实验三 进线程切换

161130118 尹浚宇
908664035@qq.com

实验目的

了解进程和线程的区别
掌握内核态进程切换的流程
掌握用户态线程切换的流程

实验内容

实现进程切换机制，完成系统调用 fork, sleep, exit
实现用户态线程库，完成 pthread_create, pthread_join, pthread_yield, pthread_exit 等函数

程序设计思路

进程切换:

从用户态通过中断进入内核态的流程之前的实验中已经接触过，此处不再赘述．这里着重讲述内核态下实现进程切换的相关函数.

timerHandle 函数

该函数首先需要将处于阻塞状态的进程的 sleep time 减 1，然后使得当前处于运行状态的进程的运行时长加 1．其中第一个操作通过简单遍历便可完成，第二个操作直接加一即可．完成上述两个操作后，要判断当前进程的时间片是否用完，若没用完则继续当前进程，该操作直接 return 便可完成．若时间片耗尽，我们需要切换至其他可运行进程继续运行.

进程切换是整个进程部分唯一的难点，流程如下:

1. 重新给当前进程分配时间片，将其状态从运行中置为可运行.

```
pcb[current].timeCount = 0;
pcb[current].state = STATE_RUNNABLE;
```

2. 从其余可运行进程中选择一个进程，注意这里 idle 进程即 0 号进程的优先级是最低的.

```
// find another runnable process, none idle processes have priority
i = (current+1) % MAX_PCB_NUM;
if (i == 0) // if next process is idle, skip it and try to find another runnable process
i = (i+1) % MAX_PCB_NUM;

while (i != current)
{
  if (pcb[i].state == STATE_RUNNABLE && i != 0)
    break;
  i = (i+1) % MAX_PCB_NUM;
}
if (i == current)
  i = 0;
```

3. 将选中进程的状态置为运行中，分配时间片，并将进程号回显在终端里.

```
pcb[i].state = STATE_RUNNING; // switch to running state
pcb[i].timeCount = 0; // set time count to 0

putChar(i + '0'); /* echo pid of selected process */
```

4. 恢复选中进程的栈顶，并填好 tss 中的 esp0，为下一次进程切换从用户态到内核态做好

准备. 这里填 tss 是为了配合硬件进行切换，因为内核数据段只有一个，所有的进程共享一个数据段，所以这里只需要填 esp0 就可以了，ss0 对于每个进程都是一样的且已经在 kvm.c 里初始化了.

```
/*XXX recover stackTop of selected process */
tmpStackTop = pcb[i].stackTop;
pcb[i].stackTop = pcb[i].prevStackTop;

// setting tss for user process
tss.esp0 = pcb[i].stackTop;
// switch to new process
current = i;
```

5. 切换内核栈为选中进程的内核栈，从选中进程的内核态恢复到用户态. 这里代码和 doirq.S 对应即可，难度不大.

```
// 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 $8, %esp");
asm volatile("iret");
```

fork 函数

该函数从空闲的进程控制块中选择一个，子进程直接复制父进程的堆栈和通用寄存器状态. 这里需要注意的是子进程的段寄存器要根据选中进程在内存中的位置选择合适的赋值，具体而言，内核代码段和数据段占了 GDT 的第 1, 2 两项，剩下 6 项作为 3 个用户进程的代码段和数据段. 具体代码如下:

```
pcb[i].regs.gs = USEL(2 * i + 2);
pcb[i].regs.fs = USEL(2 * i + 2);
pcb[i].regs.es = USEL(2 * i + 2);
pcb[i].regs.ds = USEL(2 * i + 2);
```

```
pcb[i].regs.cs = USEL(2 * i + 1);
```

```
pcb[i].regs.ss = USEL(2 * i + 2);
```

这里我 fork 之后继续运行父进程，且使用 LinuxV2.6 的调度算法分配父子进程的时间片，即父进程和子进程平分剩余时间片，若剩余时间片为奇数，则父进程多得到一个时间片.

sleep 函数

这个函数用于一个进程主动阻塞自己，阻塞自己的时间长度由 sf->ecx 指出，之后的流程和和 timerHandle 切换进程一致，不再赘述.

exit 函数

这个函数用于一个进程使自己消亡，并回收系统分配给它的资源. 这里回收资源是逻辑上的，即只需要切换线程控制块里的状态即可，不需要真正的回收给他分配的内存. 之后的流程和 timerHandle 切换进程一致，不再赘述.

线程切换:

这个部分主要的难点在于如何在初始化时伪造一个函数调用形式的栈状态，及每次切换线程时如何获取被切换进程的返回地址.

对于第一个问题，我们需要明白每个 tcb 里的 stack 就是每个线程真实的 stack，这一点是通过 makefile 保证的. 且这里的 tcb 是全局变量，应该在全局变量区，但 stack 又是栈，应该在栈区，那么他们为什么能在内存的同一块区域呢？这是因为这里的分段机制只分了代码段和数据段，并没有加以进一步的区分. 明白这个问题之后，我们就可以通过填 stack 来伪造一个函数调用现场，具体在 pthread_create 部分讲述.

对于第二个问题，当一个线程被切换的时候，除了通用寄存器，堆栈需要被保存之外，我们还需要知道它下一次被调度的时候需要回到那一句代码继续执行，可是底层汇编的机制使得我们不能直接读写 eip，我们需要通过间接方式来获得这个 eip，并把它保存在 tcb 的 cont 变量里. 这里我的做法是将切换线程的功能封装为一个函数 void switch_to(int)，因为函数调用的时候会将一个函数的返回地址放在该函数堆栈的 ebp+4 位置，这样我们就能获得该 eip.

switch_to 函数

这个函数首先保存除 esp ebp 和 eip 的所有寄存器，然后单独处理 esp，ebp 和 eip，之后再通过压栈弹栈的方式切换至新线程.

该函数的堆栈形如:

参数 i
switch_to 函数的返回地址
old ebp
xxx
xxx
xxx
xxx
xxx

其中 ebp 指向 old ebp 所在的位置.

这里需要注意的是，我们需要保存的堆栈和寄存器状态是调用该函数之前的状态，调用该函数主要目的是为了获得 eip. 所以保存时处理如下:

```
uint32_t ebp;
// save current regs
asm volatile("movl %%edi, %0":"=m"(tcb[current].cont.edi));
asm volatile("movl %%esi, %0":"=m"(tcb[current].cont.esi));
asm volatile("movl %%ebx, %0":"=m"(tcb[current].cont.ebx));
asm volatile("movl %%edx, %0":"=m"(tcb[current].cont.edx));
asm volatile("movl %%ecx, %0":"=m"(tcb[current].cont.ecx));
asm volatile("movl %%eax, %0":"=m"(tcb[current].cont.eax));
```

```
asm volatile("movl %%ebp, %0":"=m"(ebp));
tcb[current].cont.ebp = *(uint32_t*)ebp;
tcb[current].cont.eip = *((uint32_t*)ebp+1);
tcb[current].cont.esp = ebp + 0xc;
```

在切换线程的时候，需要注意的是对于局部变量和参数的寻址都是基于 ebp 的，所以我们需要通过安全的方式切换线程. 这里我通过 push 和 pop 的方式搭配 ret 进行跳转，避免了使

用 ebp 寻址，这是因为 pop 和 push 是通过 esp 来完成的.

```
current = i;
// switch regs jmp to pthread i
asm volatile("movl %0, %%esp"::"m"(tcb[i].cont.esp));

asm volatile("pushl %0"::"m"(tcb[i].cont.eip));
asm volatile("pushl %0"::"m"(tcb[i].cont.eax));
asm volatile("pushl %0"::"m"(tcb[i].cont.ecx));
asm volatile("pushl %0"::"m"(tcb[i].cont.edx));
asm volatile("pushl %0"::"m"(tcb[i].cont.ebx));
asm volatile("pushl %0"::"m"(tcb[i].cont.ebp));
asm volatile("pushl %0"::"m"(tcb[i].cont.esi));
asm volatile("pushl %0"::"m"(tcb[i].cont.edi));

asm volatile("popl %edi");
asm volatile("popl %esi");
asm volatile("popl %ebp");
asm volatile("popl %ebx");
asm volatile("popl %edx");
asm volatile("popl %ecx");
asm volatile("popl %eax");
asm volatile("ret");
```

pthread_create 函数

在知道了切换线程的方法之后，初始化的填法就相当明确了. 这里我们把 eip 指向该线程绑定的函数的入口位置，然后把参数填进栈的顶部. 这里需要注意的时候进入函数的时候会执行 push ebp, movl esp ebp. 而我的切换方法是通过 ret 切换的，所以这里把 esp 放在了参数的下一个位置，具体代码如下:

```
tcb[i].cont.eip = (uint32_t)start_routine;
tcb[i].cont.ebp = (uint32_t)&(tcb[i].cont);
tcb[i].stack[MAX_STACK_SIZE - 1] = (uint32_t)arg;
tcb[i].cont.esp = (uint32_t)&(tcb[i].stack[MAX_STACK_SIZE - 2]);
```

pthread_exit 函数

该函数使得调用该函数的线程消亡，并释放所有因为等待该线程而阻塞的线程，之后从可运行线程了选择一个进行切换. 这里并不需要保存现场，所以只需要执行 switch_to 函数的下半段切换操作即可.

pthread_join 函数

该函数使得调用该函数的线程阻塞，转而执行该函数指定的一个线程. 线程切换调用 switch_to 函数即可.

pthread_yield 函数

该函数使得调用该函数的线程阻塞，与 join 的区别是这里没有指定线程，所以我们需要从其余可运行线程中选择一个进行切换，线程切换调用 switch_to 函数即可.

实验心得

阅读 makefile 可以帮助理解程序内存的分布，画出堆栈情况有助于理解调用流程，通过反汇编用户程序可以找到一些意向之外的 bug.