实验四：调度器

实验目的

- 熟悉 ucore 的系统调度器框架，以及内置的 Round-Robin 调度算法。
- 基于调度器框架实现一个调度器算法(Stride Scheduling)

调度算法

此次实验，需要实现 Stride Scheduling 算法，后面的部分会给出该算法的大体描述。这里给出 Stride 调度器的一些相关的资料（目前网上中文的资料比较欠缺）。

- `http://wwwagss.informatik.uni-kl.de/Projekte/Squirrel/stride/node3.html`
- `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.138.3502&rank=1`
- Please GOOGLE “Stride Scheduling” ...

1 程序清单

此次实验中，主要有如下需要关注的文件：

```
lab4
|-- html
|   |-- lab4.html
|   `-- ...
|-- lab4.pdf
|-- lab4-handin
`-- proj13.1
    |-- ...
    |-- libs
    |   |-- ...
    |   |-- skew_heap.h
    |   `-- list.h
    |-- kern
    |   |-- ...
    |   `-- schedule
    |       |-- sched.h
    |       |-- sched.c
    |       |-- sched_RR.h
    |       |-- sched_RR.c
    |       |-- sched_stride.h
    |       `-- sched_stride.c
    `-- user
        |-- ...
        |-- threadfork.c
        |-- threadtest.c
        |-- matrix.c
        |-- primer.c
        |-- priority.c
        `-- ...
```

简单说明如下：

- libs/list.h: 提供了双向列表相关的定义和函数。
- libs/skew_heap.h: 提供了基本的优先队列数据结构，为本次实验提供了抽象数据结构方面的支持。
- kern/schedule/{sched.h,sched.c}: 定义了 ucore 的调度器框架，其中包括相关的数据结构（包括调度器的接口和运行队列的结构），和具体的运行时机制。
- kern/schedule/{sched_RR.h,sched_RR.c}: 具体的 round-robin 算法，在本次实验中你需要了解其实现。

- kern/schedule/{sched_stride.h,sched_stride.c}: Stride 调度器的基本框架，在此次实验中你需要填充其中的空白部分以实现一个完整的 Stride 调度器。
- user/{threadfork.c,threadtest.c,matrix.c,primer.c,priority.c,...}: 相关的一些测试用户程序，测试调度算法的正确性，user 目录下包含但不限于这些程序。在完成实验过程中，建议阅读这些测试程序，以了解这些程序的行为，便于进行调试。

与 lab3 相比，ucore 框架进行了适当的修改。前面的实验，创建了第一个用户进程，并让它正确运行。这中间也使用了一些调度策略。你可以阅读 lab3 下的 kern/schedule/sched.c 的 schedule 函数的实现，在此次实验中 sched.c 只是实现了一个调度器框架，而不再涉及具体的调度实现。

除此之外，实验中还涉及了 idle 进程的概念。

当 cpu 没有进程可以执行的时候，系统应该如何工作？在 scheduler 实现中，内核可以不断的遍历进程池，直到找到第一个 runnable 状态的 process，调用并执行它。也就是说，当系统没有进程可以执行的时候，它会把所有 cpu 时间用在搜索进程池，以实现 idle 的目的。但是这样的设计不被大多数操作系统所采用，原因在于它将进程调度和 idle 进程两种不同的概念混在了一起，而且，当调度器比较复杂时，schedule 函数本身也会比较复杂，这样的设计结构很不清晰而且难免会出现错误。所以在此次实验中，ucore 建立了一个单独的进程(kern/process/proc.c 中的 idleproc) 作为 cpu 空闲时的 idle 进程，这个程序是通常一个死循环。你需要了解这个程序的实现。

2 准备知识

2.1 数据结构

在理解框架之前，需要先了解一下调度器框架所需要的数据结构。

- 通常的操作系统中，进程池是很大的（虽然在 ucore 中，MAX_PROCESS 很小）。在 ucore 中，调度器引入 run-queue 的概念，通过链表结构管理进程。
- 由于目前 ucore 设计运行在单 CPU 上，其内部只有一个全局的 run-queue，用来管理系统内全部的进程。
- run-queue 通过链表的形式进行组织。链表的每一个节点是一个 list_entry_t, 每个 list_entry_t 又对应到了 struct proc_struct *, 这其间的转换是通过宏 le2proc 来完成的。具体来说，我们知道在 struct proc_struct 中有一个叫 run_link 的 list_entry_t，因此可以通过偏移量逆向找到对因某个 run_list 的 struct proc_struct。即

  进程结构指针 = le2proc(链表节点指针, run_link);

- 为了保证调度器接口的通用性，ucore 调度框架定义了如下接口：

```
1 struct sched_class {
2   // 调度器的名字
3   const char *name;
4   // 初始化运行队列
```

```
5   void (*init) (struct run_queue *rq);
6   // 将进程 p 插入队列 rq
7   void (*enqueue) (struct run_queue *rq, struct proc_struct *p);
8   // 将进程 p 从队列 rq 中删除
9   void (*dequeue) (struct run_queue *rq, struct proc_struct *p);
10  // 返回 run-queue 中下一个可执行的进程
11  struct proc_struct* (*pick_next) (struct run_queue *rq);
12  // timetick 处理函数
13  void (*proc_tick)(struct run_queue* rq, struct proc_struct* p);
14 };
```

该类中，几乎全部成员变量均为函数指针。具体的功能会在后面的框架说明中介绍。

- 此外，proc.h 中的 struct proc_struct 中也记录了一些调度相关的信息：

```
1  struct proc_struct {
2    // ...
3    // 该进程是否需要调度，只对当前进程有效
4    volatile bool need_resched;
5    // 该进程的调度链表结构，该结构内部的连接组成了 run-queue 列表
6    list_entry_t run_link;
7    // 该进程剩余的时间片，只对当前进程有效
8    int time_slice;
9    // round-robin 调度器并不会用到以下成员
10   // 该进程在优先队列中的节点，仅在 LAB4 使用
11   skew_heap_entry_t lab4_run_pool;
12   // 该进程的调度优先级，仅在 LAB4 使用
13   uint32_t lab4_priority;
14   // 该进程的调度步进值，仅在 LAB4 使用
15   uint32_t lab4_stride;
16 };
```

在此次实验中，你需要了解 sched_RR.c 中的函数。在该文件中，你可以看到 ucore 已经为 RR 调度算法创建好了一个名为 RR_sched_class 的调度策略类。

- 通过数据结构 struct run_queue 来描述完整的 run-queue。它的主要结构如下：

```
1  struct run_queue {
2    //其运行队列的哨兵结构，可以看作是队列头和尾
3    list_entry_t run_list;
4    //优先队列形式的进程容器，只在 LAB4 中使用
5    skew_heap_entry_t *lab4_run_pool;
6    //表示其内部的进程总数
7    unsigned int proc_num;
8    //每个进程一轮占用的最多时间片
9    int max_time_slice;
10 };
```

在 ucore 框架中，run-queue 存储的是当前可以调度的程序，所以，只有状态为 runnable 的进程才能够进入 run-queue。当前正在运行的进程并不会在 run-queue 中，这一点需要注意。

2.2 计时器的原理和实现

在传统的操作系统中，计时器是其中一个基础而重要的功能.它提供了基于时间事件的调度机制。在 ucore 中，timer 中断(irq0) 给操作系统提供了有一定间隔的时间事件，操作系统将其作为基本的调度和计时单位（我们记两次时间中断之间的时间间隔为一个时间片，timer splice）。

基于此时间单位，操作系统得以向上提供基于时间点的事件，并实现基于时间长度的等待和唤醒机制。在每个时钟中断发生时，操作系统产生对应的时间事件。应用程序或者操作系统的其他组件可以以此来构建更复杂和高级的调度。

sched.h, sched.c 定义了有关 timer 的各种相关接口来使用 timer 服务，其中主要包括:

- typedef struct {...} timer_t: 定义了 timer_t 的基本结构，其可以用 sched.h 中的 timer_init 函数对其进行初始化。
- void timer_init(timer_t *timer, struct proc_struct *proc, int expires): 对某计时器进行初始化，让它在 expires 时间片之后唤醒 proc 进程
- void add_timer(timer_t *timer): 向系统添加某个初始化过的 timer_t，该计时器在指定时间后被激活，并将对应的进程唤醒至 runnable（如果当前进程处在等待状态）
- void del_timer(timer_t *time): 向系统删除（或者说取消）某一个计时器。该计时器在取消后不会被系统激活并唤醒进程。
- void run_timer_list(void): 更新当前系统时间点，遍历当前所有处在系统管理内的计时器，找出所有应该激活的计数器，并激活它们。该过程在且只在每次计时器中断时被调用。在 ucore 中，其还会调用调度器事件处理程序。

一个 timer_t 在系统中的存活周期可以被描述如下：

1. timer_t 在某个位置被创建和初始化，并通过 add_timer 加入系统管理列表中
2. 系统时间被不断累加，直到 run_timer_list 发现该 timer_t 到期。
3. run_timer_list 更改对应的进程状态，并从系统管理列表中移除该 timer_t。

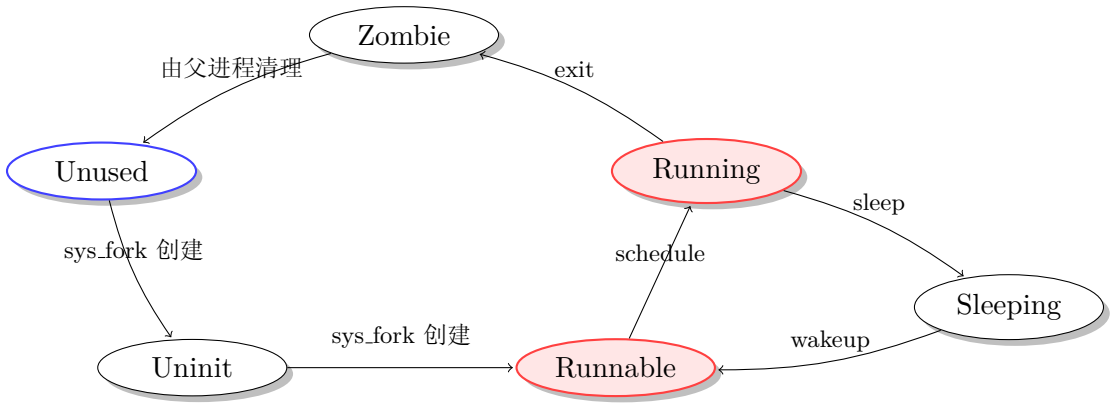
尽管本次实验并不需要填充计时器相关的代码，但是作为系统重要的组件（同时计时器也是调度器的一个部分），你应该了解其相关机制和在 ucore 中的实现方法。接下来的实验描述将会在一定程度上忽略计时器对调度带来的影响，即不考虑基于固定时间点的调度。

2.3 进程状态设计

在此次实验中，进程的状态之间的转换需要有一个更为清晰的表述，参见下图：

如图所示，在 ucore 中，runnable 的进程会被放在 run-queue 中。值得注意的是，在具体实现中，ucore 的 struct proc_struct 中, running 和 runnable 共享同一个状态(state)值(PROC_RUNNABLE)。不同的是 running 的进程不会放在 run-queue 中。

进程的正常生命周期如下：



- 进程首先在 cpu 初始化或者 sys_fork 的时候被创建，当为该进程分配了一个进程描述符之后，该进程进入 uninit 态(在 proc.c 中 alloc_proc)。
- 当进程完全完成初始化之后，该进程转为 runnable 态。
- 当到达调度点时，由调度器 sched_class 根据 rq 的内容来判断一个进程是否应该被运行，即转换成 running。
- running 态进程被阻塞主动调用 sleep 变成 sleeping 态。
- sleep 态被 wakeup 变成 runnable。
- running 态进程主动 exit 变成 zombie 态，然后由其父进程完成对其资源的释放, 成为 unused。

所有从 running 态变成其他状态的进程都要出 run-queue，反之，被放入某个 run-queue 中。

2.4 进程调度设计

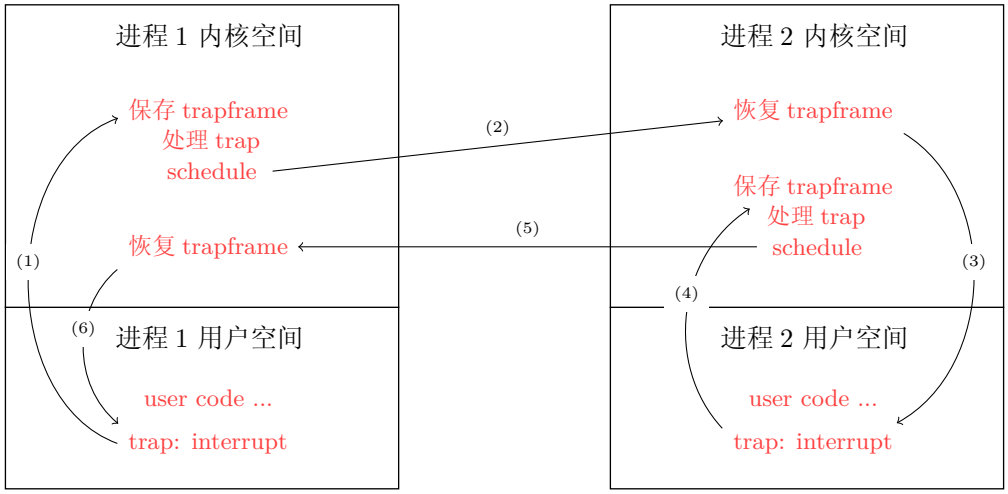
通过对上一个实验，以及 scheduler，你应该已经对进程调度和上下文切换有了初步的认识。在本次实验中，结合调度器的设计，你应该对 ucore 中，进程切换以及堆栈的维护和使用，有更加深刻的认识。

下图为进程切换流程图：

解释如下：

首先在执行进程 1 的用户代码时，出现了一个 trap (例如是一个 Time interrupt)，这个时候就会从进程 1 的用户态切换到内核态(过程(1))，并且保存好进程 1 的 trapframe；当内核态处理中断时发现需要进行进程切换时，ucore 要通过 schedule 先切换到进程 2 的内核态(过程(2))，继续进程 2 上一次在内核态的操作，并最终将执行权转交给进程 2 的用户空间((3))。

当进程 2 由于某种原因发生中断之后(过程(4))，并发现需要切换到进程 1。再次切换到进程 1 时(过程(5))，会执行进程 1 上一次在内核调用 schedule (具体还要跟踪到 switch_to 函数) 函数后的下一行代码，这行代码当然还是在进程 1 的上一次中断处理流



程中。最后当进程 1 的中断处理完毕的时候，执行权又会反交给进程 1 的用户代码(过程(6))。

这就是只有两个进程的情况下，进程切换间的大体流程。

几点需要强调的是：

a) 需要弄清楚并小心的是，进程切换以后，程序是从哪里开始执行的？虽然还是同一个 cpu，但是此时使用的资源已经完全不同了。

b) 内核在第一个程序运行的时候，需要进行哪些操作？有了前一个 lab 的经验，可以确定，内核运行第一个程序的过程，实际上是从启动时的内核状态切换到该程序的内核状态的过程，而用户程序的起始状态的入口，应该是 forkret 等。请参考代码，了解启动第一个进程（idle）的过程（kern/init/init.c:kern_init 中）。

2.5 框架说明

2.5.1 *调度点*

- 参加下图，了解实验中都有哪些调度点，以及调度器框架函数的使用：

  各个调度点说明如下：

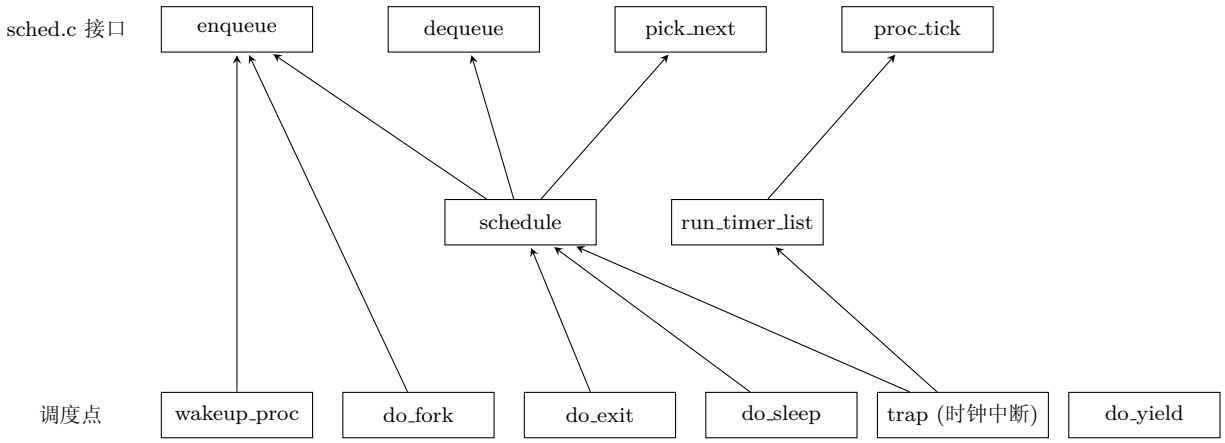
  wakeup_proc: 只需要将需要唤醒的进程设置成 runnable，调用 enqueue，将其加入队列。这个函数不会引起进程调度。

  do_fork: 只需要将新增的进程设置成 runnable，调用 enqueue_proc，将进程加入队列。这个函数不会引起进程调度。

  do_exit: 将当前进程设置成 zombie，并且引起调度。

  do_sleep: 将当前进程设置成 sleeping，并且引起调度。

  trap: 调用 run_timer_list 进行计时器的事件处理，并结合 proc_tick 判断判断当前进程是否需要调度(proc_struct 结构中的 need_resched 标记是否为 1)，如果需要则调用 schedule。



do_yield: do_yield 并不直接调用 schedule，而是将当前进程的 need_resched 标记置为 1，由此在下一个时钟中断时强制引发进程调度。（当然，在这里直接执行调度也是可以的）

2.5.2 *框架以及 RR 调度器设计*

调度器整体框架如下：

PART A:

```
int kern_main(void)
    void proc_init(void)
    void init_main(void)
        switch_to(c->context, idle->context)
```

系统启动调度过程

a) 系统启动之后，会通过 proc_init 来对进程池（包括 idleproc 和系统初始进程）以及 cpu 的 run-queue 和采用的调度策略(struct sched_class *sched_class, *sub_sched_class) 进行初始化。

b) 在进程出初始化完成之后，kern_main 函数自动执行成 idleproc 的内容，并让调度器开始工作。

c) 系统初始进程会继续初始化其他系统进程（比如 kswap 进程）和用户进程，并开始执行用户进程。

PART B:

```
void trap(struct trapframe *tf)
    void process_timer_event(void)
        void sched_class_proc_tick(struct run_queue *rq, struct proc_struct
            *p)
    void schedule()
        void sched_class_enqueue(struct run_queue *rq, struct proc_struct *p
            )
        void sched_class_pick_next(struct run_queue *rq)
```

```
        void sched_class_dequeue(struct run_queue *rq, struct proc_struct *p
            )
        switch_to(prev->contex, next->contex)
trapret
```

a) trap 函数实际上是由 trapasm.S 调用的，并且如果 trap 直接返回，会执行到__trapret 的位置，也就是继续执行刚刚发生中断的程序。

b) 在 trap 中，会对时钟中断进行处理。

c) 时钟中断会尝试处理系统里的时钟事件，关于这部分在实验中暂不涉及，但我们知道由于时钟事件的存在，可能会有进程由于被唤醒而加入到当前的 run-queue 中来。

d) 时间事件的处理程序也会使用调度器的 sched_class_proc_tick 进行当前进程的时钟 tick 处理。

e) 在之后，调度器会检查是否需要在当前 tick 进行调度操作，如果不需要则直接返回__trapret。

f) schedule 会来判断当前运行的 process 是否需要重新回到 run-queue 中去。

g) schedule 会通过接口函数 pick_next 来获得下一个可以执行的 process，并将它从 run-queue 中取出。通过 proc_run（会调用 switch_to）来进行切换和运行它。如果找不到下一个可调度的进程，schedule 则会选择 idleproc 作为下一个运行的进程。

此外类似的调度点还有许多，包括 do_fork、do_exit、do_sleep 等等。上述过程不做过多的说明，你可以通过阅读代码来理解。

PART C:

round-robin 调度器是所有调度器里面相对基础和简单的调度器之一。其调度思想是让所有 runnable 的进程轮流使用 CPU 时间。调度器维护当前 runnable 进程的有序队列。当当前进程的时间片用完之后，调度器将当前进程放置到调度队列的尾部，再从其头部取出进程进行调度。具体对应到调度器的接口上，我们有：

- init: 初始化当前的 run-queue（初始化为一个空的有序队列）。
- enqueue: 将 proc 插入队列尾部。
- dequeue: 从队列中删除相应的元素（注意该元素不一定处在队列头部，而可以在队列的任何位置）。
- pick_next: 返回 run-queue 队列的头部元素。
- proc_tick: 检测当前进程是否已用完分配的时间片。如果时间片用完，应该正确设置进程结构的相关标记来引起进程切换。一个 process 最多可以连续运行 rq.max_time_slice 个时间片。

proj13.1/kern/schedule/sched_RR.c 实现了一个完整的 round-robin 调度器，你可以参考代码理解其实现。

2.6 用户测试程序

user 目录下定义了若干用户程序。提供的实验框架内，OS 会先创建 idle 进程，然后再加载指定的用户代码来创建用户进程。可以在跟目录下指定在本次实验中 OS 需要加载的用户程序。比如说，假设需要 OS 加载和执行用户程序 matrix，则在项目跟目录下使用

```
make build-matrix
```

则可以生成一个对应的操作系统镜像，该镜像会加载 matrix.c 作为初始用户程序。接下来就和之前的 LAB 一样，比如

```
make qemu
```

则可运行这个 OS 镜像。

2.7 Stride Scheduling

考察 round-robin 调度器，在假设所有进程都充分使用了其拥有的 CPU 时间资源的情况下，所有进程得到的 CPU 时间应该是相等的。但是有时候我们希望调度器能够更智能地为每个进程分配合理的 CPU 资源。假设我们为不同的进程分配不同的优先级，则我们有可能希望每个进程得到的时间资源与他们的优先级成正比关系。基于这种想法，各种调度算法被提出。其中 Stride 调度是其中一个较为典型和简单的算法。除了简单易于实现以外，它还有如下的特点：

- 可控性：如我们之前所希望的，可以证明 Stride Scheduling 对进程的调度次数正比于其优先级。
- 确定性：在不考虑计时器事件的情况下，整个调度机制都是可预知和重现的。

该算法的基本思想可以考虑如下：

1. 为每个 runnable 的进程设置一个当前状态 stride，表示该进程当前的调度权。另外定义其对应的 pass 值，表示对应进程在调度后，stride 需要进行的累加值。
2. 每次需要调度时，从当前 runnable 进程中选择 stride 最小的进程调度。
3. 对于获得调度的进程 P，将对应的 stride 加上其对应的步长 pass（只与进程的优先权有关系）。
4. 在一段固定的时间之后，回到 2. 重新调度当前 stride 最小的进程。

可以证明，如果令

$$\text{P.pass} = \frac{\text{BigStride}}{\text{P.priority}}$$

，其中 P.priority 表示进程的优先权（大于 1），而 BigStride 表示一个预先定义的大常数，则该调度方案为每个进程分配的时间将与其优先级成正比。证明过程我们在这里略去，有兴趣的同学可以在网上查找相关资料。

将该调度器应用到 ucore 的调度器框架中来，则需要将调度器接口实现如下：

- init:
  - 初始化调度器类的信息（如果有的话）。
  - 初始化当前的 run-queue 初始化为一个空的容器结构。（比如和 round-robin 一样，为一个有序列表）
- enqueue
  - 初始化刚进入 runnable 队列的进程 proc 的 stride 属性。
  - 将 proc 插入放入 run-queue 中去（注意：这里并不要求放置在队列头部）。
- dequeue
  - 从 run-queue 中删除相应的元素。
- pick_next
  - 扫描整个 run-queue，返回其中 stride 值最小的对应进程。
  - 更新对应进程的 stride 值，即`pass = BIG_STRIDE / P->priority; P->stride += pass`。
- proc_tick:
  - 检测当前进程是否已用完分配的时间片。如果时间片用完，应该正确设置进程结构的相关标记来引起进程切换。
  - 一个 process 最多可以连续运行 rq.max_time_slice 个时间片。

在具体实现时，有一个需要注意的地方：stride 属性的溢出问题，在之前的实现里面我们并没有考虑 stride 的数值范围，而这个值在理论上是不断增加的，在 stride 溢出以后，基于 stride 的比较可能会出现错误。

比如假设当前存在两个进程 A 和 B，stride 属性采用 16 位无符号整数进行存储。当前队列中元素如下（假设当前运行的进程已经被重新放置进 run-queue 中）：

A.stride (实际值)	A.stride (理论值)	A.pass ($=\frac{\text{BigStride}}{\text{A.priority}}$)
65534	65534	100
B.stride (实际值)	B.stride (理论值)	B.pass ($=\frac{\text{BigStride}}{\text{B.priority}}$)
65535	65535	50

此时应该选择 A 作为调度的进程，而在一轮调度后，队列将如下：

A.stride (实际值)	A.stride (理论值)	A.pass ($=\frac{\text{BigStride}}{\text{A.priority}}$)
98	65634	100
B.stride (实际值)	B.stride (理论值)	B.pass ($=\frac{\text{BigStride}}{\text{B.priority}}$)
65535	65535	50

可以看到，由于溢出，进程间 stride 的理论比较和实际比较结果出现了偏差。

我们首先在理论上分析这个问题：令PASS_MAX 为当前所有进程里最大的步进值。则我们可以证明如下结论：对每次 Stride 调度器的调度步骤中，有其最大的步进值STRIDE_MAX 和最小的步进值STRIDE_MIN 之差：

```
STRIDE_MAX – STRIDE_MIN <= PASS_MAX
```

提问 1：如何证明该结论？

有了该结论，在加上之前对优先级有Priority > 1 限制，我们有

```
STRIDE_MAX – STRIDE_MIN <= BIG_STRIDE
```

于是我们只要将 BigStride 取在某个范围之内，即可保证对于任意两个 Stride 之差都会在机器整数表示的范围之内。而我们可以通过其与 0 的比较结构，来得到两个 Stride 的大小关系。在上例中，虽然在直接的数值表示上 98 < 65535，但是 98 - 65535 的结果用带符号的 16 位整数表示的结果为 99, 与理论值之差相等。所以在这个意义下 98 > 65535。基于这种特殊考虑的比较方法，即便 Stride 有可能溢出，我们仍能够得到理论上的当前最小 Stride，并做出正确的调度决定。

提问 2：在 ucore 中，目前 Stride 是采用无符号的 32 位整数表示。则 BigStride 应该取多少，才能保证比较的正确性？

2.8 使用优先队列实现 Stride Scheduling

在刚刚的实现描述中，对于每一次 pick_next 函数，我们都需要完整地扫描来获得当前最小的 stride 及其进程。这在进程非常多的时候是非常耗时和低效的，有兴趣的同学可以在实现了基于列表扫描的 Stride 调度器之后比较一下 prime 程序在 Round-Robin 及 Stride 调度器下各自的运行时间。考虑到其调度选择于优先队列的抽象逻辑一致，我们考虑使用优化的优先队列数据结构实现该调度。

优先队列是这样一种数据结构：使用者可以快速的插入和删除队列中的元素，并且在预先指定的顺序下快速取得当前在队列中的最小（或者最大）值及其对应元素。可以看到，这样的数据结构非常符合 Stride 调度器的实现。

本次实验提供了 libs/skew_heap.h 作为优先队列的一个实现，该实现定义相关的结构和接口，其中主要包括：

```
1  // 优先队列节点的结构
2  typedef struct skew_heap_entry skew_heap_entry_t;
3  // 初始化一个队列节点
4  void skew_heap_init(skew_heap_entry_t *a);
5  // 将节点 b 插入至以节点 a 为队列头的队列中去，返回插入后的队列
6  skew_heap_entry_t *skew_heap_insert(skew_heap_entry_t *a,
7                                      skew_heap_entry_t *b,
8                                      compare_f comp);
9  // 将节点 b 插入从以节点 a 为队列头的队列中去，返回删除后的队列
10 skew_heap_entry_t *skew_heap_remove(skew_heap_entry_t *a,
11                                     skew_heap_entry_t *b,
12                                     compare_f comp);
```

其中优先队列的顺序是由比较函数 comp 决定的，sched_stride.c 中提供了 proc_stride_comp_f 比较器用来比较两个 stride 的大小，你可以直接使用它。

当使用优先队列作为 Stride 调度器的实现方式之后，run-queue 结构也需要作相关改变，其中包括：

- struct run_queue 中的 lab4_run_pool，在使用优先队列的实现中表示当前优先队列的头元素，如果优先队列为空，则其指向空指针（NULL）。
- struct proc_struct 中的 lab4_run_pool 结构，表示当前进程对应的优先队列节点。

本次实验已经修改了系统相关部分的代码，使得其能够很好地适应 LAB4 新加入的数据结构和接口。而在实验中我们需要做的是用优先队列实现一个正确和高效的 Stride 调度器，如果用较简略的伪代码描述，则有：

- `init(rq)`:
  - `Initialize rq->run_list`
  - `Set rq->lab4_run_pool to NULL`
  - `Set rq->proc_num to 0`
- `enqueue(rq, proc)`
  - `Initialize proc->time_slice`
  - `Insert proc->lab4_run_pool into rq->lab4_run_pool`
  - `rq->proc_num ++`
- `dequeue(rq, proc)`
  - `Remove proc->lab4_run_pool from rq->lab4_run_pool`
  - `rq->proc_num --`
- `pick_next(rq)`
  - `If rq->lab4_run_pool == NULL, return NULL`
  - `Find the proc corresponding to the pointer rq->lab4_run_pool`
  - `proc->lab4_stride += BIG_STRIDE / proc->lab4_priority`
  - `Return proc`
- `proc_tick(rq, proc)`:
  - `If proc->time_slice > 0, proc->time_slice --`
  - `If proc->time_slice == 0, set the flag proc->need_resched`

3 实验任务以及帮助

- 实验任务 1: 回答之前 2.7 末尾提出的问题。
- 实验任务 2: 根据 2.7 和 2.8 提供的描述，仿照 round-robin 调度器提供的实现 sched_RR.c 的模式，完成基于优先队列实现的 stride 调度器 sched_stride.c
- 在已经存在的文件中，所有需要补完的代码位置可以通过如下命令找到：

```
grep "LAB4: YOUR CODE" * -ir --color
```

STEP 1 : 根据之前对 Stride 调度器的相关描述，完成其实现。

STEP 2 : 修改 sched.c 使其选择你所实现的 stride 调度器作为使用的调度器。

4 扩展实验

在 ucore 的调度器框架下实现下面若干种调度器：

- Linux 的 CFS 调度器 or O(1)调度器 or SD 调度器。
- Solaris 的多级反馈队列调度器
- FreeBSD 的 ULE 调度器
- Windows 的调度器

5 实验报告要求

在代码目录 lab4 下完成实验。回答实验指导书中的问题，在实践中完成实验中的练习。在报告中简要描述如何完成练习，练习中碰到的问题/困难，已经如何解决的等。实验报告文档命名为 lab4-学生 ID.txt，在 proj13.1 目录下执行：

```
make handin
```

会得到 proj13.1-handin.tar.gz 的文件。将此文件和实验报告放在 lab4-handin 目录下，并在 Lab4 根目录下执行

```
tar jcf lab4-StudentID.tar.bz2 lab4-handin
```

即可生成 lab4-*StudentID*.tar.bz2 压缩文件，请注意把*"Student ID"*替换成你的学号。比如你的学号是 2008011999，则对应的命令是

```
tar jcf lab4-2008011999.tar.bz2 lab4-handin
```

如果你做了 Challenge，除了要在报告上写明外，还要在网络学堂提交窗口的作业内容一栏注明。