PRJ 3

TASK 1 简易终端&基础指令实现

1. 思路说明

（1）SHELL输入解析

退格键的解析：

- 内核给出的函数对于任何用户输入，cursor都是单调递增的，故自己封装一个函数，对于输入的退格键，会进行如下操作：

- 将cursor -1
- 在原位上填充一个空格，并reflush更新屏幕显示。

输入指令的解析：

- buffer暂存读入的数据，以换行分割
- 解析输入时空格分割，并根据首个参数判断操作类型

> Note：unix下的回车会被解析成 `\r` 而非 `\n` 。

（2）内核函数的补全

> 本部分只记录较为重要的tip，函数的具体功能和参数已经在讲义中给出

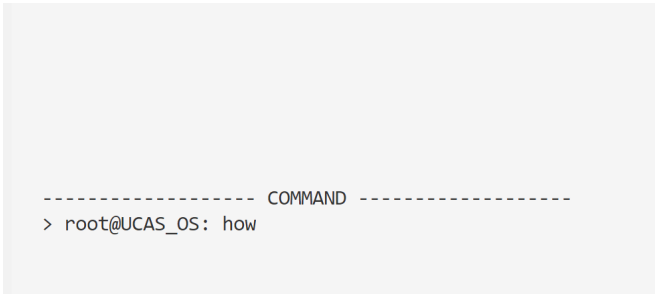
关于Exit和Kill的资源释放：

- exit不在 `do_exit` 中就释放PCB（修改其状态为 `UNUSED` 并释放相关资源），而在其后续调用调度器后统一进行资源释放；
- kill若遇到待kill进程正在执行的情况（含自己kill自己与后续多核的情况），将当前PCB置为 `TASK_EXITED` 状态，这样在后续调度器调用时可同 `exit` 的情况一并判断并回收资源。

2. debug记录

（1）解析输入&处理回显

可以正常输入普通字符：



但输入回车和退格键似乎都会被解析成空格。原本对删除键的实现方式如下：

```
    // TODO [P3-task1]: call syscall to read UART port
    while((tmp = sys_getchar())==-1);
    out[0]=tmp;
    printf("%s", out);
    // TODO [P3-task1]: parse input
    // note: backspace maybe 8('\b') or 127(delete)
    if(tmp=='\b'|| tmp==127){
        if(i>0)
            i--;
        buffer[i]='\0';
    }
```

意识到对于退格，简单地构造一个内容为 `{'\b', '\0'}` 的数组是不得行的……

> (6条消息) (初学者)关于C语言中退格键(\b)的初步了解*partmentXHC*的博客-CSDN博客c语言退格

按照上述思路构造内容为 `{'\b', ' ', '\b', '\0'}` 的字符串，实现如下：

```
    out[0]=tmp;
    // TODO [P3-task1]: parse input
    // note: backspace maybe 8('\b') or 127(delete)
    if(tmp=='\b'|| tmp==127){
        if(i>0)
            i--;
        buffer[i]='\0';
        out[1]=' ';
        out[2]='\b';
    }
    printf("%s", out);
```

输出更加离谱：

```
------------------- COMMAND -------------------
> root@UCAS_OS: how   h
```

查看 `sys_write` 对应的内核函数实现：

```
void screen_write(char *buff)
{
  int i = 0;
  int l = strlen(buff);

  for (i = 0; i < l; i++)
  {
```

```
    screen_write_ch(buff[i]);
}
}
static void screen_write_ch(char ch)
{
if (ch == '\n')
{
    current_running->cursor_x = 0;
    current_running->cursor_y++;
}
else
{
    new_screen[SCREEN_LOC(current_running->cursor_x,
current_running->cursor_y)] = ch;
    current_running->cursor_x++;
}
}
```

从 `current_running->cursor_x++` 可知其会使得cursor的x坐标单调递增。也尝试使用 `bios_putchar` 也无法正常处理退格。似乎只能自己封装一个函数了……在screen.c里加上如下函数：

```
void screen_putchar(char ch){
if (ch == '\n')
{
    current_running->cursor_x = 0;
    current_running->cursor_y++;
}
else if(ch == '\b' || ch== 127){
    current_running->cursor_x --;
    new_screen[SCREEN_LOC(current_running->cursor_x,
current_running->cursor_y)] = ' ';  //打印出空格覆盖原来输入
}
else
{
    new_screen[SCREEN_LOC(current_running->cursor_x,
current_running->cursor_y)] = ch;
    current_running->cursor_x++;
}
screen_reflush();   //输出
}

```

额外封装一个系统调用 `sys_putchar`，但发现还是无法解析换行，gdb跟踪看输入换行时发生了什么：

```
Continuing.

Breakpoint 3, main () at test/shell.c:54
54              sys_putchar(tmp);
(gdb) p/c tmp
$2 = 13 '\r'
(gdb)
```

意识到unix中换行会被解析成 `\r` 而不是 `\n` ，修改后可以输出换行，同时注意到printf似乎无法输出换行 `\n` ，一样换成 `\r` 试试：

```
------------------- COMMAND -------------------
> root@UCAS_OS: hi
Error: Unknown command hi> root@UCAS_OS:
```

查看printf的源码，对于 `%s` ，其是用 `strlen` 获取参数的长度后搬运到一个小buffer里，在shell.c中打印出来解析的args的长度：

```
------------------- COMMAND -------------------
> root@UCAS_OS: exec hi
5
3
Fail to find the task! Please try again!
```

不知道因为什么原因使得其都比实际长度大1，gdb跟踪一下：

```
Breakpoint 2, strlen (
    src=0x52001c02 <args+10> "hi")
    at tiny_libc/string.c:26
26          int i = 0;
(gdb) s
27          while (src[i++] != '\0')
 {
(gdb) p src[0]
$1 = 104 'h'
(gdb) p i
$2 = 0
(gdb) s
30          return i;
(gdb) p i
$3 = 3
(gdb) p src[2]
$4 = 0 '\000'
(gdb) p src[3]
$5 = 0 '\000'
(gdb) p src[1]
$6 = 105 'i'
(gdb)
```

救命……为啥它不停下？？？查看 `strlen` 源码：

```
3
4   int strlen(const char *src)
5   {
6       int i = 0;
7       while (src[i++] != '\0') {
8           ;
9       }
0       return i;
1   }
2
```

好家伙，i++应该放在循环里面？修改后就可以舒适地使用`%s`输出了：

```
------------------- COMMAND -------------------
> root@UCAS_OS: hi
Error: Unknown command hi
> root@UCAS_OS:
```

（2）补全内核相关函数

要完成进程的管理，需要给PCB再添加一个状态`UNUSED`表示当前PCB未被使用。同时为了更好地复用创建PCB stack的函数，引入参数argv和argc（在do_exec和do_thread_create都可以更好地使用之）。

注意到先前的loader打印会覆盖当前行：

```
------------------- COMMAND -------------------
Fail to find the task! Please try again!
Error: exec failed!
```

先前使用的是bios_putchar逐个输出，修改为`screen_write`：

```
1  char *output_str = "Fail to find the task! Please try again!\n";
2  screen_write(output_str);
```

可正常输出：

```
------------------- COMMAND -------------------
> root@UCAS_OS: exec hey
Fail to find the task! Please try again!
Error: exec failed!
```

而若要能支持`&`判断是否等待该函数执行完毕，要实现`waitpid`，同时要记得初始化pcb的wait_list，否则会在访问其前驱后继时出问题。还有值得注意的是在讲义中规定了函数的原型，不能额外引入参数告诉内核此时是否需要等待进程，故只能在创建进程返回后再进行系统调用进行wait。

> 还有记得实时调整定义的task_info数组的大小以及task_name字符数组的长度，避免溢出而导致task找不到。

同时注意到此时还发生了很吊诡的事情：

```
> [TASK] I want to wait task (pid=4) to exit.
> [TASK] I am task with pid 3, I have acquired two mutex locks. (999
> [TASK] I want to acquire mutex lock1 (handle=0).
```

`ready_to_exit`已经结束了，但task还是没有打印出已经获得锁。而用ps查看进程状态：

```
------------------- COMMAND -------------------
> root@UCAS_OS: ps
[Process table]:
[0] PID : 1  STATUS : RUNNING
[1] PID : 2  STATUS : BLOCKED
[2] PID : 3  STATUS : EXIST
[3] PID : 4  STATUS : RUNNING
> root@UCAS_OS:
```

而我的do_exit实现如下：

```
void do_exit(void){
  current_running->status = TASK_EXITED;
  // 清空被阻塞的进程
  free_block_list(&(current_running->wait_list));
  do_scheduler();
}
```

gdb跟踪一下，实际上在跑的是shell进程。再查看锁相关实现：

```
void do_mutex_lock_acquire(int mlock_idx)
{
  /* TODO: [p2-task2] acquire mutex lock */
  // 成功获得锁
  if(spin_lock_try_acquire(&mlocks[mlock_idx].lock))
      return;
  // 获取锁失败
  do_block(&current_running->list,
&mlocks[mlock_idx].block_queue);
  pcb_t *prior_running = current_running;
  current_running  = get_pcb_from_node(seek_ready_node());
  current_running->status = TASK_RUNNING;
  switch_to(prior_running, current_running);
}
```

实际上在`do_block`时已经进行了调度，后续无需再做。修改后功能正常：

```
> [TASK] Task (pid=4) has exited.
> [TASK] I am task with pid 3, I have acquired two mutex locks. (9999
> [TASK] I have acquired mutex lock2 (handle=1).
```

```
------------------- COMMAND -------------------
> root@UCAS_OS: exec  waitpid &
Info: excute waitpid successfully, pid = 2
> root@UCAS_OS: ps
[Process table]:
[0] PID : 1  STATUS : RUNNING
[1] PID : 2  STATUS : EXIST
[2] PID : 3  STATUS : EXIST
[3] PID : 4  STATUS : EXIST
> root@UCAS_OS:
```

（3）调用内核函数

```
------------------- COMMAND -------------------
> root@UCAS_OS: exec wait_locks 0 1 2
Fail to find the task! Please try again!
Error: exec failed!
> root@UCAS_OS: exec wait_locks
Info: excute wait_locks successfully, pid = 2
> root@UCAS_OS:
```

发现只要带了参数就报错？gdb跟踪一下：

```
Breakpoint 1, load_task_img (
  taskname=0x52001bb2 "wait_locks1")
  at ./kernel/loader/loader.c:15
15          for(i=0;i<TASK_MAXNUM;i++){
```

发现解析的时候2，3两个参数连在了一起，gdb看一看：

```
$17 = 0x0
<p args[0]
$18 = "exec\000\00
0\000\000\000"
(gdb) p args[1]
$19 = "wait_locks"
(gdb) p args[2]
$20 = "2\000\000\0
00\000\000\000\000
\000"
<p args[2]
```

意识到存参数的数组太短了……直接连在一起了。修改长度后总算起了个进程：

```
> [TASK] I have acquired mutex lock2 (handle=0).  Assertion failed at main in te
 t/test_project3/S_CORE/wait_locks.c:8
```

```
------------------- COMMAND -------------------
> root@UCAS_OS: exec wait_locks 1 2 3
Info: excute wait_locks successfully, pid = 2
> root@UCAS_OS:
```

仔细查看测试用例，访问都是从下标1开始，得知其argv把当前进程名也传了进来（而我之前把之排除了）。修改后出bug：

```
------------------- COMMAND -------------------
> root@UCAS_OS: exec wait_locks 1 2 3
exception code: 5 , Load access fault , epc 520b0828 , ra 520b0a30
### ERROR ### Please RESET the board ###
```

gdb跟踪一下：

```
    argv=0x52001bc4)
    at test/test_project3/wait_locks.c:8
8           assert(argc >= 4);
(gdb) n
9           int print_location = atoi(argv[1]);
(gdb) s
atoi (
    str=0x736b "")
    at tiny_libc/atoi.c:72
72          return (int)atol(str);
(gdb) s
atol (
    str=0x736b "")
    at tiny_libc/atoi.c:9
9           long ret = 0;
(gdb) p str
$6 = 0x736b ""
(gdb) x str
0x736b: 0 '\000'
(gdb) s
10          int negative = 0;
(gdb)
11          int base = 10;
(gdb)
15          if (NULL == str) {
(gdb)
20          while (isspace(*str)) {
(gdb)
```

这是`char**`和`char*[]`和`char[][]`不完全一样导致的（死去的C语言知识开始攻击）：

> (6条消息) char **和char* []区别，char *和char []区别夏风之羽的博客-CSDN博客char*[]

改用char*[]存储起始地址：

但是打印出了抢锁信息后很快又抛出了assert：

```
> [TASK] I have acquired mutex lock2 (handle=3).  Assertion failed at main in te
 t/test_project3/wait_locks.c:8

------------------- COMMAND -------------------
> root@UCAS_OS: exec wait_locks 1 2 3
Info: excute wait_locks successfully, pid = 2
> root@UCAS_OS:
```

gdb跟踪发现第一把进去的时候参数总算传对了：

```
Breakpoint 1 at 0x520b007a: file test/test_proj
ect3/wait_locks.c, line 8.
(gdb) c
Continuing.

Breakpoint 1, main (argc=4,
    argv=0x52001c78)
    at test/test_project3/wait_locks.c:8
8           assert(argc >= 4);
(gdb) n
9           int print_location = atoi(argv[1]);
(gdb)
10          int handle1 = atoi(argv[2]);
(gdb)
11          int handle2 = atoi(argv[3]);
```

但是此后执行流如下：

```
        sys_mutex_release(handle2);
)
        return 0;
)
    }
)
rt ()
at arch/riscv/crt0/crt0.S:24
        ld ra, (sp)
)
        ld s0, 8(sp)
)
        addi sp, sp, 16
)
rt ()
at arch/riscv/crt0/crt0.S:27
        jr ra
)
rt ()
at arch/riscv/crt0/crt0.S:7
        addi sp, sp, -16
)
        sd s0, 8(sp)            // store frame pointer in stack
)
        sd ra, (sp)             // store return address in stack
)
        addi s0, sp, 16          // frame pointer points to original


)
        la x5, __bss_start         // bss起始地址
```

```
)
        la x6, __BSS_END__          // bss终止地址
)
        sw  x0, (x5)           // store 0 to x5
)
        add x5, x5, 0x04       // point to next word
)
        ble x5, x6, do_clear
)
        call main
)

kpoint 1, main (argc=0,
argv=0x0)
at test/test_project3/wait_locks.c:8
        assert(argc >= 4);
(gdb)
```

意识到先前的crt0中是又让其回到了kernel：

```
/* TODO: [p1-task3] finish task and return to the kernel */
/* NOTE: You need to replace this with new mechanism in p3-task2! */
ld ra, (sp)
ld s0, 8(sp)
addi sp, sp, 16
jr ra
```

现在应该通过系统调用使得进程exit。

TASK 2

1. 思路说明

(1) 屏障

- 初始设置一个目标值 `goal`
- 当其等待队列中的进程个数 `wait_num` 达到目标值后，就释放 `wait_list` 所有进程，以此达到"同步"的效果。

```c
typedef struct barrierq
{
    // TODO [P3-TASK2 barrier]
    int goal;
    int wait_num;
    list_head wait_list;
    int key;
    use_status_t usage; // 记录是否被使用
} barrier_t;
```

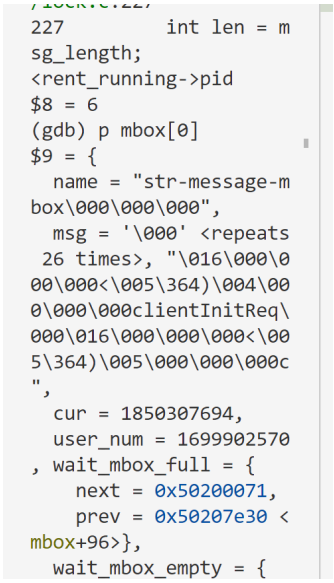
（2）条件变量

2. Debug 记录

condition和barrier未遇到十分难搞的bug，此部分主要记录mailbox部分遇到的bug。

（1）邮箱≈读写指针皆单调递增的循环数组

跟踪邮箱初始化的时候发现如下问题，其`user_num`和`cur`域皆不正常：



```
227         int len = m
sg_length;
<rent_running->pid
$8 = 6
(gdb) p mbox[0]
$9 = {
  name = "str-message-m
box\000\000\000",
  msg = '\000' <repeats
 26 times>, "\016\000\0
00\000<\005\364)\004\00
0\000\000clientInitReq\
000\016\000\000\000<\00
5\364)\005\000\000\000c
",
  cur = 1850307694,
  user_num = 1699902570
, wait_mbox_full = {
    next = 0x50200071,
    prev = 0x50207e30 <
mbox+96>},
  wait_mbox_empty = {
```

gdb跟踪发现是在一次信息copy时覆盖了后面的域：

```
ad 1 hit Breakpoint 2, do_mbox_send (
mbox_idx=0,
msg=0x52504f04,
msg_length=26)
```

```
at ./kernel/locking/lock.c:227
        int len = msg_length;
) p mbox[0]
= {
me = "str-message-mbox\000\000\000",
g = '\000' <repeats 26 times>,
6\000\000\000<\005\364)\004\000\000\000clientInitReq", '\000'
eats 13 times>, cur = 26,
er_num = 3,
it_mbox_full = {
next = 0x50207e30 <mbox+96>,
prev = 0x50207e30 <mbox+96>},
it_mbox_empty = {
next = 0x50207e40 <mbox+112>,
prev = 0x50207e40 <mbox+112>}}
) n
        while((tmp_cur= mbox[mbox_idx].cur + len)>MAX_MBOX_LENGTH){
) n
        mbox[mbox_idx].cur = tmp_cur;
) n
        memcpy(mbox[mbox_idx].msg + mbox[mbox_idx].cur, msg, len);
) p mbox[0]
= {
me = "str-message-mbox\000\000\000",
g = '\000' <repeats 26 times>,
6\000\000\000<\005\364)\004\000\000\000clientInitReq", '\000'
eats 13 times>, cur = 52,
er_num = 3,
it_mbox_full = {
next = 0x50207e30 <mbox+96>,
prev = 0x50207e30 <mbox+96>},
it_mbox_empty = {
next = 0x50207e40 <mbox+112>,
prev = 0x50207e40 <mbox+112>}}
) n
        free_block_list(&mbox[mbox_idx].wait_mbox_empty);   // 唤醒
因邮箱空而阻塞的进程
) p mbox[0]
= {
me = "str-message-mbox\000\000\000",
g = '\000' <repeats 26 times>,
6\000\000\000<\005\364)\004\000\000\000clientInitReq\000\016\000\000
<\005\364)\005\000\000\000c",
r = 1850307694,
er_num = 1699902569, wait_mbox_full = {
next = 0x50200071,
prev = 0x50207e30 <mbox+96>},
it_mbox_empty = {
next = 0x50207e40 <mbox+112>,
```

```
prev = 0x50207e40 <mbox+112>}}
)
```

意识到如下两句应该互换顺序：

```
        free_block_list(&mbox[mbox_idx].wait_mbox_empty);   // 唤醒所有因邮箱空而阻塞
        do_block(&current_running->list, &mbox[mbox_idx].wait_mbox_full);
    }
    // 进行数据拷贝
    mbox[mbox_idx].cur = tmp_cur;
    memcpy(mbox[mbox_idx].msg + mbox[mbox_idx].cur, msg, len);
    free_block_list(&mbox[mbox_idx].wait_mbox_empty);   // 唤醒所有因邮箱空而阻塞的进
    return 0;
}
```

改后仍旧跑不出预期效果，gdb跟踪：

```
ecksum = adler32(msgBuffer,
 header.length);
(gdb) p msgBuffer
$73 = "ientInitReq\000q", '
\000' <repeats 50 times>
(gdb)
```

发现 server收到的client init信息不对。猜测是memcpy时起始没搞对：

```
int do_mbox_send(int mbox_idx, void * msg, int msg_length){
    int len = msg_length;
    int tmp_cur;
    // 邮箱已满，阻塞
    while((tmp_cur= mbox[mbox_idx].cur + len)>MAX_MBOX_LENGTH){
        // 拷贝部分数据
        len = MAX_MBOX_LENGTH - mbox[mbox_idx].cur;
        memcpy(mbox[mbox_idx].msg + mbox[mbox_idx].cur, msg, len);
        mbox[mbox_idx].cur = MAX_MBOX_LENGTH;
        msg += len;
        len = tmp_cur - mbox[mbox_idx].cur;
        free_block_list(&mbox[mbox_idx].wait_mbox_empty);   // 唤醒所有因邮箱空而阻塞的进程
        do_block(&current_running->list, &mbox[mbox_idx].wait_mbox_full);
    }
    // 进行数据拷贝
    memcpy(mbox[mbox_idx].msg + mbox[mbox_idx].cur, msg, len);
    mbox[mbox_idx].cur = tmp_cur;
    free_block_list(&mbox[mbox_idx].wait_mbox_empty);   // 唤醒所有因邮箱空而阻塞的进程
    return 0;
}
int do_mbox_recv(int mbox_idx, void * msg, int msg_length){
    int len = msg_length;
    int tmp_cur;
    // 邮箱读空，阻塞
    while((tmp_cur = mbox[mbox_idx].cur - len)<0){
        len = mbox[mbox_idx].cur;
```

```
        mbox[mbox_idx].cur = 0;
        memcpy(msg, mbox[mbox_idx].msg + mbox[mbox_idx].cur, len);
        msg += len;
        len = mbox[mbox_idx].cur - tmp_cur;
        free_block_list(&mbox[mbox_idx].wait_mbox_full);   // 唤醒所有因邮箱满而阻塞的进程
        do_block(&current_running->list, &mbox[mbox_idx].wait_mbox_empty);
    }
    // 进行数据拷贝
    mbox[mbox_idx].cur = tmp_cur;
    memcpy(msg, mbox[mbox_idx].msg + mbox[mbox_idx].cur, len);
    free_block_list(&mbox[mbox_idx].wait_mbox_full);   // 唤醒所有因邮箱满而阻塞的进程
    return 0;
}
```

发现server一次recv操作中请求len异常：

```
Breakpoint 6, do_mbox_recv (mbox_idx=0,
    msg=0x52502f5a,
    msg_length=1953391977)
    at ./kernel/locking/lock.c:247
247         int len = msg_length;
(gdb) c
Continuing.
```

查看源码：

```
        blockedCount += sys_mbox_recv(handle_mq, &header, sizeof(MsgHeader_t));
        blockedCount += sys_mbox_recv(handle_mq, msgBuffer, header.length);
```

猜测是header信息有误。gdb跟踪：

```
47              blockedCount += sys_mbox_recv(handle_mq, msgBuffer, header.length);
(gdb) p header
$92 = {length = 14,
  checksum = 703857980, sender = 6}
```

第一次读出正确，而第二次开始读出有误。意识到应该分别维护一个读写cur，保证二者单调递增，否则会导致后写入的信息被先读出，顺序有误。同时意识到此时邮箱相当于一个循环数组，不能简单地使用原本的 `memcpy` ，自定义一个循环数组的memcpy：

```
void myMemcpy(char *dest, char *src, int vcur, int len, int arr_size, int mode){
    int pcur;   //cur的实际值
    // 写操作中dest是循环数组
```

```
    if(mode==MODE_W){
        for(int i=0; i<len; i++){
            pcur = i % arr_size;
            dest[pcur] = src[i];
        }
    }
    // 读操作中src是循环数组
    else{
        for(int i=0; i<len; i++){
            pcur = i % arr_size;
            dest[i] = src[pcur];
        }
    }
}
```

（2）邮件 = 包

起初我纠结于一次发不完是否应该分批发：

- 如果整个不写，要是出现信息比邮箱size还大的话，就会整个卡死
- 如果只写一半，就有可能导致下一次开始写的不是原本写了一半的人，导致信息有误。此时可能需要额外增加一把锁，使得当前写进程持有锁地被阻塞。

这种疑问是由于我最早未理解基于信箱的传输是包传输，故一次发不完就不可传输。

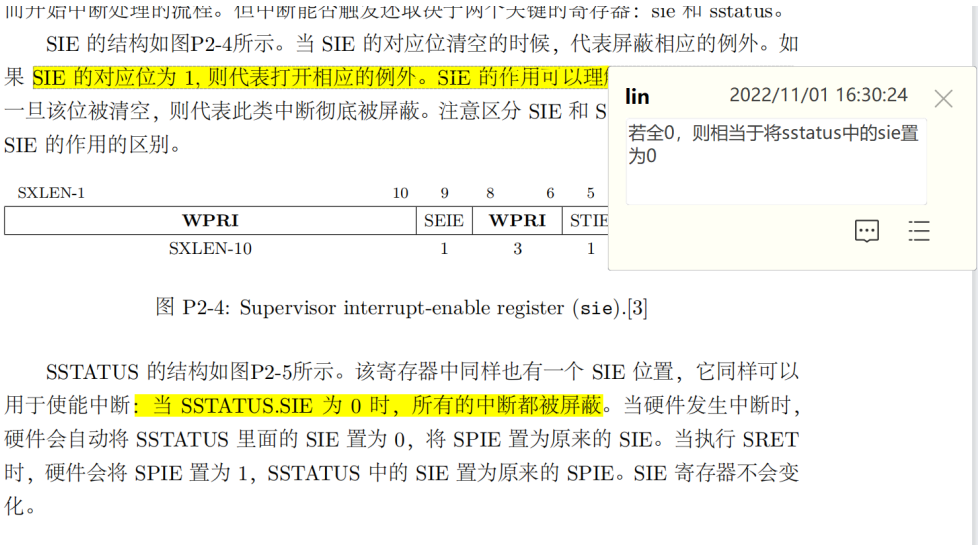
此外，希望后续讲义中就**注明recv和send返回值的含义**，后才意识到对于发包or收包失败的情况再阻塞后重发/收的情况需要返回被阻塞的次数。

TASK 3

1. 思路说明

（1）开关中断如何实现?

bootloader：关全局中断？查看P2的讲义得知需要修改sie寄存器：


而开始中断处理的流程。但中断能否触发还取决于两个关键的寄存器：sie 和 sstatus。

SIE 的结构如图P2-4所示。当 SIE 的对应位清空的时候，代表屏蔽相应的例外。如果 SIE 的对应位为 1, 则代表打开相应的例外。SIE 的作用可以理
一旦该位被清空，则代表此类中断彻底被屏蔽。注意区分 SIE 和 S
SIE 的作用的区别。

SXLEN-1	10	9	8	6	5
WPRI		SEIE	WPRI		STIE
SXLEN-10		1	3		1

图 P2-4: Supervisor interrupt-enable register (sie).[3]

SSTATUS 的结构如图P2-5所示。该寄存器中同样也有一个 SIE 位置，它同样可以用于使能中断：当 SSTATUS.SIE 为 0 时，所有的中断都被屏蔽。当硬件发生中断时，硬件会自动将 SSTATUS 里面的 SIE 置为 0，将 SPIE 置为原来的 SIE。当执行 SRET 时，硬件会将 SPIE 置为 1，SSTATUS 中的 SIE 置为原来的 SPIE。SIE 寄存器不会变化。

lin 2022/11/01 16:30:24
若全0，则相当于将sstatus中的sie置为0

entry.S中已经有现成的函数 `disable_preempt` 。

（2）主从核初始化时任务的划分

如何在main中划分任务？→根据ID，主核来做共享变量的初始化→如何获得当前核的信息？

- 思路1：在boot_loader中将之当作参数传入（我们已经知道在刚进入bootloader时a0寄存器存储的便是cpuid，只需将之再传给kernel）；
- 思路2：更本质的，我们直接从 `mhartid` 读出（xv6是直接将该信息存在了CPU结构体里），我们可以封装一个函数，使用汇编代码从该寄存器中读出当前核的id，这后续便于我们便捷地获取cpuid的信息。

正要开写的时候惊奇地发现框架已经提供了该函数 `get_current_cpu_id` ，直接调用即可。

再划分主从核该各自干什么：我们先参考一下xv6的代码：

对于xv6，主核和从核的函数入口是不一致的，对于主核：

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
```

```
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}

```

我们也可以参考这种执行的flow，对主核而言，先进行必要资源的初始化再唤醒从核，之后再起首个用户进程。

对于从核，其入口是`mpenter`：

```
// Other CPUs jump here from entryother.S.
static void
mpenter(void)
{
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}
```

可见每个核都会调用`segini`设置每个cpu自己的GDT全局描述符表，并且使用`lapicinit`设置时间中断，最后也调用`mpmain`打印一些身份信息，并进行调度。（这么看来在我们的设计中从核应该不会干什么主核没干的事）。

我们的设计中要让从核做什么呢?

- 抢个内核锁;
- 设立其中断入口

（3）内核态区分主从核

维护cpuid的一个全局变量，在抢到内核锁后更新该值，则在整个内核中该值都是不变的√，考虑构造cpu结构体，三核及以上时更好扩展。

刚进入内核：有两个case

- 从用户态刚进入内核
  - ecall进入内核态
  - 进入`exception_handler_entry`→在此处调用抢内核锁!

> `ret_from_exception`回到用户态，相应在此释放内核锁

    - 此后进入`interrupt_helper`，在此处更新cpuid的值，并更新current_running的信息。
- 从bootloader刚刚进入内核

只需要在上述该两种情况做id信息的更新。

2. Debug记录

（1）主核收到软件中断

跑起来后报错在奇怪的地方：

```
U-Boot 2019.07UCAS_OS DASICS v2.0.4 (Aug 29 2022 - 10:34:31 +0000)

CPU:   rv64imafdcnsu
Model: riscv-virtio,qemu
DRAM:  256 MiB
In:    uart@60000000
Out:   uart@60000000
Err:   uart@60000000
Net:   No ethernet found.
>ero  : 0000000000000000  ra   : 0000000050201134  sp   : 000000005050
> gp   : 00000000501fede0  tp   : 0000000000000000  t0   : 00000000002
  t1   : 0000000000000002  t2   : 0000000000000002 s0/fp : 00000000500
  s1   : 0000000000010020  a0   : 0000000000000000  a1   : 00000000500
  a2   : 0000000000000000  a3   : 0000000000000073  a4   : 00000000000
  a5   : 0000000000000001  a6   : 0000000000000000  a7   : 00000000000
  s2   : 0000000000000001  s3   : 0000000000000000  s4   : 00000000000
  s5   : 0000000000000000  s6   : 0000000000000000  s7   : 00000000000
  s8   : 0000000000000000  s9   : 0000000000000000  s10  : 00000000000
  s11  : 0000000000000000  t3   : 0000000000000000  t4   : 00000000000
  t5   : 0000000000000000  t6   : 0000000000000000
 sstatus: 0x120 sbadaddr: 0x0 scause: )
 sepc: 0x502011ce
QEMU: Terminated:atxhandle_other in ./kernel/irq/irq.c:77
```

gdb跟踪：

```
Thread 2 hit Breakpoint 1, main (
    app_info_loc=1,
    app_info_size=0,
    seq_end_loc=0,
    seq_start_loc=115)
    at ./init/main.c:173
173         uint64_t cpu_id = get_current_cpu_i
d();
(gdb) n
Remote connection closed
```

在从核尝试获取id时就会报错，gdb跟入函数内部：

```
)
    at ./arch/riscv/kernel/smp.S:5
5         csrr a0, CSR_MHARTID
<p $sstatus
$2 = 288
(gdb) /xp $ssta>>
Undefined command: "".  Try "help".
(gdb) p/x $ssta>
$3 = 0x120
<p/b sstatus
Size letters are meaningless in "print" command
.
<p $mhartid
Could not fetch register "mhartid"; remote fail
ure reply 'E14'
(gdb)
```

意识到此时处于S态，gdb的时候访问M态寄存器会报错，重开后又在后面才报错：

```
(gdb) s
main (app_info_loc=1, app_info_size=0,
    seq_end_loc=0, seq_start_loc=115)
    at ./init/main.c:175
175         if(cpu_id==0){
(gdb) s
224             lock_kernel();
(gdb) n
225             s_current_running->status
 = TASK_RUNNING;
(gdb) n
```

而且奇怪的是上述出错位置没有打印scause，gdb跟踪：

```
中断
<nt_running
$16 = (pcb_t * volatile) 0x50205a58 <m_pid0_p
cb>
(gdb) n
20              irq_table[scause & ~SCAUSE_IR
Q_MASK](regs, stval, scause);
(gdb) p $scause
$17 = -9223372036854775807
(gdb) p/x $scause
$18 = 0x8000000000000001
(gdb) n
Remote connection closed
(gdb)
```

查看头文件定义，发现主核收到了软中断：

```
#define IRQ_U_SOFT      0
#define IRQ_S_SOFT      1
#define IRQ_M_SOFT      3
#define IRQ_U_TIMER     4
```

查看FAQ得知可以通过将SIP置零达到屏蔽的效果，gdb跟踪：

```
Thread 1 hit Breakpoint 1, wakeup_other_hart () at ./kernel/s
mp/smp.c:16
16          send_ipi(NULL);
(gdb) p $sip
$1 = 0
(gdb) n
17      }
(gdb) p $sip
$2 = 2
(gdb)
```

在 `send_ipi` 之后主核的SIP会变为非零值表示有未处理的中断，故应该在wakeup other之后再将SIP清零，此后再打开全局中断。也就是说我们需要将 `init_exception` 的工作，更准确地说，是 `setup_exception` 的工作放到主核唤醒从核之后。考虑将 `setup_expception` 从 `init` 中拆出，单独进行，在唤醒从核后将 `SIP` 置为0。

（2）bss段清理：主核的task

Three types of interrupts are defined: software interrupts, timer interrupts, and external interrupts. A supervisor-level software interrupt is triggered on the current hart by writing 1 to its supervisor software interrupt-pending (SSIP) bit in the `sip` register. A pending supervisor-level software interrupt can be cleared by writing 0 to the SSIP bit in `sip`. Supervisor-level software interrupts are disabled when the SSIE bit in the `sie` register is clear.

从核抢锁时会报错?

```
(gdb)
spin_lock_acquire (
    lock=0x502091e8 <kloc
k>)
    at ./kernel/locking/l
ock.c:35
35          while(atomic_
swap(LOCKED, &lock->statu
s)==LOCKED);
(gdb)
Remote connection closed
(gdb)
```

将断点打在抢锁入口处：

```
22          spin_lock_acquire(&klock);
(gdb) p klock
$2 = {
  status = UNLOCKED}
(gdb) s
spin_lock_acquire (
    lock=0x502091e8 <klock>)
    at ./kernel/locking/lock.c:35
35          while(atomic_swap(LOCKED, &lock->status
)==LOCKED);
(gdb)
atomic_swap (val=1,
    mem_addr=1344311784) at ./arch/riscv/include/at
omic.h:13
13              : "=r"(ret), "+A" (*(uint32_t*)mem_
addr)
(gdb)
11          __asm__ __volatile__ (
(gdb)
16          return ret;
(gdb)
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 3, lock_kernel ()
    at ./kernel/smp/smp.c:22
22          spin_lock_acquire(&klock);
(gdb) p klock
$3 = {status = LOCKED}
(gdb)
```

发现主核在唤醒从核后立刻抢到锁，导致从核抢不到锁。但难道不应该是被阻塞吗？为什么一整个就报错呢？

gdb跟踪：

```
    at ./init/main.c:235
235     t_ticks()+TIMER_INTERVAL);
<rent_running
$6 = (pcb_t * volatile) 0x0
<rent_running
$7 = (pcb_t * volatile) 0x0
<rent_running
$8 = (pcb_t * volatile) 0x0
(gdb) p s_pid0_pcb
```

主从核的current_running未初始化，但主核已在init_pcb中进行？

```
    // 创建首个用户进程shell（主从核共用？）
    bios_putchar(do_exec("shell", 0, NULL));    //创建不成功则exit
    /* TODO: [p2-task1] remember to initialize 'current_running' */
    m_current_running = &m_pid0_pcb;
    s_current_running = &s_pid0_pcb;
    current_running = m_current_running;
}

```

gdb查看发现唤醒从核之后所有信息似乎都置空了：

```

Thread 1 hit Breakpoint 3, lock_kernel ()
    at ./kernel/smp/smp.c:22
22          spin_lock_acquire(&klock);
(gdb) n
23      }
(gdb)
main (
    app_info_loc=118108, app_info_size=448,
    seq_end_loc=118605, seq_start_loc=118556)
    at ./init/main.c:181
181             init_jmptab();
(gdb)
184             init_task_info(app_info_loc, app_info_size);
(gdb)
187             init_pcb(seq_start_loc, seq_end_loc);
(gdb)
190             time_base = bios_read_fdt(TIMEBASE);
<rent_running
$14 = (pcb_t * volatile) 0x50205aa0 <m_pid0_pcb>
<rent_running
$15 = (pcb_t * volatile) 0x50205af8 <s_pid0_pcb>
(gdb) c
Continuing.

```

```
Thread 1 hit Breakpoint 3, lock_kernel ()
    at ./kernel/smp/smp.c:22
22          spin_lock_acquire(&klock);
<rent_running
$16 = (pcb_t * volatile) 0x50205af8 <s_pid0_pcb>
(gdb) i threads
  Id   Target Id                    Frame
* 1    Thread 1.1 (CPU#0 [running]) lock_kernel
    ()
    at ./kernel/smp/smp.c:22
  2    Thread 1.2 (CPU#1 [running]) 0x00000000500011aa in ?? ()
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 3, lock_kernel ()
    at ./kernel/smp/smp.c:22
22          spin_lock_acquire(&klock);
<rent_running
$17 = (pcb_t * volatile) 0x0
(gdb) thread 1
[Switching to thread 1 (Thread 1.1)]
#0  mini_itoa (
    value=0, radix=10,
    uppercase=0,
    unsig=0,
    buffer=0x504ffda8 "\001", zero_pad=0)
    at ./libs/printk.c:74
74              int digit = value % radix;
<rent_running
$18 = (pcb_t * volatile) 0x0
(gdb) p syscall
$19 = {
  0x0 <repeats 64 times>}
(gdb)
```

仔细阅读讲义，意识到二者共用代码段和数据段，从核刚进入时不应该再做一次bss段的清空。

（3）内核锁上锁的位置

```
[U-BOOT] ERROR: truly_illegal_insn
exception code: 2 , Illegal instruction , epc 5020112a , ra 5020112a

### ERROR ### Please RESET the board ###
QEMU: Terminated
```

去掉内核锁跑单核还是能过，保留调用，将上锁解锁的过程直接变为return还是会出错，意识到可能是其调用位置不对：

```
ENTRY(ret_from_exception)
  /* TODO: [p2-task3] restore context via provided macro and return to sepc */
  /* HINT: remember to check your sp, does it point to the right address? */
  RESTORE_CONTEXT
  // 释放内核锁
  call unlock_kernel
  sret
ENDPROC(ret_from_exception)

ENTRY(exception_handler_entry)
  // 上内核锁
  call lock_kernel
  /* TODO: [p2-task3] save context via the provided macro */
  SAVE_CONTEXT
```

意识到在恢复上下文前应该打开内核锁，否则会在用户栈上跑，同样的应该在保存上下文之后才上锁。

TASK 4

这一部分的思路比较直观，并无遇到太难de的bug，但是我未做出预期的加速效果，可能与我采取的处理思路有关：

- 未为主从核单独设置调度队列，
- 采取RR调度，每次检查mask是否匹配

因此被绑核的进程可能反而运行得更慢（如，核2检测到一个ready的进程被绑定给核1，其会跳过之寻找下一个进程，同时核1在绕了一圈后再pick该进程时估计cache也被替换得差不多了……，这估计还和cache的容量和算法有关），在理论课上的绑核实验我也是获得了和理论相反的效果。

最终运行时，taskset似乎只能确保其在该核运行，而无明显加速作用：

```
 [3] integer test (86/150) access: 5 auipc: 520f010c
 [4] integer test (102/150) access: 3 auipc: 520f010c
 [5] integer test (89/150) access: 3 auipc: 520f010c
 [6] integer test (87/150) access: 3 auipc: 520f010c
 [7] integer test (93/150) access: 2 auipc: 520f010c









------------------- COMMAND -------------------
> root@UCAS_OS: ps
[Process table]:
[0] PID : 1  STATUS : RUNNING mask 0x3 Running on core 1
[1] PID : 2  STATUS : BLOCKED mask 0x3
[2] PID : 3  STATUS : READY mask 0x1
[3] PID : 4  STATUS : RUNNING mask 0x3 Running on core 2
[4] PID : 5  STATUS : READY mask 0x3
[5] PID : 6  STATUS : READY mask 0x3
[6] PID : 7  STATUS : READY mask 0x3
> root@UCAS_OS: ps
```