

操作系统Lab3实验报告

211300082 谭荣熙 rongxitan@smail.nju.edu.cn

1. 实验要求

本次实验主要完成自制简单操作系统的进程管理功能, 通过实现一个简单的任务调度, 介绍基于时间中断进程切换完成任务调度的全过程, 主要涉及到fork, exit, sleep等库函数和对应的处理例程实现.

1.1 完成库函数

1.2 时钟中断处理

1.3 实现syscallFork, syscallSleep, syscallExit等系统调用例程

2. 实验过程

在Lab3中, 我完成了除选做部分外的全部内容, 接下来我将从三个阶段来介绍完成实验的过程.

2.1 填补fork, sleep, exit库函数

- 根据Lab2中有关内容的理解, 直接进行相应系统调用即可

```
pid_t fork()                { return syscall(SYS_FORK, 0, 0, 0, 0, 0);  
}  
int sleep(uint32_t time) { return syscall(SYS_SLEEP, (uint32_t)time,  
0, 0, 0, 0); }  
int exit()                  { return syscall(SYS_EXIT, 0, 0, 0, 0, 0);  
}
```

2.2 时钟中断处理: 填写timeHandle函数

- 将切换到内核进程的代码抽象成宏定义: 注意到进程切换在多个函数中均会被使用到, 故抽象成宏定义switch_to_kernel().

```

#define switch_to_kernel() do {\
    uint32_t tmpStackTop = pcb[current].stackTop;\
    pcb[current].stackTop = pcb[current].prevStackTop;\
    tss.esp0 = (uint32_t)&(pcb[current].stackTop);\
    asm volatile("movl %0, %%esp"::"m"(tmpStackTop));\
    asm volatile("popl %gs");\
    asm volatile("popl %fs");\
    asm volatile("popl %es");\
    asm volatile("popl %ds");\
    asm volatile("popal");\
    asm volatile("addl $8, %esp");\
    asm volatile("iret");} while(0)

```

- 更新阻塞进程的sleepTime: 遍历pcb, 将状态为STATE_BLOCKED的进程的sleepTime减一, 若其变为0, 则重新设为STATE_RUNNABLE.

```

for (int i = 0; i < MAX_PCB_NUM; i++) {
    if (pcb[i].state == STATE_BLOCKED) {
        if (pcb[i].sleepTime > 0) pcb[i].sleepTime--;
        if (pcb[i].sleepTime == 0) pcb[i].state =
STATE_RUNNABLE;
    }
}

```

- 更新当前进程的timeCount: 将当前进程的timeCount加一, 若时间片用完且有其它状态为STATE_RUNNABLE的进程则切换, 否则继续执行.

```

// 若没用完, 则pcb[current].timeCount自增
if (pcb[current].state == STATE_RUNNING &&
pcb[current].timeCount < MAX_TIME_COUNT) {
    pcb[current].timeCount++;
    return;
}
// 重置pcb[current]的状态
pcb[current].timeCount = 0;
pcb[current].state = STATE_RUNNABLE;
// 寻找可用进程并切换
for (int i = (current + 1) % MAX_PCB_NUM; i != current; i =
(i+1) % MAX_PCB_NUM) {
    if (pcb[i].state == STATE_RUNNABLE && i != 0) {
        current = i;
        break;
    }
}
pcb[current].state = STATE_RUNNING;
switch_to_kernel();

```

2.3 实现相关系统调用例程

- 填写syscallFork函数:

- 首先, 寻找一个可用pcb块, 并根据fork() 系统调用的内涵, 让子进程复用父进程的地址空间

```
int new_index = 0;
// 寻找可用进程
for (new_index = 0; new_index < MAX_PCB_NUM &&
pcb[new_index].state != STATE_DEAD; new_index++);
// 找不到, 则返回异常
if (new_index == MAX_PCB_NUM) {
    pcb[current].regs.eax = -1;
    return;
}
// 由于I/O有开销, 故打开中断允许抢占, 并依次复制地址空间的内容
enableInterrupt();
for (int i = 0; i < 0x100000; i++) {
    *(unsigned char *) (i + (new_index + 1) * 0x100000) = *
(unsigned char *) (i + (current + 1) * 0x100000);
    asm volatile("int $0x20");
}
disableInterrupt();
```

- 设置子进程的基本信息, 并复制当前进程的部分寄存器信息. 注意到, pcb块中的prevStackTop和stackTop暂存的是前序进程和当前进程栈基址esp的相对地址, 故需要对其进行相对地址的计算转换. 同时, 在kvm.c文件中, 注意到initSeg函数在填写gdt表时, 为每个pcb块分配两个段, 分别赋予了不同的权限. 结合initProc中对pcb[1]的设置, 可推断每个新创建的子进程均占用两个段, 其中代码段为2*i+1, 其余使用第2*i+2个段. 最后修改状态, 设置返回值即可

```
pcb[new_index].pid = new_index;
pcb[new_index].sleepTime = 0;

pcb[new_index].prevStackTop = pcb[current].prevStackTop -
(uint32_t)&(pcb[current]) + (uint32_t)&(pcb[new_index]);
pcb[new_index].stackTop = pcb[current].stackTop -
(uint32_t)&(pcb[current]) + (uint32_t)&(pcb[new_index]);

pcb[new_index].state = STATE_RUNNABLE;
pcb[new_index].timeCount = 0;

pcb[new_index].regs.edi = pcb[current].regs.edi;
... // esi, ebp, xxx, edx, ebx, ecx, eax, irq, error, eip,
esp, eflags are similar

pcb[new_index].regs.cs = USEL(1 + 2*new_index);
pcb[new_index].regs.ss = USEL(2 * (new_index+1));
... // ds, es, fs, gs are the same as ss
```

```
pcb[current].state = STATE_RUNNABLE;
pcb[current].regs.eax = new_index;
pcb[new_index].regs.eax = 0;
```

- 将进程选择过程进行抽象: 在本次实验要求中, 并没有明确提出分级就绪队列的建立与维护. 故简单地, 在本次实验的实现中, 直接从current开始, 用循环队列的方式对所有的pcb块遍历, 选择下一个就绪的进程进行运行. 若全部pcb都已占满且无就绪进程, 则将控制权交回给内核. 将此过程抽象成宏定义.

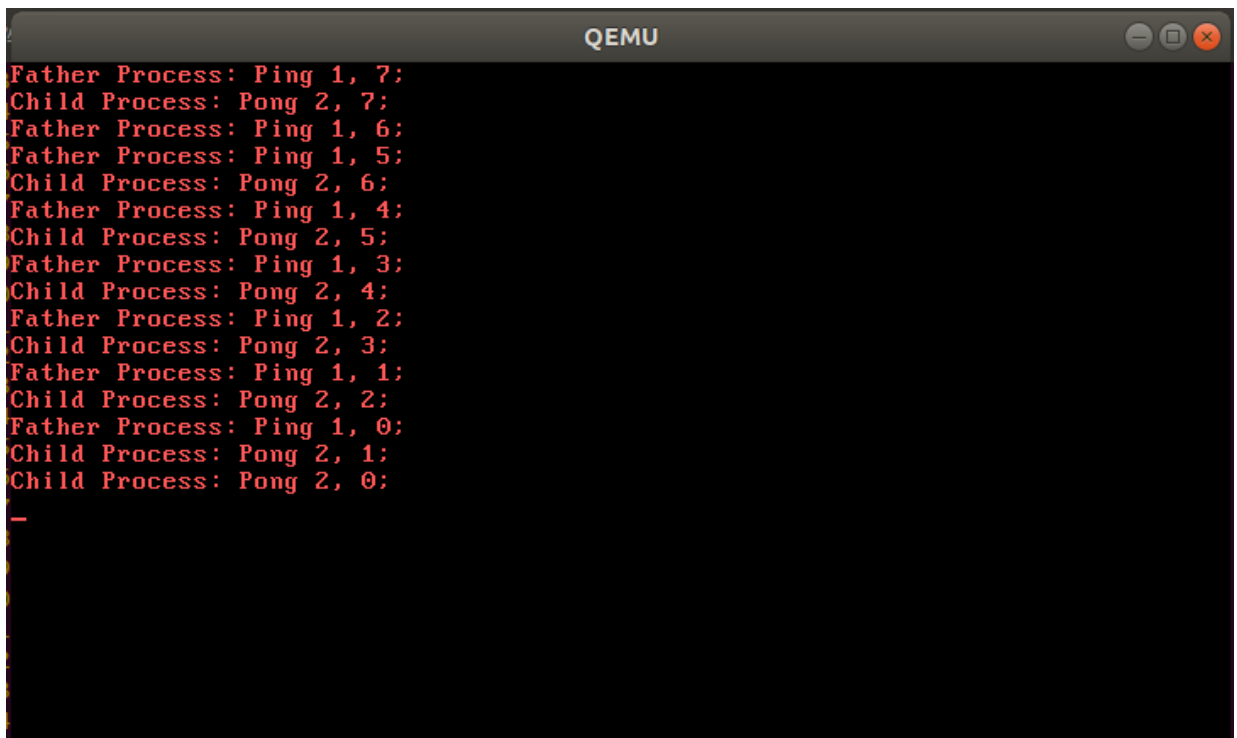
```
#define find_new_running_pcb() do {\
    int find_runnable_pcb = 0; \
    for (int i = (current + 1) % MAX_PCB_NUM; i != current; i =\
        (i+1) % MAX_PCB_NUM) {\
        if (pcb[i].state == STATE_RUNNABLE && i != 0) {\
            find_runnable_pcb = 1;\
            current = i;\
            break;\
        }\
    }\
    if (find_runnable_pcb == 0) current = 0;\
    pcb[current].state = STATE_RUNNING;\
} while(0)
```

- 实现syscallSleep和syscallExit函数: 由于相关工具已封装好, 故直接调用即可. 二者的区别仅在于对current进程状态的更改.

```
void syscallSleep(struct StackFrame *sf) {\
    pcb[current].state = STATE_BLOCKED;\
    find_new_running_pcb();\
    switch_to_kernel();\
}\
void syscallExit(struct StackFrame *sf) {\
    pcb[current].state = STATE_DEAD;\
    find_new_running_pcb();\
    switch_to_kernel();\
}
```

3. 实验结果

- 上述实现顺利通过用户程序的测试.

A screenshot of a QEMU terminal window. The window title is "QEMU". The terminal output shows a sequence of "Ping" and "Pong" messages between a "Father Process" and a "Child Process". The father process sends pings with values from 7 down to 0, and the child process responds with pongs of the same values. The sequence ends with a hyphen "-" on a new line.

```
QEMU
Father Process: Ping 1, 7;
Child Process: Pong 2, 7;
Father Process: Ping 1, 6;
Father Process: Ping 1, 5;
Child Process: Pong 2, 6;
Father Process: Ping 1, 4;
Child Process: Pong 2, 5;
Father Process: Ping 1, 3;
Child Process: Pong 2, 4;
Father Process: Ping 1, 2;
Child Process: Pong 2, 3;
Father Process: Ping 1, 1;
Child Process: Pong 2, 2;
Father Process: Ping 1, 0;
Child Process: Pong 2, 1;
Child Process: Pong 2, 0;
-
```

由此可知, 实验大致正确.

4. 问题回答

4.1 linux下进程的创建及运行有两个命令fork和exec, 请说明他们的区别?

fork和exec都是创建和运行进程的系统调用. 其区别主要体现在:

- fork创建一个新进程, 子进程是父进程的一个副本. 这两个进程会在fork()调用的地方开始执行代码, 且父进程和子进程的pid不同, 二者后续独立运行, 各自管理自己的资源
- exec将当前进程替换为一个新的进程, 会将原进程的代码段, 数据段, 堆栈段等全部替换为新进程的相应部分, 但进程的pid不变.

代码示例如下:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pid = fork(); // 创建子进程
    if (pid < 0) {
        printf("fork error\n");
    } else if (pid == 0) {
        printf("This is child process, PID=%d\n", getpid());
        char* args[] = {"ls", "-l", NULL}; // 执行ls命令
        execvp(args[0], args); // 用ls命令替换子进程
        printf("exec error\n"); // 如果execvp()执行失败则会执行到这里
    } else {
        printf("This is parent process, PID=%d\n", getpid());
        wait(NULL); // 等待子进程结束
    }
}
```

```

        printf("Child process finished\n");
    }
    return 0;
}

```

代码实现了用fork创建子进程,并在子进程独立的内存空间上调用exec执行ls命令,替换掉子进程的内容的过程。

4.2 请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析,并分析 fork/exec/wait/exit在实现中是如何影响进程的执行状态的?

- fork函数: 寻找可用进程块,为其分配空间,将父进程的所有资源复制给子进程,父进程和子进程独立运行,通过不同的pid进行标识.
- exec函数: 在当前进程块将有关资源设置为需要调用的进程的内容.
- wait函数: 阻塞父进程,等待子进程的结束并返回子进程状态,实现进程间同步. 故阻塞过程中也需要进行进程间的切换.
- exit函数: 退出当前进程,并寻找下一个可用进程进行切换.

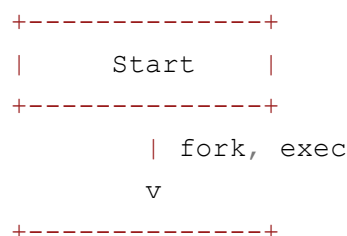
实现中:

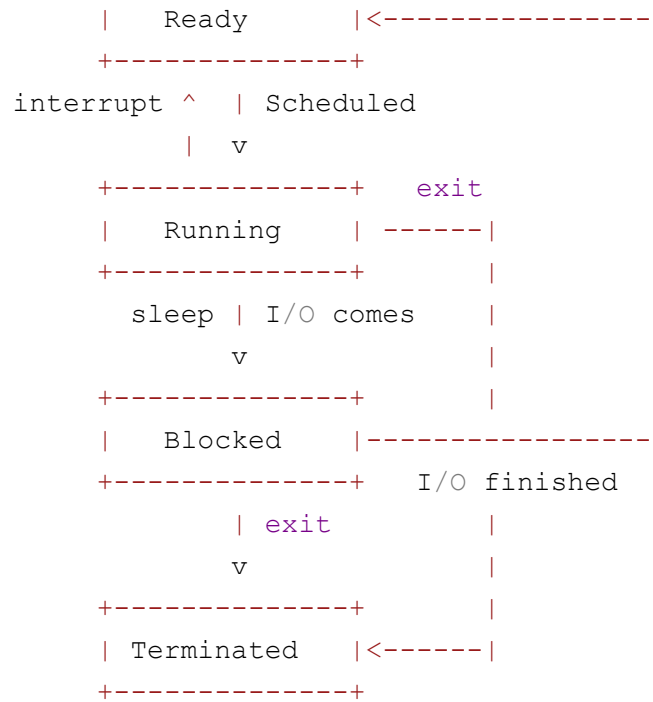
- fork函数: 父进程执行状态不变,子进程执行状态设为STATE_RUNNABLE.
- exec函数: 替换后,不改变当前进程状态,即维持在STATE_RUNNING.
- wait函数: 阻塞父进程过程中,不更改子进程状态,父进程状态设为STATE_BLOCKED. 直到子进程结束,状态为STATE_DEAD时,才将父进程状态转为STATE_RUNNABLE继续执行.
- exit函数: 当前进程状态设为STATE_DEAD,寻找下一个就绪进程执行.

4.3 描述当创建一个用户态进程并加载了应用程序后,CPU是如何让这个应用程序最终在用户态执行起来的. 即这个用户态进程被OS选择占用CPU执行(RUNNING态)到具体执行应用程序第一条指令的整个经过.

- 当创建一个用户态进程并加载了应用程序后,操作系统会将其设置为就绪态,并加入到就绪队列中,此时处于等待CPU执行的状态.
- 操作系统按照调度算法从就绪队列中选择一个进程,将其设置为运行态,加入到CPU的运行队列中. 此时,CPU开始执行进程的代码.
- 若前序执行过程涉及I/O读写,当前进程会被抢占,设置为阻塞态. 直到I/O读写操作结束后,重新设置为就绪态加入就绪队列中,等待操作系统的调度.
- CPU从应用程序的入口开始执行第一条指令.

4.4 请给出一个用户态进程的执行状态生命周期图(包执行状态,执行状态之间的变换关系,以及产生变换的事件或函数调用). (字符方式画即可)





5. 实验心得

本次实验让我了解了计算机系统中不同进程之间的切换, 以及常用的一些进程切换接口. 同时通过手动实现系统调用服务例程以及轮转调度策略, 我对这些系统调用的实现有了更深的理解. 这次实验的难点在于`fork`部分的正确理解与实现. 但最终能顺利完成该实验, 我也收获满满.