直观方便。

内核配置过程中，arch/arm/configs/xxx_defconfig文件包含了许多电路板的默认配置，只需要运行make ARCH=arm xxx_deconfig 就可以为xxx开发板配置内核。

编译内核和模块的方法是：

make zImage
make modules

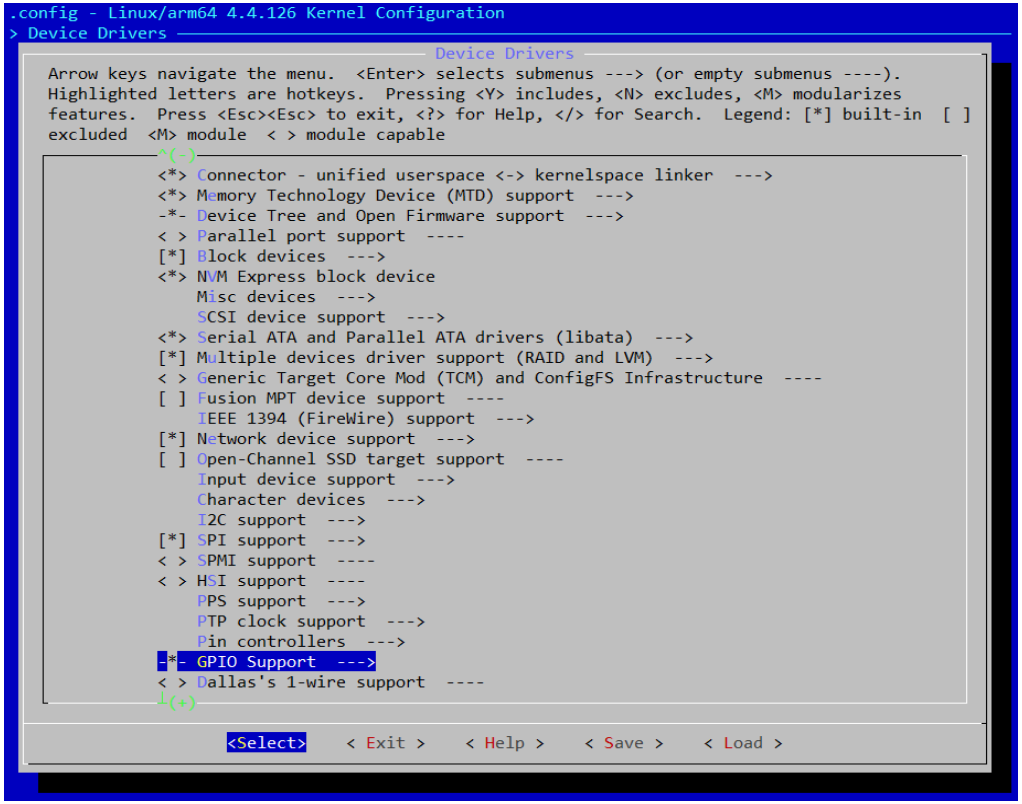
若对于arm系列芯片，需要ARCH = arm作为环境变量导出，执行命令后在源代码的根目录下会得到未压缩的内核映像vmlinux和内核符号表文件System.map，在arch/arm/boot/目录下会得到压缩的内核映像zImage，在内核各对应目录内得到选中的内核模块。

Linux内核的配置系统由以下3个部分组成：

Makefile: 分布在Linux内核源代码中，定义编译规则。

Kconfig（配置文件）：给用户提供配置选择的功能。

配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供字符界面和图形界面），这些配置工具使用的都是脚本语言，如用Tcl/TK、Perl等。



4.3.6 Linux驱动程序

Linux支持三种类型的硬件设备：

字符设备（character device）：字符设备是能够像字节流（比如文件）一样被访问的设备，对它的读写是以字节为单位的，是没有缓冲直接读写的设备，字符设备的驱动程序中实现了open、close、read和write等系统调用，应用程序可以通过设备文件/dev/ttySAC0来访问字符设备。

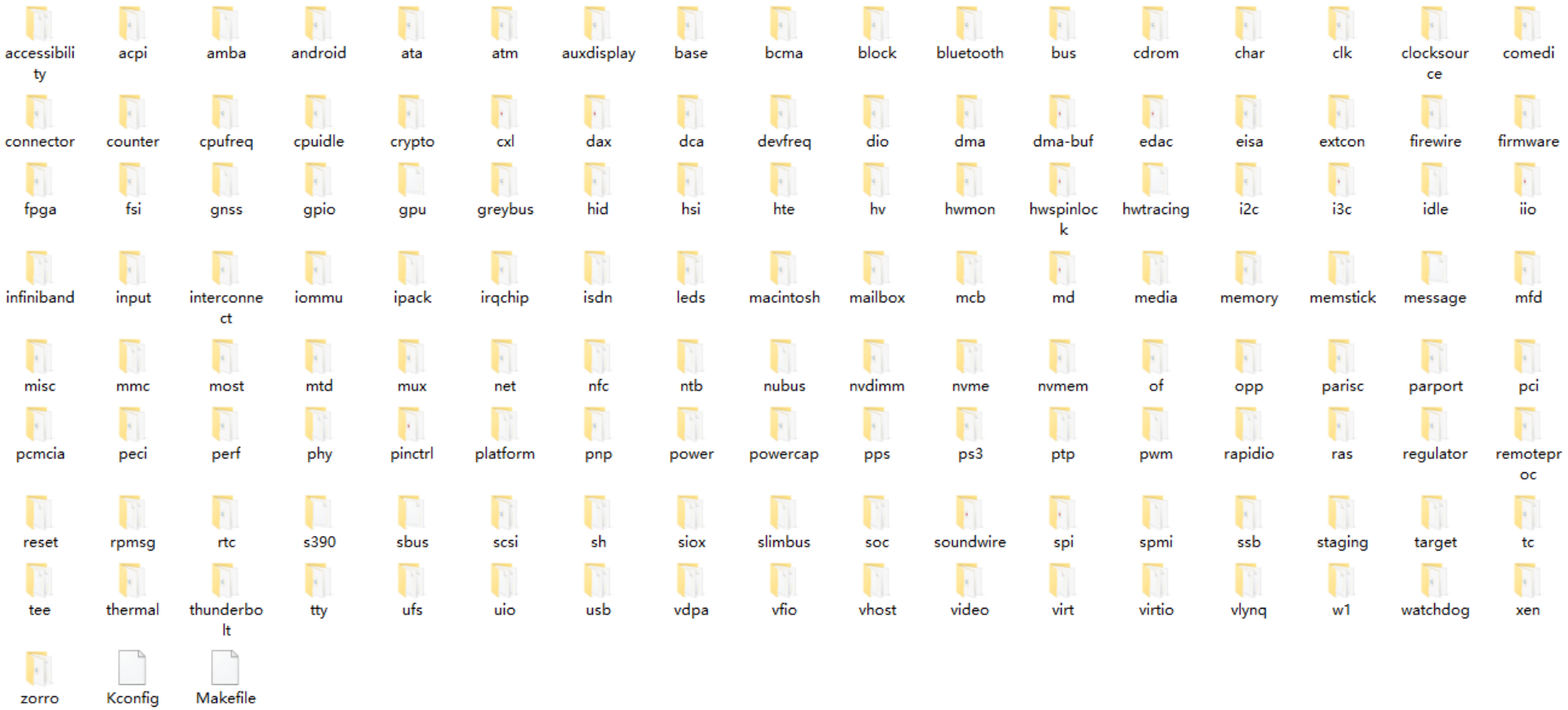
块设备（block device）：块设备上的数据以块的形式存放，只能按照一个块（一般是512字节或者1024字节）的倍数进行读写的设备，块设备通过buffer cache访问，可以随机存取，就是说任何块都可以读写，不必考虑它在设备的什么地方。块设备可以通过相应的设备文件访问，例如/dev/mtdblock0、/dev/hda1等，应用程序可以通过相应的设备文件实现open、close、read和write等系统调用，与块设备传送任意字节的数据。

网络设备（network interface）：是通过网络socket接口访问的设备，无法归纳到前面两类，如果说它是字符设备，它的输入/输入出却是有结构的、成块的（报文、包、帧）；如果说它是块设备，它的“块”又不是固定大小的，大到数百甚至数千字节，小到几字节。访问网络接口的方法是给它们分配一个唯一的名字（比如eth0），但这个名字在文件系统中（比如/dev目录下）不存在对应的节点项，应用程序、内核和网络驱动程序间的通信完全不同于字符设备、块设备，库、内核提供了一套和数据包传输相关的函数，而不是open、read和write等。

Linux内核源码中有大约85%是各种驱动程序的代码，种类齐全，可以在同类型驱动的基础上进行修改以符合具体的要求。一般编写Linux设备驱动程序的大致流程如下：1）查看原理图、数据手册，了解设备的操作方法；2）在内核中找到相近的驱动程序，以它为模板进行开发；3）实现驱动程序的初始化：比如向内核注册驱动程序，这样应用程序传入文件名时，内核才能找到相应的驱动程序。4）设计所要实现的操作，比如open、close、read和write函数等；5）对需要中断的程序，实现中断服务；6）编译该驱动程序到内核中，或者用insmod命令加载；7）测试驱动程序。

设备树（DTS，Device Tree Source）是一种描述硬件的数据结构，在老的Linux版本如2.6，Arm架构的板级硬件细节过多地被硬编码在arch/arm/plat-xxx和arch/arm/mach-xxx中，采用设备树后，硬件的细节可以直接通过它传递给Linux，而不再需要内核中大量的冗余代码。设备树由一系列被命名的节点（Node）和属性（Property）组成，其中节点本身可包括子节点，属性是成对出现的名称和值，在设备树中可描述的信息包括：CPU的数量和类别；内存基地址和大小；总线和桥；外设连接；中断控制器和中断使用情况；GPIO控制器和GPIO使用情况；时钟控制器和时钟使用情况。它基本上画一颗电路板上CPU、总线、设备组成的树，内核可以识别这棵树，并根据它展开出Linux内核中的platform_device等设备，而这些设备用到的内存、IRQ等资源，也被传递给内核，内核会将这些资源绑定给展开的相应的设备。

linux-6.0.3\drivers目录文件

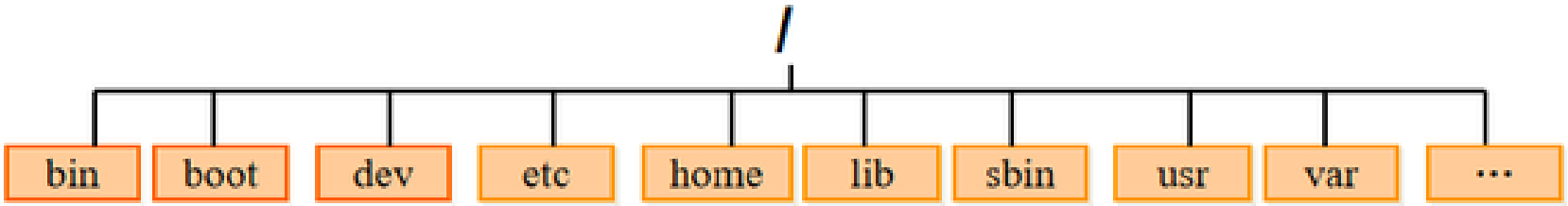


4.3.7 Linux根文件系统

Linux需要在一个分区上存放系统启动所必需的文件，比如内核映像文件、内核启动后运行的第一个程序（init）、给用户提供操作界面的shell程序、应用程序所依赖的库等。这些基本的文件合称为根文件系统，它们存放在一个分区中。

根文件系统首先是一种文件系统，该文件系统不仅具有普通文件系统的存储数据文件的功能，但是相对于普通的文件系统，它的特殊之处在于，它是内核启动时所挂载（mount）的第一个文件系统，内核代码的映像文件保存在根文件系统中，系统引导启动程序会在根文件系统挂载之后从中把一些初始化脚本（如rcS,inittab）和服务加载到内存中去运行。

根文件系统包含系统启动时所必须的目录和关键性的文件，以及使其他文件系统得以挂载（mount）所必要的文件。例如：init进程的应用程序必须运行在根文件系统上；根文件系统提供了根目录“/”；linux挂载分区时所依赖的信息存放于根文件系统/etc/fstab这个文件中；shell命令程序必须运行在根文件系统上，譬如ls、cd等命令；总之：一套linux体系，只有内核本身是不能工作的，必须要rootfs（上的etc目录下的配置文件、/bin /sbin等目录下的shell命令，还有/lib目录下的库文件等等）相配合才能工作。



- bin (binaries)存放二进制可执行文件
- sbin (super user binaries)存放二进制可执行文件，只有root才能访问
- etc (etcetera)存放系统配置文件
- usr (unix shared resources)用于存放共享的系统资源
- home 存放用户文件的根目录
- root 超级用户目录
- dev (devices)用于存放设备文件
- lib (library)存放跟文件系统中的程序运行所需要的共享库及内核模块
- mnt (mount)系统管理员安装临时文件系统的安装点
- boot 存放用于系统引导时使用的各种文件
- tmp (temporary)用于存放各种临时文件
- var (variable)用于存放运行时需要改变数据的文件

4.4 LiteOS操作系统

OpenHarmony整体遵从分层设计，从下向上依次为：内核层、系统服务层、框架层和应用层，本章内容主要关注其内核层。在内核层上，OpenHarmony支持如下几种类型的操作系统：

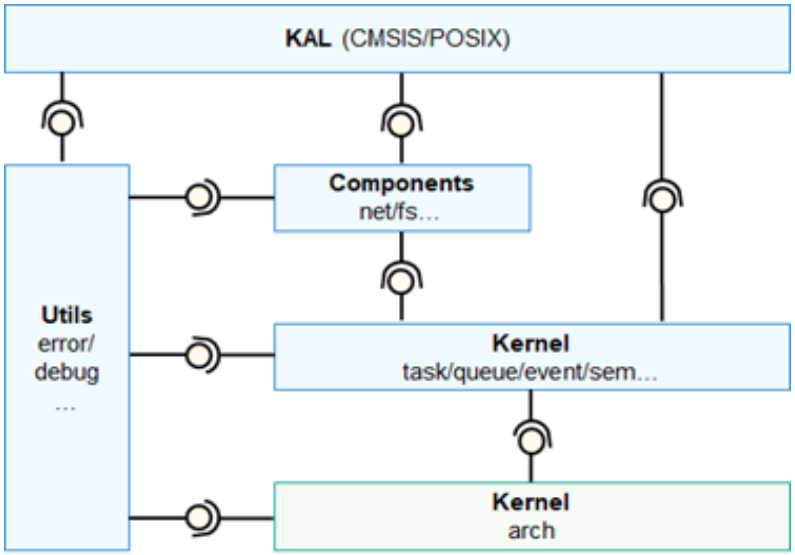
（1）LiteOS-M轻量系统，面向MCU类处理器例如Arm Cortex-M、RISC-V 32位的设备，硬件资源极其有限，支持的设备最小内存为128KiB，可以提供多种轻量级网络协议，轻量级的图形框架，以及丰富的IOT总线读写部件等。可支撑的产品如智能家居领域的连接类模组、传感器设备、穿戴类设备等。

（2）LiteOS-A小型系统，面向应用处理器例如Arm Cortex-A的设备，支持的设备最小内存为1MiB，可以提供更高的安全能力、标准的图形框架、视频编解码的多媒体能力。可支撑的产品如智能家居领域的IP Camera、电子猫眼、路由器以及智慧出行域的行车记录仪等。

（3）Linux内核标准系统，面向应用处理器例如Arm Cortex-A的设备，支持的设备最小内存为128MiB，可以提供增强的交互能力、3D GPU以及硬件合成能力、更多控件以及动效更丰富的图形能力、完整的应用框架。可支撑的产品如高端的冰箱显示屏。

4.4.1 LiteOS-M操作系统

LiteOS-M内核是面向IoT领域构建的轻量级物联网操作系统内核，具有小体积、低功耗、高性能的特点。其代码结构简单，主要包括内核最小功能集、内核抽象层、可选组件以及工程目录等。



在硬件抽象层，主要是各种硬件架构提供支持，LiteOS-M已经支持ARM Cortex-M3、ARM Cortex-M4、ARM Cortex-M7、ARM Cortex-M33、RISC-V等主流架构。

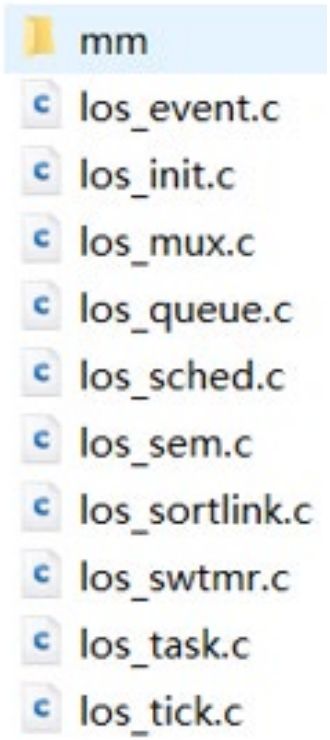
CPU体系架构分为通用架构定义和特定架构定义两层，通用架构定义层为所有体系架构都需要支持和实现的接口，特定架构定义层为特定体系架构所特有的部分。在新增一个体系架构的时候，必须需要实现通用架构定义层，如果该体系架构还有特有的功能，可以在特定架构定义层来实现。CPU体系架构规则如下表。

规则	通用体系架构层	特定体系架构层
头文件位置	kernel/arch/include	kernel/arch/<arch>/<arch>/<toolchain>/
头文件命名	los_<function>.h	los_arch_<function>.h
函数命名	Halxxxx	Halxxxx

LiteOS-M基础内核（Kernel task/queue/event/sem...）

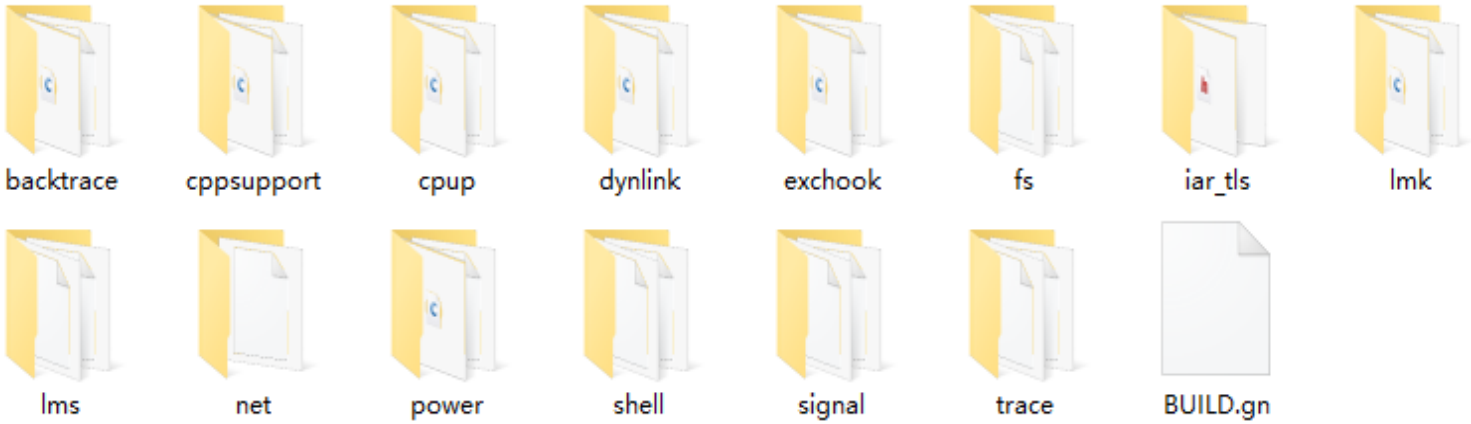
LiteOS-M基础内核主要包括：任务管理、中断管理、内存管理、消息队列、信号量、时间管理，事件管理和定时器等。基础内核的文件内容如下图所示。

```
/kernel/liteos_m
├── components              # 可选组件
│   ├── backtrace           # 回溯栈支持
│   ├── cppsupport          # C++支持
│   ├── cpup                # CPUP功能
│   ├── dynlink             # 动态加载与链接
│   ├── exchook             # 异常钩子
│   ├── fs                  # 文件系统
│   └── net                 # Network功能
├── kal                     # 内核抽象层
│   ├── cmsis               # cmsis标准接口支持
│   └── posix               # posix标准接口支持
├── kernel                  # 内核最小功能集支持
│   ├── arch                # 内核指令架构层代码
│   │   ├── arm             # arm32架构的代码
│   │   └── include         # 对外接口存放目录
│   ├── include             # 对外接口存放目录
│   └── src                 # 内核最小功能集源码
├── targets                 # 板级工程目录
├── utils                   # 通用公共目录
```



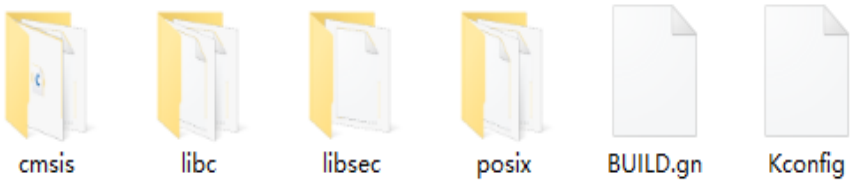
扩展组件（Components）

扩展组件，主要包括文件系统、网络模块、功耗模块、动态链接和调测工具等模块。主要文件如下图所示，每个模块的说明可参见：https://gitee.com/openharmony/docs/blob/master/zh-cn/device-dev/kernel/kernel-mini-extend.md



KAL（Kernel Abstraction Layer）层

KAL层是为了让应用层在不同的操作系统上实现统一的API调用功能而产生的，提供统一的标准接口，支持ARM内核的CMSIS库函数和POSIX接口。



Utils模块

Utils模块提供错误处理、调测等能力。

在开发板配置文件target_config.h配置系统时钟、每秒Tick数，可以对任务、内存、IPC、异常处理模块进行裁剪配置。系统启动时，根据配置进行指定模块的初始化。内核启动流程包含外设初始化、系统时钟配置、内核初始化、操作系统启动等。

4.4.2 LiteOS-A操作系统

LiteOS-A内核主要应用于小型系统，面向设备一般是M级内存，可支持MMU隔离，业界类似的内核有Zircon或Darwin等。

轻量级内核LiteOS-A重要的新特性如下：

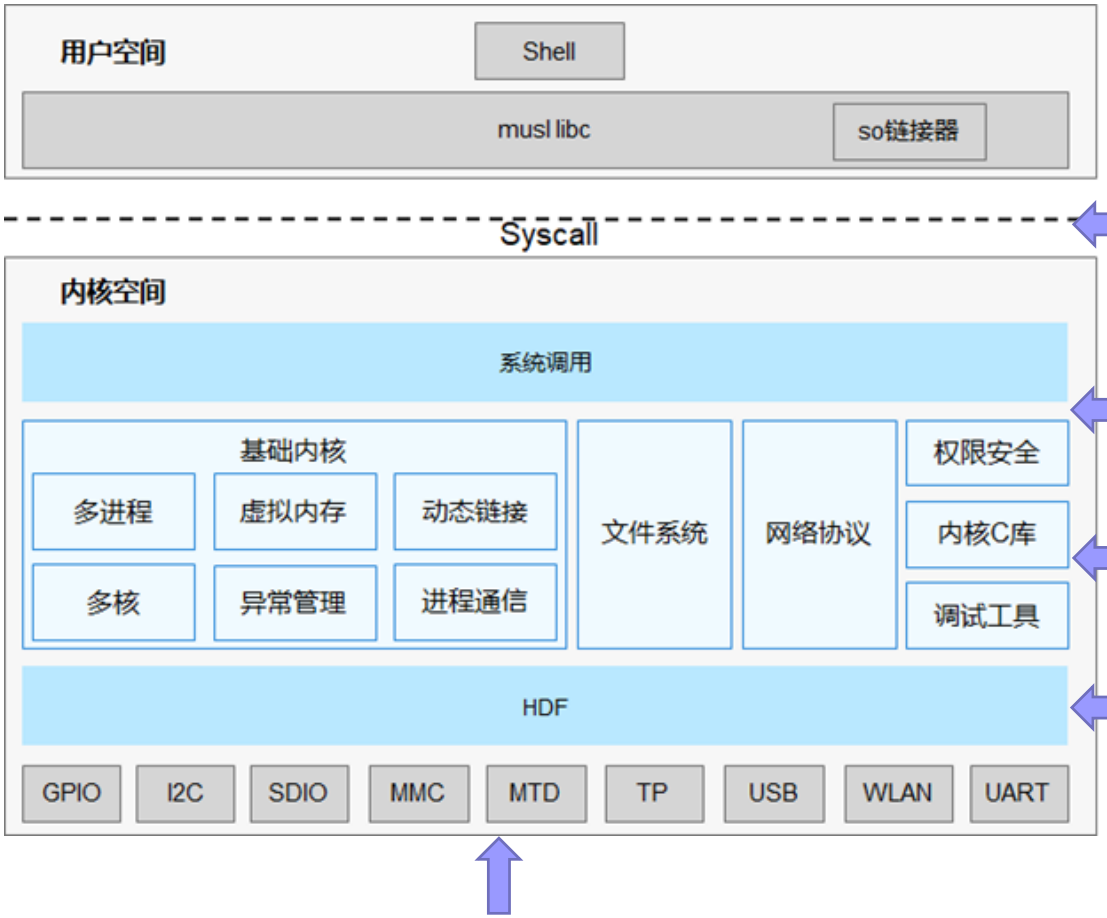
（1）新增了丰富的内核机制：新增虚拟内存、系统调用、多核、轻量级IPC（Inter-Process Communication，进程间通信）、DAC（Discretionary Access Control，自主访问控制）等机制，丰富了内核能力；为了更好的兼容软件和开发者体验，新增支持多进程，使得应用之间内存隔离、相互不影响，提升系统的健壮性。

（2）引入统一驱动框架HDF（Hardware Driver Foundation）：统一驱动标准，为设备厂商提供了更统一的接入方式，使驱动更加容易移植，力求做到一次开发，多系统部署。

（3）支持1200+标准POSIX接口，使得应用软件易于开发和移植，给应用开发者提供了更友好的开发体验。

（4）轻量级内核主要由基础内核、扩展组件、HDF框架、POSIX接口组成。轻量级内核的文件系统、网络协议等扩展功能（没有像微内核那样运行在用户态）运行在内核地址空间，主要考虑组件之间直接函数调用比进程间通信或远程过程调用要快得多。

LiteOS-A内核架构如下图



Syscall是用户调用内核的接口。

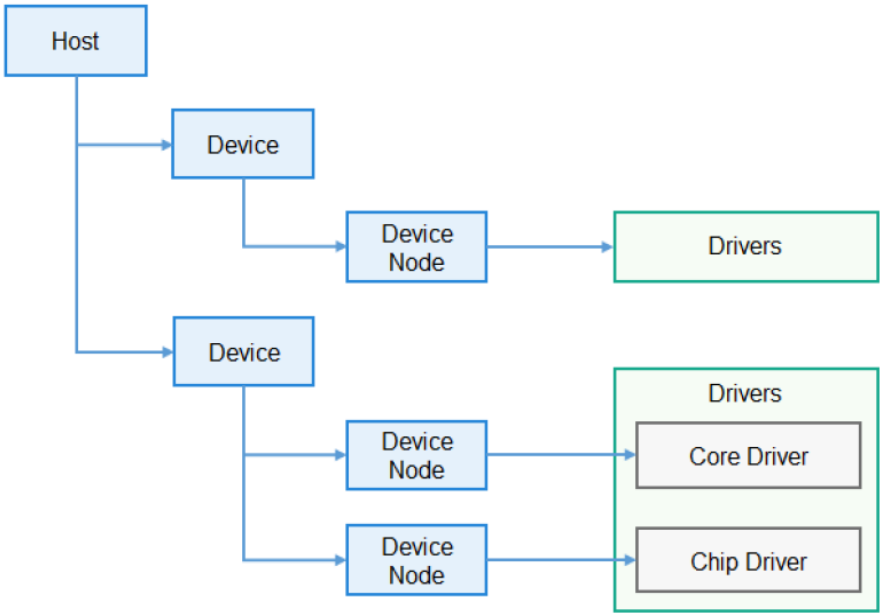
基础内核就是通用操作系统所具有的功能模块。

内核扩展模块主要提供文件系统、网络协议、权限安全、内核C库和调试工具等。

LiteOS-A内核引入了统一的驱动框架HDF，为驱动开发者提供驱动框架能力，包括驱动加载、驱动服务管理和驱动消息机制管理。

专有硬件服务子系统提供设备操作接口有：FLASH、GPIO、I2C、PWM、UART、WATCHDOG等，用户可以直接调用这些接口。

HDF框架以组件化的驱动模型作为核心设计思路，为开发者提供更精细化的驱动管理，让驱动开发和部署更加规范。HDF框架将一类设备驱动放在同一个host里面，开发者也可以将驱动功能分层独立开发和部署，支持一个驱动多个node，HDF框架管理驱动模型如下图所示。



HDF驱动框架的配置文件包含驱动设备描述与驱动私有信息，HDF框架定义的驱动按需加载方式的策略是由配置文件中的preload字段来控制。驱动的按序加载是通过配置文件中的priority（取值范围为整数0到200）来决定的，priority值越小，表示的优先级越高。驱动服务是HDF驱动设备对外提供能力的对象，由HDF框架统一管理。驱动服务管理主要包含驱动服务的发布和获取。HDF框架定义了驱动对外发布服务的策略，是由配置文件中的policy字段来控制。

谢谢！