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

k_schedule

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 k_schedule(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 k_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{
        k_block(&(*current_running)->list,&lock->block_queue);
        lock->lock.guard = 0;
        k_schedule();
    }
}

void k_mutex_lock_release(mutex_lock_t *lock)
{
    /* TODO */
    while (atomic_cmpxchg_d(UNGUARDED, GUARDED, (ptr_t)&(lock->lock.guard)) == GUARDED)
    {
        ;
    }
    if(list_is_empty(&lock->block_queue)){
        lock->lock.flag = 0;
    }
    else{
        k_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。

```
void k_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 k_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);
}
```

The whole project

Bootblock

work on physical address.

master core:

1. read input to decide which kernel to boot.
2. calculate kernel size and offset.
3. SBI read kernel from SD to memory.
4. jump to kernel start

slave core do not do 1~3, just IDLE and wait for IPI, then execute 4.

kernel entry

head.S: clear bss, setup C environment and jump to boot kernel

boot.c: setup virtual memory, load kenel elf and jump to main

SV-39: user low address, 3-level page table, kernel high address, 2-level page table

kernel init

master init:

- kernel lock

- exception
- syscall
- network card
- file system
- pcb & current_running_task
- screen

slave init:

- cancel kenel direct map
- pcb & current_running_task

kernel schedule

Check ready_queue queue: if empty, just return; or goto bubble task from boot process

priority schedule: consider block time and priority to choose which task in ready_queue to be scheduled

simple schedule: always pop the front task from ready_queue

block queue: lock
timer list: sleep syscall

syscall: spawn, exit, kill, waitpid, process show, taskset, get pid

enter or exit kernel: save and load all registers (gp/csr)
switch between tasks: only load store saved registers. (kernek function call)

exception

After exception, hardware enter kernel, goto interrupt handler. Use handle table to find handler function directly, indexed by exception code reported by hardware.

Then goto handle clock interrupt, SIP, pagefault or syscall. After handle, return to user mode, goto recovery point.

lock

spin lock: atomic_swap, check old value returned. If busy, spin.
mutex_lock: atomic_cmpxchg to get operation to lock, use flag to judge lock status. If busy, block and schedule

communication

semaphore

data structure:

```
typedef struct Semaphore{
    basic_info_t sem_info;
    int sem;
    list_head wait_queue;
} Semaphore_t;
```

p operation:

```
    sem_list[key]->sem--;
    if(sem_list[key]->sem < 0){
        k_block(&(*current_running)->list,&sem_list[key]->wait_queue);
        k_schedule();
    }
```

v operation:

```
    sem_list[key]->sem++;
    if(sem_list[key]->sem <= 0 && !list_is_empty(&sem_list[key]->wait_queue)){
        k_unblock(sem_list[key]->wait_queue.next,2);
    }
```

condition

data structure:

```
typedef struct cond{
    basic_info_t cond_info;
    int num_wait;       // initialized to 0
    list_head wait_queue;
} cond_t;
```

wait operation:

```
    cond_list[key]->num_wait++;
    k_block(&(*current_running)->list,&cond_list[key]->wait_queue);
    k_mutex_lock_release(lock_id,operator);
    k_schedule();
    k_mutex_lock_acquire(lock_id,operator);
```

signal operation:

```
    if(cond_list[key]->num_wait > 0){
        k_unblock(cond_list[key]->wait_queue.next,2);
        cond_list[key]->num_wait--;
    }
```

broadcast operation:

```
    while(cond_list[key]->num_wait > 0){
        k_unblock(cond_list[key]->wait_queue.next,2);
        cond_list[key]->num_wait--;
    }
```

barrier

data structure:

```
typedef struct barrier{
    basic_info_t barrier_info;
    int count;
    int total;
    int mutex_id;
    int cond_id;
} barrier_t;
```

wait operation:

```
    k_mutex_lock_acquire(barrier_list[key]->mutex_id - 1,operator);
    if((++barrier_list[key]->count) == barrier_list[key]->total){
        barrier_list[key]->count = 0;
        k_cond_broadcast(barrier_list[key]->cond_id - 1,operator);
    }
    else{
        k_cond_wait(barrier_list[key]->cond_id - 1, barrier_list[key]->mutex_id - 1,ope
    }
    k_mutex_lock_release(barrier_list[key]->mutex_id - 1,operator);
```

mailbox

```
typedef struct mbox{
    basic_info_t mailbox_info;
    char name[MBOX_NAME_LEN];
    char buff[MBOX_MSG_MAX_LEN];
    int read_head, write_tail;
    int used_units;
    int mutex_id;
    int full_cond_id;
    int empty_cond_id;
    int cited_num;
    int cited_pid[MBOX_MAX_USER];
} mbox_t;
```

send operation:

- if there is enough space to send (space are used in cycle)
  - put message in
  - maintain the write tail pointer and fields related
  - wake up read process
- else cond wait

recv operation:

- if buffer is empty then cond wait
- else read message, maintain read_head pointer and fields related
- wake up blocked

file system

```
#define SUBLK_SIZE          1
#define BLK_MAP_SIZE        8
#define INO_MAP_SIZE        1
#define INO_SIZE            512
#define DATA_BLK_SD_SIZE    FS_SIZE_BLK - DATA_SD_START - FS_START_BLK
#define DATA_BLK_SIZE       DATA_BLK_SD_SIZE >> 3
```

soft and hard link

hard link point to inode, while soft link point to file name.

if rename original file, soft link will loss target while hard link won't.

if delete original file, soft link will point to non-existing file, while hard link will become original file's new name.