操作系统实验报告

author:高梓源 Stu. Num: 2019K8009929026

TASK 1

按照程序执行的逻辑对不同文件进行修改。

main

首先是 `init/main.c` 中初始化的实现， 认为所谓任务test就对应一个PCB，针对每个任务及其 `type` ，分配虚拟地址，初始化 `list, pid, type, status` 变得顺理成章，而后去初始化PCB_stack，需要将所有寄存器（终端处理用）和上下文切换所需保存的(callee saved)寄存器初始化，因为一般寄存器初始化从0开始，因此这里较为简单。

初始化完毕后，将PCB对应的 `list` 加入 `ready_queue` 。

对于光标等的初始化包含在以上，因为每个任务在固定位置输出时都要系统调用获取光标位置，因此为每个PCB设置一个光标指示

do_scheduler

设置好任务后可以开始运行和调度，调度过程就是停止当前任务的运行（或已经运行到一定阶段完毕，主动调用调度函数，非抢占式，使用该方法），保存当前任务上下文，寻找 `ready_queue` 中的一个任务（直接取队列对头，第一个进入的任务）并删除该任务，取该任务的上下文加载到寄存器中，开始执行。

如此循环

```
void do_scheduler(void)
{
    // TODO schedule
    // Modify the current_running pointer.
    pcb_t *curr = current_running;
    if(curr->status == TASK_RUNNING && curr->pid != 0){
        list_add_tail(&(curr->list), &ready_queue);
        curr->status = TASK_READY;
    }
    pcb_t *next_pcb = dequeue(&ready_queue);
    next_pcb->status = TASK_RUNNING;
    current_running = next_pcb;
    process_id = next_pcb->pid;

    // restore the current_runnint's cursor_x and cursor_y

    // TODO: switch_to current_running
    switch_to(curr,next_pcb);
}
```

switch_to

上下文切换的汇编宏，这里有两种实现思路：

- 将kernel_sp看作动态变化的栈指针，context信息始终存储在它下方，因此也是动态变化的，最新xv_6采取这种方式
- 将kernel_sp看作context位置的指示，context始终在全部寄存器信息下方，保持不变，旧版xv_6和linux都采取这种方式

上一种方式需要利用当前sp存储上下文有关信息，而后将sp也存储
而后一种方式需要先临时存储sp，加载kernel_sp(context位置指示)信息，利用固定的sp进行存储全部上下文，最终无需重写sp

list

利用linux中实现的精妙设计，PCB中的list域减去结构体中的偏移量即可的出PCB的地址，进而可以得到该进程信息。

实现方法是定义宏：

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)

#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})


#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)
```

可利用list_entry去根据list索引PCB

而后根据双向链表本身的性质，实现一些增删改查、判断空操作，可以灵活利用实现队列

杂

因为在tiny_libc中定义了很多未有操作系统实现的 `syscall` 因此需要转换成我们实现的，比如 `sys_yield, sys_move_cursor` 等

TASK 2

首先，利用在OSTEP中学到的互斥锁的实现完善 `lock.c`

改变了结构体，将锁拆成2元 `guard, flag` ，前者用来使每个请求锁的任务都能够跳出循环，提升性能， `flag` 即标志着当前锁是否被占有，若当前锁未被占有，则请求成功的锁 `flag` 为1，否则进入 `block_queue` 。

释放锁的过程同样美妙，若当前 `block_queue` 为空，则代表锁仍然可以被当前任务占有， `flag` 设置为0,在下一次请求锁时可以成功获取锁；否则，当前进程从 `block_queue` 中将一个进程解救出来，放到 `ready_queue` 中

有一个trick，事实上解救出来执行的任务并不会再次 `acquire` 获得锁，因此不改变 `flag` 是明智的，新任务开始执行则自动获得了锁，其他任务的见到 `flag` 为1则按部就班进入 `block_queue` ，因此这种trick保证了结果的正确性。

```
void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    /* TODO */
    while (atomic_cmpxchg_d(UNGUARDED, GUARDED, (ptr_t)&(lock->lock.guard)) == GUARDED)
    {
        ;
    }
    if(lock->lock.flag == 0){
        lock->lock.flag = 1;
        lock->lock.guard = 0;
    }
    else{
        do_block(&current_running->list,&lock->block_queue);
        lock->lock.guard = 0;
        do_scheduler();
    }
}

void do_mutex_lock_release(mutex_lock_t *lock)
{
    /* TODO */
    while (atomic_cmpxchg_d(UNGUARDED, GUARDED, (ptr_t)&(lock->lock.guard)) == GUARDED)
    {
        ;
    }
    if(list_empty(&lock->block_queue)){
        lock->lock.flag = 0;
    }
    else{
        do_unblock(&lock->block_queue);
    }
    lock->lock.guard = 0;
}
```

进入 `block_queue` 只需要利用 `list.h` 中实现的简单函数 `list_add_tail()` 即可，而后将该进程状态改为BLOCK，注意避免抢占式调度引起的中断使得变量赋值错误，需要在guard拉低之后再调用调度函数

而去解救 `block_queue` 中的任务，利用 `dequeue` 函数，取出对首并且删去，将状态改为READY，而后加入 `ready_queue` 即可。

```c
void do_block(list_node_t *pcb_node, list_head *queue)
{
    // TODO: block the pcb task into the block queue
    list_add_tail(pcb_node,queue);
    current_running->status = TASK_BLOCKED;
}

void do_unblock(list_head *queue)
{
    // TODO: unblock the `pcb` from the block queue
    pcb_t *fetch_pcb = dequeue(queue);
    fetch_pcb->status = TASK_READY;
    list_add_tail(&fetch_pcb->list,&ready_queue);
}
```