《操作系统》实验报告

LAB3 进线程切换

姓名：陈攀岭

学号：171860516

邮箱：171860516@smail.nju.edu.cn

一、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。

内核：实现进程切换机制，并提供系统调用 fork、sleep、exit。

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。

用户：对上述库函数进行测试。

二、实验原理

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（Interrupt Descriptor Table，中断描述符表），初始化 GDT，初始化 TSS（Task State Segment，任务状态段），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。

三、实验步骤

1.实验任务

（1）完善 kernel/kernel/irqHandle.c 中的 timerHandle、syscallFork、syscallSleep 和 syscallExit 函数；

（2）完善 lib/pthread.c 中的 pthread_create、pthread_exit、pthread_join 和 pthread_yield 函数

2.代码实现

（1）timerHandle 函数：

首先扫描进程表，对于进程阻塞（STATE_BLOCKED）的进程时间片（sleepTime）-1，时间片减为 0 后，重新设置进程为等待状态（STATE_RUNNABLE）。

```c
// make blocked processes sleep time -1, sleep time to 0, re-run
for(int j = 0; j < MAX_PCB_NUM; j++) {
    if(pcb[j].sleepTime > 0 && pcb[j].state == STATE_BLOCKED) {
        pcb[j].sleepTime--;
        if(pcb[j].sleepTime == 0)
            pcb[j].state = STATE_RUNNABLE;
    }
}
```

对于当前进程，若处理时间片（timeCount）未达到最大（MAX_TIME_COUNT）,则时间片+1，进程继续；否则，转换为另一进程。

```c
// time count not max, process continue
if(pcb[current].timeCount < MAX_TIME_COUNT)
    pcb[current].timeCount++;
// else switch to another process
else {
    pcb[current].state = STATE_RUNNABLE;
    pcb[current].timeCount = 0;
    pcb[current].regs = *sf;
```

将当前进程重置为等待状态（STATE_RUNNABLE）。从当前进程往后依次查找等待进程，直到找到该进程。更改进程状态（STATE_RUNNABLE→STATE_RUNNING），echo 返回进程号（pid），更改 tss，切换进程堆栈。

```
int j;
for(j = 0; j < MAX_PCB_NUM; j++) {
    if(pcb[(i + j) % MAX_PCB_NUM].state == STATE_RUNNABLE)
        break;
}
current = (i + j) % MAX_PCB_NUM;
pcb[current].state = STATE_RUNNING;
pcb[current].timeCount = 0;
/* echo pid of selected process */
putChar(pcb[current].pid + '0');
/*XXX recover stackTop of selected process */
tss.esp0 = (uint32_t) & (pcb[current].stackTop);
tss.ss0 = KSEL(SEG_KDATA);
// setting tss for user process
// switch kernel stack
asm volatile("movl %0, %%esp" :: "r" (&pcb[current].regs));
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");
```

（2）syscallFork 函数：

首先扫描进程表，找到空的进程表项（STATE_DEAD）。

```
// find empty pcb
int i, j;
for (i = 0; i < MAX_PCB_NUM; i++) {
    if (pcb[i].state == STATE_DEAD)
        break;
}
```

如果找到正确表项（i != MAX_PCB_NUM），复制用户空间，设置进程表项。

```
/*XXX set pcb
  XXX pcb[i]=pcb[current] doesn't work
*/
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;
for(int j = 0; j < MAX_STACK_SIZE; j++)
    pcb[i].stack[j] = pcb[current].stack[j];
pcb[current].state = STATE_RUNNABLE;
```

设置表项寄存器，对于段寄存器，GDT 中除去内核代码段和数据段的 1、2 项，空的 0 项以及指向 TSS 的 9 项外，有 6 项（即 3、4、5、6、7、8）可用于 3 个用户进程的代码段与数据段（即进程号为 1、2、3 的用户进程），于是以进程号分配，即代码段为（pid ∗ 2 + 1）项，数据段为（pid ∗ 2 + 2）项。对于其余通用寄存器等寄存器，直接复制父进程寄存器内容。

```
/*XXX set regs */
pcb[i].regs.gs = USEL(pcb[i].pid * 2 + 2);
pcb[i].regs.fs = USEL(pcb[i].pid * 2 + 2);
pcb[i].regs.es = USEL(pcb[i].pid * 2 + 2);
pcb[i].regs.ds = USEL(pcb[i].pid * 2 + 2);
pcb[i].regs.edi = pcb[current].regs.edi;
pcb[i].regs.esi = pcb[current].regs.esi;
pcb[i].regs.ebp = pcb[current].regs.ebp;
pcb[i].regs.xxx = pcb[current].regs.xxx;
pcb[i].regs.ebx = pcb[current].regs.ebx;
pcb[i].regs.edx = pcb[current].regs.edx;
pcb[i].regs.ecx = pcb[current].regs.ecx;
pcb[i].regs.eax = pcb[current].regs.eax;
pcb[i].regs.irq = pcb[current].regs.irq;
pcb[i].regs.error = pcb[current].regs.error;
pcb[i].regs.eip = pcb[current].regs.eip;
pcb[i].regs.cs = USEL(pcb[i].pid * 2 + 1);
pcb[i].regs.eflags = pcb[current].regs.eflags;
pcb[i].regs.esp = pcb[current].regs.esp;
pcb[i].regs.ss = USEL(pcb[i].pid * 2 + 2);
```

最后设置返回值，创建成功情况下，父进程返回子进程进程号（pid）（即进程表号 i），子进程返回值置 0；否则，返回值为-1。

```
    /*XXX set return value */
    pcb[i].regs.eax = 0;
    pcb[current].regs.eax = i;
}
else {
    pcb[current].regs.eax = -1;
}
```

（3）syscallSleep 函数

由 lib/syscall.c 中 syscall 函数可知，时间片参数存放在 ecx 寄存器中。

```
int sleep(uint32_t time) {
    return syscall(SYS_SLEEP, (uint32_t)time, 0, 0, 0, 0);
}
```

```
int32_t syscall(int num, uint32_t a1,uint32_t a2,
        uint32_t a3, uint32_t a4, uint32_t a5)
```

```
asm volatile("movl %0, %%ecx"::"m"(a1));
```

设置时间片，将进程状态设置为进程阻塞（STATE_BLOCKED）。通过调度函数 schedule()切换至其他 STATE_RUNNABLE 进程。

```
void syscallSleep(struct StackFrame *sf) {
    pcb[current].sleepTime = sf->ecx;
    pcb[current].state = STATE_BLOCKED;
    schedule();
}
```

（4）syscallExit 函数

将进程状态设置为 STATE_DEAD，通过调度函数 schedule()切换至其他 STATE_RUNNABLE 进程。

```
void syscallExit(struct StackFrame *sf) {
    pcb[current].state = STATE_DEAD;
    schedule();
}
```

（5）schedule 函数

从当前进程向后依次查找等待状态（STATE_RUNNABLE）进程，状态设置为运行（STATE_RUNNING），设置时间片，echo 进程号，设置 tss，转换进程堆栈。

```
void schedule() {
    int i = (current + 1) % MAX_PCB_NUM;
    int j;
    for(j = 0; j < MAX_PCB_NUM; j++) {
        if(pcb[(i+j)%MAX_PCB_NUM].state == STATE_RUNNABLE)
            break;
    }
    current = (i + j) % MAX_PCB_NUM;
    pcb[current].state = STATE_RUNNING;
    pcb[current].timeCount = 0;
    /* echo pid of selected process */
    putChar(pcb[current].pid + '0');
    /*XXX recover stackTop of selected process */
    tss.esp0 = (uint32_t) &(pcb[current].stackTop);
    tss.ss0 = KSEL(SEG_KDATA);
    // setting tss for user process
    // switch kernel stack
    asm volatile("movl %0, %%esp" :: "r" (&pcb[current].regs));
    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");
}
```

（6）pthread_create 库函数

首先取得当前所有通用寄存器所需的值。

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, v
    uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax, eip;
    getRegs(edi, esi, ebp, esp, ebx, edx, ecx, eax, eip);
```

扫描线程表找到空表项。

```
int i;
for(i = 0; i < MAX_TCB_NUM; i++) {
    if(tcb[i].state == STATE_DEAD)
        break;
}
```

成功找到后，设置线程表项，当前线程状态设为等待（STATE_RUNNABLE），保存上下文。

```
if(i != MAX_TCB_NUM) {
    *thread = (pthread_t)i;
    tcb[i].state = STATE_RUNNING;
    tcb[i].pthid = i;
    tcb[i].pthArg = (uint32_t)arg;
    tcb[i].cont.eip = (uint32_t)start_routine;
    tcb[i].cont.ebp = (uint32_t)(tcb[i].stack + MAX_STACK_SIZE -
    tcb[i].cont.esp = tcb[i].cont.ebp;
    tcb[i].stackTop = 0;
    tcb[current].state = STATE_RUNNABLE;
    saveTCBcont(edi, esi, ebp, esp, ebx, edx, ecx, eax, eip)
```

切换线程，设置指针 EBP、ESP，根据 man page，arg 作为唯一 start_routine()唯一参数，压入栈，call 线程 eip（即 start_routine()函数）。成功返回值 0，否则返回-1；

```
        current = i;
        asm volatile("movl %0, %%ebp":: "r" (tcb[current].cont.ebp))
        asm volatile("movl %ebp, %esp");
        asm volatile("pushl %0":: "m" (tcb[current].pthArg));
        asm volatile("call %0":: "r" (tcb[current].cont.eip));
        return 0;
}
else
        return -1;
```

（7）pthread_join 函数

首先取当前所有通用寄存器的值，当 thread 指定线程结束（即线程状态为 STATE_RUNNABLE），将当前线程设置为 STATE_BLOCKED，保存线程上下文，切换为指定线程，状态更改为 STATE_RUNNING，恢复线程上下文并跳转执行。成功返回 0，否则返回-1。

```
int pthread_join(pthread_t thread, void **retval){
    uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax, eip;
    getRegs(edi, esi, ebp, esp, ebx, edx, ecx, eax, eip);
    if(tcb[thread].state == STATE_RUNNABLE) {
        tcb[current].state = STATE_BLOCKED;
        saveTCBcont(edi, esi, ebp, esp, ebx, edx, ecx, eax, eip)
        current = thread;
        tcb[current].state = STATE_RUNNING;
        recover(tcb[current].cont.edi,
            tcb[current].cont.esi,
            tcb[current].cont.ebp,
            tcb[current].cont.esp,
            tcb[current].cont.ebx,
            tcb[current].cont.edx,
            tcb[current].cont.ecx,
            tcb[current].cont.eax,
            tcb[current].cont.eip);
        return 0;
    }
    else
        return -1;
}
```

（8）pthread_yield 函数

首先保存当前通用寄存器内容；将当前线程置为等待（STATE_RUNNABLE），保存线程上下文；从当前线程向后寻找等待中线程（STATE_RUNNABLE），切换线程，恢复上下文，跳转执行。成功返回 0。

```
int pthread_yield(void){
    uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax, eip;
    getRegs(edi, esi, ebp, esp, ebx, edx, ecx, eax, eip);
    tcb[current].state = STATE_RUNNABLE;
    saveTCBcont(edi, esi, ebp, esp, ebx, edx, ecx, eax, eip)
    int i = (current + 1) % MAX_TCB_NUM;
    int j;
    for(j = 0; j < MAX_TCB_NUM; j++) {
        if(tcb[(i+j)%MAX_TCB_NUM].state == STATE_RUNNABLE)
            break;
    }
    current = (i + j) % MAX_TCB_NUM;
    tcb[current].state = STATE_RUNNING;
    recover(tcb[current].cont.edi,
            tcb[current].cont.esi,
            tcb[current].cont.ebp,
            tcb[current].cont.esp,
            tcb[current].cont.ebx,
            tcb[current].cont.edx,
            tcb[current].cont.ecx,
            tcb[current].cont.eax,
            tcb[current].cont.eip);
    return 0;
}
```

（9）pthread_exit 函数

关闭当前线程（置为 STATE_DEAD），寻找等待线程（STATE_RUNNABLE），切换线程，恢复上下文并跳转执行。

```
void pthread_exit(void *retval){
    tcb[current].state = STATE_DEAD;
    int i = (current + 1) % MAX_TCB_NUM;
    int j;
    for(j = 0; j < MAX_TCB_NUM; j++) {
        if(tcb[(i+j)%MAX_TCB_NUM].state == STATE_RUNNABLE)
            break;
    }
    current = (i + j) % MAX_TCB_NUM;
    tcb[current].state = STATE_RUNNING;
    recover(tcb[current].cont.edi,
            tcb[current].cont.esi,
            tcb[current].cont.ebp,
            tcb[current].cont.esp,
            tcb[current].cont.ebx,
            tcb[current].cont.edx,
            tcb[current].cont.ecx,
            tcb[current].cont.eax,
            tcb[current].cont.eip);
    return;
}
```

（10）宏定义即恢复函数

因为对于取上下文的值、保存上下文、恢复上下文并跳转执行相同内容多次出现，因此使用宏定义及函数简化代码。

取当前寄存器内容宏定义，对于 EDI,ESI,EBX,EDX,ECX,EAX，得到寄存器中内容。对于指针 EBP，ESP，EIP，由函数调用命令可知，此前 EIP 已被 push 进栈，之后 EBP 被 push 进栈，随后“mov %esp, %ebp”，因此，此时 EBP 和 ESP 指针相同，且该地址内容为旧 EBP，于是取指针指向地址的值为保存的 EBP 内容，“指针地址+8”为旧 ESP 内容，则取 ESP，再+=8 得到保存的 ESP 内容，EBP 指针+4 地址内容则为压入栈的 EIP 内容，于是得到 EIP 的保存值。

```
#define getRegs(edi, esi, ebp, esp, ebx, edx, ecx, eax, eip) \
    asm volatile("movl %%edi, %0": "=m" (edi)); \
    asm volatile("movl %%esi, %0": "=m" (esi)); \
    asm volatile("movl %%ebx, %0": "=m" (ebx)); \
    asm volatile("movl %%edx, %0": "=m" (edx)); \
    asm volatile("movl %%ecx, %0": "=m" (ecx)); \
    asm volatile("movl %%eax, %0": "=m" (eax)); \
    asm volatile("movl (%%ebp), %0": "=r" (ebp)); \
    asm volatile("movl %%esp, %0": "=m" (esp)); \
    esp += 8; \
    asm volatile("movl 0x4(%%ebp), %0": "=r" (eip));
```

保存上下文，将保存的寄存器内容拷贝到相应的线程上下文中。

```
#define saveTCBcont(edi, esi, ebp, esp, ebx, edx, ecx, eax, eip) \
    tcb[current].cont.edi = edi; \
    tcb[current].cont.esi = esi; \
    tcb[current].cont.ebp = ebp; \
    tcb[current].cont.esp = esp; \
    tcb[current].cont.ebx = ebx; \
    tcb[current].cont.edx = edx; \
    tcb[current].cont.ecx = ecx; \
    tcb[current].cont.eax = eax; \
    tcb[current].cont.eip = eip;
```

恢复及跳转执行，将寄存器内容恢复。对于最后的跳转，由于跳转在 EBP 指针恢复之后，即 EBP 指针不能再用来作为参数 EIP 的寻址，于是在之前使用 EAX 寄存器保存 EIP 的值，最后使用“jmp *%eax”进行跳转至新线程执行。

```
void recover(uint32_t edi, uint32_t esi, uint32_t ebp, uint32_t esp, uint32_t ebx, uin
    asm volatile("movl %0, %%edi":: "m" (edi));
    asm volatile("movl %0, %%esi":: "m" (esi));
    asm volatile("movl %0, %%ebx":: "m" (ebx));
    asm volatile("movl %0, %%edx":: "m" (edx));
    asm volatile("movl %0, %%ecx":: "m" (ecx));
    asm volatile("movl %0, %%eax":: "m" (eax));
    asm volatile("movl %0, %%esp":: "m" (esp));
    asm volatile("movl %0, %%eax":: "m" (eip));
    asm volatile("movl %0, %%ebp":: "m" (ebp));
    asm volatile("jmp %eax");
}
```

3. 实验运行

（1）给 utils 文件夹下两个.pl 文件权限

```
cpl@debian:~/OSlab/lab3-171860516陈攀岭/lab$ cd utils/
cpl@debian:~/OSlab/lab3-171860516陈攀岭/lab/utils$ ls
genBoot.pl  genKernel.pl
cpl@debian:~/OSlab/lab3-171860516陈攀岭/lab/utils$ chmod 777 genBoot.pl
cpl@debian:~/OSlab/lab3-171860516陈攀岭/lab/utils$ chmod 777 genKernel.pl
```

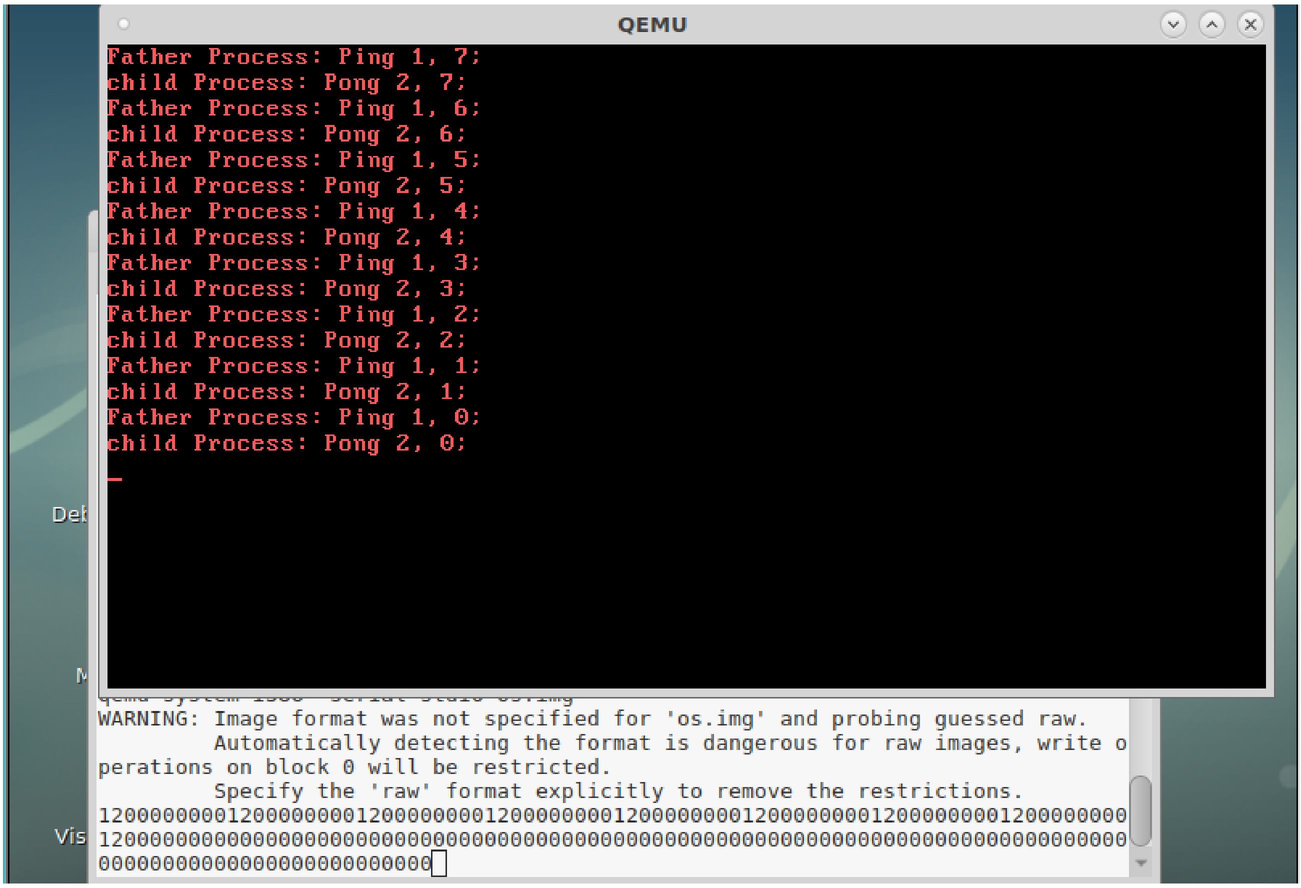
（2）进程测试

```
#include "lib.h"
//#include "types.h"
//#include "pthread.h"

int data = 0;
//int main(void);

int uEntry(void) {
    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(128);
        }
        exit();
    }
    else if(ret != -1){
        data = 1;
        while( i != 0) {
            i --;
            printf("Father Process: Ping %d, %d;\n", data, i);
            sleep(128);
        }
        exit();
```

make play



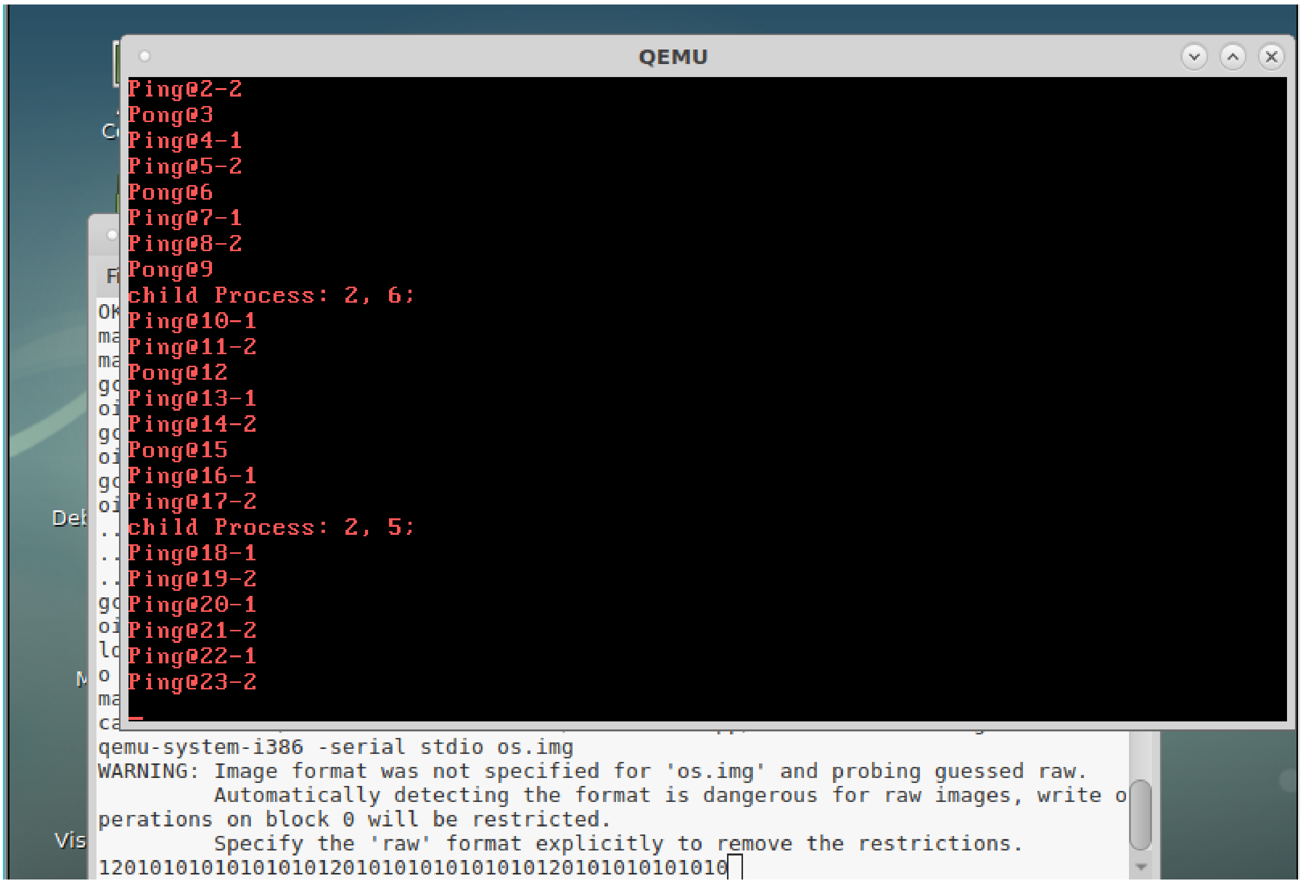
（3）线程测试

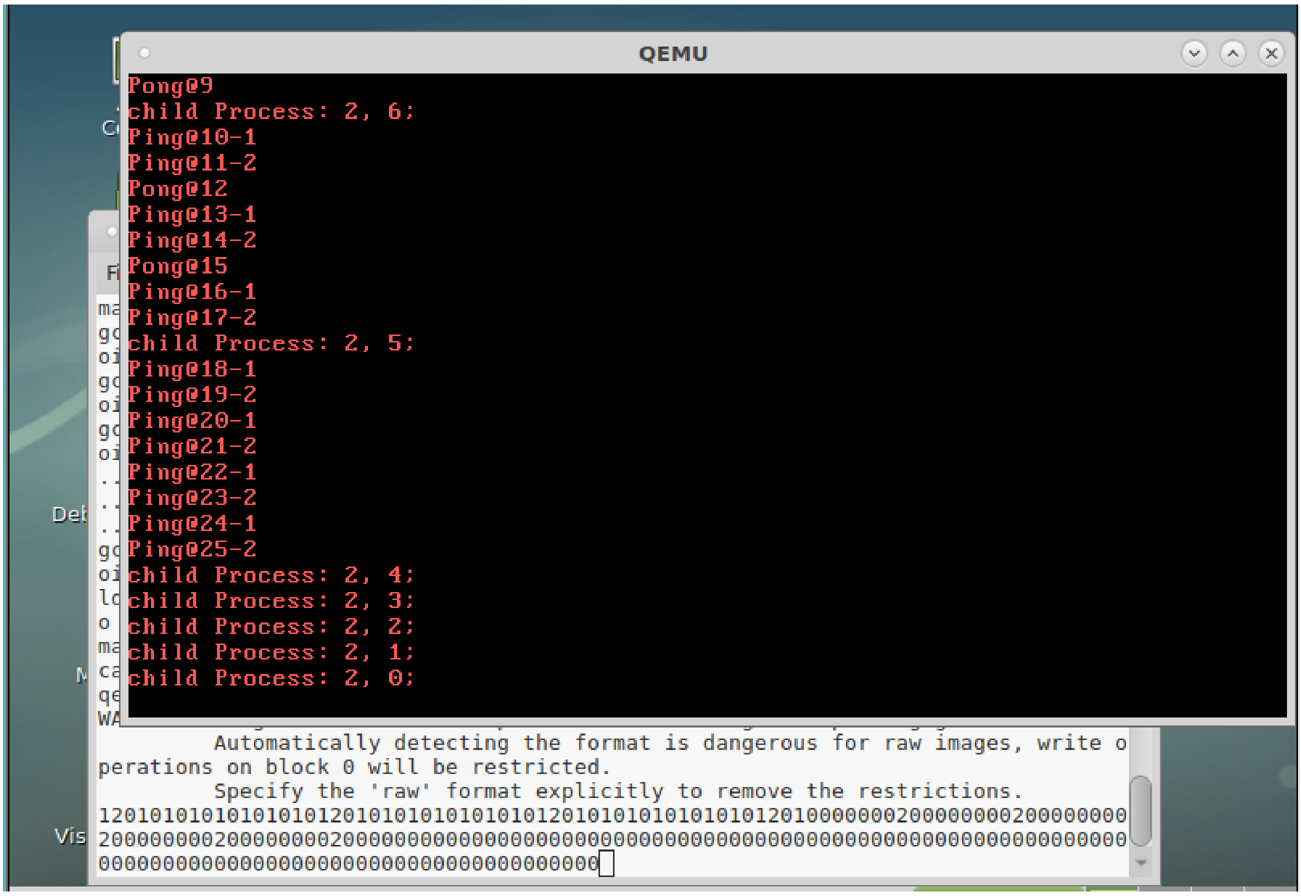
```
#include "lib.h"
#include "types.h"
#include "pthread.h"

int data = 0;
int main(void);

int uEntry(void) {
    int ret = fork();
    int i = 8;
    if (ret == 0){
        data = 2;
        while(i != 0){
            i --;
            printf("child Process: %d, %d;\n", data, i);
            sleep(128);
        }
        exit();
    }
    else if(ret != -1){
        pthread_initial();
        main();
    }
    return 0;
}
```

make play


QEMU
Ping@2-2
Pong@3
Ping@4-1
Ping@5-2
Pong@6
Ping@7-1
Ping@8-2
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
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
1201010101010101012010101010101010120101010101010


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;
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
120101010101010101201010101010101012010101010101010120100000000200000000200000000
200000000200000000200000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000