

Lab3 进线程切换

俞星凯 171830635

2651904866@qq.com

实验目的

1. 实现一个简单的任务调度。
2. 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。

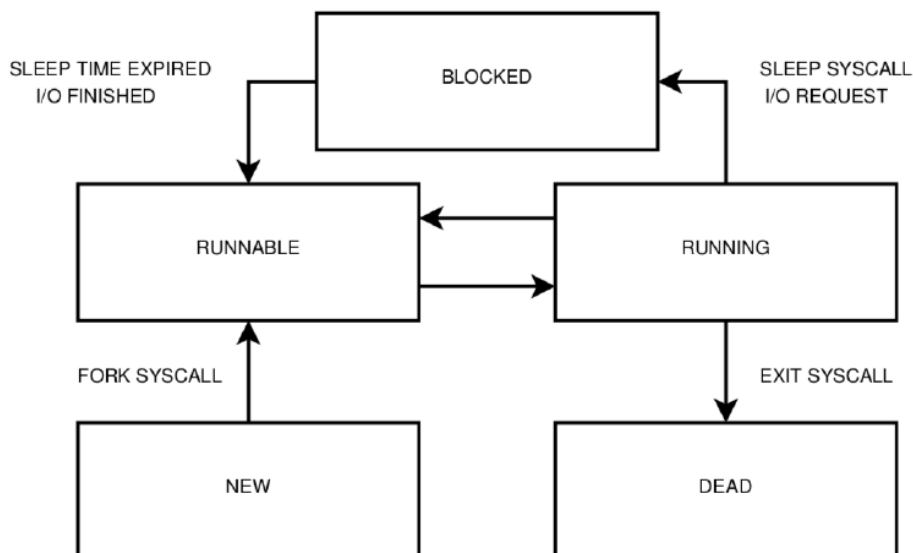
实验内容

1. 内核：实现进程切换机制，并提供系统调用 `fork`、`sleep`、`exit`。
2. 库：对上述系统调用进行封装；实现一个用户态的线程库，完成 `pthread_create`、`pthread_join`、`pthread_yield`、`pthread_exit` 等接口。
3. 用户：对上述库函数进行测试。

背景知识

1. 进程与线程

进程为操作系统资源分配的单位，每个进程都有独立的地址空间（代码段、数据段），独立的堆栈，独立的进程控制块；线程作为任务调度的基本单位，与进程的唯一区别在于其地址空间并非独立，而是与其他线程共享。以下为一个广义的进程（包括进程与线程）生命周期中的状态转换图

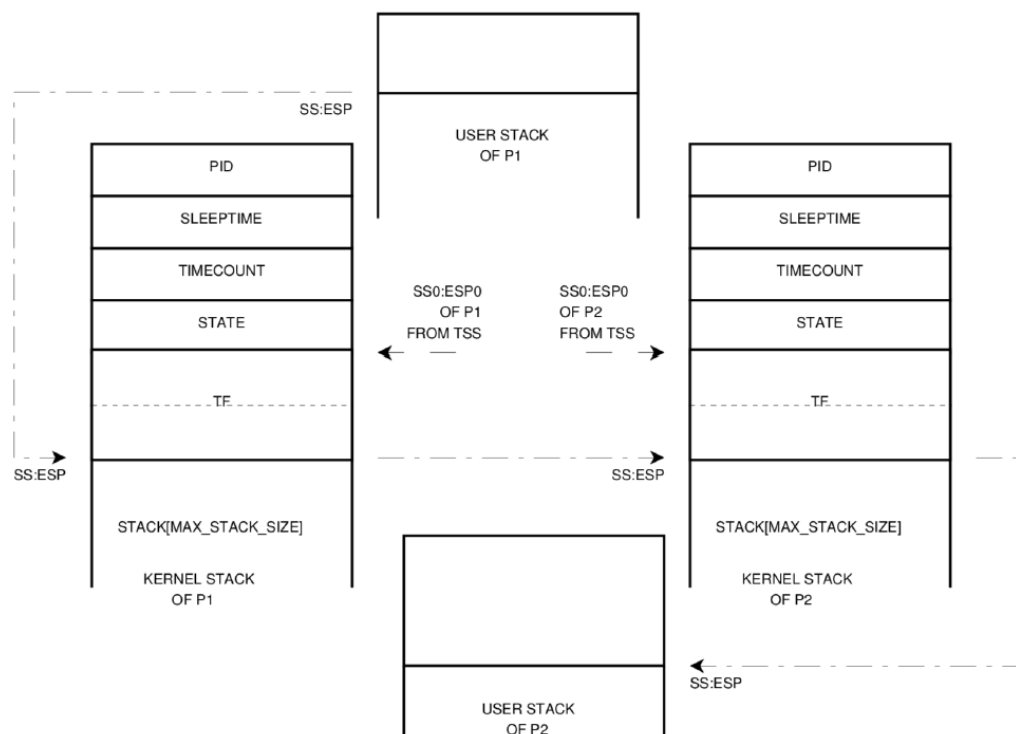


- (1) 进程由其父进程利用 **FORK** 系统调用创建，则该进程进入 **RUNNABLE** 状态
- (2) 时间中断到来，**RUNNABLE** 状态的进程被切换到，则该进程进入 **RUNNING** 状态

- (3) 时间中断到来，**RUNNING** 状态的进程处理时间片耗尽，则该进程进入 **RUNNABLE** 状态
- (4) **RUNNING** 状态的进程利用 **SLEEP** 系统调用主动阻塞；或利用系统调用等待硬件 I/O，则该进程进入 **BLOCKED** 状态
- (5) 时间中断到来，**BLOCKED** 状态的进程的 **SLEEP** 时间片耗尽；或外部硬件中断表明 I/O 完成，则该进程进入 **RUNNABLE** 状态
- (6) **RUNNING** 状态的进程利用 **EXIT** 系统调用主动销毁，则该进程进入 **DEAD** 状态

2. 进程切换与堆栈切换

下图为产生时间中断后，时间中断处理程序为两个用户态进程 **P1**、**P2** 进行进程切换的过程中，堆栈切换的简单图示



- (1) 进程 **P1** 在用户态执行，8253 可编程计时器产生时间中断
- (2) 依据 TSS 中记录的进程 **P1** 的 **SS0:EPS0**，从 **P1** 的用户态堆栈切换至 **P1** 的内核堆栈，并将 **P1** 的现场信息压入内核堆栈中，跳转执行时间中断处理程序
- (3) 进程 **P1** 的处理时间片耗尽，切换至就绪状态的进程 **P2**，并从当前 **P1** 的内核堆栈切换至 **P2** 的内核堆栈
- (4) 从进程 **P2** 的内核堆栈中弹出 **P2** 的现场信息，切换至 **P2** 的用户态堆栈，从时间中断处理程序返回执行 **P2**

实验过程

1. 启动阶段

与 Lab2 类似，具体流程如下：

- (1) **Bootloader** 从实模式进入保护模式,加载内核至内存，并跳转执行

- (2) 内核初始化 IDT, 初始化 GDT, 初始化 TSS, 初始化串口, 初始化 8259A, ...
- (3) 启动时钟源, 开启时钟中断处理
- (4) 加载用户程序至内存
- (5) 初始化内核 IDLE 线程的进程控制块 (Process Control Block), 初始化用户程序的进程控制块
- (6) 切换至用户程序的内核堆栈, 弹出用户程序的现场信息, 返回用户态执行用户程序

2. 进程

在本次实验中, 进程相关的数据结构是 ProcessTable, 具体内容如下:

```
struct ProcessTable {
    uint32_t stack[MAX_STACK_SIZE];
    struct StackFrame regs;
    uint32_t stackTop;
    uint32_t prevStackTop;
    int state;
    int timeCount;
    int sleepTime;
    uint32_t pid;
    char name[32];
};
```

其中 stack[MAX_STACK_SIZE] 为每个进程独立的内核堆栈, regs 记录每个进程从用户态陷入内核态时压入内核堆栈的信息 (即所有寄存器信息、中断号、Error Code), state 记录每个进程的状态 (即 RUNNING、RUNNABLE、BLOCKED、DEAD 等), timeCount 记录每个进程的处理 (RUNNING) 时间片, sleepTime 记录每个进程阻塞 (BLOCKED) 的时间片, pid 记录每个进程的进程号。

(1) timerHandle

```
void timerHandle(struct StackFrame *sf) {
    int i;
    uint32_t tmpStackTop;
    // make blocked processes sleep time -1, sleep time to 0, re-run
    for (i = 0; i < MAX_PCB_NUM; ++i)
        if (pcb[i].state == STATE_BLOCKED)
            if (--pcb[i].sleepTime == 0)
                pcb[i].state = STATE_RUNNABLE;
    // time count not max, process continue
    pcb[current].timeCount++;
    // else switch to another process
    if (pcb[current].timeCount > MAX_TIME_COUNT) {
        pcb[current].state = STATE_RUNNABLE;
        pcb[current].timeCount = 0;
        for (i = (current + 1) % MAX_PCB_NUM; i != current; i = (i + 1) % MAX_PCB_NUM)
            if (pcb[i].state == STATE_RUNNABLE)
                break;
        current = i;
        pcb[current].state = STATE_RUNNING;
    }
    /* echo pid of selected process */
    putchar('T'); putchar('i'); putchar('m'); putchar('e'); putchar('r');
    putchar('0' + pcb[current].pid); putchar('\n');
    /* XXX recover stackTop of selected process */
    tmpStackTop = pcb[current].stackTop;
    pcb[current].stackTop = pcb[current].prevStackTop;
    // setting tss for user process
    tss.esp0 = pcb[current].stackTop;
    tss.ss0 = KSEL(SEG_KDATA);
}
```

```

// switch kernel stack
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 $0, %esp");
asm volatile("iret");
}

```

当时钟中断到来，先将每个 BLOCKED 状态进程的 sleepTime 减一，如果到达 0，则可以进入 RUNNABLE 状态。接着将当前进程的 timeCount 加一，如果超过最大可运行时间，则需要对进程和堆栈切换。切换的步骤是，在 pcb 包含的进程中寻找下一个处于 RUNNABLE 状态的进程，设置为 RUNNING 状态，然后将 tss 的 esp 变成该进程的 stackTop。在 stackTop 所指内存地址之上是以下内容。

```

struct StackFrame {
    uint32_t gs, fs, es, ds;
    uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
    uint32_t irq, error;
    uint32_t eip, cs, eflags, esp, ss;
};

```

所以，利用 pop 汇编代码来还原 gs、fs、es、ds 等段寄存器，然后用 popa 还原各个通用寄存器，接着就由硬件完成，硬件会弹出 irq 和 error，并将栈中的 eip、cs、eflags、esp、ss 赋值给对应寄存器，这就完成了进程与堆栈切换。

(2) syscallFork

FORK 系统调用用于创建子进程，内核需要为子进程分配一块独立的内存，将父进程的地址空间、用户态堆栈完全拷贝至子进程的内存中，并为子进程分配独立的进程控制块，完成对子进程的进程控制块的设置。若子进程创建成功，则对于父进程，该系统调用的返回值为子进程的 pid，对于子进程，其返回值为 0；若子进程创建失败，该系统调用的返回值为-1。

```

void syscallFork(struct StackFrame *sf) {
    // find empty pcb
    int i, j;
    for (i = 0; i < MAX_PCB_NUM; i++) {
        if (pcb[i].state == STATE_DEAD)
            break;
    }
    if (i != MAX_PCB_NUM) {
        /* XXX copy userspace
        XXX enable interrupt */
        enableInterrupt();
        for (j = 0; j < 0x100000; j++) {
            *((uint8_t *) (j + (i+1)*0x100000)) = *((uint8_t *) (j + (current+1)*0x100000));
            //asm volatile("int $0x20"); //XXX Testing irqTimer during syscall
        }
        /* XXX disable interrupt */
        disableInterrupt();
        /* XXX set pcb
        XXX pcb[i]=pcb[current] doesn't work */
        // copy kernel stack
        for (j = 0; j < sizeof(ProcessTable); ++j)
            *((uint8_t *) (&pcb[i]) + j) = *((uint8_t *) (&pcb[current]) + j);
        pcb[i].stackTop = (uint32_t) &(pcb[i].regs);
        pcb[i].prevStackTop = (uint32_t) &(pcb[i].stackTop);
        pcb[i].state = STATE_RUNNABLE;
        pcb[i].timeCount = 0;
        pcb[i].sleepTime = 0;
        pcb[i].pid = i;
        /* XXX set regs */
        pcb[i].regs.ss = USEL(2+2*i);
        pcb[i].regs.cs = USEL(1+2*i);
    }
}

```

```

    pcb[i].regs.ds = USEL(2+2*i);
    pcb[i].regs.es = USEL(2+2*i);
    pcb[i].regs.fs = USEL(2+2*i);
    pcb[i].regs.gs = USEL(2+2*i);
    /* XXX set return value */
    pcb[i].regs.eax = 0;
    pcb[current].regs.eax = i;
    putchar('F');putchar('o');putchar('r');putchar('k');
    putchar('0' + pcb[i].pid);putchar('\n');
}
else {
    pcb[current].regs.eax = -1;
}
// return to user space
return;
}

```

首先查找 `pcb` 中是否有可用未分配的进程，即处于 `DEAD` 态的进程，若不存在则创建子进程失败。否则，将核心栈和 `ProcessTable` 全部复制，然后将栈指针指向核心栈顶，设置 `state`、`timeCount`、`sleepCount` 和各个段寄存器。最后是设置 `eax`，这里需要注意的是，对于父进程，即 `current`，将 `eax` 设置为子进程号，对于子进程，即 `i`，将 `eax` 设置为 0。

(3) `syscallSleep` 和 `syscallExit`

`SLEEP` 系统调用用于进程主动阻塞自身，内核需要将该进程由 `RUNNING` 状态转换为 `BLOCKED` 状态，设置该进程的 `SLEEP` 时间片，并切换运行其他 `RUNNABLE` 状态的进程。

`EXIT` 系统调用用于进程主动销毁自身，内核需要将该进程由 `RUNNING` 状态转换为 `DEAD` 状态，回收分配给该进程的内存、进程控制块等资源，并切换运行其他 `RUNNABLE` 状态的进程。

在本次实验中，这两个函数实现是类似的，改变当前进程的 `state`，随后查找一个 `RUNNABLE` 状态的进程并完成进程和堆栈切换，切换的步骤在 `timerHandle` 中已经给出。

3. 线程

在本次实验中，进程相关的数据结构是 `ThreadTable`，具体内容如下：

```

struct ThreadTable {
    uint32_t stack[MAX_STACK_SIZE];
    struct Context cont;
    uint32_t retPoint;
    uint32_t pthArg;
    uint32_t stackTop;
    int state;
    uint32_t pthid;
    uint32_t joinid;
};

```

其中，`stack` 是线程栈，用来存放局部变量，`cont` 储存各通用寄存器，`state` 表示线程状态，其他参数在本次实验中不使用。

(1) `pthread_create`

`pthread_create` 调用成功会返回 0；出错会返回错误号，并且 `*thread` 的内容未定义；本实验要求出错返回 -1。

```

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
{
    int i;
    for (i = 0; i < MAX_TCB_NUM; ++i)
        if (tcb[i].state == STATE_DEAD)
            break;
    if (i == MAX_TCB_NUM)
        return -1;
    *thread = i;
    tcb[i].stack[MAX_STACK_SIZE - 1] = (uint32_t)arg;
    tcb[i].cont.esp = (uint32_t)&tcb[i].stack[MAX_STACK_SIZE - 2];
    tcb[i].cont.eip = (uint32_t)start_routine;
    tcb[i].pthArg = (uint32_t)arg;
    tcb[i].state = STATE_RUNNABLE;
    tcb[i].pthid = i;
    return 0;
}

```

首先查找一个可用的 tcb，即处于 DEAD 状态的 tcb，然后设置 esp、eip、pthid、state 等，其他不必要设置。这里需要注意的是，将 arg 放入 stack 最高处作为参数，并设置 esp 为 stack 的第二高位，这样当下次运行该线程时，在汇编代码中，先是执行 push %esp 和 movl %esp,%ebp，于是 arg 就处于 ebp+8 的地址，随后便可以在线程栈中执行代码。

(2) pthread_yield

pthread_yield 函数会使得调用此函数的线程让出 CPU。实验要求 pthread_yield 调用成功返回 0；出错返回-1。实际测试只考虑调用成功的情况。

```

int pthread_yield(void)
{
    asm volatile("movl 4(%%ebp), %0":"=r"(tcb[current].cont.eip));
    asm volatile("pusha");
    asm volatile("popl %0":"=m"(tcb[current].cont.edi));
    asm volatile("popl %0":"=m"(tcb[current].cont.esi));
    asm volatile("popl %0":"=m"(tcb[current].cont.ebp));
    asm volatile("popl %0":"=m"(tcb[current].cont.esp));
    asm volatile("popl %0":"=m"(tcb[current].cont.ebx));
    asm volatile("popl %0":"=m"(tcb[current].cont.edx));
    asm volatile("popl %0":"=m"(tcb[current].cont.ecx));
    asm volatile("popl %0":"=m"(tcb[current].cont.eax));
    asm volatile("movl (%%ebp), %0":"=r"(tcb[current].cont.ebp));
    asm volatile("leal 8(%%ebp), %0":"=r"(tcb[current].cont.esp));
    printf("");
    tcb[current].state = STATE_RUNNABLE;
    int i;
    for (i = (current+1)%MAX_TCB_NUM; i != current; i = (i+1)%MAX_TCB_NUM)
        if (i != 0 && tcb[i].state == STATE_RUNNABLE)
            break;
    current = i;
    tcb[i].state = STATE_RUNNING;
    //printf("");
    asm volatile("movl %0, %%esp":"=r"(&tcb[current].cont.edi));
    asm volatile("popa");
    asm volatile("movl %0, %%esp":"=m"(tcb[current].cont.esp));
    asm volatile("jmp *%0":"=m"(tcb[current].cont.eip));
    printf("");
    return 0;
}

```

第一步是保存现场信息，在内存地址 ebp+4 处存放的是 yeild 函数的返回地址，将它赋值给当前 tcb 的 eip，接着通过 pusha 压入各个通用寄存器，一个一个 pop 到 tcb 的 cont 中，需要注意此处对 cont 中 ebp 和 esp 的设置都是无效的。由于在 yeild 的汇编代码中首先执行的两条语句是 push %esp 和 movl %esp,%ebp，所

以把 `cont` 中的 `ebp` 设置成内存地址 `ebp` 中的内容，即 `yield` 调用前寄存器 `ebp` 的值，把 `cont` 中的 `esp` 设置成寄存器 `ebp+8`，即 `yield` 调用前寄存器 `esp` 的值，这样做的目的是伪造一种当前线程已经执行 `yield` 的“假象”，下次调度该线程时便可以从 `yield` 函数的下一条指令执行。

第二步是查找一个处于 `RUNNABLE` 状态，把当前线程设置成 `RUNNABLE`，被选中进程设置为 `RUNNING`，然后 `current` 赋值为被选中线程。

第三步是恢复被选中线程信息，这里通过把 `esp` 对齐到 `cont` 的最低处，然后利用 `popa` 恢复各个通用寄存器，再对 `esp` 进行一次赋值(因为 `popa` 会舍弃弹出的 `esp`)，最后使用 `jmp` 跳转执行被选中线程。

(3) `pthread_join` 和 `pthread_exit`

`pthread_join` 函数会等待 `thread` 指向的线程结束。实验要求 `pthread_join` 调用成功返回 0；出错返回 -1。实际测试只考虑调用成功的情况。

`pthread_exit` 函数会结束当前线程。

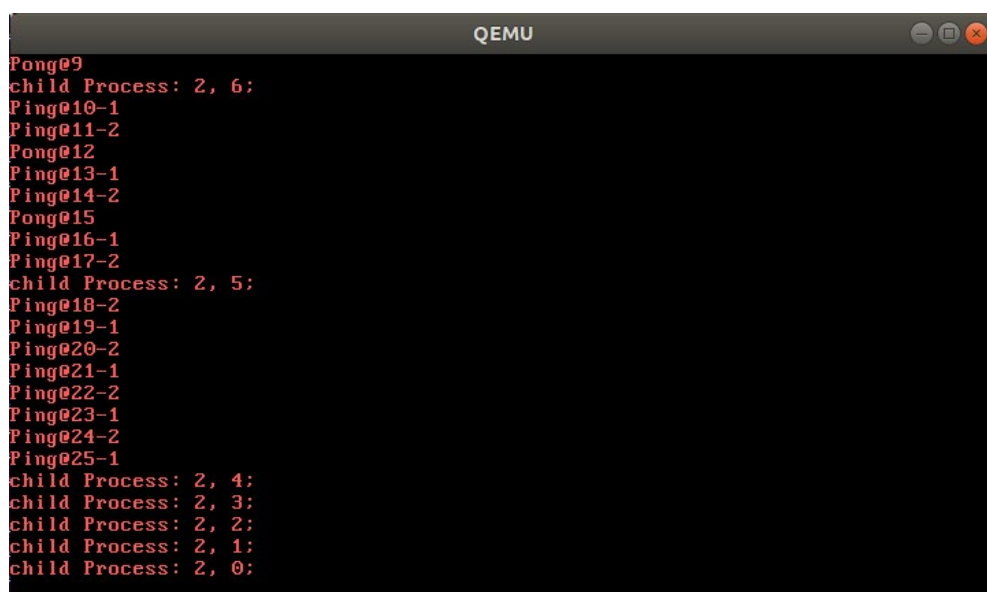
在本次实验中，这两个函数实现是类似的，`join` 使用 `while` 等待 `thread` 指向的线程结束，在循环体内调度运行 `thread` 指向的线程，而 `exit` 则将当前线程置为 `DEAD` 状态，然后调度运行一个 `RUNNABLE` 状态的线程。调度的代码在 `yield` 中已经给出，不再赘述。

实验疑惑

出现乱码和无限等待两种 BUG，在线程代码中加入 `printf("")` 便解决，但不明白为何可以解决。

实验收获

了解了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。



```
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@18-2
Ping@19-1
Ping@20-2
Ping@21-1
Ping@22-2
Ping@23-1
Ping@24-2
Ping@25-1
child Process: 2, 4:
child Process: 2, 3:
child Process: 2, 2:
child Process: 2, 1:
child Process: 2, 0:
```