Project 3 Interactive OS and Process Management 设计文档

中国科学院大学
王嵩岳
2020 年 11 月 16 日

1. Shell 设计

在管理 shell 的命令时，我借鉴了南京大学 ProjectN: ICS-PA (Introduction to Computer System, Programming Assignment)中 nemu 项目内嵌的一个简易调试器。我将实现命令的所有子属性全部打包到一个结构数组中。

子属性	描述	类型
name	命令头，用于解析索引	Char *
description	功能描述	Char *
usage	用法描述	Char *
func	对应函数	函数指针
arg_num	最少命令参数个数	int

```
typedef void (*function)(void *arg0, void *arg1, void *arg2);
static struct {
    char *name;
    char *description;
    char *usage;
    function func;
    int arg_num;
} cmd_table [] = {
    {"help", "Display informations about all supported commands", "help [NO ARGS]", &shell_help, 0},
    {"exec", "Execute task n in testbench (mode: 0 parent, 1 auto)", "exec [n] [mode]", &shell_exec, 2},
    {"taskset", "Set process or task(default) only runs on a certain core", "taskset [task id] [mask]\n
    {"kill", "Kill process n (pid)", "kill [n]",&shell_kill, 1},
    {"clear", "Clear the screen", "clear", &shell_clear, 0},
    {"ps", "Show all process", "ps", &shell_ps, 0}
};
```

这样，解析命令之后，直接根据数组下标索引，调用函数指针域，即可调用函数，极大地简化了判断逻辑。

```
cmd_table[man_id].func(arg[1], arg[2], arg[3]);
```

其中，arg 通过自编的 strtok 函数进行解析，实现在用户态的 C 库中。

```
while ((parse = strtok(arg[i++], parse, ' ')) != NULL);
```

2. kill 和 wait 内核实现的设计

（1） Kill 时锁的释放

我的做法是，当一个进程拿到锁后，在 PCB 的 lock 域里储存这个锁（二元信号量）的 id。当释放锁后，将标记的 id 记为-1。下图是获取锁的内核函数，红色部分为记录锁 id 的

过程，蓝色部分为清理标记的过程。

```
int kernel_binsemop(int binsem_id, int op)
{
    current_running = (get_current_cpu_id() == 0) ? &current_running_core0 : &current_running_core1;
    binsem_t *bin = sem_list[binsem_id];
    if (op == BINSEM_OP_LOCK){
        if(bin->status == BIN_LOCKED)
            do_block(*current_running, &bin->block_queue);
        else{
            current_running = (get_current_cpu_id() == 0) ? &current_running_core0 : &current_running_core1;
            bin->status = BIN_LOCKED;
            ((pcb_t *)(*current_running))->lock[((pcb_t *)(*current_running))->lock_num ++] = binsem_id;
        }
    } else if (op == BINSEM_OP_UNLOCK){
        if(list_empty(&bin->block_queue)){
            current_running = (get_current_cpu_id() == 0) ? &current_running_core0 : &current_running_core1;
            for(int i = 0; i < ((pcb_t *)(*current_running))->lock_num; i++)
                if (((pcb_t *)(*current_running))->lock[i] == binsem_id)
                {
                    ((pcb_t *)(*current_running))->lock[i] = -1;
                    break;
                }
            bin->status = BIN_UNLOCKED;
```

当 kill 的时候，在方法中将所有持有的 lock 释放。

```
// Release all locks it handled
  for (int i = 0; i < pcb[pid - 1].lock_num; i++)
       if (pcb[pid - 1].lock[i] > 0)
          kernel_binsemop(pcb[pid - 1].lock[i],BINSEM_OP_UNLOCK);
```

（2）Wait 的处理

我在 PCB 里新增了 parent 域，用于记录其父进程（或者是等待她执行完的进程）。

```
int do_waitpid(pid_t pid)
{
    current_running = (get_current_cpu_id() == 0) ? &current_running_core0 : &current_running_core1;
    if (pcb[pid - 1].pid != 0){
        if (pcb[pid - 1].status != TASK_EXITED){
            pcb[pid - 1].parent = ((pcb_t *)(*current_running))->pid;
            do_block(((pcb_t *)(*current_running)), &wait_queue);
        }
    }
    return pid;
}
```

当父进程系统调用 waitpid 时，若等待的进程还未结束，则把等待的进程的 parent 域记录为自己的 pid，然后被阻塞到全局等待队列 wait_queue。

每当一个进程退出/被杀死时，如果其 mode 为检查 zombie，那么会调用 zombie_check 方法，这个方法就是将其 pcb 中的 parent 的 pid 对应的 pcb 从阻塞队列 unblock 出来。这样就实现了退出后，父进程再次被调度并返回。

```
void zombie_check(pid_t pid)
{
    if (pid == 0)
        return ;
    else
    {
        do_unblock(&pcb[pid - 1].list);
    }
}
```

3. 同步原语设计

这里我将条件变量和屏障都实现在了内核态，可以支持进程间的同步。在内核态均使用信号量这同一种数据结构进行存储：

```
typedef struct sem {
    int id;
    int key;
    int status;
    list_head block_queue;
} sem_t;

typedef sem_t binsem_t;
typedef sem_t cond_t;
typedef sem_t barrier_t;
```

其中，id 是序列下标，key 值对用户公开，status 用于记录状态（对条件变量和屏障，用于记录调用了 wait 进程的数目），block_queue 即为每个条件变量和屏障的阻塞队列，wait 时将 PCB 的 list 域加入。

对用户态的接口数据结构，我们只能记录对应内核中数据结构的 id 和 key，而不能将阻塞队列的地址暴露给用户。

```
typedef struct mthread_barrier
{
    // TODO:
    int key;
    int id;
    int count;
} mthread_barrier_t;
```

```
typedef struct mthread_cond
{
    // TODO:
    int key;
    int id;
} mthread_cond_t;
```

每次申请信号量时，用户端提供 key 值，由内核从指针数组中找一个空闲的 id，使用 kmalloc 在堆中动态分配信号量和屏障具体数据结构的空间，释放时回收，以尽可能节约内存。

```
#define MAX_BINSEM 256
#define MAX_NUM 256
sem_t *sem_list[MAX_BINSEM];
```

```
// Generate id
for(id = id_gen(); sem_list[id] != NULL; id = id_gen());
// Alloc memory space
sem_t *new = (sem_t *) kmalloc(sizeof(sem_t));
```

具体操作的实现与讲义一致，无需赘述。

4. mailbox 设计

Mailbox 作为进程间的通信，需要经过内核，用户不应当（也无法看到，上了虚存以后）得知 mailbox 的具体地址，故数据结构应实现在内核中，仅仅给用户一个 id 的索引。

```
typedef struct mailbox
{
    int  id;
    char name[64];
    char buffer[MSG_MAX_SIZE];
    int  head, tail;
    int  cited;
    int  used_size;
    int  full_id;
    int  empty_id;
    int  lock_id;
}mbox_t;
```

其中，id 即为邮箱号，name 为邮箱名，buffer 为内容（消息体），head 和 tail 分别指示消息体中有效信息的头和尾，这样可以循环利用消息体。Used_size 用于标志已经使用的空间大小，full_id 和 empty_id 是两个条件变量，分别用于阻塞正在读等待和写等待的进程。

在用户端，用户只能看到 mailbox 的 id。

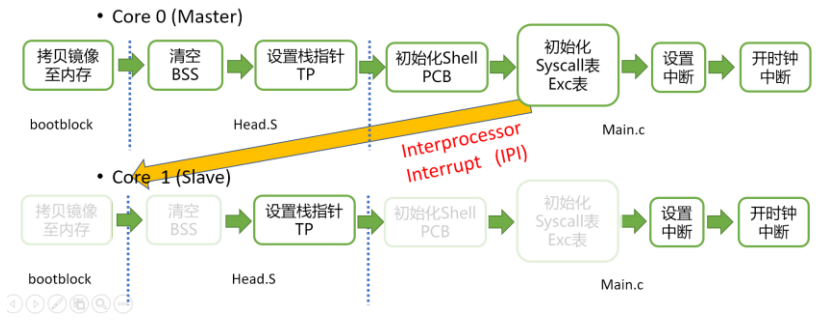
```
typedef int mailbox_t;
```

当出现多 comsumer-productor 的时候，邮箱完全可以正常运行，原因是我们采用了条件变量作为并发保护。当很多进程同时读一个邮箱时，若没有信息，都会被阻塞，待写入信息后，再进行广播，将所有被阻塞等待的进程释放出来。（以发消息为例）

```
void kernel_mbox_send(mailbox_t mailbox, void *msg, int msg_length)
{
    mbox_t *box = mbox_list[mailbox];
    kernel_binsemop(box->lock_id, BINSEM_OP_LOCK);
    while(MSG_MAX_SIZE - box->used_size < msg_length)
        kernel_sem_wait(box->empty_id, box->lock_id);
    if(MSG_MAX_SIZE - box->tail < msg_length)
    {
        memcpy((char *)(box->buffer + box->tail), (char *)msg, MSG_MAX_SIZE - box->tail);
        box->tail = msg_length - (MSG_MAX_SIZE - box->tail);
        memcpy((char *)box->buffer, (char *)(msg + msg_length - box->tail), box->tail);
    }
    else
    {
        memcpy((char *)(box->buffer + box->tail), (char *)msg, msg_length);
        box->tail += msg_length;
    }
    box->used_size += msg_length;
    kernel_sem_broadcast(box->full_id);
    kernel_binsemop(box->lock_id, BINSEM_OP_UNLOCK);
}
```

5. 双核使用设计

双核启动过程可以用如下的流程图进行表示：（灰色部分为从核无需完成的）



总体上看，双核启动需要注意的问题就是：寄存器相关的，每个核心都要做；而对内存相关的初始化，两个核心只需要做一次。比如清空 BSS，只需主核清空即可，否则从核启动后，主核初始化的 syscall 表就被清了。

我们需要对主核和从核启动的先后顺序进行设置：在 bootblock.S 里，我们让主核正常运行，从核只开 S 态软件中断等待 IPI。当主核做完内存相关初始化时，发送 IPI，唤醒从核。

另外，pid0 对应的内核线程需要双核给两个不同的栈。因为两个核心是同时跑这个进程的，他们不可以同时跑在一个栈上。于是我将 pid0_pcb 设置了两个，也用于后续当没有进程可以调度时，处理器核总是可以调度 pid0_pcb。

对于 current_running，这是唯一需要两份的数据结构，因为每个核心都有在跑的进程。因此我设置了 current_running 为二阶指针，每次需要使用 current_running 时，通过 get_cpuid 进行判断，指向 current_running_core0 或 current_running_core1。

注意，我们的设置中，ready_queue 只有一个，每个核都可以调度处于 ready_queue 的进程。每当一个核心时钟中断到来时，会从 ready_queue 中找一个可以被调度（mask 符合）的进程唤醒在本核上。绑核操作本质上也是通过调整 mask 来实现的，调整完 mask 后，下次调度就会被唤醒在对应的核心上。

多核间同步原语实现与单核一致，因为同步原语的数据结构均存放在内核，且我们能够通过自旋锁保证，同一时刻仅有一个核心陷入内核，因此每次对同步原语的操作都是连贯的 、不会被打断的。

参考文献

[1] NJU-ProjectN：简易调试器 https://nju-projectn.github.io/ics-pa-gitbook/ics2020/1.4.html