Project 3 Interactive OS and Process Management 设计文档

中国科学院大学

[姓名] 蔡洛姗

[日期] 2020/11/23

1. kill 和 wait 内核实现的设计

（1）kill 处理过程中如何处理锁，是否有处理同步原语，如果有处理，请说明。

对于内核锁，我们在每个进程中设置一个 lock 数组，用来存放该进程已经获得的锁，每次获得一个锁时就将锁的地址放在该数组中，并且记录获得锁的数量。若进程被 kill，则将 pcb 种所有对应的锁释放。处理同步原语主要就是锁来实现，因此 kill 时释放锁，由于整个 kill 是一个系统调用，不会中断，不需要考虑用原语。

（2）wait 实现时，等待的进程的 PCB 用什么结构保存？

在 pcb 中设置了一个等待队列 wait_list，用于存放等待该 pcb 完成的进程地址。

（3）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）需要特别注意栈空间和 pcb 的回收，以及 ZOMBIE 状态。

2. 同步原语设计

（1）信号量和屏障实现的各自数据结构的包含内容

```
typedef struct mthread_barrier
{
    unsigned total_num;                 //需要在屏障处等待共同执行的任务总数
    unsigned wait_num;                  //当前到达屏障处的任务数
    list_head barrier_queue;            //屏障的等待队列
} mthread_barrier_t;

typedef struct mthread_cond
{
    list_head wait_queue;               //条件变量的等待队列
} mthread_cond_t;
```

3. mailbox 设计

（1）mailbox 的数据结构以及主要成员变量的含义

用户态可见的 mailbox 只有一个信箱 id，用于到内核态访问信箱，防止信箱内容在用户态暴露。

内核态的信箱结构如下：

```
typedef struct mailbox_k
```

```
{
    char name[25];                  //name of mailbox
    char msg[MAX_MBOX_LENGTH];      //content of message
    int index;                      //ptr of msg_buf
    int visited;                    //count when the box is used
    mailbox_status_t status;        //信箱状态：open/close
    mthread_cond_t empty;     //条件变量：如果信箱为空，则进入 empty 的等待队列
    mthread_cond_t full;     //条件变量：如果信箱为满，则进入 full 的等待队列
} mailbox_k_t;
```

（2）你在 mailbox 设计中如何处理 producer-consumer 问题，使用哪种同步原语进行并发访问保护？你的实现是否支持多 producer 或多 consumer，如果有，你是如何处理的。

我将 mailbox 操作也封装成系统调用，从而保证对信箱的互斥访问。我的实现使用了条件变量处理生产-消费者问题，支持多 producer 和多 consumer，只要缓冲区不是满或空的。

（3）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

这部分实验主要遇到的问题更多是之前的进程 kill, spawn 操作没有完善的问题。

4. 双核使用设计（没有做双核的同学不用写）

（1）你在启用双核时遇到的问题有什么，如何解决的？

在主核通过核间中断唤醒从核之后，由于我没有固定中断哪个核，所以需要清 sip 寄存器防止主核也被中断，在这里我使用了 C 语言的嵌入汇编，并使用了宏定义里的 CSR_SIP 偏移，导致编译总是不过，而且编译器并没有报错。后来在老师的电脑上编译了一下，报出了这个错误，可能是因为嵌入汇编不支持用偏移来对应寄存器的操作，因此把它直接写成 sip 就能通过了。

在启动过程中我给内核加了大锁来保证只有一个核进入内核，当主核启动进入内核之后，kernel_lock 已经锁上了，但是当从核也启动之后，再申请锁发现它的状态还是 unlock，也就是被清零了。后来发现是因为 kernel_lock 是一个全局变量，保存在内存的 bss 段上，而从核启动时跳入 kernel_main 执行了清空 bss 段的代码，也就把 lock 给释放了。应该在 head.S 清空 bss 段之前就判断是主核还是从核，走不同的分支。

接着在成功启动两个核后，进程卡住，一个在内核中做调度，一个在申请内核锁自旋，为什么会重复做调度而不中断呢？其实是因为一开始 pcb 中只有一个 test_shell 进程，主核先进入内核把这个进程拿走执行，退出内核态，然后从核进入内核做调度，发现没有 pcb 可以执行，而主核这时候执行系统调用需要进入内核却被锁住，因此陷入死循环。

（2）你是如何让不同的任务在不同的核上运行的？

我让两个核共用了一个 pcb 数组和一个 ready_queue，而 current_running 有两个，且内核被一个大锁保护起来，这样只有一个核能进入内核态做调度，从 ready_queue 中选出一个 pcb 任务执行，这样不同任务就会在不同核上运行。

（3）你在双核上如何保证同步原语和 kill 的正确性？（做 C-Core 的同学回答）

5. 关键函数功能

```
/* 回收内存 */
```

```
void recycle(long *stack_base){
    *stack_base = recycle_queue;
    recycle_queue = (ptr_t)stack_base;
}
/* reuse the free-stack space */
ptr_t reuse(){
    long *new_stack_base;
    if(recycle_queue==(ptr_t)&recycle_queue){
        return 0;
    }else{
        new_stack_base = (long *)recycle_queue;
        recycle_queue = *new_stack_base;
    }
    return (ptr_t)new_stack_base;
}

//create a process with given info, return the pid
pid_t do_spawn(task_info_t *task, void* arg, spawn_mode_t mode){
    pcb_t *new_pcb;
    if (!list_empty(&exit_queue))
    {
        new_pcb = list_entry(exit_queue.prev, pcb_t, list);
        list_del(exit_queue.prev);
    }else
    {
        new_pcb = &pcb[process_num++];
    }

    if((new_pcb->kernel_stack_base=reuse())==0){
        new_pcb->kernel_stack_base = allocPage(1) + PAGE_SIZE;
    }
    if((new_pcb->user_stack_base=reuse())==0){
        new_pcb->user_stack_base = allocPage(1) + PAGE_SIZE;
    }
    new_pcb->kernel_sp =  new_pcb->kernel_stack_base;
    new_pcb->user_sp = new_pcb->user_stack_base;
    init_pcb_stack(new_pcb->kernel_sp, new_pcb->user_sp,
task->entry_point, arg, new_pcb);
    new_pcb->kernel_sp = new_pcb->kernel_sp -sizeof(regs_context_t) -
sizeof(switchto_context_t);
    new_pcb->pid = process_id++;
    new_pcb->type = task->type;
    new_pcb->status = TASK_READY;
    new_pcb->mode = mode;
```

```
    new_pcb->cursor_x = 1;
    new_pcb->cursor_y = 1;
    new_pcb->lock_num = 0;
    list_add(&new_pcb->list, &ready_queue);
    init_list_head(&new_pcb->wait_list);

    return new_pcb->pid;
}

//exit from the current_running pcb, recycle the pcb/stack/lock
void do_exit(void){
    uint64_t cpu_id;
    cpu_id = get_current_cpu_id();
    pcb_t *exit_pcb = current_running[cpu_id];

    /* 释放 wait 队列 */
    while(!list_empty(&exit_pcb->wait_list)){
        pcb_t *wait_pcb;
        wait_pcb = list_entry(exit_pcb->wait_list.prev, pcb_t, list);
        if(wait_pcb->status!=TASK_EXITED){
            do_unblock(exit_pcb->wait_list.prev);
        }
    }

    /* 释放锁 */
    int i=exit_pcb->lock_num;
    while(i){
        i--;
        do_mutex_lock_release(exit_pcb->locks[i]);
    }

    /* 回收内存资源 */
    recycle((long *)exit_pcb->kernel_stack_base);
    recycle((long *)exit_pcb->user_stack_base);

    /* 回收 pcb */
    list_add(&exit_pcb->list, &exit_queue);

    /* 修改状态 */
    exit_pcb->status = TASK_EXITED;
    do_scheduler();
}

//kill process[pid], recycle the pcb/stack/lock, get away from queues
```

```
int do_kill(pid_t pid){
    int i;
    for(i=0; (pcb[i].pid!=pid) && i<NUM_MAX_TASK; i++);
    if (i==NUM_MAX_TASK)
    {
        return 0;
    }
    pcb_t *killing_pcb = &pcb[i];

    /* 移出所在的队列 */
    list_del(&killing_pcb->list);         //就绪队列，block 队列
    list_del(&(killing_pcb->timer.list)); //timers 队列

    /* 释放 wait 队列 */
    while(!list_empty(&killing_pcb->wait_list)){
        pcb_t *wait_pcb;
        wait_pcb = list_entry(killing_pcb->wait_list.prev, pcb_t, list);
        if(wait_pcb->status!=TASK_EXITED){
            do_unblock(killing_pcb->wait_list.prev);
        }
    }
    /* 释放锁 */
    i = killing_pcb->lock_num;
    while(i){
        i--;
        do_mutex_lock_release(killing_pcb->locks[i]);
    }

    /* 回收内存资源 */
    recycle((long *)killing_pcb->kernel_stack_base);
    recycle((long *)killing_pcb->user_stack_base);

    /* 回收 pcb */
    list_add(&killing_pcb->list, &exit_queue);

    /* 修改状态 */
    killing_pcb->status = TASK_EXITED;
    killing_pcb->pid = 0;
    uint64_t cpu_id;
    cpu_id = get_current_cpu_id();
    if(killing_pcb==current_running[cpu_id])
        do_scheduler();
    return 1;
}
```

```
//current_running task should wait until process[pid] finished
int do_waitpid(pid_t pid){
    int i;
    for(i=0; (pcb[i].pid!=pid) && i<NUM_MAX_TASK; i++);
    if (i==NUM_MAX_TASK) return 0;
    uint64_t cpu_id;
    cpu_id = get_current_cpu_id();
    current_running[cpu_id]->status = TASK_BLOCKED;
    do_block(&current_running[cpu_id]->list, &pcb[i].wait_list);
    return 1;
}

void do_process_show(){
    prints("[PROCESS TABLE]\n");
    int i,j;
    i=j=0;
    for(;i<process_num;i++){
        if(pcb[i].status==TASK_RUNNING){
            prints("[%d] PID : %d STATUS : RUNNING\n",j,pcb[i].pid);
            j++;
        }else if(pcb[i].status==TASK_READY){
            prints("[%d] PID : %d STATUS : READY\n",j,pcb[i].pid);
            j++;
        }else if(pcb[i].status==TASK_BLOCKED){
            prints("[%d] PID : %d STATUS : BLOCKED\n",j,pcb[i].pid);
            j++;
        }
    }
}

pid_t do_getpid(){
    uint64_t cpu_id;
    cpu_id = get_current_cpu_id();
    return current_running[cpu_id]->pid;
}
//P3-task2-----------------------------------------------------------------
int do_cond_wait(mthread_cond_t *cond, mthread_mutex_t *mutex){
    uint64_t cpu_id;
    cpu_id = get_current_cpu_id();
    current_running[cpu_id]->status = TASK_BLOCKED;
    list_add(&current_running[cpu_id]->list,&cond->wait_queue);
    do_binsemop(mutex->lock_id, BINSEM_OP_UNLOCK);
    do_scheduler();
```

```
    do_binsemop(mutex->lock_id, BINSEM_OP_LOCK);
    return 1;
}
int do_cond_signal(mthread_cond_t *cond){
    if(!list_empty(&cond->wait_queue)){
        do_unblock(cond->wait_queue.prev);
    }
    return 1;
}
int do_cond_broadcast(mthread_cond_t *cond){
    while(!list_empty(&cond->wait_queue)){
        do_unblock(cond->wait_queue.prev);
    }
    return 1;
}
int do_barrier_wait(mthread_barrier_t *barrier){
    barrier->wait_num++;
    if(barrier->total_num==barrier->wait_num){
        while(!list_empty(&barrier->barrier_queue))
            do_unblock(barrier->barrier_queue.prev);
        barrier->wait_num=0;
    }else{
        uint64_t cpu_id;
        cpu_id = get_current_cpu_id();
        current_running[cpu_id]->status = TASK_BLOCKED;
list_add(&current_running[cpu_id]->list,&barrier->barrier_queue);
        do_scheduler();
    }
    return 1;
}

//P3-task3-----------------------------------------------------------------
mailbox_k_t mailbox_k[MAX_MBOX_NUM]; //kernel's mail box
int do_mbox_open(char *name)
{
    int i;
    for(i=0;i<MAX_MBOX_NUM;i++){
        if(kstrcmp(name,mailbox_k[i].name)==0){
            return i;
        }
    }
    for(i=0;i<MAX_MBOX_NUM;i++){
        if(mailbox_k[i].status==MBOX_CLOSE){
            mailbox_k[i].status=MBOX_OPEN;
```

```
            int j=0;
            while(*name){
                mailbox_k[i].name[j++]=*name;
                name++;
            }
            mailbox_k[i].name[j]='\0';
            return i;
        }
    }
    prints("No mailbox is available\n");
    return -1;
}
void do_mbox_close(int mailbox_id){
    mailbox_k[mailbox_id].status = MBOX_CLOSE;
}
void do_mbox_send(int mailbox_id, void *msg, int msg_length){
  if((mailbox_k[mailbox_id].index+msg_length)>MAX_MBOX_LENGTH){
        //mailbox is full, block the task unil box is not full
        uint64_t cpu_id;
        cpu_id = get_current_cpu_id();
        current_running[cpu_id]->status = TASK_BLOCKED;
        list_add(&current_running[cpu_id]->list,
                &mailbox_k[mailbox_id].full.wait_queue);
        do_scheduler();
    }
    //put msg in mailbox
    int i;
    for(i=0;i<msg_length;i++){
        mailbox_k[mailbox_id].msg[mailbox_k[mailbox_id].index++] =
        ((char*)msg)[i];
    }
    do_cond_broadcast(&mailbox_k[mailbox_id].empty); //release wait_tasks
}
void do_mbox_recv(int mailbox_id, void *msg, int msg_length){
    if((mailbox_k[mailbox_id].index-msg_length)<0){ //mailbox is empty
        //block the task unil box is not empty
        uint64_t cpu_id;
        cpu_id = get_current_cpu_id();
        current_running[cpu_id]->status = TASK_BLOCKED;
        list_add(&current_running[cpu_id]->list,
                &mailbox_k[mailbox_id].empty.wait_queue);
        do_scheduler();
    }
    //get msg from mailbox
```

```
    int i;
    for(i=msg_length-1;i>=0;i--){
        ((char*)msg)[i] =
               mailbox_k[mailbox_id].msg[--mailbox_k[mailbox_id].index];
    }
    do_cond_broadcast(&mailbox_k[mailbox_id].full);  //release all tasks
waiting for send msg
}
```