OS Lab 3 - Process

在本实验中，我们将在系统中引入进程的概念，并实现如下图所示的进程状态转化逻辑。

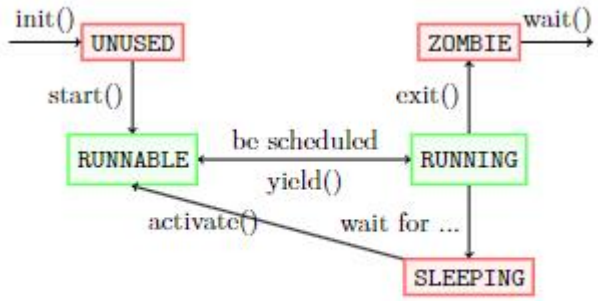

图 1: enum procstate 状态转化图

与经典的进程概念不同，我们将首先实现**内核进程**，或者说线程、协程都可以。

内核进程是一种运行在内核态、执行内核代码的进程。它可以像用户进程一样被调度，有着与用户进程相同的状态转化逻辑。

> 因为内核进程可以自由开关中断，既有抢占式的成分又有非抢占式的成分，此外内核进程与内核共用内存空间，因此叫线程或协程更贴切，叫进程是为了与后续内核进程进入用户态成为用户进程保持一致。

我们将先在内核进程上实现全套的进程状态转化关系，在下个lab中再引入用户进程。与用户进程相比，实现内核进程需要考虑的细节更少，有利于大家把重点集中于进程状态的转化流程。

Processes

在内核中，我们使用PCB（struct proc）来表示一个进程。struct proc中包含了该进程相关的所有控制信息。进程之间通过父子关系形成进程树，每个进程都有自己的子进程链表。树的根节点是一个特殊的进程，称为root_proc，将是除idle外运行的第一个进程，执行rest init后转入用户态执行/init。

进程退出时，将其子进程的父进程改为root_proc。确保除了root的进程都有父进程，也就是所有的进程能够构成一棵树。

进程树需要通过锁来同步。简单起见，我们建议直接使用全局的进程树锁。

每个进程都有一个pid。进程树中存在的所有进程的pid不能重复。

Scheduler

调度器维护CPU和进程的调度信息，在进程请求调度时决定下一个运行什么进程，并执行进程切换。进程切换需要更新相关的调度信息，并进行上下文切换。

一般情况下，调度器会为处于RUNNING和RUNNABLE状态的进程维护一个调度队列（也可能为每个CPU分别维护一个队列，我们统称调度队列）。当进程状态更改为SLEEPING或者ZOMBIE时，将被从调度队列中移除。当UNUSED或SLEEPING状态的进程被激活（activate_proc）时，将被加入调度队列。

每个CPU都有一个专属的idle进程。idle进程不进入调度队列，或进入调度队列但优先级永远最低。当没有其他进程可以调度时，调度器将选择idle进程，保证CPU上总有进程可以运行。

调度队列可能需要锁来同步。

设计调度算法时，请注意考虑负载均衡和公平调度问题。在lab3中，因为只有内核进程，甚至有点像协程，我们只要求调度算法不要过于离谱即可。但你可以提前思考一下，lab4中有了用户进程之后，如何设计你的调度算法?

The Life of A Process

本段将带大家过一遍进程的从创建到退出的整个流程。

涉及的具体理论知识请参考elearning上进程相关理论课内容，调用的实验框架函数请参考API Reference，可以在elearning上或点击链接查看。

当一段内核代码需要创建一个内核进程时，首先它应该处于early init和init阶段之后，因为init阶段才完成进程树和调度器的初始化。

要创建进程的内核代码调用`create_proc`，分配空间并初始化进程结构体。此时进程处于`UNUSED`状态。

在进程启动之前，还可以对进程结构体的一些内容进行修改，如修改其父进程（一般情况下，只能选择root_proc和当前进程为新进程的父进程，为什么？），修改其调度信息，修改初始寄存器值等。

随后调用`start_proc`启动进程。启动进程时，将为进程设置入口函数，并将其加入调度队列，状态更新为`RUNNABLE`。此时进程已经可以被调度。

进程被调度后，进入指定的进程入口函数，执行进程代码。（真正的入口函数是`proc_entry`，然后才进入指定的入口函数，为什么要这样设计？）

进程可以调用`wait`、`wait_sem`等函数，这些函数会在条件不满足时令进程陷入`SLEEPING`状态。他们都是通过配置好相关信息后调用`sched(SLEEPING)`实现的。（直接调用`sched(SLEEPING)`会怎么样？）

其他进程可以通过调用`activate_proc`唤醒处于`SLEEPING`状态的进程，这会将进程的状态更改为`RUNNABLE`并加入调度队列。当进程收到`signal`时，也会调用`activate_proc`。（如何分辨进程是因为`signal`唤醒还是正常唤醒？）

进程执行完毕后，应调用`exit`退出。（直接return会怎么样？）`exit`将释放一些资源，将子进程全部转移给`root_proc`，然后调用`sched(ZOMBIE)`。此时进程处于`ZOMBIE`状态，不再执行，只保留一些基础的数据等待父进程调用`wait`回收。

进程的父进程可以调用`wait`释放`ZOMBIE`状态子进程的剩余资源，并释放进程结构体。`wait`将向父进程反馈子进程的退出代码和PID。

Trap

异常与中断的概念我们不再赘述。通过配置相关寄存器，我们将所有trap的入口设定为trap_entry。trap_entry中需要保存trap的上下文，并调用trap_global_handler。

trap_global_handler会根据trap的类型进行相应的处理。本lab中，我们只启用了时钟中断，并且对于时钟中断不做处理直接返回。（即只要求大家正确书写基础代码，不实现具体功能）

提示

- 关于信号量Semaphore
  - 用于同步与信息传递，分P、V操作（对应wait、post），内部有一个值val。post使val++，wait使val--。
  - 通俗的理解post是生产、wait是消费，当val=0时，再进程进行消费将会进入sleeping，另外的进程进行了post到val>0时，sleeping的进程会被唤醒。
- swtch、KernelContext（进程上下文）请参考课件2.3.3，以及API reference。
  - KernelContext不包含caller_save register的原因是在C语言中调用switch时由caller保存了。
- trap_，UserContext（处理器上下文）请参考课件2.3.3，以及API reference
- proc.c部分是关于进程的生命周期的。
- sched.c部分是关于进程调度的，需要与schinfo配合，定义合适的数据结构。
- 这个lab需要进程树锁与调度锁，API reference对于哪些函数、数据需要注意并发问题进行了提示。

- 时钟中断相关的可以先忽略，应该不影响测试，下一个实验进行细节补充。
- 不仅是需要填TODO，也需要自己完成一些初始化操作。
- 虎鲸视频

任务与提交

完成下列内容

- `aarch64/swtch.S: swtch`
- `aarch64/trap.S: trap_entry trap_return`
- `kernel/proc.c: set_parent_to_this exit wait start_proc init_proc`
- `kernel/proc.h: UserContext KernelContext`
- `kernel/sched.c: thisproc init_schinfo _acquire_sched_lock _release_sched_lock activate_proc _sched(update_this_state pick_next update_this_proc)`
- `kernel/schinfo.h: sched schinfo`

我们已经在kernel_entry中编写了调用proc_test的代码。如果一切顺利，将输出proc_test PASS。

提交：将实验报告提交到 elearning 上，格式为`学号-lab3.pdf`。

从lab2开始，用于评分的代码以实验报告提交时为准。如果需要使用新的代码版本，请重新提交实验报告。

截止时间：2023年10月13日 11:59。逾期提交将扣除部分分数。

报告中可以包括下面内容

- 代码运行效果展示
- 实现思路和创新点
- 思考题
- 对后续实验的建议
- 其他任何你想写的内容

> ~~你甚至可以再放一只可爱猫猫~~

报告中不应有大段代码的复制。如有使用本地环境进行实验的同学，请联系助教提交代码（最好可以给个git仓库）。使用服务器进行实验的同学，助教会在服务器上检查，不需要另外提交代码。