**浙江大学实验报告**

**课程名称**： 操作系统

**实验项目名称**： RV64 内核线程调度

**学生姓名**： **学号**：

**电子邮件**：

**实验日期**： 2023 年 10 月 20 日

1. **实验内容**

1.1准备工程

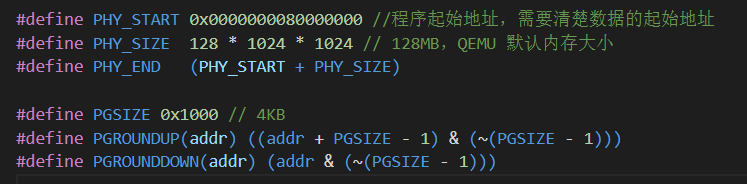
本实验基于lab1所实现的代码进行，先repo同步以下代码:rand.h/rand.c，string.h/string.c，mm.h/mm.c，proc.h/proc.c，test.h/test\_schedule.h，schedule\_null.c/schedule\_test.c以及新增的一些 Makefile文件变化。

将这些文件正确放置后，可以直接复制lab1中的文件至lab2对映的文件夹中。确保工程可以正常运行，有可能需要同学自己调整修改一些Makefile文件以及在一些源代码文件中添加头文件的引入。

1.2 宏定义的添加

在lab2中我们需要使用一些物理内存管理接口，在此我们使用了kalloc接口。使用kalloc函数可以申请4KB的物理页。

由于引入了简单的物理内存管理，我们需在\_start的适当位置调用mm\_init，来初始化内存管理系统，并且在初始化时需要使用一些自定义的宏，因而需要在defs.h文件中添加如下内容：



1.3 线程调度的预备知识

1.3.1 线程的结构属性

在不同的操作系统中，操作系统会为每个线程保存的信息都不同。在本次实验中，我们仅提供一种基础的实现，因而每个线程会包括：

线程ID：用于唯一确认一个线程的标志。

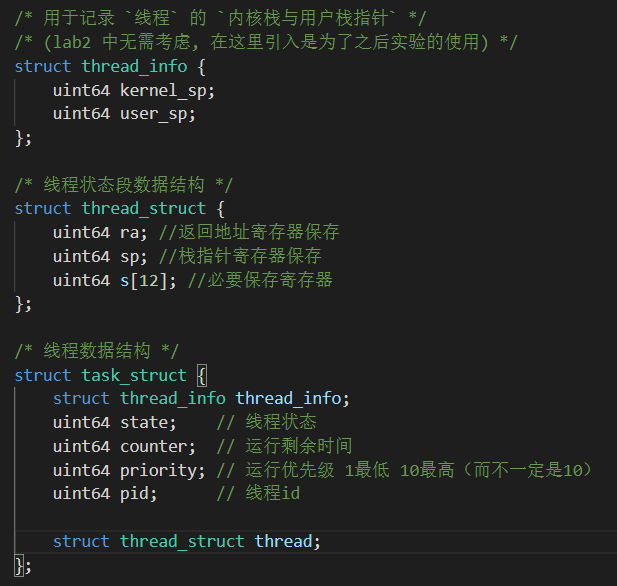
运行栈：每个线程都必须有一个独立的运行栈，用于线程保存运行时的数据。

执行上下文：当线程不在执行状态时，我们需要保存其上下文---一些寄存器的当前数值，这样在线程运行时间到后才能将其正常恢复，继续运行。

运行时间片：为每个线程分配的当前总运行时间。

优先级：在优先级相关调度时，配合调度算法，来选出下一个执行的线程。

具体的数据结构实现如下：



1.3.2 线程切换/调度

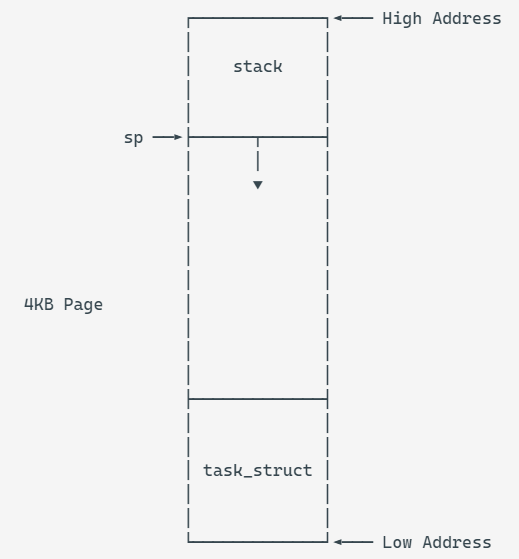
在每次处理时钟中断时，操作系统首先会将当前线程的运行剩余时间减少一个单位。再根据调度算法来确定是继续运行当前线程还是调度其他线程来执行。

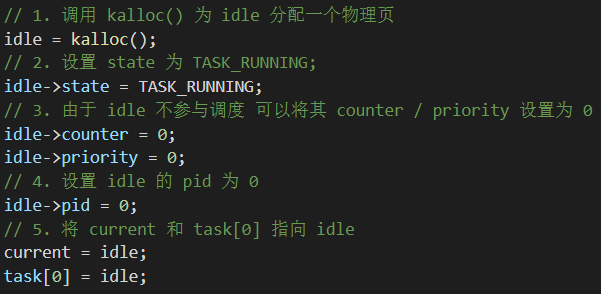
在进程调度时，操作系统会遍历所有可运行的线程，按照一定的调度算法选出下一个执行的线程，本实验实现的是最短作业优先算法和优先级调度算法。最终将选择得到的线程与当前线程进行切换。

在切换的过程中，首先我们需要保存当前线程的执行上下文，再将将要执行线程的上下文载入到相关寄存器中，至此我们就完成了线程的调度与切换。

1.4 线程调度功能实现

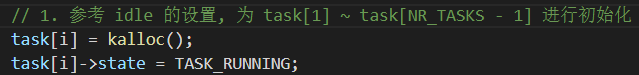
1.4.1 线程初始化

在初始化线程时，我们使用kalloc函数为每个线程分配一个4KB的物理页，并将对映的task\_struct存放在该页的低地址部分，将线程的栈指针sp指向该页的高地址部分。具体结构示意图见下：

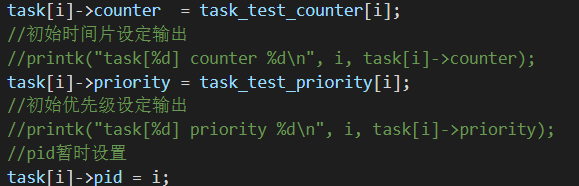
当我们的操作系统开始运行时，其本身就是一个线程，在本实验中被我们成为“idle”线程（本实验中不参与二次调度），但此时我们并没有为它设计task\_struct。所以第一步我们要为idle线程设置task\_struct。并将current（当前运行的线程），task[0]（系统支持线程的首个地址）都指向idle。

具体而言，我们需要将idle的pid设置为0，并同样将其时间片与优先级置0（表示基础进程，不参与线程调度）；另外，我们还需要将idle的线程状态设置为运行态。

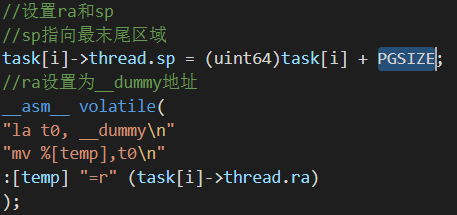
在此之后，我们要为操作系统中可能运行的其他线程赋予同样的属性信息。首先，我们效仿idle线程的设置，为这些线程分配地址设置状态。



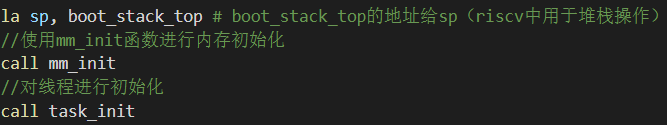
接着，根据test\_init(NR\_TASKS)函数调用的赋值语句，我们为每个线程设置对映的初始化时间片大小以及对映的优先级高低。然后，我们为每个线程设置对映的pid信息。



最后，我们使用内联汇编语句，为线程的ra、sp寄存器存储至附初值。其中ra存储\_\_dummy全局标签的地址信息，sp根据原理定义，指向线程拥有空间的最上端（使用PGSIZE偏移）。



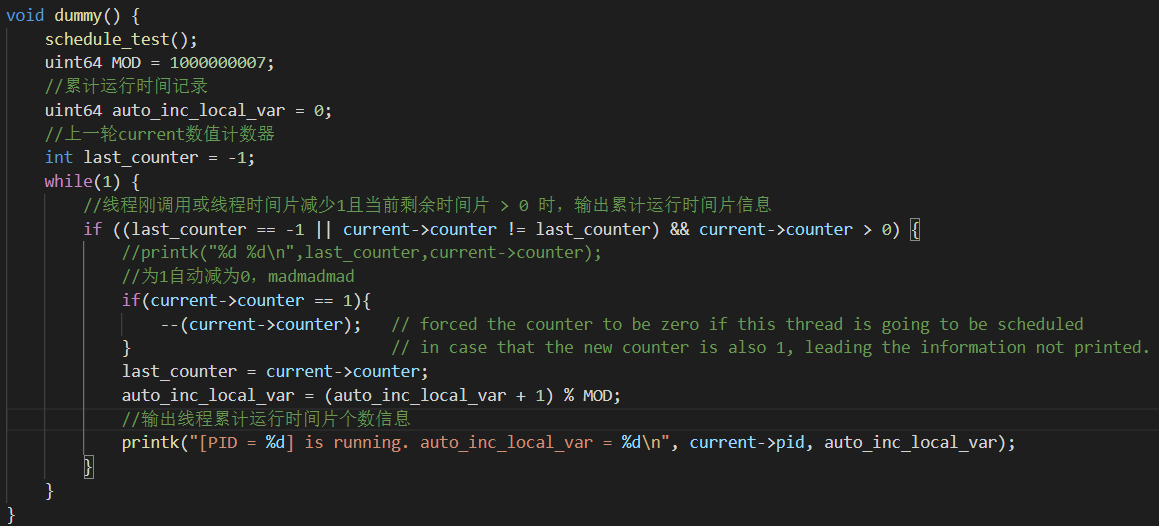
最后，我们要在\_start（head.S文件中）全局标签的适当位置调用task\_init函数。



1.4.2 dummy函数

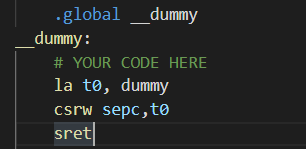
在本实验中，我们所用的线程均为模拟线程，并没有实质的运行状态。因而在实验的实现中，我们采用调用dummy函数，并应用while(1)这一无限循环，来模拟线程的最初运行过程（为线程调度的ra返回设置合适的地址信息）。

dummy函数的具体实现信息如下：



首先，我们要在entry.S文件中添加全局标签\_\_dummy，并在\_\_dummy中将 sepc 设置为dummy函数的地址，便于使用sret从中断中返回。因为当我们创建一个新的线程，此时线程的栈为空，当这个线程被调度时，并没有上下文可以或需要被恢复的，所以我们为线程第一次调度提供了一个特殊的返回函数\_\_dummy）。

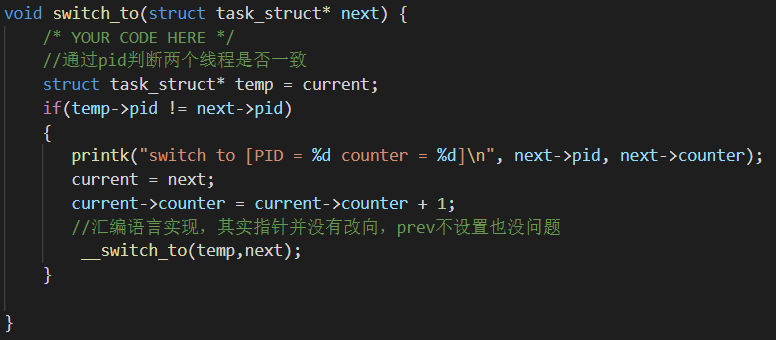
\_\_dummy与\_traps中断的restore过程部分相比，只是省略了从栈上恢复上下文的过程（因为第一次调度，没有上下文需要恢复），但\_\_dummy仍具有手动设置sepc的必要。



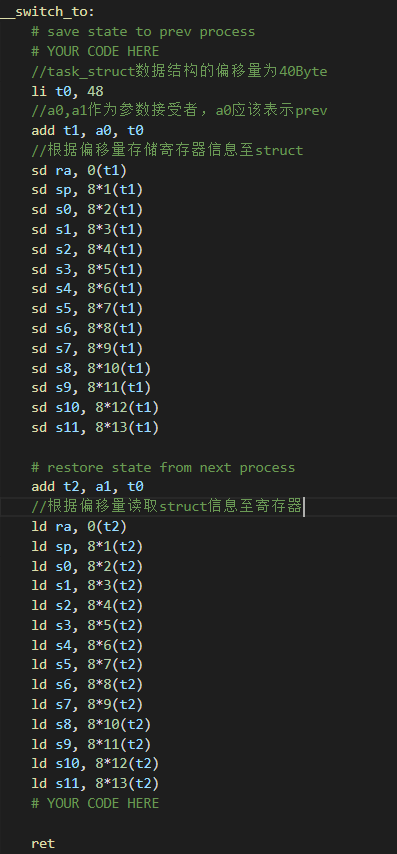
1.4.3实现线程切换

在实现线程切换时，我们需要先判断下一个进程的来源。若下一个需要执行的线程next与当前的线程current为同一个线程，则我们无需做任何处理。否则，我们需要调用\_\_switch\_to函数来进行线程的切换。

这一具体实现我们在调用switch\_to函数时完成。我们可以通过对比线程的pid来比较其是否一致，并输出一定的pid及时间片设定信息以助于我们调试运行。



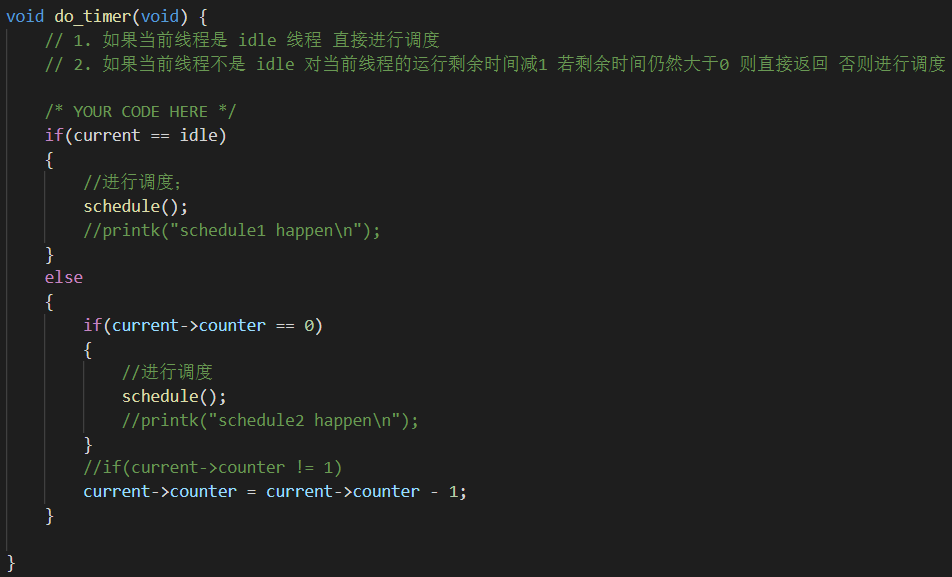
另外，我们要在entry.S文件中实现线程上下文切换函数\_\_switch\_to。\_\_switch\_to接受两个task\_struct指针作为参数（往往依靠a0，a1寄存器传递），保存当前线程的ra、sp、s0-s11寄存器内容到当前线程的thread\_struct结构中，并将下一个线程在thread\_struct结构中的相关数据载入到ra、sp、s0-s11寄存器中。具体实现如下：



1.4.4 实现调度入口函数

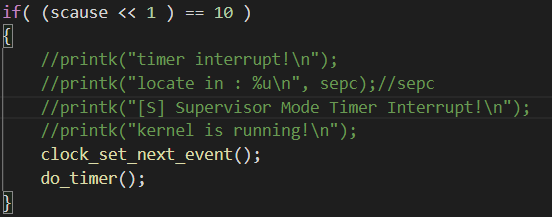
在实现了线程切换的判断及实现函数后，我们需要为其设置一个调度入口，由此介导调度函数的运行，从而间接引导线程切换。

在本实验中，我们实现do\_timer函数来成为线程调度入口，并在时钟中断处理函数中调用其，使调度问询与时钟中断同步。具体实现如下：



若当前运行线程为idle，我们可以直接启动线程调度函数；若不为idle线程，我们在当前线程时间片不为0时，仅仅进行对其时间片减1的操作；若时间片为0，则我们需要进行相应的线程调度工作。线程调度函数schedule我们将在后续模块中分析。

最后，我们要把do\_timer函数放置于时钟中断处理函数的适当位置。



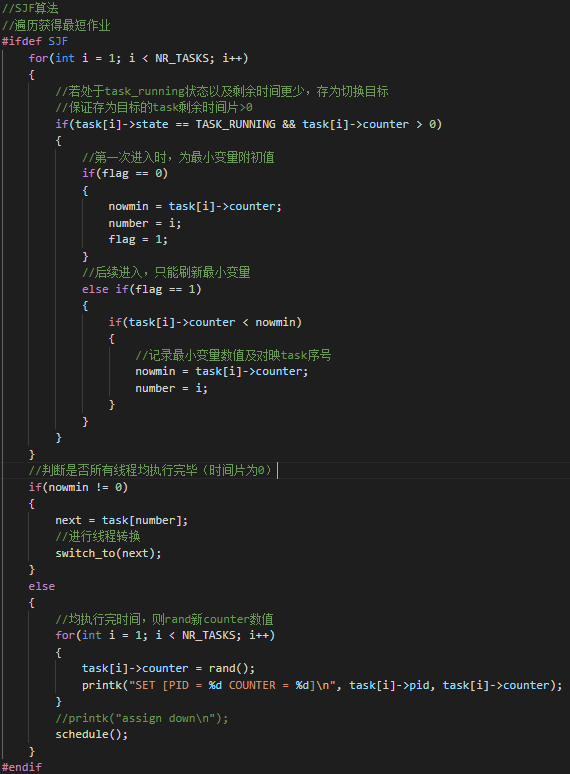
1.5 线程调度

本次实验我们需要实现两种调度算法：1.短作业优先调度算法，2.优先级调度算法。

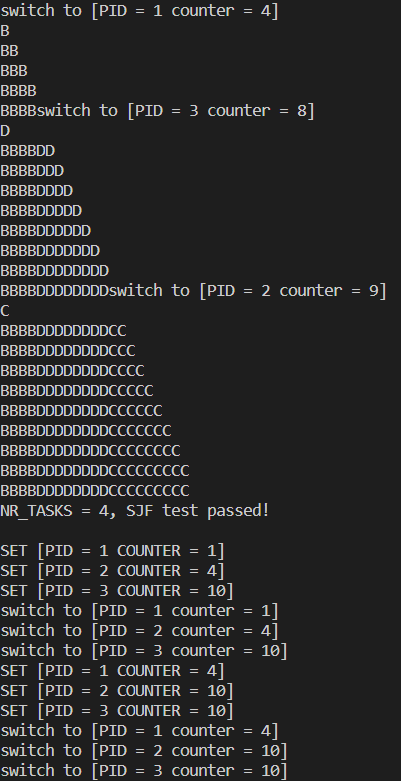
1.5.1短作业优先调度算法

在短作业优先调度算法中，当我们需要进行线程调度时，我们先遍历线程指针数组task（不包括idle线程，即task[0]），在所有为运行状态下的线程中寻找运行剩余时间片最少的线程，让其作为下一个执行的线程。

如果所有运行状态下的线程运行剩余时间片都已为0，我们则对所有task线程（除idle线程）的运行剩余时间片重新赋值（本实验使用rand函数随机赋值），在此之后再重新进行线程调度工作。程序具体实现见下：



对于短作业优先调度算法，我们使用make test-run进行task总量为1 + 3的测试。测试结果如下：



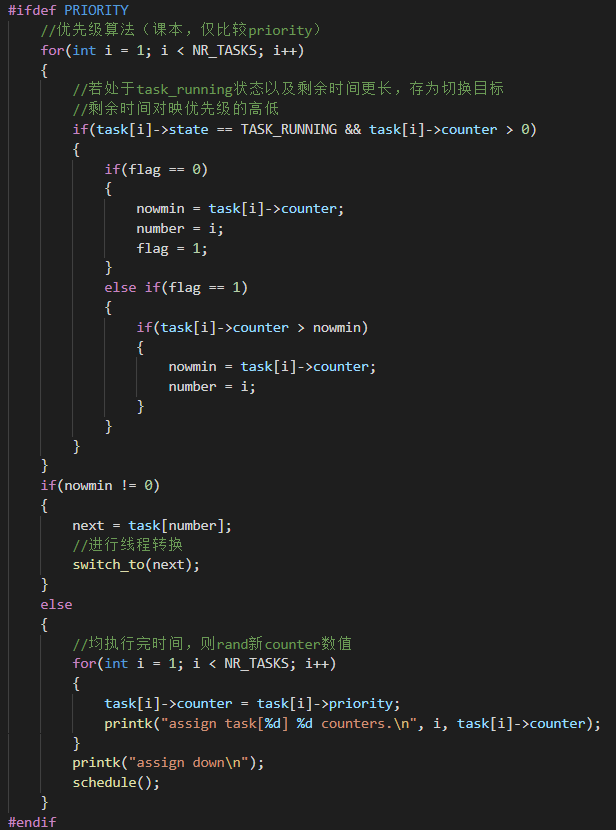
可以看到，当我们第一次实行线程调度时，线程从idle调度到剩余时间片最短的线程；因为这是第一次调度，对映的ra返回地址为dummy函数。所以在第一轮调度之后，线程总体按照剩余时间片的长短进行任务处理，并在dummy程序中进入schedule\_test函数，陷入死循环（打印出结果中呈现的字符串）。

由结果显示我们可以看到，我们成功通过了NR\_TASKS为4时的test样例。且在第二次随机赋予各线程时间片后，我们的调度程序并不会在此于schedule\_test函数中不停的打印字符。由具体的输出结果我们可以看出，线程的运行顺序还是按照最短作业顺序进行优先挑选的，符合我们的设计要求。

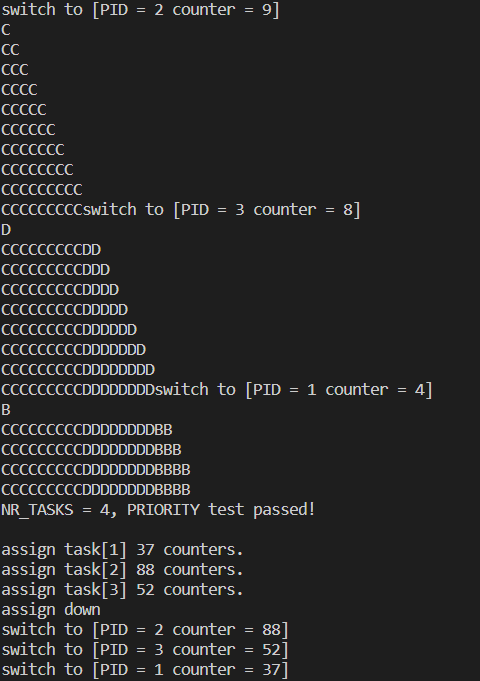
1.5.2 优先级调度算法

在优先级调度算法中，我们根据不同线程的优先级，来挑选安排线程的运行顺序。在本实验中，我们仿照“[Linux v0.11 调度算法](https://elixir.bootlin.com/linux/0.11/source/kernel/sched.c" \l "L122)”进行实现。

其实现的本质为，当所有线程的时间片均为0后，我们根据每个线程的优先级大小，将其优先级的具体数值加至其对映时间片上；在调度线程时，我们可以选择时间片最长的线程进行运行。这样，优先级较大（较高）的线程，可以获得更多的时间片以及更优先的运行情况。具体的代码实现见下：



在此，我们也对于优先级调度算法，使用make test-run进行task总量为1 + 3的测试。测试结果如下：

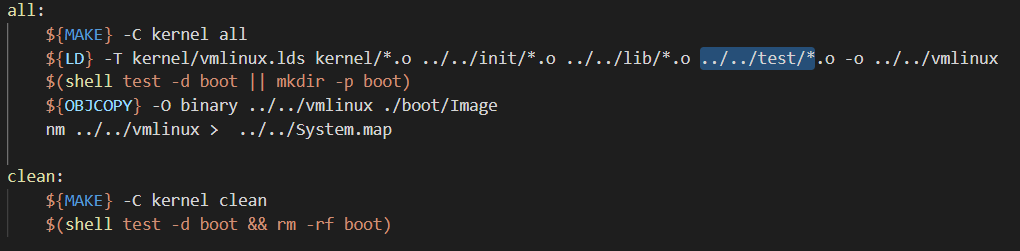


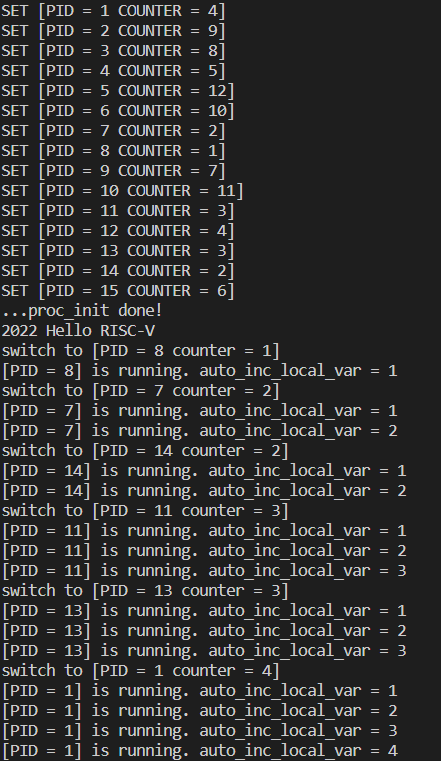
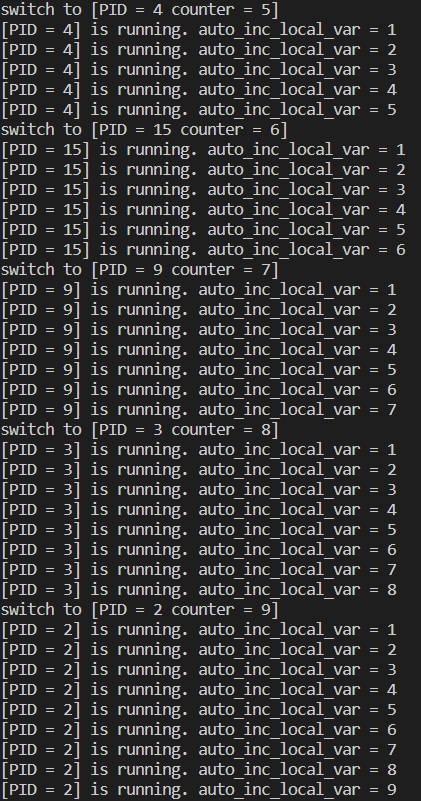
从结果截图我们可以看到，在第一次实行线程调度时，线程从idle调度到剩余时间片最多的线程（这与我们优先级优先调度算法的设计有关，时间片长等价于优先级高）；因为这是第一次调度，对映的ra返回地址为dummy函数。所以在第一轮调度之后，线程总体按照剩余时间片的长短进行任务处理，并在dummy程序中进入schedule\_test函数，陷入死循环（打印出结果中呈现的字符串）。

由接下来的结果我们可以得到，我们成功通过了NR\_TASKS为4时的test样例。且在第二次根据各线程优先级赋予各线程不同数量的时间片后，我们的调度程序并不会再次于schedule\_test函数中不停的打印字符。且由具体的输出结果我们可以看出，线程的运行顺序还是按照最高优先级线程先运行的顺序进行优先挑选的，符合我们的设计要求。

1.6 编译与测试

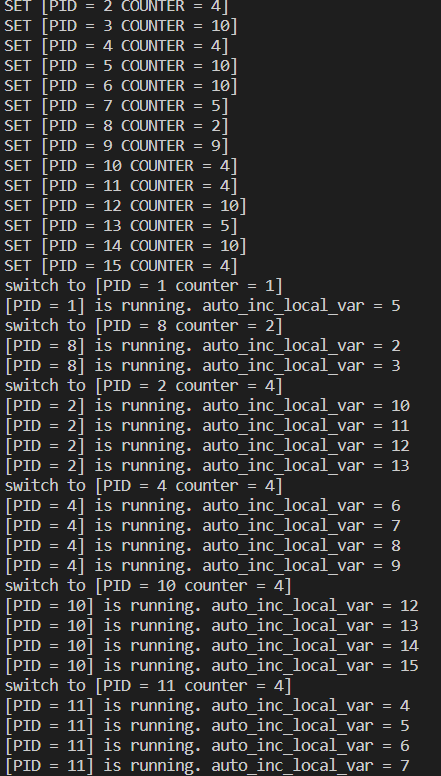
为了成功运行工程，我们需要在makefile文件中包括test文件夹中的各个文件，可以在makefile文件中添加“../../test/\*”指令。



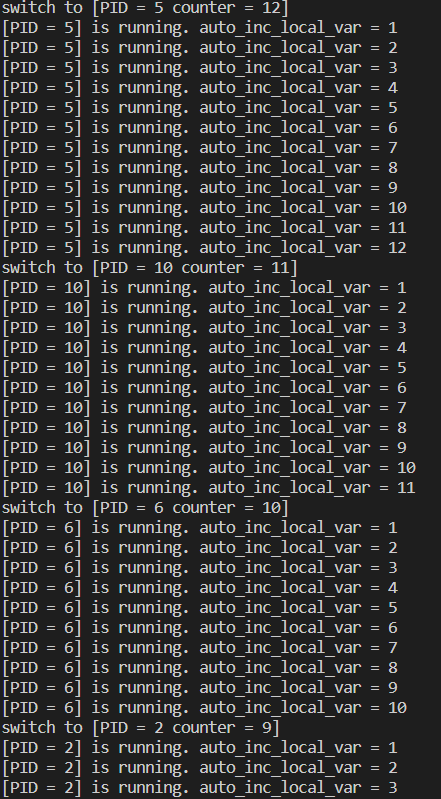
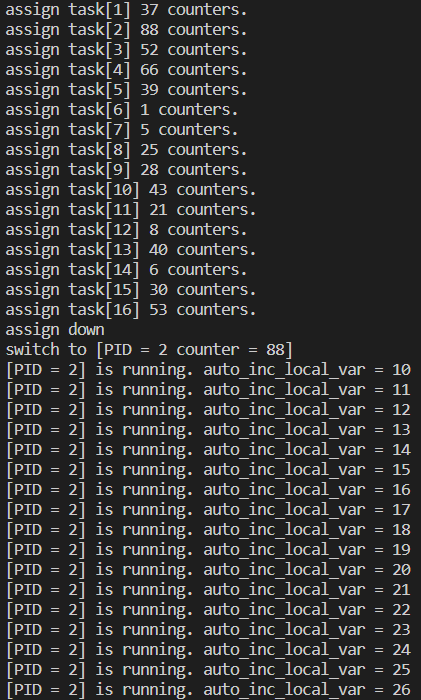
另外，对于“最短作业优先调度算法”或是“优先级调度算法”的选择，我们采用了 #ifdef和 #endif 的选择编译方法，并在对映的makefile文件中，添加了“CFLAG = ${CF} ${INCLUDE} -DPRIORITY”或“CFLAG = ${CF} ${INCLUDE} -DSJF”指令，用于在编译时选择使用不同的调度算法。

最后，我们使用make run指令判断本实验设计的正确性，最短作业优先算法的输出结果见上，为便于展示，我们这里仅展示NR\_TASKS = 16情况下的部分结果。我们可以看到，首先，对于第一轮线程运行而言，各个线程都运行了等价于时间片大小的时间；并且对照各个线程初始化的时间片，可以看出，这个调度是按照最短作业对映的顺序进行线程优先调度运行的。

在经过一次调度后，我们可以看到，各个线程的时间片输出说明，并不是重新从0开始计算，而是输出线程运行的累积时间片结果，并对映相应的新一轮时间片大小。



对于优先级调度算法，我们也可以得出正确的结果。其结果表现大体上与最短作业优先级算法保持一致；仅在线程调度时，由选择最短作业线程转换为选择优先级最高的线程。具体输出结果见下：



1. **思考题**

1.在 RV64 中一共用 32 个通用寄存器，为什么context\_switch中只保存了14个?

在context\_switch函数中，我们仅保存了ra、sp、s0 - s11这十四个寄存器。首先对于ra和sp寄存器而言，我们一定需要保存其中的内容，因为ra寄存器关系到线程切换后相应的返回地址，而sp寄存器关系到对映线程栈的数据信息使用。

另外，s0 - s11寄存器，本质上属于callee寄存器；在系统程序运行的过程中，调用者会默认被调用程序会使用这些寄存器，因而需要主动保存这些寄存器中的内容。对于其他寄存器而言，他们属于caller寄存器；调用者无需在意这类寄存器，因而在调用范式中，被调用者必须为调用者存储这些寄存器中的内容，以至于在调用结束后将其恢复原貌。对于调用者而言，caller类型的寄存器在使用前后不会发生变化。因而其只需要保存callee类型的寄存器即可。

2.当线程第一次调用时，其ra所代表的返回点是 \_\_dummy。那么在之后线程对映的线程切换中，context\_switch中的ra保存/恢复的函数返回点是什么呢?

第一次ra调用返回点：\_\_dummy标签。

a7f84f6ce4fe54679fe3f774777d46e

第二次ra调用返回点：switch\_to + 120（即switch函数的结尾，表示进程切换结束）。



第三次ra调用返回点：依然为switch函数的结尾。

**三、讨论、心得**

本次实验总体而言算是顺利，特别是第一遍code时，感觉跟着实验指导一步步往下异常合理。但在最后的test与调试阶段，还是出现了一些花费时间且令人讨厌的小bug。

首先是test环境与run环境不同所带来的困扰。在run环境中，我们的current->counter变量会被自动多减一次（当其为1时，会被自动减1置0）。首先，这在一定程度上会导致程序的错误---64bit无符号数0 - 1导致数据下溢；其次，test与run环境的作减次数不同，可能导致我们不能同时满足两个环境的正确运行结果。在满足run环境正确运行的前提下，可能导致test failed的情况。

另外，在时钟中断处理程序中，我们需要先设置下一次时钟中断的时间，再执行do\_timer函数。若我们先执行了do\_timer函数，则在\_\_dummy标签段运行sret语句时，会重新将“允许时钟中断”位打开，此时时钟中断值并没有被重新设置，从而会导致再次进入时钟中断的结果，导致程序运行错误。