OS Lab3 实验报告

姓名：孙文博

学号：201830210

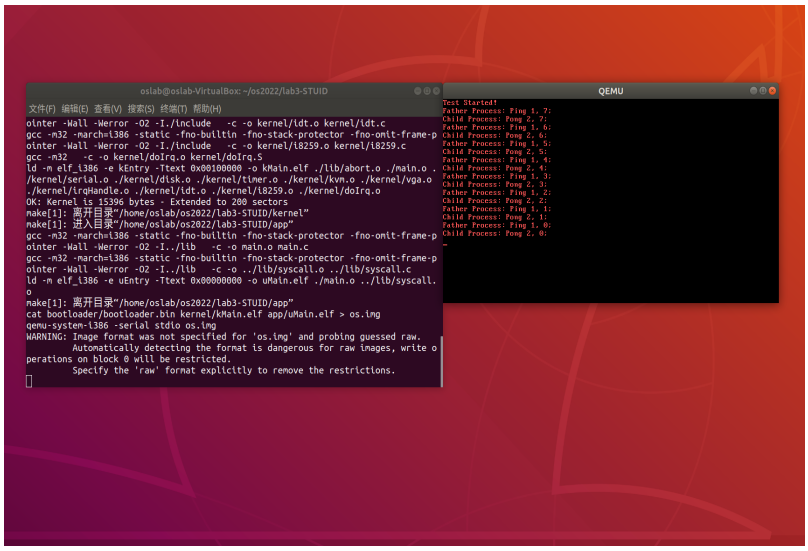
邮箱：201830210@smail.edu.nju.cn

1. 实验进度

完成了实验 `lab3` 的内容及思考题。

2. 实验结果

实现了三个库函数 `fork()`，`exit()`，`sleep()` 及对应的处理例程，通过了测试用例。



3. 实验修改代码

3.1 完成库函数

框架代码中已经给出 `fork` 的库函数，仿照其实现 `sleep` 和 `exit` 库函数：

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

```
    return syscall(SYS_EXIT, 0, 0, 0, 0, 0);
}
```

3.2 时钟中断处理

首先将手册中给的进程切换代码封装成 `ChangeStack()` 函数:

```
void changeStack(){
    // set esp to pcb[current]'s stack:
    uint32_t tempStackTop = pcb[current].stackTop;
    pcb[current].stackTop = pcb[current].prevStackTop;
    tss.esp0 = (uint32_t)&(pcb[current].stackTop);
    asm volatile("movl %0, %%esp" : :"m"(tempStackTop));
    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");
}
```

接着通过 `schedule()` 函数实现进程调度，找到当前运行进程:

```
void schedule(){
    int found_idle = 0;
    for(int i = 1; i < MAX_PCB_NUM; ++i){
        if (computeMod(i)!=0 && pcb[computeMod(i)].state == STATE_RUNNABLE){
            // this needs to be locked to protect from concurrency
            current = computeMod(i);
            found_idle = 1;
            break;
        }
    }
    if (!found_idle) {
        // switch to idle
        current = 0;
    }
    ... //输出current有关信息
    changeStack();
}
```

最后根据时钟中断的逻辑实现 `timeHandle()` 函数:

```
void timerHandle(struct StackFrame *sf) {
    updateSleepTime();
    pcb[current].timeCount++;
```

```
    if(pcb[current].timeCount >= MAX_TIME_COUNT){
        pcb[current].timeCount = 0;
        if (pcb[current].state == STATE_RUNNING){
            pcb[current].state = STATE_RUNNABLE;
        }
        schedule();
    }
}
```

3.3 系统调用例程

有了3.2中实现的时钟中断，接下来只要根据手册中的要求补充 `syscallFork()`, `syscallSleep()`, `syscallExit()` 函数即可，由于篇幅原因仅展示 `syscallFork()` 中核心部分代码：

```
void syscallFork(struct StackFrame *sf) {
    int availPos = -1;
    for(int i = 1; i < MAX_PCB_NUM; ++i){
        if(pcb[i].state == STATE_DEAD){
            availPos = i;
        }
    }
    if(availPos == -1) {
        // FORK FAILED
        pcb[current].regs.eax = -1;
    }
    else {
        // FORK SUCCESSFUL
        enableInterrupt();
        for (int j = 0; j < 0x100000; j++) {
            *(uint8_t *)(j + (availPos+1)*0x100000) = *(uint8_t *)(j +
(current+1)*0x100000);
        }
        disableInterrupt();
        for (int j = 0; j < sizeof(ProcessTable); ++j)
            *((uint8_t *)(&pcb[availPos]) + j) = *((uint8_t *)(&pcb[current]) +
j);

        ... //涉及pcbd内容复制等操作
    }
}
```

---

4. 实验思考题

Exercise1

答：函数 `fork()` 用来创建一个新的进程，该进程几乎是当前进程的一个完全拷贝；函数族 `exec()` 用来启动另外的进程以取代当前运行的进程。

先来看 fork() 的框架代码：

```
void main()
{
   int i;
   if ( fork() == 0 )
   {
      /* 子进程程序 */
        for ( i = 1; i <1000; i ++ )
            printf("This is child process\n");
   }
   else
   {
      /* 父进程程序*/
        for ( i = 1; i <1000; i ++ )
            printf("This is process process\n");
   }
}
```

在这个过程中，fork() 函数启动一个新的进程，这个进程几乎是当前进程的一个拷贝：子进程和父进程使用相同的代码段；子进程复制父进程的堆栈段和数据段。对于父进程，fork() 函数返回了子程序的进程号，而对于子程序，fork() 函数则返回零。

再来看 exec() 等函数:

一个进程一旦调用 exec 类函数，它本身就"死亡"了，系统把代码段替换成新的程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段与堆栈段，唯一留下的，就是进程号，也就是说，对系统而言，还是同一个进程，不过已经是另一个程序了。

如果程序想启动另一程序的执行但自己仍想继续运行的话，可以结合 fork 与 exec 的使用:

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

char command[256];
void main()
{
  int rtn; /*子进程的返回数值*/
  while(1) {
      /* 从终端读取要执行的命令 */
      printf( ">" );
      fgets( command, 256, stdin );
      command[strlen(command)-1] = 0;
      if ( fork() == 0 ) {/* 子进程执行此命令 */
         execlp( command, NULL );
         /* 如果exec函数返回，表明没有正常执行命令，打印错误信息*/
         perror( command );
         exit( errno );
      }
```

```
        else {/* 父进程， 等待子进程结束，并打印子进程的返回值 */
            wait ( &rtn );
            printf( " child process return %d\n", rtn );
        }
    }
}
```

Exercise2

答：

- fork 执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork 函数返回 0，在父进程中，fork 返回新创建子进程的进程 ID。我们可以通过 fork 返回的值来判断当前进程是子进程还是父进程。fork 不会影响当前进程的执行状态，但是会将子进程的状态标记为 RUNNABLE，使得可以在后续的调度中运行起来；

- exec 完成用户进程的创建工作。首先为加载新的执行码做好用户态内存空间清空准备。接下来的一步是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。exec 不会影响当前进程的执行状态，但是会修改当前进程中执行的程序；

- wait 是等待任意子进程的结束通知。wait_pid 函数等待进程 id 号为 pid 的子进程结束通知。这两个函数最终访问 sys_wait 系统调用接口让 ucore 来完成对子进程的最后回收工作。wait 系统调用取决于是否存在可以释放资源（ZOMBIE）的子进程，如果有的话不会发生状态的改变，如果没有的话会将当前进程置为 SLEEPING 态，等待执行了 exit 的子进程将其唤醒；

- exit 会把一个退出码 error_code 传递给 ucore，ucore 通过执行内核函数 do_exit 来完成对当前进程的退出处理，主要工作简单地说就是回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作。exit 会将当前进程的状态修改为 ZOMBIE 态，并且会将父进程唤醒（修改为RUNNABLE），然后主动让出 CPU 使用权；

鸣谢博客：https://www.cnblogs.com/ECJTUACM-873284962

Exercise3

答：在经过调度器占用了 CPU 的资源之后，用户态进程调用了 exec 系统调用，从而转入到了系统调用的处理例程；

在经过了正常的中断处理例程之后，最终控制权转移到了 syscall.c 中的 syscall 函数，然后根据系统调用号转移给了 sys_exec 函数，在该函数中调用了 execve 函数来完成指定应用程序的加载；

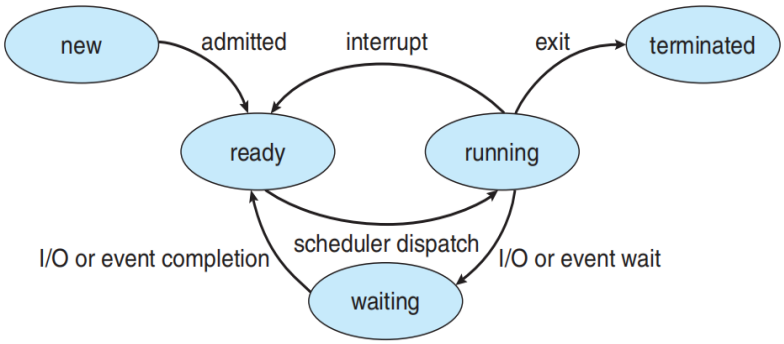
在 execve 中进行了若干设置，包括推出当前进程的页表，换用 kernel 的 PDT 之后，使用 load_icode 函数，完成了对整个用户线程内存空间的初始化，包括堆栈的设置以及将 ELF 可执行文件的加载，之后通过 current->tf 指针修改了当前系统调用的 trapframe，使得最终中断返回的时候能够切换到用户态，并且同时可以正确地将控制权转移到应用程序的入口处；

在完成了 exec 函数之后，进行正常的中断返回的流程，由于中断处理例程的栈上面的 eip 已经被修改成了应用程序的入口处，而 CS 上的 CPL 是用户态，因此 iret 进行中断返回的时候会将堆栈切换到用户的栈，并且完成特权级的切换，并且跳转到要求的应用程序的入口处；

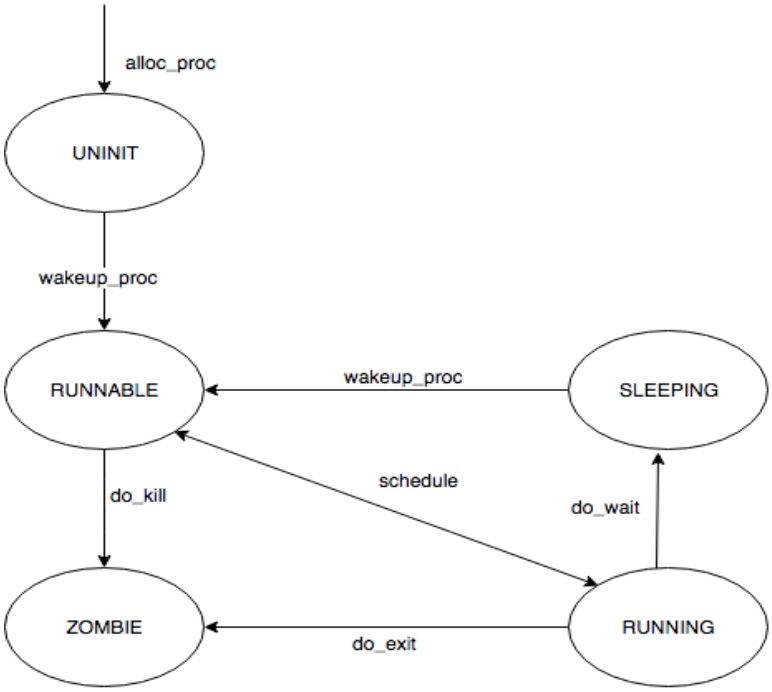
接下来开始具体执行应用程序的第一条指令；

Exercise4

答：用户态进程执行流程：



用户态进程执行状态生命周期图如下：



---

5. 实验心得

本次实验历时三周，实验重点在于对进程切换及时钟中断的理解和具体实现，实验中实现了 `fork()`, `sleep()`, `exit()` 三个系统调用操作，正好也对应了课本2.5章节进程间通信的相关内容，对课本知识进行进一步巩固。

实验主要的challenge在于 `fork` 中子进程对父进程内容的复制过程，细节需要仔细考虑，并通过手册给出的参考资料最终成功实现。另外本次思考题内容较多，回答内容也较多，所以报告篇幅有一点点超标，希望助教哥哥可以理解~

最后感谢老师和助教提供的悉心指导的实验手册，希望之后的实验也可以一帆风顺！ 🤞