

# 操作系统实验报告

学号：201220139

姓名：艾冷州

邮箱：[201220139@smail.nju.edu.cn](mailto:201220139@smail.nju.edu.cn)

日期：2022年4月24日

## 1. Exercises

1. 请把上面的程序，用 gcc 编译，在你的 Linux 上面运行，看看真实结果是啥（有一些细节需要改，请根据实际情况进行更改）。

```
int uEntry(void) {
    int data = 0;
    int ret = fork();
    int i = 8;
    if (ret == 0) {
        data = 2;
        while( i != 0) {
            i--;
            printf("Child Process: Pong %d, %d;\n", data, i);
            sleep(1);
        }
        exit();
    }
    else if (ret != -1) {
        data = 1;
        while( i != 0) {
            i--;
            printf("Father Process: Ping %d, %d;\n", data, i);
            sleep(1);
        }
        exit();
    }
    return 0;
}
```

答：运行结果如图：

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

```
Father Process: Ping 1, 2;  
Father Process: Ping 1, 1;  
Child Process: Pong 2, 1;  
Father Process: Ping 1, 0;  
Child Process: Pong 2, 0;
```

运行此程序时，可见程序每 1 秒在控制台上打印出 `Father Process: Ping %d, %d;` 和 `Child Process: Pong %d, %d;` 的字符串。但打印这两个字符串的顺序并不确定。此外，每次运行此程序时，打印字符串的顺序都不相同。

2. 请简单说说，如果我们想做虚拟内存管理，可以如何进行设计（比如哪些数据结构，如何管理内存）？

答：(1) 在内核区域分配一块空间，大小为整个内存的空间大小的  $1/4K$ 。其中每个字节映射到内存的某一 4K 空间，用于保存空间的分配情况（如分配给内核、分配给用户堆区、分配给用户栈区等）；(2) 在内核区域同样分配一块空间，用于保存进程号及其对应的代码段和栈区的位置；堆区的位置信息则用链表形式存储，存在堆栈区。在进程结束销毁进程时，先销毁进程自己的空间，再销毁内核用于保存该进程的堆区地址的空间。

3. 我们考虑这样一个问题：假如我们的系统中预留了 100 个进程，系统中运行了 50 个进程，其中某些结束了运行。这时我们又有一个进程想要开始运行（即需要分配给它一个空闲的 PCB），那么如何能够以  $O(1)$  的时间和  $O(n)$  的空间快速地找到这样一个空闲 PCB 呢？

答：用队列的形式储存空闲 PCB 的 index。每次分配空闲 PCB 时，将队头的 index 取出即可。

4. 请你说说，为什么不同用户进程需要对应不同的内核堆栈？

答：因为从用户态进入内核态时，需要在内核堆栈中保存现场信息。而每个用户进程对应不同用户堆栈，且保存现场时它们的现场信息都不同。因此当恢复现场时，要想直接取出对应进程的现场信息，只能为每个进程分配对应的内核堆栈。

5. `stackTop` 有什么用？为什么一些地方要取地址赋值给 `stackTop`？

答：`stackTop` 保存内核栈的栈顶。因为在切换进程时需要保存对应的栈顶地址。

6. 请说说在中断嵌套发生时，系统是如何运行的？（把关键的地方说一下即可，简答）

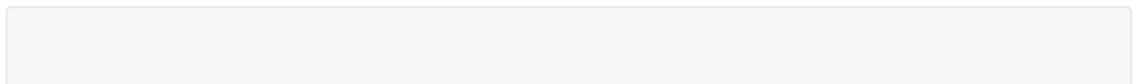
答：(1) 保存被中断程序的现场；(2) 分析中断源，判断中断原因，当同时有多个中断同时请求的时候还要考虑中断的优先级；(3) 转去执行相应的处理程序；(4) 恢复被中断程序现场，继续执行被中断程序。

7. 线程为什么要以函数为粒度来执行？

答：更大的粒度是程序，而程序一般不需要共享全局变量和堆区；更小的粒度是指令，而执行单条指令耗费的时间远远小于创建线程的时间。

8. 请用 `fork`, `sleep`, `exit` 自行编写一些并发程序，运行一下，贴上你的截图。（自行理解，不用解释含义，帮助理解这三个函数）

答：程序如下：



```

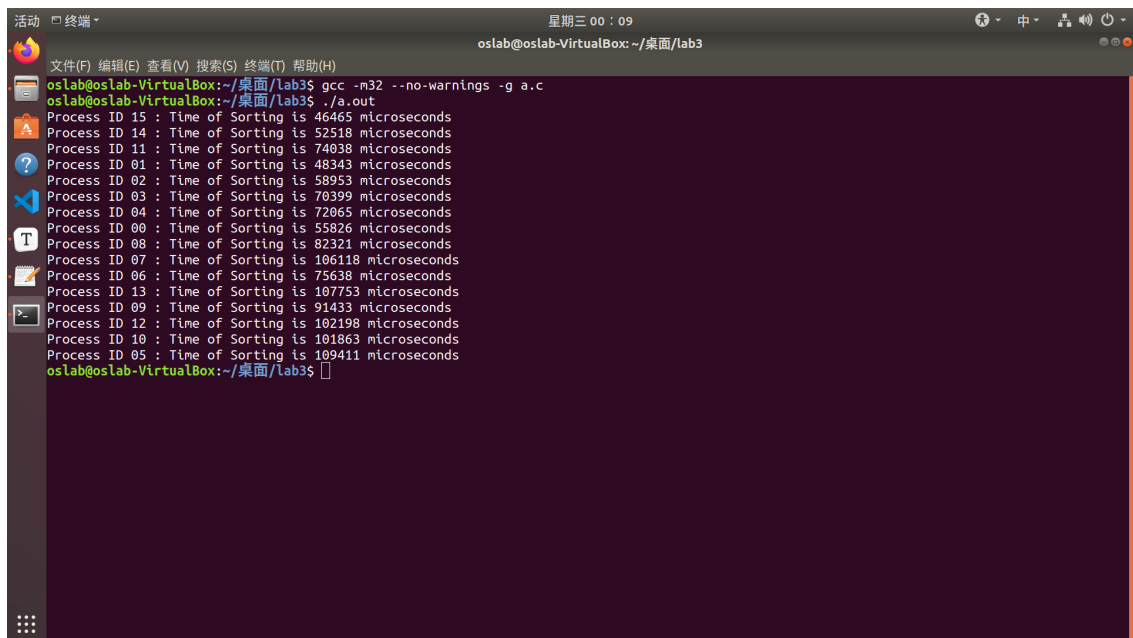
// 此程序的功能为，创建 16 个并发进程，同时对 65536 个随机数进行排序操作，并获得不同进程的排序时间。
#include<unistd.h>
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<assert.h>
#include<time.h>

int cmp_int(const void* _a , const void* _b) {
    int* a = (int*)_a;
    int* b = (int*)_b;
    return *a - *b;
}

int main() {
    int p0 = fork();
    int p1 = fork();
    int p2 = fork();
    int p3 = fork();
    int pid = (p0 != 0) + ((p1 != 0) << 1) + ((p2 != 0) << 2) + ((p3 != 0) << 3);
    srand(pid);
    int* arr = (int*)malloc(65536 * sizeof(int));
    for (int i = 0; i < 65536; i++) {
        arr[i] = rand();
    }
    struct timeval start, end;
    gettimeofday(&start, NULL);
    qsort(arr, 65536, sizeof(int), cmp_int);
    gettimeofday(&end, NULL);
    int diffTime = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec);
    printf("Process ID %02d : Time of Sorting is %d microseconds\n", pid, diffTime);
    sleep(1);
    exit(0);
    return 0;
}

```

运行截图如下：



9. 请问, 我使用 `loadelf` 把程序装载到当前用户程序的地址空间, 那不会把我 `loadelf` 函数的代码覆盖掉吗?

答: 不会。因为 `loadelf` 程序在内核段, 而被装载的程序在用户段。

## 2. Tasks

1. 这一部分算是对实验 2 的复习, 我们在 `lab3/lib/syscall.c` 中留下了三个库函数待完成, 你需要调用 `syscall` 完善库函数。

- 完成 `fork`
- 完成 `exec`
- 完成 `sleep`
- 完成 `exit`

2. 完成时钟中断处理函数。

3. 完成系统调用处理函数。

- `syscallFork`
- `syscallExec`
- `syscallSleep`
- `syscallExit`

## 3. Challenges

5. 你是否可以完善你的 `exec`, 第三个参数接受变长参数, 用来模拟带有参数程序执行。

举个例子, 在 `shell` 里面输入 `cat a.txt b.txt`, 实际上 `shell` 会 `fork` 出一个新的 `shell` (假设称为 `shell0`), 并执行 `exec("cat", "a.txt", "b.txt")` 运行 `cat` 程序, 覆盖 `shell0` 的进程。

不必完全参照 `Unix`, 可以有自己的思考与设计。

答: 如下:

```
int exec(uint32_t sec_start, uint32_t sec_num, uint32_t argc, ...) {
    /*TODO:call syscall*/
}
```

```

    return syscall(SYS_EXEC, sec_start, sec_num, argc, (uint32_t)&argc + 4,
0);
}

void syscallExec(struct StackFrame *sf) {
    // TODO 完成exec
    // hint: 用loadelf, 已经封装好了
    uint32_t entry = 0;
    uint32_t secstart = sf->ecx;
    uint32_t secnum = sf->edx;
    uint32_t argc = sf->ebx;
    char** argv = (void*)sf->esi;
    pcb[current].regs.esp = 0x000FFFFC;
    uint32_t base = 0x00100000 * (current + 1);
    loadelf(secstart, secnum, base, &entry);
    pcb[current].regs.eip = entry;
    // 将变长参数压栈
    pcb[current].regs.esp -= argc * 4;
    for(int i = 0; i < argc; i++) {
        *(char*)(base + pcb[current].regs.esp + i) = argv[i];
    }
    // 将参数个数压栈
    pcb[current].regs.esp -= 4;
    *(char*)(base + pcb[current].regs.esp) = argc;
}

```