# 绪论

## 总序

操作系统的“Page Fault”异常处理例程首先会判断，该数据的虚拟地址是否是该任务的合法地址？根据之前操作系统的设置，答案是“Yes”；然后取出保存在对应页表项中的扇区地址，把存储设备上交换区上对应的虚拟页内容读入到某空闲的物理内存页中；接着更新页表项内容，即把对应的物理页号写入页表项，把页表项中的存在位（Present Bit）置“1”；最后是返回该任务，让该任务重新执行访问数据的指令。这次处理器再次执行这条指令时，TLB还是会没有命中，但由于对应页表项内容合法，所以TLB会缓存该页表项，并完成虚拟地址到物理地址的转换，完成访问数据指令的执行。

相对于内存访问，交换区的扇区访问要慢很多。为了进一步提高系统的执行效率，当操作系统在让存储设备进行I/O访问时，可将当前任务设置为阻塞状态，并切换其他可以运行的任务继续执行。通过这种任务调度方式，可以充分发挥多道程序和分时多任务的整体执行效率

这是2022年暑假参加rCore社区第二阶段的课题， 我接着课题（异步操作系统）继续尝试做的一些工作。课题主要探索利用Rust先天对异步支持的优势来重新设计rCore并尝试为其添加异步的特性。 操作系统设计的意义：

1. 库操作系统（Library Operating System， 简称 LibOS）是根据某类应用的特殊需求，由某一高级编程语言将原本属于操作系统内核的某些资源管理功能，如文件磁盘 I/O、网络通信等，按照模块化的要求，以库的形式提供给应用程序的特殊操作系统。 它能代替操作系统内核合理地管理和控制所涉及的计算机资源，并将所涉及的计算机
2. 资源直接暴露给应用程序， 让应用程序直接访问底层（虚拟）硬件， 以便应用程序能够高效地运行。特别地，它与应用程序在编译时被链接到一起，形成一个只有单地址空间的二进制文件并工作在应用层，是构建 Unikernel 的一个必不可少的系统组件。
   * Linux 作为主流的开源操作系统发展已经走过三十多个年头了，历史的包袱已经略显沉重， 随着物联网的快速袭来， 单片机的开发也日益增多。 Linux Kernel内核中的许多module可能略显多余甚至很多存在功能冗余， 这样不急增加单片机的设计难度还为增加单片机的内存压力。可供定制和模块化良好的库操作系统可以缓解单片机开发中系统的选择匮乏。
3. 异步的另外一种含义是计算机多线程的异步处理。 与同步处理相对， 异步处理不用阻塞当前线程来等待处理完成，而是允许后续操作，直至其它线程将处理完成，并回调通知此线程。
   * 异步的需求主要来自任务的并发，而且是发生大量的任务请求。如果同步进行系统响应，那将是一件十分可怕的事情，系统以阻塞的方式对请求进行响应，那可能是一段很长的等待， 尤其可怕的是， CPU存在极大可能是一个空闲的状态。 例如， 存在一个I/O事件， 在请求I/O的控制权之后，CPU将无需为此事件提供服务，那么CPU就会空闲下来，从CPU资源使用的角度看这是十分不可取的，CPU的资源十分宝贵，一定需要让CPU尽可能处于忙的工作环境下。异步，异步的另外一种含义是计算机多线程的异步处理。与同步处理相对，异步处理不用阻塞当前线程来等待处理完成，而是允许后续操作，直至其它线程将处理完成， 并通过全局的唤醒机制回调通知此线程。 从而使得核心资源处于一个忙碌的状态。

ring\_scheduler: 自己的工作是继续tronado-os的工作， 实现一个类似shared\_ scheduler的ring\_scheduler，将其接入rCore并使得其可以工作在rCore中，并试着将其从内核模块中正交出来，形成一个独立的模块（Crate）。 而异步的设计往往需要底层硬件的支持， 因此需要完成对底层硬件的异步工作，而磁盘的读写被抽象为文件的读写，当硬件的异步完成时，需要对文件系统进行合理的改造使其能完成异步相应的工作需求。ring\_scheduler的接入需要在内核中实例化，提供相应的系统调用。

模块化的设计：预先的设想只是将模块在语言静态编译时就完成所有模块的载入工作，但是得益于Theseus的启发，有考虑在所有模块正交完成之后，尝试动态的内核模块的载入，其核心是对每个模块的elf的解析，从而在内核运行时动态载入。上述两者的模块化思路，前者主要依赖于语言的编译时，其实质只是将杂糅在一起的模块通过正交，变成独立且具有层次的的依赖树，模块之间的依赖通过各自之间ABI的调用形成约束；后者主要依赖语言的运行时，是对模块的进一步抽象，因为依赖于语言的运行时，所以需要独立的可以管理模块的内核单元对模块进行树状管理。

## 动机

很多程序必须执行大量独立且不需要串行化的任务。例如，数据库服务器可以监听和处理大量的客户端请求。因为这些请求没必要按照某个特定的顺序被处理，他们可以被当作独立的执行单元，原则上他们可以并行地执行。如果系统可以提供子任务并发执行地机制，这些应用程序将会运行得更好。

操作系统中，相对于内存访问，交换区的扇区访问要慢很多。为了进一步提高系统的执行效率，当操作系统在让存储设备进行I/O访问时，可将当前任务设置为阻塞状态，并切换其他可以运行的任务继续执行。通过这种任务调度方式，可以充分发挥多道程序和分时多任务的整体执行效率。

## 研究背景与意义

研究目的：异步的提出是为了提高程序在热点事件下的并发性能。

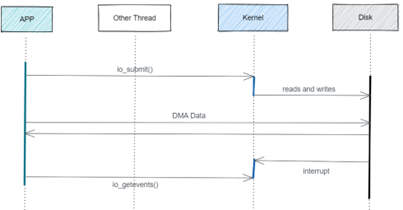
研究意义： 编译器与操作系统进行密切配合，用户态的异步系统调用会同编译器自动生成相应的系统调用请求代码和协程控制块数据结构；只有在第一次系统调用和最后一次系统调用时需要进入内核空间；中间各次的系统调用只进行系统调用结果查询和进程、线程或协程切换。

研究好处： 异步调度下，操作系统应对高并发的任务时，可以获得更好的响应表现，CPU可以一直处在忙碌的状态。无栈的协程，内部的状态转换通过状态机图实现，无需函数主动调用yield，内部也只是状态的转换，无需多余的栈空间开销。

## 国内外研究现状

### 国外现状

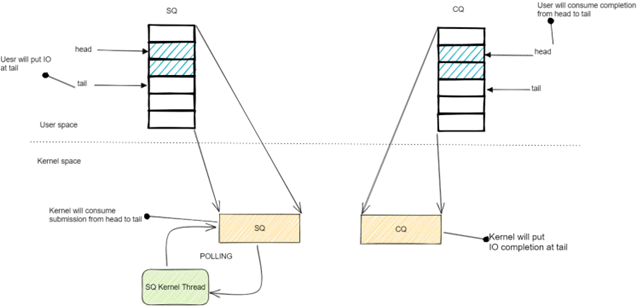
#### Libaio



Libaio的工作流

Libaio的工作流如[[figure:c1libaio]](#figure:c1libaio)所示， 以read的请求来说， direct IO的模式会把从磁盘上读取的数据直接返回给了用户态的内存空间，不会在内核中缓存，当存在多次重复读取的场景，每次都需要读取磁盘，这样会增加磁盘的压力。

#### IO\_URING



io\_uring的工作流

io\_uring的工作流如[[figure:c1iouring]](#figure:c1iouring)所示，io\_uring的基本逻辑与linux-aio是类似的，提供两个接口，一个将I/O请求提交到内核，一个从内核接收完整的事件。但是在设计上是真正的异步。只要设置了合适的参数，在系统调用上下文中只是将请求放入队列，而不会做额外的工作，保证了应用不会被阻塞。io\_uring无需像aio那样使用 poll+read/write来处理sockets，只需要提交一个阻塞式的度，请求完成之后，就会出现在CQ（completion ring）。io\_uring实例有两个环形队列，在内核和用户空间之间共享， 这两个队列都是单生产者、单消费者，提供无锁的接口，内部使用内存屏障同步。

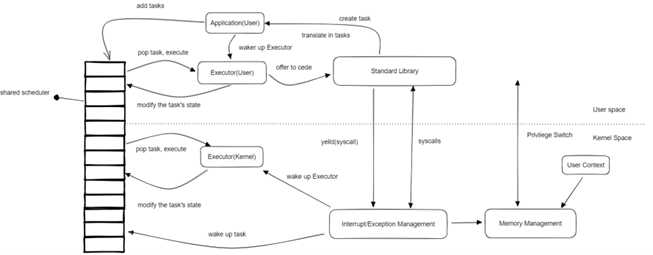
**io\_uring可以工作在三种模式上**

1. 中断驱动的默认模式。通过提交相应的I/O请求，然后直接通过CQ状态判断是否完成。
2. 轮询模式，这种模式需要文件系统和块设备支持轮询功能，相比中断驱动模式，这种虽然延迟低，但可能会消耗更多CPU资源。当一个读或写请求提交给轮询上下文之后，应用必须轮询CQ队列，判断请求是否已经完成。
3. 内核轮询模式，这种模式会创建一个内核线程来执行SQ的轮询工作。

使用这种模式的io\_uring实例， 应用无需切到到内核态 就能触发（issue）I/O 操作。通过SQ来提交SQE，以及监控CQ的完成状态，应用无需任何系统调用，就能提交和收割 I/O。

### 国内现状

#### tornado-os的共享调度器



shared\_scheduler的工作流

调度器直接将所使用的代码、任务池资源都共享到用户，由用户运行和内核相同的代码，以此与内核以相同的逻辑处理任务池中的任务，从而从共享的任务池中得到下一个任务。

总结一下，基本上，异步系统调用的执行过程应当符合：

1. 第一次异步系统调用时
   * 用户进程准备系统调用参数、发出系统调用请求；
   * 内核进程将映射共享内存、发起相应服务协程的异步执行；
   * 内核进程执行完服务协程后，再响应队列保存返回值，并通过用户态中断通知应用进程；
2. 第二次异步系统调用时
   * 用户进程再请求队列转杯系统调用参数；在共享内存的响应队列中查看一次系统调用的结果；
   * 内核进程在完成第一个服务协程后，在共享内存的响应队列中保存返回值，主动查询新的系统调用请求，并执行；如果没有新的请求，则让出CPU；

发展：异步的调度的研究应该会是未来网络系统服务的一个方向，随着网络的普及和发展，异步是一种可行的高并发设计思路，是有其研究的价值和意义。

# 前置知识

## Rust：依赖语言的程序设计

Rust 是一门十分高效的编程语言， 其设计目的是在编译时提供强大的类型和内存安全保证，它将高级托管语言的强大功能和表达能力与类似c语言的无垃圾收集或底层运行时的效率相结合。编写操作系统可以利用Rust的许多特性来实现语言内、安全的操作系统设计，并使用crate (Rust的项目容器和翻译单元)实现源代码级模块化。而Cargo.toml包含源代码和依赖项清单。编写操作系统是没有使用Rust的标准库，但使用了它的基本核心和alloc库。

Rust的所有权模型是其编译时内存安全和管理的关键。所有权基于仿射类型，其中一个值最多只能使用一次。在Rust中，每个值都有一个所有者，例如，下面的代码中分配的字符串值“hello!”由hello变量拥有。在一个值被移动之后，例如，如果“hello!”在第5行中被从hello移动到owned\_string (第14行)，它的所有权将被转移，之前的所有者(hello)将不能再使用它。

fn main() {  
 let hel: &str;  
 {  
 let hello = String::from("hello!");  
 // consume(hello); // -> "value moved" error in here.  
 let borrowed\_str: &str = &hello;  
 hel = substr(borrowed\_str);  
 // println!("{}", hel); // -> lifetime error  
 }  
}  
fn substr<'a> (input\_str: &'a str) -> &'a str {  
 &input\_str[0..3] // return value has lifetime 'a  
}  
fn consume(owned\_string: String) { }

当所有者的作用域结束时，例如，在词法块的末尾，通过编译器插入对其析构函数的调用，所有者的值将被丢弃(释放)。Rust中的析构函数是通过实现给定类型的Drop特征来实现的，其中自定义的Drop处理程序可以执行除释放内存之外的任意操作。在上面的第8行中，hello字符串超出了作用域，并由它的删除处理程序自动释放。

还可以通过 borrow 来获得对它们的引用(第6行)，而且这些引用的生命周期不能超过所拥有值的生命周期。第11行中的语法将input\_str参数的生命期命名为’a，并指定返回的&str引用具有相同的生命期’a。返回的&str引用在第7行中被赋值给hel，这将在第9行中由于生命周期的原因而被释放，因为在第8行中，hel将在它最初从(hello)借来的拥有值被删除后使用。Rust的编译器包括一个借用检查器来执行这些生命周期规则，以及别名异或可变性的核心原则，其中可以有多个不可变引用，也可以有单个对值的可变引用，但不能同时存在这两个引用。这允许它静态地确保堆栈和堆上的值的内存安全。

系统设计还广泛利用了Rust泛型的特征，是一种抽象类型的声明，它指定了类型必须实现的方法集，类似于OOP语言中的多态接口。trait可用于在泛型类型参数上设置边界。例如，函数fn print\_str<T: Into<String>>(s: T)，其中<T: Into<String>约束了入参s的类型，其必须实现Into<String>的泛型特征, 如果s没有符合类型约束则在代码编译时就会被抛出错误。

## 操作系统

### Riscv

#### Riscv的机器模式

risc-v架构定义了3种工作模式，又称为特权模式（privileged mode）。

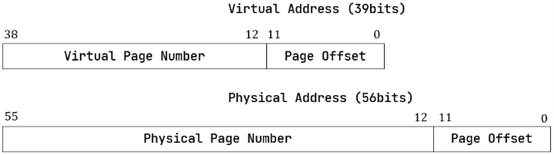
* 机器模式（machine mode），简称M模式；
* 监督模式（supervisor mode），简称S模式；
* 用户模式（user mode），简称U模式。

Riscv架构定义机器模式为必选模式，另外两种模式为可选模式，通过不同的模式组合可以实现不同的系统。其中机器模式拥有最高的权限，而用户模式的权限则是最低的，权限高的模式对于权限低的模式是透明的，权限低的无法获得权限高的所拥有的资源。从而达到了层与层之间的隔离，设计上指令集是安全的。

SBI（Supervisor Binary Interface）是Supervisor Execution Environment (SEE)和supervisor之间的接口。 它允许主管通过使用 ecall 指令执行一些特权操作。 SEE 和 supervisor 的示例有：Unix 类平台上的 M-Mode 和 S-Mode，其中 SBI 是它们之间的唯一接口，以及 Hypervisor extended-Supervisor (HS) 和 Virtualized Supervisor (VS)。

#### Sv39

**地址格式与组成**

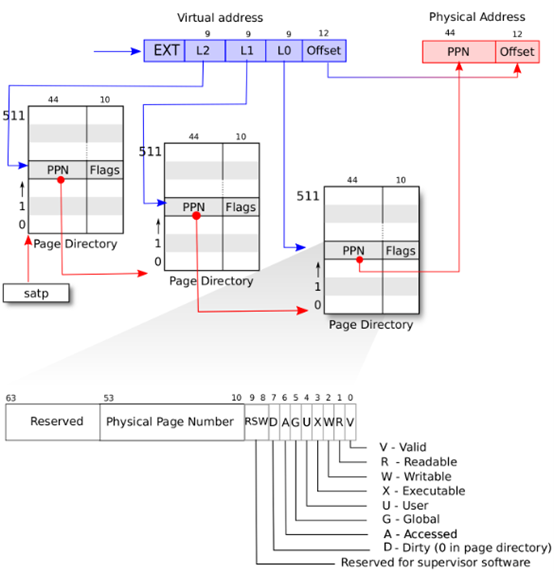


Sv39的地址格式

采用分页管理，单个页面的大小设置为 4KiB ，每个虚拟页面和物理页帧都对齐到这个页面大小，也就是说虚拟/物理地址区间 [0,4KiB) 为第 0 个虚拟页面/物理页帧，而 [4KiB,8KiB) 为第 1 个，以此类推。 4KiB 需要用 12 位字节地址来表示，因此虚拟地址和物理地址都被分成两部分：它们的低 12 位，即 [11:0] 被称为 页内偏移 (Page Offset) ，它描述一个地址指向的字节在它所在页面中的相对位置。而虚拟地址的高 27 位，即 [38:12] 为它的虚拟页号 VPN，同理物理地址的高 44 位，即 [55:12] 为它的物理页号 PPN，页号可以用来定位一个虚拟/物理地址属于哪一个虚拟页面/物理页帧。

地址转换是以页为单位进行的，在地址转换的前后地址的页内偏移部分不变。可以认为 MMU 只是从虚拟地址中取出 27 位虚拟页号，在页表中查到其对应的物理页号（如果存在的话），最后将得到的44位的物理页号与虚拟地址的12位页内偏移依序拼接到一起就变成了56位的物理地址。

**地址转换过程**



Sv39地址转换过程

在 SV39 模式中，采用三级页表，即将 27 位的虚拟页号分为三个等长的部分，第 26-18 位为一级页索引 VPN0 ，第 17-9 位为二级页索引 VPN1 ，第 8-0 位为三级页索引 VPN2 。

页表分为一级页表（多级页表的根节点），二级页表，三级页表（多级页表的叶节点）。每个页表都用 9 位索引，因此有 个页表项，而每个页表项都是 8 字节，因此每个页表大小都为 。正好是一个物理页的大小。从而可以把一个页表放到一个物理页中，并用一个物理页号来描述它。事实上，一级页表的每个页表项中的物理页号可描述一个二级页表；二级页表的每个页表项中的物理页号可描述一个三级页表；三级页表中的页表项内容则和先前提到的页表项一样，包含物理页号，即描述一个要映射到的物理页。

具体来说，假设有虚拟地址： 首先会记录装载「当前所用的一级页表的物理页」的页号到 satp 寄存器中；

* 把 作为偏移在一级页表的物理页中找到二级页表的物理页号；
* 把 作为偏移在二级页表的物理页中找到三级页表的物理页号；
* 把 作为偏移在三级页表的物理页中找到要访问位置的物理页号；

物理页号对应的物理页基址（即物理页号左移12位）加上 offset 就是虚拟地址对应的物理地址。

这样处理器通过这种多次转换，终于从虚拟页号找到了一级页表项，从而得出了物理页号和虚拟地址所对应的物理地址。刚才我们提到若页表项满足 R,W,X 都为 0，表明这个页表项指向下一级页表。在这里三级和二级页表项的 R,W,X 为 0 应该成立，因为它们指向了下一级页表。

### API与ABI

从使用操作系统的角度会比较容易对操作系统内核的功能产生初步的认识。操作系统内核是一个提供各种服务的软件，其服务对象是应用程序，而用户是通过应用程序的服务间接获得操作系统的服务的，因此操作系统内核在一般用户面前是一个透明地。但应用程序需要访问操作系统获得操作系统的服务，这就需要通过操作系统的接口才能完成。操作系统与运行在用户态软件之间的接口形式就是应用程序二进制接口 (ABI, Application Binary Interface)。

操作系统不能只提供面向单一编程语言的函数库的编程接口 (API, Application Programming Interface) ，它的接口需要考虑对基于各种编程语言的应用支持，以及访问安全等因素，使得应用软件不能像访问函数库一样的直接访问操作系统内部函数，更不能直接读写操作系统内部的地址空间。为此，操作系统设计了一套安全可靠的二进制接口，即系统调用接口 (System Call Interface)。系统调用接口通常面向应用程序提供了 API 的描述，但在具体实现上，还需要提供 ABI 的接口描述规范。

在现代处理器的安全支持（特权级隔离，内存空间隔离等）下，应用程序就不能直接以函数调用的方式访问操作系统的函数，和直接读写操作系统的数据变量。不同类型的应用程序可以通过符合操作系统规定的系统调用接口，发出系统调用请求，来获得操作系统的服务。操作系统提供完服务后，返回应用程序继续执行。

#### API 与 ABI 的区别

应用程序二进制接口ABI是不同二进制代码片段的连接纽带。ABI定义了二进制机器代码级别的规则，主要包括基本数据类型、通用寄存器的使用、参数的传递规则、以及堆栈的使用等等。ABI与处理器和内存地址等硬件架构相关，是用来约束链接器 (Linker) 和汇编器 (Assembler) 的。在同一处理器下，基于不同高级语言编写的应用程序、库和操作系统，如果遵循同样的ABI定义，那么它们就能正确链接和执行。

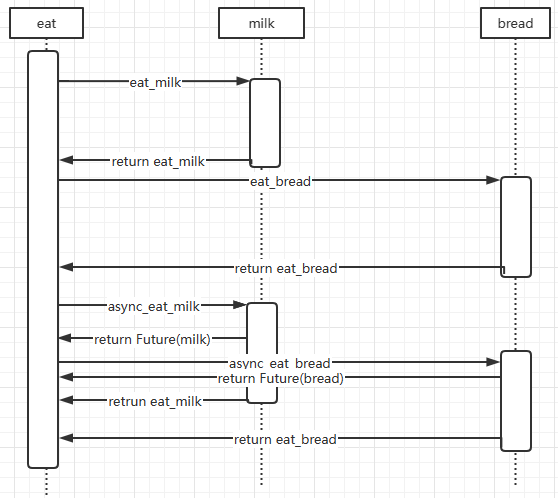
应用程序编程接口API是不同源代码片段的连接纽带。API定义了一个源码级（如C语言）函数的参数，参数的类型，函数的返回值等。因此API是用来约束编译器 (Compiler) 的：一个API是给编译器的一些指令，它规定了源代码可以做以及不可以做哪些事。API与编程语言相关，如libc是基于C语言编写的标准库，那么基于C的应用程序就可以通过编译器建立与libc的联系，并能在运行中正确访问libc中的函数。

## 异步

Rust 语言以async/await的形式为协作式多任务处理提供了一流的支持。

### Futrue

如下我们可以通过下面这个图理解Future



Breakfirst的同步和异步

这张序列图显示了一个函数，该函数为eat函数。先执行了同步的eat\_milk和eat\_bread函数，又执行了async\_eat\_milk和async\_eat\_bread函数。先后两次同步和两次异步。

在同步调用的时候，函数是排队执行的，从牛奶到面包是依次进行，在牛奶喝完之前，不可以啃面包，事情只能一步一步完成，而完成的信号就是相应函数的返回。

在异步调用的时候，喝牛奶时，可以返回Futre，牛奶的工作会被置换到后台，然后掉起啃面包的行为，这个动作同样是异步的，换句话说，当面包的工作出现阻塞时，面包同样可以被置换到后台，让出资源。这样的工作使得相应函数可以更早地被执行，然后形成宏观上相应动作地并发执行，而无需在接收函数返回之前长时间的等待。

### Rust中的Future

|  |  |
| --- | --- |
| pub trait Future {  type Output;  fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>; } | pub enum Pool<T> {  Ready(T),  Pending, } |

Rust’s Future

在上述的代码中，Future的trait接口，Output将会被返回一个异步的值。例如在时序图中提到的async\_eat\_milk的返回值。 而poll函数返回的是一个可以被轮询检查自身状态的值，其具体的结构如上图右部分。

当poll返回的值Ready(T)时，函数将完成的值进行Poll的封装，返回其状态完成的变体。否则，将返回其其他状态的变体，向其范围内全局的调用者发出该工作尚未完成的信号。

poll方法采用两个参数： self:Pin<&mut Self>和cx: &mut Context。前者的行为类似普通引用，只是将值固定到内存的位置。后者存在的目的时为了Waker的实例可以被异步任务接收。这使得异步任务可以发出信号，表面其此时的状态（完成或一部分已完成）。由于主任务知道其将在准备就绪时收到通知，因此不需要一遍一遍调用。

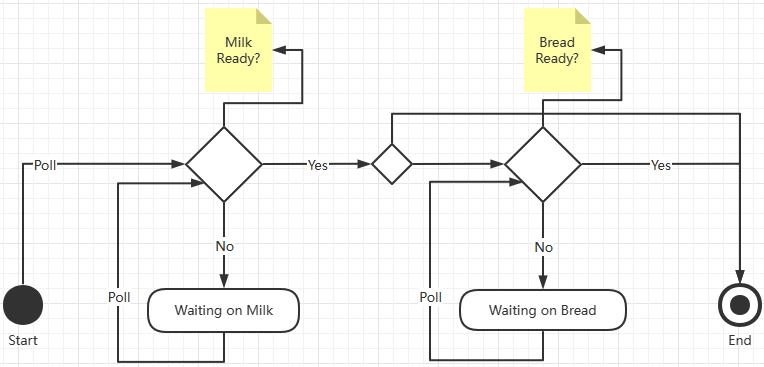
### Future使用的例子

下面的代码可以看成是早餐的Rust伪码描述，伪码中省略了不必要的细节，只给了能够说明问题的框架

[ caption = BREAKFIRST ]  
enum FoodState {  
 Cooked,  
 Eating,  
 Finished,  
};  
async fn milk () { /\* ... \*/ }  
async fn bread () { /\* ... \*/ }  
async fn eat () {  
 /\* ... \*/  
 milk.await;  
 bread.await;  
 /\* ... \*/  
}

#### 状态转换

在编译器的后台，上述的代码将会被转化为状态机，每个调用将会形成不同的状态，根据上述代码可以转化出下面的状态。每个状态表示函数中不同的暂停点。“开始”和“结束”状态表示函数在其执行的开始和结束时。“Waiting on Milk”状态表示函数当前正在等待第一个结果。同样，“Waiting on Bread”状态表示函数等待第二个结果的暂停点。状态机通过使每个调用成为可能的状态转换变为Rust的Future。



Breakfirst的状态机图

该图使用箭头表示状态开关，使用菱形表示替代方法。该图使用箭头表示状态开关，使用菱形表示替代方法。例如，牛奶没有喝完，则采用标记为“no”的路径，并达到“Waiting on Milk”状态。否则，将采用“是”路径。没有标题的小红色菱形表示函数eat的分支。

我们看到第一次调用启动函数poll并让它运行，直到它到达尚未准备好的Future。如果路径上的所有Future都准备就绪，则该函数可以运行到“结束”状态，在该状态中，它返回包装在 Poll::Ready.否则，状态机将进入等待状态并返回Poll::Pending。在下一次调用时，状态机将从上次等待状态开始，并重试上次操作。

#### 状态保存

为了能够从上一个等待状态继续，状态机必须在内部跟踪当前状态。此外，它必须保存下次调用时继续执行所需的所有变量。这就是编译器真正可以发光的地方：因为它知道何时使用哪些变量，因此它可以自动生成具有所需变量的结构。

|  |  |
| --- | --- |
| async fn eating(min\_energy: usize) -> String {  let food = async\_eat\_milk().await;  if food.energy() < min\_energy {  content + &async\_eat\_bread().await  } else {  content  } } | enum BreakfirstStateMachine {  State(StartState),  WaitingOnMilk(WaitingOnMilkState),  WaitingOnBread(WaitingOnBreadState),  End(EndState), } |

Breakfirst的状态机描述和与之对应的可能状态

在编译器中， 可以尝试这样理解， eating函数将会生成上图右边的状态。在Start和WaitingOnMilk的状态下，需要将参数存储下来，为food.energy()和min\_energy的对对比做准备。WaitingOnMilk的状态将会被WaitingOnMilkState所记录，其将会被全局的调度器所调用。当状态机继续运行时，async\_eat\_milk需要再次轮询WaitingOnMilkState，因此需要保存它。

为了使得描述方便， 每个状态都被定义一个单独的枚举变体（如上图右半部分），并将相应的状态结构作为字段添加到每个变体中。为了实现状态转换，编译器基于以下函数生成特征的实现：

impl Future for BreakfirstStateMachine {  
 type Output = String;  
 fn poll(self: Pin<&mut Self>, en: &mut Energy) -> Poll<Self::Output> {  
 loop {  
 match self {  
 BreakfirstStateMachine::Start(state) => {}  
 BreakfirstStateMachine::WaitingOnMilk(state) => {}  
 BreakfirstStateMachine::WaitingOnBread(state) => {}  
 BreakfirstStateMachine::End(state) => {}  
 }  
 }  
 }  
}

Future的类型是因为它是函数的返回类型。为简单起见，我们只显示简化的代码，不处理Pin、所有权、生存期等。因此，此代码和以下代码应被视为伪代码，而不是直接使用。当然，真正的编译器生成的代码可以正确处理所有内容，尽管可能以不同的方式处理。

为了使代码摘录保持较小，我们分别显示每个臂的代码。让我们从Start状态开始

BreakfirstStateMachine::Start(state) => {  
 let eat\_milk\_future = async\_eat\_milk();  
 let state = WaitingOnMilkState {  
 min\_energy: state.min\_energy,  
 eat\_milk\_future,  
 }  
 \*self = BreakfirstStateMachine::WaitingOnMilk(state);  
}

状态机处于函数开头的状态Start。在这种情况下，执行从函数体到第一个.await为了处理该操作，将状态机的状态更改为WaitingOnMilk，其中包括结构的构造。

由于语句是在循环中执行的，因此执行跳转到接下来的状态：

BreakfirstStateMachine::WaitingOnMilk(state) => {  
 match state.eat\_milk\_future.poll(en) {  
 Poll::Pending => return Poll::Pending,  
 Poll::Ready(energy) => {  
 if food.energy() < state.min\_energy {  
 let eat\_bread\_future = async\_eat\_bread();  
 let state = WaitingOnBreadState {  
 energy,  
 eat\_bread\_future,  
 }  
 \*self = BreakfirstStateMachine::WaitingOnBread(state);  
 } else {  
 \*self = BreakfirstStateMachine::End(EndState);  
 return Poll::Ready(energy);  
 }  
 }  
 }  
}

在这个分支中，首先调用Poll::Pending如果还没有准备好，我们退出循环并返回self, 由于在这种情况下保持该状态 WaitingOnMilk，因此状态机上的下一个调用将进入同一分支并再次轮询。

准备就绪后，我们将结果分配给变量并继续执行函数的代码： 如果小于保存在状态结构中的值，则异步开始啃食面包。此时再次将操作转换为状态更改，这次转换为WaitingOnBread状态。由于正在执行内部循环，因此执行之后直接跳转到新状态的分支，然后再次轮询。

如果我们进入条件的另一条分支，则不会发生进一步的操作。从而到达函数的末尾并返回包装在 Poll::Ready，并将当前状态更改为End状态。

WaitingOnBread的代码如下:

BreakfirstStateMachine::WaitingOnBread(state) => {  
 match state.async\_eat\_bread.poll(en) {  
 Poll::Pending => return Poll::Pending,  
 Poll::Ready(bread\_energy) => {  
 \*self = BreakfirstStateMachine::End(EndState);  
 return Poll::Ready(state.energy + &bread\_energy);  
 }  
 }  
}

与状态WaitingOnMilk类似，我们首先轮询async\_eat\_bread如果它仍然挂起，我们退出循环并返回Poll::Pending否则，我们可以执行函数的最后一个操作：将变量与将来的结果连接起来。我们将状态机更新为End状态，然后返回包装在Poll::Ready中的结果。

最后的代码为

BreakfirstStateMachine::End(state) => {  
 panic!("poll called after Poll::Ready was returned");  
}

Future返回后不应再次轮询，因此，如果在对于已经处于该状态时被调用，编译器应当抛出panic。

# 系统设计

## 系统内核

### 内存管理

### 进程通信

#### 哲学家就餐

## 异步I/O

### ring\_scheduler

### 异步

## 模块设计

### 静态模块

在动态模块的设计中，系统模块设计应当遵循以下原则：

1. 最小正交化的模块
2. 树状依赖和空间隔离

#### 最小正交的模块

#### 树状依赖和空间隔离

### 动态模块

在动态模块的设计中，系统的模块设计应当遵循以下三个原则：

1. 需要获取所有模块的运行时的持久边界
2. 最大限度地发挥语言（Rust）和编译器的作用
3. 最小化模块之间的状态溢出

#### 需要获取所有模块的运行时的持久边界

系统内核中的模块组件具有明确定义的边界（严格化的正交模块），并在整个运行时保持不变： 在实现时，系统模块组件以Rust独立的Crate的形式存在；在编译时，系统模块组件以一组加载的内存区域的形式存在；在运行时，系统模块组件以一组加载内存区域的形式存在，内存区域具有每部分的边界和依赖元数据。

上述的设计原则每一个内核的系统模块组件都需要遵循。运行时，可以显示识别系统模块组件的边界是系统内核中组件隔离和状态管理的基础。

在运行时，系统内核根据需要将所有系统模块组件加载并链接到系统中。简而言之，这需要找到并解析系统模块组件对象，将其部分加载到内存中，解析其依赖，根据依赖树，以写入连接器重定位条目，根据需要递归加载任何丢失的系统模块组件，并向符号映射添加新的公共符号。基于此可以为内核进化和故障恢复提供理论基础。加载的系统模块组件集定义了一个系统模块组件空间，一个包含所有系统模块组件公共符号的真正的名称空间，用于快速解析单元格之间的依赖关系。每个加载的系统模块组件节点跟踪其组成部分和存储区域包含它们。每个系统模块组件中的部分对应于其Crate的目标文件中的部分，例如，可执行文件、只读数据和读写数据部分。每个加载的分段节点跟踪其大小、在存储器中的位置以及双向依赖性(输入和输出)；额外的元数据用于加速系统模块组件交换和其他系统功能。

系统模块组件边界的持久性降低了复杂性: 系统内核的持久性系统模块组件边界在其存在的所有阶段提供了一致的系统结构抽象。这将会降低了开发者对系统的理想模型的复杂性，并简化了故障恢复和演化逻辑，因为系统内核可以在运行时从相同的面向系统模块组件的角度自省和管理它自己的代码。从顶层应用程序和库到核心内核组件的一切都可以作为系统模块组件来观察。这使得系统内核能够

* 实现统一适用于任何单元的单一机制，即模块交换，以及
* 以安全的方式从多个系统层(例如，应用和内核组件)联合进化模块。

#### 最大限度地发挥语言（Rust）和编译器的作用

通过使编译器能够最大限度地检查安全性和正确性不变量来最大限度发挥语言的力量。

将系统内核的执行环境与该语言的运行时模型相匹配，并在Rust等现代语言提供的强大的静态类型系统中实现操作系统概念。这将编译器检查的不变量(例如，没有悬空引用)扩展到了所有类型的资源，而不仅仅是语言中内置的那些。

依赖语言设计有两个主要好处：

* 首先，它使编译器能够接管资源管理职责，减少了操作系统必须维护的状态，从而减少了状态溢出并加强了隔离。
* 它使编译器能够在理解代码行为的过程中应用安全检查，从应用程序到核心内核组件实现端到端安全，并将语义运行时错误转化为编译时错误。

相比之下，传统的非语言方法依赖于硬件保护和运行时检查来维护安全性、隔离性和正确性的不变量。这些特性对编译器是透明的，需要不安全的代码。甚至现有的安全语言操作系统。在语言级别的安全代码和底层的不安全核心之间有一个缺口，后者将语言所需的抽象实现为一个黑盒。

#### 最小化模块之间的状态溢出

由于系统内核的组件结构是模块化的，因此状态溢出只能发生在跨越模块边界并导致接收单元状态改变的交互(例如，函数调用)中。

# 系统实现

## 内核

### KernelHartInfo

#[repr(C)]  
pub struct KernelHartInfo {  
 hart\_id: usize,  
 // ...  
 asid\_alloc: (LinkedList<usize>, usize), // (空余的编号回收池，目前已分配最大的编号)  
 user\_mm\_sets: (LinkedList<MemorySet>, usize), // (注册的用户地址空间映射，上一次进入的用户地址空间编号)  
}

KernelHartInfo游走在内核态和用户态之间。记录了如CPU的核心编号， 用户空间的地址映射信息，地址分配的信息等等。

### 地址空间的映射

pub async fn prepare\_user<S: Into<String>>(user: S, kernel\_stack\_top: usize) {  
 let user: String = user.into();  
 let mut user\_memory = load\_user(&user).await;  
 let user\_asid = user\_memory.address\_space\_id.into\_inner();  
 let swap\_cx\_va = VirtualAddress(swap\_contex\_va(user\_asid));  
 let swap\_cx\_vpn = VirtualPageNumber::floor(swap\_cx\_va);  
 let swap\_cx\_ppn = user\_memory  
 .mapping  
 .translate(swap\_cx\_vpn)  
 .unwrap()  
 .page\_number();  
 let swap\_cx = unsafe {  
 (swap\_cx\_ppn .start\_address() .0 .wrapping\_add(KERNEL\_MAP\_OFFSET) as \*mut trap::SwapContext) .as\_mut() .unwrap()  
 };  
 let \_user\_satp = user\_memory.mapping.get\_satp(user\_memory.address\_space\_id);  
 let user\_stack\_handle = user\_memory .alloc\_page\_range(STACK\_SIZE, Flags::READABLE | Flags::WRITABLE | Flags::USER) .expect("alloc user stack");  
  
 let user\_stack\_top = user\_stack\_handle.end.0;  
 assert!( KernelHartInfo::load\_user\_mm\_set(user\_memory), "try load memory set with exited");  
 let kernel\_satp = satp::read().bits();  
 let tp = hart::read\_tp();  
 \*swap\_cx = trap::SwapContext::new\_to\_user( kernel\_satp, 0, tp, kernel\_stack\_top, user\_stack\_top, user\_trap\_handler as usize,);  
 swap\_cx.set\_gp(crate::SHAREDPAYLOAD\_BASE);  
 swap\_cx.set\_tp(user\_asid);  
}

### 内核态与用户态之间的转化

pub unsafe extern "C" fn supervisor\_to\_user() -> ! {  
 core::arch::asm!(  
 "csrw satp, a1  
 sfence.vma", // 刷新页表  
 // 从 SwapContext 中恢复用户的上下文  
 // 将用户的 a0 寄存器保存在 sscratch 寄存器中，  
 // 这样子可以在最后一步将它和 a0（ctx） 进行交换  
 "  
 ld t0, 9\*8(a0)  
 csrw sscratch, t0  
 ",  
 "  
 // ... 恢复通用寄存器的上下文 ...  
 ",  
 // 恢复用户的 a0 寄存器，并且保存交换栈顶在 sscratch 寄存器中  
 "csrrw a0, sscratch, a0",  
 // 返回到用户态  
 "sret",  
 options(noreturn)  
 )  
}

用户态切换到内核态，用户态从这里开始陷入。函数指针在从内核态返回到用户态之前被写到 stvec 寄存器里面去，但是目前页表仍然还是用户态的页表。先对用户态的上下文进行保存，然后在进行页表的切换。

pub unsafe extern "C" fn user\_to\_supervisor() -> ! {  
 core::arch::asm!(  
 // 交换 a0 和 sscratch（原先保存着交换栈的栈顶指针）  
 "csrrw a0, sscratch, a0",  
 "  
 // ... 保存 SwapContext ..  
 ",  
 // 保存用户的 a0 寄存器  
 "csrr t0, sscratch  
 sd t0, 9\*8(a0)",  
 // 写 sepc 寄存器到 SwapContext 中相应位置  
 "csrr t0, sepc  
 sd t0, 34\*8(a0)",  
 // 恢复内核栈指针  
 "ld sp, 32\*8(a0)",  
 // todo: 如何处理 tp 寄存器  
 "ld tp, 35\*8(a0)",  
 // 将用户中断处理函数指针放到 t0 寄存器  
 "ld t0, 33\*8(a0)",  
 // // 将用户的 satp 寄存器放到 t2 寄存器里面去  
 // "csrr t2, satp",  
 // 恢复内核页表  
 "ld t1, 31\*8(a0)  
 csrw satp, t1",  
 "sfence.vma",  
 // 跳转到中断处理函数  
 "jr t0",  
 options(noreturn)  
 );  
}

## 异步I/O

### 异步事件的底层依赖: Event

需要设计一个事件监听的工具，使得可以将同步的数据结构转化为异步的数据结果，为异步提供基础的依赖。

异步事件的内部同步的数据结构为:

struct Inner {  
 notified: AtomicUsize,  
 list: Mutex<List>,  
 cache: UnsafeCell<Entry>,  
}

Inner中的notified用于通知已被通知的Entry的数目，如果所有条目都被通知了，或者没有条目被通知，该值都会被设置为usize::MAX。而list则是用于指向已被注册的监听的链表， 而Mutex提供在多线程中list资源的锁机制，以解决资源竞争带来的数据冲突。

#### 异步事件的一些原语

pub fn listen(&self) -> EventListener {  
 let inner = self.inner();  
 let listener = EventListener {  
 inner: unsafe { Arc::clone(&ManuallyDrop::new(Arc::from\_raw(inner))) },  
 entry: Some(inner.lock().insert(inner.cache\_ptr())),  
 };  
  
 // Make sure the listener is registered before whatever happens next.  
 full\_fence();  
 listener  
}

pub fn notify(&self, n: usize) {  
 full\_fence();  
 if let Some(inner) = self.try\_inner() {  
 if inner.notified.load(Ordering::Acquire) < n {  
 inner.lock().notify(n);  
 }  
 }  
}

pub fn notify\_additional(&self, n: usize) {  
 full\_fence();  
  
 if let Some(inner) = self.try\_inner() {  
 if inner.notified.load(Ordering::Acquire) < usize::MAX {  
 inner.lock().notify\_additional(n);  
 }  
 }  
}

fn full\_fence() {  
 core::sync::atomic::fence(Ordering::SeqCst);  
}

full\_fence 该函数会阻止编译器和CPU围绕对某些类型的内春操作重新排序。使其可以在一些原子操作中创建同步关系。

#### Event异步事件的状态

enum State {  
 /// 刚刚被创建  
 Created,  
 /// 已经接收到一个通知  
 ///  
 /// 如果这是个 `additional` 通知，`bool` 将为 `true`  
 Notified(bool),  
 /// 正在被一个异步任务 `poll`，保存了任务的 `waker`  
 Polling(Waker),  
}

只用当状态为‘State::Notified(bool)‘的时候，事件才是被通知， 因此有如下的事件监听机制：

impl Future for EventListener {  
 type Output = ();  
  
 fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'\_>) -> Poll<Self::Output> {  
 let mut list = self.inner.lock();  
  
 let entry = match self.entry {  
 None => unreachable!("cannot poll a completed `EventListener` future"),  
 Some(entry) => entry,  
 };  
 let state = unsafe { &entry.as\_ref().state };  
  
 match state.replace(State::Notified(false)) {  
 State::Notified(\_) => {  
 list.remove(entry, self.inner.cache\_ptr());  
 drop(list);  
 self.entry = None;  
 return Poll::Ready(());  
 }  
 State::Created => {  
 state.set(State::Polling(cx.waker().clone()));  
 }  
 State::Polling(w) => {  
 if w.will\_wake(cx.waker()) {  
 state.set(State::Polling(w));  
 } else {  
 state.set(State::Polling(cx.waker().clone()));  
 }  
 }  
 }  
  
 Poll::Pending  
 }  
}

EventListener中的poll函数,

## 模块化

# 总结

## 异步锁的思路

pub struct AsyncMutex<T: ?Sized> {  
 state: AtomicUsize,  
 lock\_ops: Event,  
 data: UnsafeCell<T>,  
}

其中 AsyncMutex 的底层依赖是 Event 。

async fn acquire\_slow(&self){  
 loop {  
 let listener = self.lock\_ops.listen();  
  
 match self  
 .state  
 .compare\_exchange(0, 1, Ordering::Acquire, Ordering::Acquire)  
 .unwrap\_or\_else(|x| x)  
 {  
 0 => return,  
  
 1 => {}  
  
 \_ => break,  
 }  
  
 listener.await;  
 }  
 // ...  
}

通过Event::listen()创建事件监听，从代码的第2行到第15行，如果异步锁没有被任何任务持有，则第一个尝试获取锁，当compare\_exchange的返回值为0时，则表示事件成功获取锁，代码第16行，listener.await则是等待锁的释放。

于是

pub async fn lock(&self) -> AsyncMutexGuard<'\_, T> {  
 if let Some(guard) = self.try\_lock() {  
 return guard;  
 }  
 self.acquire\_slow().await;  
 AsyncMutexGuard(self)  
 }

异步的lock函数返回一个异步的锁结构，依赖于Rust的生命周期，其会在生命周期结束的时候被释放。

## 文件系统的异步改造

### 如何泛化异步接口

Rust 的编译器默认并不支持 async trait function。 编译器会推荐使用 async-trait，但是经过其改装的 async trait并不是零开销的。其会将

|  |  |
| --- | --- |
| #[async\_trait] pub trait AsyncBlockDevice {  async fn read(&self, block\_id: usize, buf: &mut [u8]);  async fn write(&self, block\_id: usize, buf: &[u8]); } | pub trait AsyncBlockDevice {  fn read(&self, block\_id: usize, buf: &mut [u8]) -> Box<dyn Future<Output = Option<&[u8], &[u8])>>>;  fn write(&self, block\_id: usize, buf: &[u8]) -> Box<dyn Future<Output = Option<&[u8], &[u8])>>>; } |

async trait 对异步接口的处理

这里就有两层开销了：

* 动态调度的开销 dyn Future。这意味着所有异步驱动设备的 read和write 函数都比较难做一些编译器的优化。
* 内存分配的开销 Box。在键值存储里，read和write应该是一个会被经常调用的函数。trait 被 async-trait 改写成新的形式之后，每次调用read和write都需要在堆上新建一个对象。这会对程序的性能造成比较大的影响。

其实，Rust本身提供了泛型编程的思路，直接拿出一个可供参考的解答，对应上部分可以写出如下代码：

pub trait AsyncBlockDevice {  
 type NextFuture<'a> = impl Future<Output = Option<(&'a [u8], &'a [u8])>>;  
  
 fn read(&self, block\_id: usize, buf: &mut [u8]) -> Self::NextFuture<'\_>;  
 fn write(&self, block\_id: usize, buf: &[u8]) -> Self::NextFuture<'\_>;  
}

解释：

* NextFuture 是由 read（或者 write）函数返回的，而正常实现的read（或者 write）函数，只能返回和&self生命周期相同的引用。在NextFuture上标注生命周期，是为了保障Self 的生命周期至少和 NextFuture 的一样长。
* read 和 write 的返回值，从代码中可以知道，两个函数的返回值是一个 Future，并不是一个常见的异步函数的返回值，因此需要使用async move返回一个闭包以满足异步函数的返回约束。
* 既然 read 和 write 返回的是一个Future，而Rust函数是无法在其编译器中被识别为一个具体的类型的，因此，此处的NextFuture需要让编译器自动推导Feature的具体类型，通过 impl 告知编译器NextFuture相关函数的返回需要产生一个 Future。

## 高阶线程通信

### Mach IPC

在Mach的微内核设计上， IPC是核心，而且是内核中最重要的组件。不同于支持IPC机制的操作系统，Mach提供了一个支持大多数操作系统的IPC机制。

Mach IPC机制，将进程之间的通行做了更一层的抽象，将通行机制的基础定为消息传递。而消息的数据量可大可小，小到几个字节，大到整个地址空间。内核需要确保进行数据传输时无需进行不必要的数据复制，同时其应当提供安全可靠的的通信，当且仅当线程获取授权时，才能发送和接收消息。

通信和内存管理时紧密相连的。IPC依赖于内存的写时复制进行大量数据的有效传递。

IPC支持用户任务之间，以及用户和内核之间的通信，且适用于那些基于客户端-服务器模型的应用程序。

可以将Mach IPC的设计思路平移到starfish中，为其可能拥有的网络服务提供更加健壮的底层通行依赖。