

# **操作系统 实验 3**

## **进线程切换**

**白晋斌**

**171860607**

**810594956@qq.com**

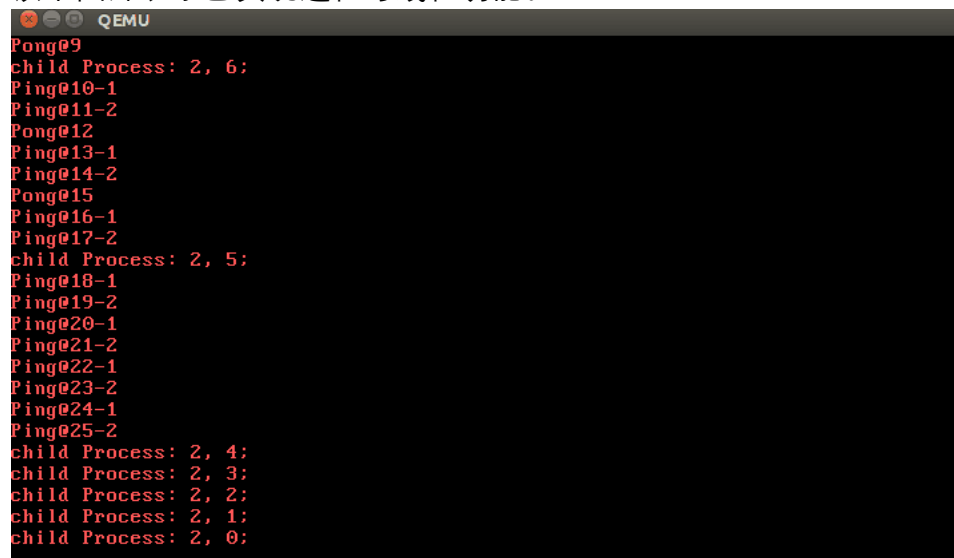
## 目录

一、重点，精华都在这里.....	3
1.基本功能 .....	3
2.拓展功能 .....	3
二、实验要求.....	5
1.1. 实现进程切换机制.....	5
1.2. 实现 FORK、SLEEP、EXIT 系统调用 .....	5
1.3. 实现 pthread 库 .....	5
三、实验过程.....	5
0.阅读框架代码.....	5
1. 进程部分 .....	5
1.0 调度算法 .....	5
1.1 补充完成 timerHandle 函数 .....	5
1.2 补充完成 fork 函数.....	5
1.3 补充完成 sleep 和 exit 函数.....	5
1.4 测试环节 .....	5
2.线程部分 .....	6
2.0.自学 .....	6
2.1 调度算法 .....	6
2.2 线程切换过程的实现 .....	6
2.3 补充完成 create、exit、join、yield 函数 .....	7
2.4 测试环节 .....	7
四、拓展功能.....	7
1.线程部分 .....	7
2.进程部分 .....	7
五、友情鸣谢.....	7

## 一、重点，精华都在这里

### 1.基本功能

以下图片表示已实现进程与线程功能。



```
QEMU
Pong@9
child Process: 2, 6:
Ping@10-1
Ping@11-2
Pong@12
Ping@13-1
Ping@14-2
Pong@15
Ping@16-1
Ping@17-2
child Process: 2, 5:
Ping@18-1
Ping@19-2
Ping@20-1
Ping@21-2
Ping@22-1
Ping@23-2
Ping@24-1
Ping@25-2
child Process: 2, 4:
child Process: 2, 3:
child Process: 2, 2:
child Process: 2, 1:
child Process: 2, 0:
```

### 2.拓展功能

新线程的属性（create 函数第二个参数）、线程返回值（join 函数第二个参数）功能均已实现。第二部分即选做任务。

#### 2.2.8. pthread\_exit 库函数

```
void pthread_exit(void *retval);
```

pthread\_exit 函数会结束当前线程并通过 **retval** 返回值，该返回值由同一进程下 **join** 当前线程的线程获得。retval 返回值在本次实验中不强制要求实现，做为选做。具体参见 [man page](#)。

实际效果如下图：

测试 1:

```
void * mytest(void *arg){
    //int i;
    int a=10086;
    pthread_exit(&a);
    return NULL;
}

int main(void) {
    pthread_t test_ID;
    //int aa=1;
    void *m;
    pthread_create(&test_ID, NULL, mytest, NULL);
    pthread_join(test_ID, &m);
    printf("my test=%d",*(int *)m);
    int a = 1, b = 2;
```

```
QEMU
my test=10086Ping@1-1
child Process: 2, 7;
Ping@2-2
Ping@3
Ping@4-1
fuPing@5-2
Ping@6
fuPing@7-1
Ping@8-2
child Process: 2, 6;
unPing@9
Ping@10-1
Ping@11-2
Ping@12
Ping@13-1
Ping@14-2
Ping@15
child Process: 2, 5;
Ping@16-1
Ping@17-2
Ping@18-1
Ping@19-2
Ping@20-1
Ping@21-2
_
```

测试 2:

```
void * mytest(void *arg){
    //int i;
    //int a=10086;
    pthread_exit(arg);
    return NULL;
}

int main(void) {
    pthread_t test_ID;
    int aa=1;
    void *m;
    pthread_create(&test_ID, NULL, mytest, &aa);
    pthread_join(test_ID, &m);
    printf("mv test=%d".*(int *)m);
}

QEMU
my test=1Ping@1-1
child Process: 2, 7;
Ping@2-2
Ping@3
Ping@4-1
Ping@5-2
Ping@6
Ping@7-1
Ping@8-2
child Process: 2, 6;
Ping@9
Ping@10-1
Ping@11-2
Ping@12
Ping@13-1
Ping@14-2
Ping@15
_
```

这证明我们的实现是正确的。

## 二、实验要求

本实验通过实现一个简单的任务调度，介绍基于时间中断进行进程切换以及纯用户态的非抢占式的线程切换完成任务调度的全过程。

- 1.1. 实现进程切换机制
- 1.2. 实现 FORK、SLEEP、EXIT 系统调用
- 1.3. 实现 pthread 库

## 三、实验过程

### 0. 阅读框架代码

#### 1. 进程部分

##### 1.0 调度算法

遍历除 0 号进程和当前进程外的所有进程，如果找到 runnable，调度即可。否则查看当前进程是否为 runnable，是的话执行当前进程，否则执行 0 号进程。这么做也是为了使各进程调度几率相近，防止发生饥饿现象。

##### 1.1 补充完成 timerHandle 函数

这一块主要通过 putchar 来 debug。代码逻辑如下。

遍历所有线程，对 blocked 线程的 sleeptime-1；如果 sleeptime==0 则转换为 runnable。

对当前线程 timecount-1，（我这里默认初始状态是 max，减到 0 转换成 runnable）。

如果当前线程 timecount==0，转换成 runnable 然后执行调度算法。

##### 1.2 补充完成 fork 函数

当我们通过 putchar 发现已经进入 1 号线程，就可以写 fork 了。

找一个 dead 的线程，复制用户空间，分别设置 stacktop, pid, state, timecount, sleeptime, 寄存器（分开设置不能直接赋值），之后通过 eax 返回即可。

##### 1.3 补充完成 sleep 和 exit 函数

这两个函数很简单。sleep 是修改 state 和 sleeptime，exit 只需要修改 state。两个函数之后执行调度算法即可。

##### 1.4 测试环节

经过对前述环节没日没夜的 debug，出现如下输出即认为第一部分完成。

```
QEMU
child Process: 2, 7;
child Process: 2, 6;
child Process: 2, 5;
child Process: 2, 4;
child Process: 2, 3;
child Process: 2, 2;
child Process: 2, 1;
child Process: 2, 0;
-
```

## 2.线程部分

### 2.0.自学

学习 create、exit、join、yield 的功能和使用方法。

学习内联汇编的相关写法。

### 2.1 调度算法

遍历所有线程，如果某线程是其他线程的爸爸，暂且忽略它。得到 runable 的可用线程。

### 2.2 线程切换过程的实现

保存当前现场。

```
asm volatile("movl %%esp,%0":"=m"(tcb[current].cont.esp));
asm volatile("movl %%ebp,%0":"=m"(tcb[current].cont.ebp));
```

eip 则通过调用一个临时函数，在函数内执行

```
asm volatile("movl 4(%%ebp), %0":"=r"(tcb[current].cont.eip));
```

这主要是因为我们发现如果直接将 eip 赋值给内存，系统会报错。

上网查询资料得知，eax、ebx、ecx、edx 等其他寄存器不需要保存亦可。

切换线程，修改状态值及 current。

恢复新线程现场。之后跳转执行即可。

```
asm volatile("movl %0, %%eax"::"r"(eip));
asm volatile("movl %0, %%esp"::"m"(esp));
asm volatile("movl %0, %%ebp"::"m"(ebp));

asm volatile("jmp *%eax");
```

## 2.3 补充完成 create、exit、join、yield 函数

create 函数通过寻找 dead 线程，之后给新线程初始化后，构造堆栈。因为 esp+8 是第一个参数的位置，我们只需要提前放到相应的位置即可。之后调度执行。

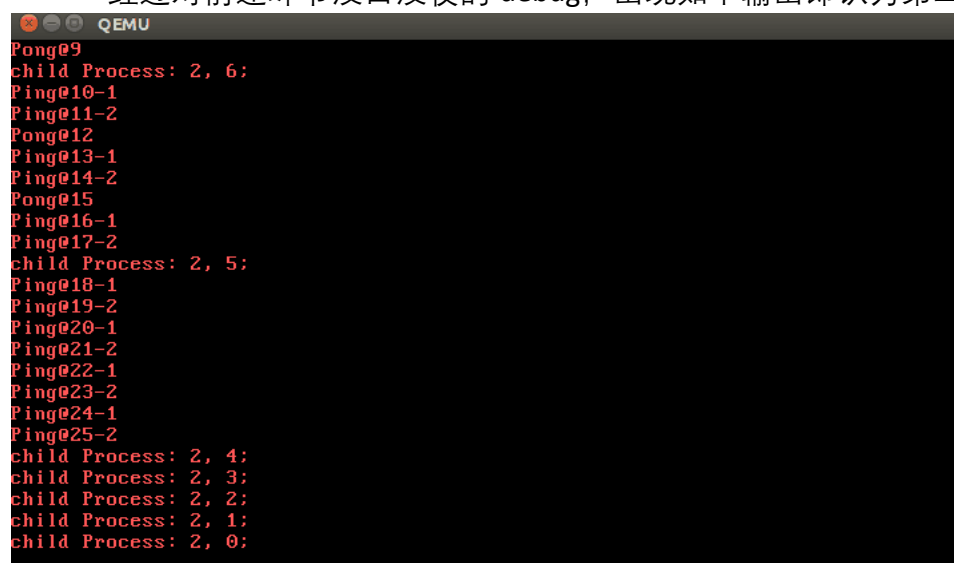
（一说 create 无需调度，这取决于操作系统，例如 creation option 参数即指建立后的线程是挂起还是直接运行。故为了提高我们的执行效率，避免 0 号线程空转，则选取创建后立即执行的策略）。

后在实现拓展功能时，发现 create 就执行容易产生时序问题，故这里改成 create 仅创建不调度执行。

exit、join、yield 主要就是改状态然后调度。特别注意的是，join 要额外设置一下 joinid。

## 2.4 测试环节

经过对前述环节没日没夜的 debug，出现如下输出即认为第二部分完成。



```
QEMU
Ping@9
child Process: 2, 6;
Ping@10-1
Ping@11-2
Ping@12
Ping@13-1
Ping@14-2
Ping@15
Ping@16-1
Ping@17-2
child Process: 2, 5;
Ping@18-1
Ping@19-2
Ping@20-1
Ping@21-2
Ping@22-1
Ping@23-2
Ping@24-1
Ping@25-2
child Process: 2, 4;
child Process: 2, 3;
child Process: 2, 2;
child Process: 2, 1;
child Process: 2, 0;
```

# 四、拓展功能

## 1.线程部分

对调度算法进行了优化，不单单是无脑遍历，而是企图寻找其他可调度进程，不行的话看自己能不能上，能上则自己上，不行的话 0 号线程接盘。

## 2.进程部分

新线程的属性（create 函数第二个参数）、线程返回值（join 函数第二个参数）功能均已实现。

# 五、友情鸣谢

感谢老师助教和各位群友鼎力相助！！