P2的整体逻辑

曾有同学问过，P1这个实验和写一个普通的C程序感觉区别不是很大。“操作系统”的概念体现在哪里？实际上P1就是一个在裸板上执行的程序，更多的是教大家在没有操作系统的环境下，应当怎样编写一个程序。对于操作系统本身的概念涉及不多。

与P1不同，P2是一个虽然简单，但是具备了最最基础的功能的内核。操作系统是负责管理硬件并提供系统服务的。因此，在P2中，内核实际上扮演了一个“提供系统服务”的角色。而我们的测试程序将使用内核提供的系统服务来完成各种任务。

测试程序更接近于我们平时在操作系统上执行的C程序，它会使用C库中的一些函数（以及一些系统服务）。为了更接近真实系统，我们做了一个超小型的C库(tiny_libc)。测试程序会使用libc中的函数执行一些任务。libc中的函数有的是简单的字符串操作等不依赖OS的功能，有的则依赖于OS提供的服务。依赖于OS的服务最终会调用到`sys/syscall.h`中的函数，其定义位于`tiny_libc/syscall.c`中。这些函数负责真正触发内核的相关功能，请求相应的系统服务。任务一和任务二中，可以直接调用内核的相关函数执行系统服务。在后面的任务中，建立了完整的例外处理机制后，需要通过触发软中断陷入内核态请求系统服务。

以printf为例，在任务一中，用户程序调用printf。printf是被定义在`tiny_libc/printf.c`中的。之后，可以调用`tiny_libc/syscall.c`中的`sys_write`函数。再由`sys_write`函数调用内核中的`port_write`服务直接写串口，从而实现输出。也可以在`printf.c`中，直接调用内核提供的`printk`函数来使用内核提供的功能。在后面任务三有了例外处理等保护机制后，触发内核服务的方式需要由对`port_write`或`printk`内核函数的直接调用的方式，改成用`ecall`陷入内核态，再通过内核的例外处理机制执行相应的内核功能的方式。

FAQ

Part1:

代码中的ready_queue等队列如何初始化？如何操作？

代码中所有对于队列的操作都需要同学们根据自己的实际情况自行实现。start code中和队列相关的代码都需要大家根据自己的具体实现方式自行调整。

早期版本的实验中，我们仿照Linux的链表实现提供了一个list.h。但是从往年的实际情况看，Linux中链表的设计更侧重于追求通用性，用到了一些和C语言内存布局方面相关的技巧，导致部分同学理解起来比较困难。我们的内核比较简单，需要的队列的类型很少，即使不做很通用的设计，也可以正常完成（虽然有可能有部分代码上的冗余）。因而，今年不再提供链表的具体实现，请大家根据自己的需求自行设计。如果需要，框架中涉及到链表类型的函数声明等都可以自行修改。

简单起见建议大家就采用自己在数据结构教科书上学到的标准的链表实现即可。**学有余力**可以稍微参考一下Linux或其他操作系统中链表的实现方式。

Linux的链表的设计更多的是追求的代码可重用。传统的教科书里定义链表时，都是把数据和指针域放在一个结构体里，然后针对这个结构体写各种插入删除等代码的。但如果有多种不同数据类型都要串成链表，代码就不好复用。例如：

```
// 一般的链表,同样的东西需要反复定义
// 特别是链表的增加/删除等操作的函数也需要跟着重新定义
struct IListNode {
    int val;
    struct IListNode* next;
};
void IListAdd(struct IListNode* head, struct IListNode *new_node); // ...
struct DListNode {
    double val;
    struct DListNode* next;
};
void DListAdd(struct DListNode* head, struct DListNode *new_node); // ...
struct SListNode {
    struct Something val;
    struct SListNode* next;
};
void SListAdd(struct SListNode* head, struct SListNode *new_node); // ...
```

对于这种问题，在一些新的编程语言中可以用范型或者面向对象的类的继承等方式解决。例如C++就可以写`list<int>`、`list<double>`等。但C语言本身没提供这类的语法。所以Linux中，把链表的指针域和相关的操作抽了出来，任何结构体想串成链表只需要嵌入相应指针域就行。

```
// Linux 里的指针域及其操作的定义
struct list_head {
    struct list_head *next, *prev;
};
static inline void list_add(struct list_head *new, struct list_head *head);
static inline void list_add_tail(struct list_head *new, struct list_head *head);
static inline void list_del_init(struct list_head *entry);
// 使用的时候把指针域嵌入到数据中
struct Val {
    int val;
    struct list_head node;
};
```

感兴趣的同学请自行查阅参考资料中的相关链接。

在任务一中，测试程序中的sys_yield()、sys_move_cursor()等是否要改成do_scheduler()、vt100_move_cursor()等内核函数?

不要将测试程序中的函数直接改成相应的内核函数！

在任务一和任务二中，sys_yield()、sys_move_cursor()等函数确实就是直接调用do_scheduler()、vt100_move_cursor()等内核函数实现的。但是我们仍然希望在形式上，测试程序中是在调用tiny_libc库中的sys_yield()、sys_move_cursor()等函数。这种设计是为了强调一个概念：do_scheduler()、vt100_move_cursor()等内核函数都是内核中的系统服务函数，用户程序不直接调用内核函数（否则就感觉测试程序好像是系统的一部分似的）。虽然我们的内核和测试程序由于种种限制编译在了一起，但本质上，内核是内核，测试程序就和大家平时写的普通的程序差不多，它俩逻辑上是分开的。测试程序是通过系统调用请求内核服务的。

任务一和任务二中由于还没有例外处理机制，在实现系统调用时只能通过普通的函数调用实现。在后面有了例外处理机制后，再将sys_yield()等函数改成用ecall指令实现的方式，实现完整的带保护的系统调用方式。这一过程对于测试程序是透明的，测试程序只知道自己是通过sys_yield()等tiny_libc中的函数去请求系统服务，具体请求系统服务的实现方式它无需关心。这样一层封装可以更明确地强调，从用户程序的角度看，用户程序和内核是分开的。这种设计是为了让大家对于操作系统内核的概念有一个更直观的感觉。

switch_to()和SAVE_CONTEXT/RESTORE_CONTEXT是什么关系?

`switch_to`函数是在内核中主动切换进程时调用的。`SAVE_CONTEXT`和`RESTORE_CONTEXT`是后面例外处理的时候保存例外现场使用的。这两种情况有明显的区别。

`switch_to`是主动切换的，软件明确的知道在调用这个函数后就就进入进程切换了。且因为是主动调用switch_to，所以遵循函数调用的相关约定。所有需要由调用者保存的寄存器已经在函数调用前都保存在栈上了。`switch_to`中只需要保存所有该由被调用者保存的寄存器即可。

`SAVE_CONTEXT`和`RESTORE_CONTEXT`是后面例外处理的时候保存例外现场使用的。例外往往都是突然发生的。例如外部的时钟触发了中断，或者某条指令发生了异常等，用户程序往往不可预知例外是在何处发生的。因此，在例外发生时，需要假定目前所有的通用寄存器都在被使用（因为是突然发生的，可能在任意一条指令处触发，所以没法确定此刻哪些寄存器是有用的，哪些是没用的，和函数调用时的情况不同）。这也就是为什么`SAVE_CONTEXT`和`RESTORE_CONTEXT`涉及所有的寄存器的原因。

printk函数崩溃是什么原因?

printk函数会试图把当前的光标保存到current_running，如果current_running为NULL的话会出错。这里根据你自己的设计可以选择删除那段保存光标的代码或者让current_running始终非空。

```
/* libs/printk.c: vprintk() */
disable_preempt();
port_write(buff);
for (int i = 0; i < ret; ++i) {
    if (buff[i] == '\n') {
        current_running->cursor_y++;
    } else if (buff[i] == '\r') {
        current_running->cursor_x = 1;
    } else {
        current_running->cursor_x++;
    }
}
enable_preempt();
```

enable_interrupt/disable_interrupt实现存在错误?

这里确实存在问题，start code中的代码是：

```
ENTRY(enable_interrupt)
  li t0, SR_SIE
  csrs CSR_SSTATUS, t0
  jr ra
ENDPROC(enable_interrupt)

ENTRY(disable_interrupt)
  li t0, SR_SIE
  csrs CSR_SSTATUS, t0
  jr ra
ENDPROC(disable_interrupt)
```

应该改成：

```
ENTRY(enable_interrupt)
  li t0, SR_SIE
  csrs CSR_SSTATUS, t0
  jr ra
ENDPROC(enable_interrupt)

ENTRY(disable_interrupt)
  li t0, SR_SIE
  csrc CSR_SSTATUS, t0
  jr ra
ENDPROC(disable_interrupt)
```

enable_preempt/disable_preempt是做什么用的?

在一些内核代码里能看到`enable_preempt`和`disable_preempt`这一对函数，这对函数原本的功能是允许发生抢占。但我们的实验中为了简化，不需要支持内核抢占。因此，所有的这对函数都可以去除。因为在内核态下是不会发生时钟中断导致抢占的。

建议可以把这两个函数直接改成：

```
ENTRY(enable_preempt)
  jr ra
ENDPROC(enable_preempt)

ENTRY(disable_preempt)
  jr ra
ENDPROC(disable_preempt)
```

相当于不执行任何功能，或者直接删掉就完了。以前有极少数同学会出现不删这两个函数就有奇怪的错误，删掉就能正常工作的情况。

pid0是做什么用的?

在start code中给了一个`pid0_stack`、`pid0_pcb`等的定义。这些定义可用可不用，与每个人自己的设计有关。代码框架中的pid0相关的变量代表的是最开始的内核。最开始刚启动的时候跑的是内核的代码。之后要切换到第一个任务中。但`switch_to`假定是从一个任务切换到另一个任务。所以对于内核代码切换到第一个任务的这一过程，不完全符合`switch_to`的假设。

`switch_to`的主要假设是`a0`和`a1`两个寄存器传入了两个pcb。pcb中记录了上次进程切换时kernel stack的位置。对于内核而言，它没有相应的pcb。为了符合这个假设，所以我们做了一个pid0的pcb。并且最开始将current_running设置为pid0_pcb。这样就可以比较自然地从代表内核的pid0_pcb切换到第一个任务的pcb。后面内核这个进程可以用作一个idle进程。当没有任何任务可以调度的时候就跑内核的死循环代码（循环调用`do_scheduler`）。

也可以对switch_to做一些小改动来完成这个。`switch_to`需要传入pcb的原因单纯是需要保存/恢复kernel_stack的位置。所以其实也可以选择把`switch_to`改成传两个指针进去。一个用于保存上一个任务的kernel_stack，一个则是指向了下一个任务的kernel_stack的值。这样，对于内核切换第一个任务的过程，调用switch_to的时候用于保存上一个任务的kernel_stack的指针随便指向一个合法的位置就行。当然，也有更为朴素的方式，就是直接判断是否是内核切第一个任务，并特殊处理即可。

参考资料

1. selfimpr1991, “ 【linux 内核数据结构】最为经典的链表 list.” https://blog.csdn.net/wenqian1991/article/details/44515713, 2015. [Online; accessed 23-September-2021].
2. full_of_bull, “Linus:利用二级指针删除单向链表.” https://coolshell.cn/articles/8990.html, 2013. [Online; accessed 23-September-2021].