基于协程的任务调度器设计方案 v4

时间：2023年12月13

目标：构建低时延服务

模块：

- [x] 任务调度
- [x] 任务唤醒
- [x] 系统调用转发

任务调度

任务控制块

进程、线程、协程均实现 `Future trait` （以下简称为 `fut`），从而可以由任务控制块（`Task`）数据结构描述。`Task` 及其内部的 `fut` 均保存在堆中，由 Rust 所有权机制进行维护，保证内存安全。

```
#[repr(C)]
pub struct Task {
    pub(crate) executor: &'static Executor,
    /// The task state field
    pub state: AtomicU32,
    /// The priority of task
    pub priority: AtomicU32,
    /// The task type field:
    ///     1. Normal
    ///     2. AsyncSyscall
    ///     3. ......
    pub task_type: TaskType,
    /// The actual content of a task.
    /// It may be a `process`, `thread` or `coroutine`.
    pub fut: AtomicCell<Pin<Box<dyn Future<Output=isize> + 'static + Send + Sync>>>,
}
```

`Task` 由 `Executor` 维护，其内部的就绪队列中维护 `TaskRef` （`Task` 指针）。

```rust
#[repr(transparent)]
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct TaskRef {
    ptr: NonNull<Task>,
}
```

任务切换

调度器的调度对象为 `Task` 。由于 Rust 将 `fut` 实现为有限状态机，将其运行所需要的上下文（包括手动保存的和编译器帮助保存的部分）保存在一块不能移动的内存中，在运行过程中恢复上下文，而当前的运行栈不保存所需要的上下文。因此，让一个 `Task` （ `fut` ） 恢复执行需要完成以下工作：

1. 调度一个处于就绪状态的 `Task` ；
2. 准备好运行栈；

上述两个工作可以 `由硬件实现的任务调度器` （以下简称 `任务调度器` ）帮助完成。完成上述两个工作后，运行函数 `execute` 将会让 `Task` 从暂停点恢复执行。

```rust
pub fn execute(task_ref: TaskRef) -> Option<TaskRef> {
    unsafe {
        let waker = waker::from_task(task_ref);
        let mut cx = Context::from_waker(&waker);
        let task = Task::from_ref(task_ref);
        task.state.store(TaskState::Running as _, Ordering::Relaxed);
        let fut = &mut *task.fut.as_ptr();
        let mut future = Pin::new_unchecked(fut.as_mut());
        match future.as_mut().poll(&mut cx) {
            Poll::Ready(_) => { ..... },
            Poll::Pending => { ..... },
        }
    }
}
```

进程、线程以及协程之间的切换由前后两个 `Task` 之间的关系界定。

任务切换	特征	上下文恢复
协程切换	同地址空间，同栈	由编译器帮助完成，在进入 execute 函数后，恢复上下文

线程切换	同地址空间，不同栈	在进入 execute 函数时，运行在当前的栈上，execute 函数在调用 poll 时，在 poll 函数内部手动实现上下文切换（被打断的上下文由 `任务调度器` 帮助保存）
进程切换	不同地址空间	在前一个 Task 执行完之后，进入调度器，调度器帮助完成进程切换（地址空间切换），切换过后，调度器取出就绪 Task，并从栈池中取出空闲栈供 execute 函数执行，后续的上下文恢复同上述两种方案

任务调度

使用协程作为任务单元，则只能使用协作式调度，但这不利于构建低时延服务。假设处理器正在运行某项耗时较长的协程，此时产生了一个需要快速响应的请求，但由于协作式调度的原因，不能及时的响应，这导致响应时延边长。在这种情况下，需要抢占才能够保证响应时延。最直接的方式是通过中断实现抢占，但这将会导致过多的上下文切换开销。还有一些其他的方式实现抢占，例如编译器插桩，这可以实现抢占式调度相接近的效果。在这里，我们可以参考这种方式，用协作式调度来近似抢占式调度。

我们在 `Task` 结构中维护优先级，调度器将根据优先级进行调度，每当产生一个新的请求时，创建或唤醒具有最高优先级的 `Task` ，保证下一次能够调度到这个 `Task` 。并且我们通过限制每个 `Task` 不会占用过长时间（过长的计算任务进行拆分，耗时的 IO 任务转化成异步的形式）。通过上述的方式实现近似的抢占式调度。

任务唤醒

`Task` 以及 `fut` 占用的空间都保存在堆中，且在进入 `execute` 函数之前不需要恢复 `fut` 的上下文。因此，调度器里需要维护的对象是 `TaskRef` （8 字节），将目标 `Task` 对应的 `TaskRef` 添加到调度器的就绪队列中这个过程即被视为任务唤醒。

如何确定任务唤醒的时机，目前主要存在以下几种方式：

1. 中断
2. 轮询
3. 混合中断轮询模式

利用中断来进行唤醒，但由于中断时机不确定，可能在 `fut` 执行过程中产生中断，这时必须保存通用寄存器等上下文，这将会产生大量的开销；若利用单独的 `fut` 进行轮询，那么，这个 `fut` 在负载高的情况下，必须单独占用一个处理器核，它将退化为线程，产生资源浪费，且上述的使用协作式调度拟合抢占式调度也不能达到应有的效果，必须将这个 `Task` 进行单独处理。

由于上述任务唤醒的过程被简化了，该过程可以由硬件电路来帮助完成。因此，我们将中断处理唤醒任务的过程从 `CPU` 卸载到 `任务调度器` 上。以网卡设备为例，当它接收到数据包后，它产生的中断信号

通过电路传递给 任务调度器 ， 任务调度器 从中断向量表中读取到对应的 TaskRef ，将 TaskRef 添加到 Executor 内的就绪队列。

系统调用转发

在 Task 执行期间，大多数上下文切换是由于系统调用造成的。此外，如果 Task 阻塞在同步的系统调用上时，那么这个 Task 也将退化成线程，其他的就绪 Task 无法在当前的栈上运行。因此需要通过系统调用转化将同步系统调用改造成异步系统调用，同时减少由系统调用导致的上下文切换开销。

这里不涉及由中断、异常导致的上下文切换优化，一方面，因为中断已经被 任务调度器 代理；另一方面，在现代的设备上，可用的物理内存不再受限，产生缺页异常的概率较小，而其他的异常处理可以直接将进程杀死。

同上任务唤醒中的描述相似，若通过共享内存进行系统调用转发，同样需要单独的 fut 轮询共享内存。为此，我们将通过 任务调度器 转发系统调用。通过将系统调用的参数写入 任务调度器 的寄存器中，向处在另一个处理器核心上的内核转发系统调用，唤醒对应的系统调用处理 Task 。

系统调用消息中除了本身的参数外，还需要 address space token 和 related user task_ref ，因为内核在处理系统调用时，涉及到读写进程的地址空间，必要时，还需要唤醒对应的用户态 Task 。具体格式如下（64 字节， arguments 占 48 字节， address space token 和 related user task_ref 各占 8 字节）：

```
high                                                    low
+---------------------+-----------------------+-----------+
| address space token | related user task_ref | arguments |
+---------------------+-----------------------+-----------+
```

难点

进程切换

1. 任务调度器 需要写页表寄存器；
2. 任务调度器 需要切换 Executor；

上述两个难点归根结底是因为进程切换的需求导致。若不需要进程切换，则问题不复存在。

针对难点 2：主要的矛盾在于 Executor 在何处维护，若在内存中维护，那么只需要用寄存器记录 Executor 在内存中的位置， 任务调度器 中需要实现的功能是读写 Executor 对应的内存，但这会导致任务切换产生多次读写内存；若直接在 任务调度器 中完全实现 Executor ，那么任务调度与切换将不需要读写内存，但缺点是进程切换时，需要将内部的数据刷新到内存中，从内存中读取目标进程 Executor 内容。除此之外，设置多个通道，限制能够并发的进程数量，不切换 Executor 。

线程切换

难点	解决方案
fut 被打断后，任务调度器 帮助保存上下文时，需要读通用寄存器；	任务调度器 直接通过电路与通用寄存器相连，或通过总线读取通用寄存器；
线程切换过程中，被打断的上下文保存在何处；	对于自定义 fut（User-implemented Future），我们在实现 Future trait 时可以保证上下文的存储位置；但对于使用 async 关键字创建的 fut，它们没有位置保存上下文。一种解决方案是在使用 async 关键字创建 fut 时，利用过程宏 #[fut] 把它转化为 User-implemented Future，见如下代码。
当 fut 被打断后，当前栈中存在函数调用链，这个栈不能用于运行其他的 Task，必须从栈池中取出一个空闲栈，当下一个取出的 Task 为线程时，则需要从当前的空闲栈切换到下一个 Task 的栈上，这些栈的状态该如何维护，实现在硬件上将会导致硬件的功能过于复杂。	

```
pub struct Thread {
    pub context: Option<Context>,
    pub inner_fut:
    ......
}

impl Future for Thread {
    fn poll(self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<Self::Output> {
        if self.context.is_some() {
            // 存在上下文，意味着它是一个线程，需要恢复上下文（手写汇编代码），
            // 这个协程执行过程中，栈将切换到被打断的状态，为线程切换
        } else {
            inner_fut.poll()
        }
    }
}

#[fut]
pub async fn() -> isize {
    0
}
```

> 假设不出现 `fut` 被打断的情况，那么问题都将不复存在。

系统调用转发

1. 填充系统调用参数时， `任务调度器` 需要读页表寄存器;

解决方案：单独设置寄存器，记录当前进程的 `address space token`