OSLab实验报告

丁保荣 171860509

1770048119@qq.com

L0:

L0中只是很简单得实现了一个滑块的移动，通过上下左右键来控制滑块，复用了框架代码的一部分。

代码的架构很简单：就是是获取屏幕大小，并将初始滑块的坐标置于屏幕正中央，然后进入一个死循环，读取键盘输入，如果输入了上下左右，就进行滑块的移动，在滑块的移动中加入边界检测。然后只有当滑块移动后，才进行屏幕的重新绘制。

在实现滑块移动的时候，一开始是每隔一秒将屏幕整个清除，重新绘制。但运行后发现屏幕的输出很诡异，完全不是想要的效果，猜测是不是重新绘制屏幕代价太大了。因此，最后修改为只有当检测到滑块移动的时候更新屏幕。而且更新的时候不是清除整个屏幕，而是只清除滑块的上一个位置，并且只绘制滑块的下一个位置，这样写屏幕的需求就降低了很多。

L1:

L1实现kalloc和free。采用链表的思想，但因为没有办法申请内存，所以直接讲链表嵌入堆区。一开始是只有一个节点，空间是所有可用空间。

kalloc调用的时候，从链表头开始寻找空闲空间，当寻找到一个空闲空间大于所需空间的时候，就停下来。如果空闲空间特别大，会将当前的节点分成两个节点，并将第一个节点的分配内存的地址返回。

free调用的时候，根据需要free的地址找到对应节点的地址，然后将这个节点管理的空间声明为可用。如果当前节点能和周围的空闲节点合并，那就将他们合并，直到不能合并为止。

在所有分配的地址中都是按照8字节对齐的。

关于锁的问题，我只实现了spin-lock,但没有关中断。基于这么几个原因。1. 如果采用内联汇编，用户程序(kalloc)是根本没有关中断的权限，我试了一下，会直接segmentation fault。这样的话，在native上根本没用办法执行。2. 网上的资料说malloc这种函数都是不可重入的，我认为在linux中断处理程序中很可能不会调用malloc,而且即使要分配内存，也是调用操作系统内部提供的接口，如brk之类的。

L2:

L2主要实现内核的线程管理以及一些锁的实现。

os->on_irq是用来实现注册中断处理程序。我主要用一个链表是存储handler，整个链表是按照seq的大小排序的，这样在调用中断处理程序的时候只要从前往后扫一遍就好了。实现的时候也没有什么bug。

os->trap是用来调用对应的handler的，所以只要把整个handler的链表从头到尾扫一遍就好了。

kmt->spinlock相关的代码是仿照xv6的实现。其中对于中断的处理，用一个栈来保存，这样就可以实现中断的嵌套。

kmt->context_save很简单，直接把传进来的context保存在current里就好了。

kmt->context_switch用来实现线程切换。线程的状态只有两个READY和SLEEP。每次切换的时候，都选择READY的切换，如果没有READY的，就继续执行当前线程(这里的实现导致我的sem_wait的实现也和标准实现有点区别)。关于进程切换的公平性。 一开始我是每次都从头到尾扫，扫到READY的，就调度它，并把它放到链表的最后，这样就能保持公平性。但后来发现这样的开销有点大，因为每次时钟中断的时候，都要拆链表。因此后面我修改了，每次扫当前线程后面的，当扫到链表尾时，再回到链表头。这样就不需要对链表进行修改了。

kmt>create实现的是线程的创建，主要内容包括绑定到某一cpu，赋予name，把状态设为READY，分配栈空间，创建上下文，加入线程链表。

kmt->teardown 把线程的栈空间释放，并把它从线程链表中移除。

kmt->sem_wait实现的过程中主要遇到的bug是，把if改成了while，这样会导致改线程不断入队列。

kmt->sem_signal把count加一，如果队列不为空，就唤醒队列里的第一个线程。