Project 3 Interactive OS and Process Management 设计文档

中国科学院大学
郑旭舟
2021 年 1 月 14 日

1. Shell 设计

设置 in_buf 存放已经输入的字符，设置 in_id 指示当前字符在 in_buf 中的下标。

```
char in_buf[IN_LEN_MAX] = {0};
int in_id = 0;
```

读取输入时使用 sys_read_keyboard()函数，该函数封装了 read_keyboard()函数，作用是根据串口状态寄存器读取串口数据寄存器并返回。

在读取输入的 while(1)循环中，以是否输入\r 字符作为“继续接收输入”和“解析输入”的分界点。

解析输入时，优化了框架内的 memcmp()函数(string.c)，增加代码可读性。

若解析后发现输入合法，则直接执行有关操作；若输入无效，则打印 error 提示。

2. kill 和 wait 内核实现的设计

2.1.1. kill 处理过程

```
int do_kill(pid_t pid){
    int i, flag = 0;
    for (i = 0; i < NUM_MAX_TASK; i++) { // 寻找需要被杀死的进程
        if (pcb[i].pid == pid && pcb[i].status != TASK_EXITED) {
            flag = 1; break;
        }
    }
    if (flag == 0) return -1; // 如果符合要求的进程不存在，退出并返回-1

    pcb_t *to_kill;
    to_kill = &pcb[i];
    // 如果需要被杀死的进程(下称 to_kill)正被阻塞，则将其从阻塞队列中释放
    if (to_kill->status == TASK_BLOCKED)
        queue_remove((queue_t *)(to_kill->block_me), to_kill);
    else
        queue_remove(&ready_queue, current_running);
    // 释放所有被 to_kill 持有的锁，同时释放因为得不到这些锁被阻塞的进程
    while (!queue_is_empty(&(to_kill->lock_queue))) {
        mutex_lock_t *t;
        t = to_kill->lock_queue.head;
        do_unblock_all(&(t->blocked));
```

```
        queue_dequeue(&(to_kill->lock_queue));
        t->status = UNLOCKED;
    }
    // 将 to_kill 的等待队列也清空
    do_unblock_all(&(to_kill->wait_queue));
    // to_kill 的状态清为 EXITED
    to_kill->status = TASK_EXITED;
    to_kill->block_me = NULL;
    // 完成 kill 操作，返回 0
    return 0;
}
```

2.1.2. kill 中如何处理锁

如上一节所示，释放这些锁，同时对每一把锁，释放被阻塞的进程。

2.1.3. kill 是否有处理同步原语

无。

2.1.4. wait 处理过程

```
int do_waitpid(pid_t pid){
    int i, find = 0;
    for (i = 0; i < NUM_MAX_TASK; i++) // 寻找需要 wait 的进程
        if (pcb[i].pid == pid) {
            find = 1; break;
        }
    // 如果找不到需要 wait 的进程(to_wait)，退出并返回-1
    if (find == 0 && i == NUM_MAX_TASK - 1) return -1;
    // 如果 to_wait 未退出，则将当前进程加入 to_wait 的阻塞队列 wait_queue
    if (pcb[i].status != TASK_EXITED) do_block(&(pcb[i].wait_queue));
}
```

2.1.5. wait 实现时，等待的进程的 PCB 用什么结构保存？

如上一节所示，将该 pcb 加入 to_wait 的 wait_queue 即可。

3. 关键函数功能

3.1. 用户态函数

3.1.1. do_spawn：装载 pcb

```
int do_spawn(task_info_t *task){
    int i;
    i = alloc_pcb();
    if (i == -1) return -1;
```

```
    set_pcb(++process_id, &pcb[i], task);
    queue_push(&ready_queue, &pcb[i]);
    pcb[i].status = TASK_READY;
    return 0;
}
```

3.1.2. do_exit：进程让自身退出

```
void do_exit(void)
{
    pcb_t *to_exit;
    to_exit = current_running;
    queue_remove(&ready_queue, current_running);

    // lock blocked
    while (!queue_is_empty(&(to_exit->lock_queue))) {
        mutex_lock_t *t;
        t = to_exit->lock_queue.head;
        do_unblock_all(&(t->blocked));
        queue_dequeue(&(to_exit->lock_queue));
        t->status = UNLOCKED;
    }
    //wait
    do_unblock_all(&(to_exit->wait_queue));

    to_exit->status = TASK_EXITED;
    do_scheduler();
}
```

3.1.3. do_kill: 杀死一个指定进程

```
int do_kill(pid_t pid){
    int i, flag = 0;
    for (i = 0; i < NUM_MAX_TASK; i++) {
        if (pcb[i].pid == pid && pcb[i].status != TASK_EXITED) {
            flag = 1;
            break;
        }
    }
    if (flag == 0) // not existed
        return -1;

    pcb_t *to_kill;
    to_kill = &pcb[i];
```

```
    if (to_kill->status == TASK_BLOCKED)
        queue_remove((queue_t *)(to_kill->block_me), to_kill);
    else
        queue_remove(&ready_queue, current_running);

    // lock blocked
    while (!queue_is_empty(&(to_kill->lock_queue))) {
        mutex_lock_t *t;
        t = to_kill->lock_queue.head;
        do_unblock_all(&(t->blocked));
        queue_dequeue(&(to_kill->lock_queue));
        t->status = UNLOCKED;
    }
    //wait
    do_unblock_all(&(to_kill->wait_queue));
    to_kill->status = TASK_EXITED;
    to_kill->block_me = NULL;

    return 0;
}
```

3.1.4. do_waitpid: 等待一个进程退出

```
int do_waitpid(pid_t pid) {
    int i, find;
    find = 0;
    for (i = 0; i < NUM_MAX_TASK; i++)
        if (pcb[i].pid == pid) {
            find = 1;
            break;
        }

    if (find == 0 && i == NUM_MAX_TASK - 1) return -1;

    if (pcb[i].status != TASK_EXITED) do_block(&(pcb[i].wait_queue));
}
```

3.1.5. do_process_show：打印进程表

```
void do_process_show()
{
    vt100_move_cursor(pcb[0].cursor_x, ++pcb[0].cursor_y);
    printk("[PROC TABLE]");
    int i, num_ps = 0;
```

```
    for (i = 0; i < NUM_MAX_TASK; i++) // show running
    {
        task_status_t status = pcb[i].status;
        switch (status) {
        case TASK_RUNNING:
            vt100_move_cursor(pcb[0].cursor_x, ++pcb[0].cursor_y);
            printk("[%d] PID = %d STATUS = RUNNING", num_ps++,
                   pcb[i].pid);
            break;
        case TASK_READY:
            vt100_move_cursor(pcb[0].cursor_x, ++pcb[0].cursor_y);
            printk("[%d] PID = %d STATUS = READY", num_ps++,
                   pcb[i].pid);
            break;
        case TASK_BLOCKED:
            vt100_move_cursor(pcb[0].cursor_x, ++pcb[0].cursor_y);
            printk("[%d] PID = %d STATUS = BLOCKED", num_ps++,
                   pcb[i].pid);
            break;
        default:
            break;
        }
    }
    screen_move_cursor(pcb[0].cursor_x, pcb[0].cursor_y);
}
```

3.1.6. do_getpid：得到当前进程的 pid

```
pid_t do_getpid() {
    return current_running->pid;
}
```

3.1.7. read_keyboard：读取键盘输入

```
char read_keyboard(void){
    while (*((char *)STAT_REG) & 0x01)
        return *((char *)DATA_REG);
    return 0;
}
```

3.1.8. (string.c)优化后的 memcmp 函数：支持指定比较字符串的前缀长度：

```
int memcmp(char *str1, char *str2, uint32_t size){
```

```
    int i;
    for (i = 0; i < size; i++) {
        if (str1[i] > str2[i])
            return 1;
        else if (str1[i] < str2[i])
            return -1;
    }
    return 0;
}
```

3.2. test_shell 函数：

```
void test_shell() {
    sys_move_cursor(pcb[0].cursor_x, pcb[0].cursor_y);
    printf("%s", command_boundary); // show boundary
    pcb[0].cursor_x = SHELL_LEFT_LOC;
    sys_move_cursor(pcb[0].cursor_x, ++pcb[0].cursor_y);
    printf("%s", user_name); // show username

    char in_buf[IN_LEN_MAX] = { 0 };
    int in_id = 0;

    while (1) {
        char in = sys_read_keyboard();
        if (in == 0) continue;
        else if (in != '\r') {
            if (in == 8) { // backspace
                if (in_id > 0) {
                    printf("%c", in); in_buf[in_id--] = '\0';
                }
            } else if (in_id < IN_LEN_MAX) { // not backspace
                printf("%c", in);
                in_buf[in_id++] = in;
            }
        } else { // parsing buffer when '\r'
            printf("%c", in);
            in_buf[in_id] = '\0';
            if (memcmp(in_buf, "ps", 2) == 0 && in_buf[2] == '\0'){ // ps
                pcb[0].cursor_x = SHELL_LEFT_LOC;
                sys_move_cursor(pcb[0].cursor_x, ++pcb[0].cursor_y);
                sys_process_show();
            } else if (memcmp(in_buf, "clear", 5) == 0
                       && in_buf[5] == '\0') { // clear
                sys_screen_clear(SHELL_BOUNDARY + 1, SCREEN_HEIGHT);
```

```
            pcb[0].cursor_y = SHELL_BOUNDARY + 1;
        } else if (memcmp(in_buf, "kill", 4) == 0) { //kill
            int be_kill = get_num(in_buf);
            int has_killed = (be_kill > 0) ?
                                sys_kill((pid_t)be_kill) : (-1);
            pcb[0].cursor_x = SHELL_LEFT_LOC;
            sys_move_cursor(pcb[0].cursor_x, ++pcb[0].cursor_y);
            if (has_killed == 0)
                printf("PROCESS (pid=%d) has been KILLED.", be_kill);
            else if (has_killed == -1)
                printf("PROCESS NOT EXISTED.");
            pcb[0].cursor_y++;
        } else if (memcmp(in_buf, "wait", 4) == 0) { //wait
            int wait_who = get_num(in_buf);
            int has_waited = (wait_who > 0) ?
                    sys_waitpid((pid_t)wait_who) : (-1);
            pcb[0].cursor_x = SHELL_LEFT_LOC;
            if (has_waited == -1) {
                sys_move_cursor(pcb[0].cursor_x, ++pcb[0].cursor_y);
                printf("PROCESS %d NOT EXISTED.", wait_who);
            }
            pcb[0].cursor_y++;
        } else if (memcmp(in_buf, "exec", 4) == 0) { //exec
            int be_exec = get_num(in_buf);
            int has_exec;
            if (be_exec > 0 && be_exec < 16)
                has_exec = sys_spawn(test_tasks[be_exec]);
            else
                has_exec = -1; // invalid task

            pcb[0].cursor_x = SHELL_LEFT_LOC;
            sys_move_cursor(pcb[0].cursor_x, pcb[0].cursor_y);
            if (has_exec == -1) {
                printf("Task %d EXEC FAILED/NOT EXISTED.", be_exec);
            } else
                printf("Task %d EXEC SUCCESS.", be_exec);
            pcb[0].cursor_y++;
        } else { // error
            printf("%c", in);
            pcb[0].cursor_x = SHELL_LEFT_LOC;
            hint_print();
        }

        pcb[0].cursor_x = SHELL_LEFT_LOC;
```

```
            sys_move_cursor(pcb[0].cursor_x, pcb[0].cursor_y);
            in_id = 0;
            memset(in_buf, 0, IN_LEN_MAX * (sizeof(char)));
            printf("%s", user_name); // show username
        }
    }
}
```