系统调用实现

1、fork（）

```
int sys_fork(struct TrapFrame *tf){

        uint32_t child=2;
        for(;PCB[child].state!=STATE_DEAD;child++);
        PCB[child].pid=child;
        PCB[child]=PCB[current];
        PCB[child].state=STATE_RUNNABLE;
        PCB[child].tf.eax=0;
        PCB[child].tf.ss=USEL(SEG_UDATA(child));
        PCB[child].tf.ds=USEL(SEG_UDATA(child));
        PCB[child].tf.cs=USEL(SEG_UCODE(child));
        uint32_t addr2=0x200000+(current-1)*4*MAX_STACK_SIZE;
        uint32_t addr1=(0x200000+(child-1)*4*MAX_STACK_SIZE);
        for(int i=0;i<4*MAX_STACK_SIZE;i++)
                *(char *)(addr1+i)=*(char *)(addr2+i);
        PCB[child].tf.eax=0;
        return child;
}
```

将父进程控制块复制给子进程控制块，并修改子进程控制块中的 ds、ss、cs，将父进程的代码区、堆栈区复制到子进程的地址空间中

对于父进程，返回子进程 pid，对于子进程，返回 0(通过将子进程控制块中 eax 设为 0 实现)

2、sleep（）

```
int sys_sleep(struct TrapFrame *tf){

        PCB[current].sleepTime=tf->ecx;
        PCB[current].state=STATE_BLOCKED;
        current=-1;
        schedule();
        return 0;
}
```

设置当前运行进程的 sleepTime，并将其状态置为阻塞态，调用进程调度函数 schedule（）

```
void schedule(){
        uint32_t index=0;
        for(int i=0;i<MAX_PCB_NUM;i++){
                if(PCB[i].state==STATE_BLOCKED && PCB[i].sleepTime<0)
                        PCB[i].state=STATE_RUNNABLE;
        }
        for(int i=1;i<MAX_PCB_NUM;i++){
                if(PCB[i].state==STATE_RUNNING){
                        index=i;break;
                }
                if(PCB[i].state==STATE_RUNNABLE){
                        index=i;break;
                }
        }
        run(index);
}
```

schedule 函数中，先遍历 PCB 数组，将其中处于阻塞态且 sleepTime 时间片耗尽的进程的状态更新为 RUNNABLE，接着再从 PCB 数组中寻找接下来要运行的进程，调用 run()函数切换至将要运行的进程

```
void run(uint32_t id){
                putChar('0'+id);
                current=id;
                PCB[id].state=STATE_RUNNING;
                tss.ss0=KSEL(SEG_KDATA);
                tss.esp0=((uint32_t)&PCB[id].state);
                uint32_t tmp = (uint32_t)&PCB[id].tf.gs;
                asm volatile("movl %0, %%esp"::"r"(tmp));
                asm volatile("popl %gs");
                asm volatile("popl %fs");
                asm volatile("popl %es");
                asm volatile("popl %ds");
                asm volatile("popal");
                asm volatile("addl $4, %esp");
                asm volatile("addl $4, %esp");
                asm volatile("iret");
}
```

run 函数中，将目标进程的状态设为 RUNNING，并修改 TSS，将 esp 指向进程控制块的 gs，pop 出段寄存器、通用寄存器，iret 跳转至目标进程执行

若 id=0，则运行 0 号进程，即 idle 进程，初始化 PCB 时将 PCB[0].eip 设为函数 idle()的首地址

```
void idle(){
        uint32_t tmp=((uint32_t)&PCB[0].tf.esp);
        asm volatile("movl %0, %%esp"::"r"(tmp));
        while(1)
                waitForInterrupt();
}
```

3、exit（）

```
int sys_exit(struct TrapFrame *tf){
        PCB[current].state=STATE_DEAD;
        schedule();
        return 0;
}
```

将当前运行进程的状态设为 DEAD，并调用 schedule 函数切换进程

程序执行流程：

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，…

修改 initSeg()函数如下：

```
void initSeg() {
	gdt[SEG_KCODE] = SEG(STA_X | STA_R, 0,		0xffffffff, DPL_KERN);
	gdt[SEG_KDATA] = SEG(STA_W,         0,		0xffffffff, DPL_KERN);
	gdt[SEG_TSS] = SEG16(STS_T32A,      &tss, sizeof(TSS)-1, DPL_KERN);
	gdt[SEG_TSS].s = 0;
	gdt[SEG_VIDEO] = SEG(STA_W,         0xb8000,       0xffffffff, DPL_KERN);

	for(int i=1;i<MAX_PCB_NUM;i++){
		gdt[SEG_UDATA(i)] = SEG(STA_W,      (i-1)*4*MAX_STACK_SIZE,       0xffffffff, DPL_USER);
		gdt[SEG_UCODE(i)] = SEG(STA_X | STA_R,      (i-1)*4*MAX_STACK_SIZE,       0xffffffff, DPL_USER);
	}
	setGdt(gdt, sizeof(gdt));

	/*
	 * 初始化TSS
	 */
	tss.ss0=KSEL(SEG_KDATA);
	tss.esp0=(uint32_t)&PCB[1].state;
	asm volatile("ltr %%ax":: "a" (KSEL(SEG_TSS)));

	/*设置正确的段寄存器*/
	asm volatile("movw %%ax, %%ds;"::"a"(KSEL(SEG_KDATA)));
	asm volatile("movw %%dx, %%ss;"::"d"(KSEL(SEG_KDATA)));
	lLdt(0);

}
```

其中，宏 SEG_UDATA(i)（定义于 memory.h 中）表示第 i 号用户进程的代码段选择符

3、启动时钟源

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，初始化用户程序的进程控制块

```
void initPCB(){

	for(int i=2;i<MAX_PCB_NUM;i++)
		PCB[i].state=STATE_DEAD;

	PCB[1].pid=1;
	PCB[1].sleepTime=-1;
	PCB[1].timeCount=-1;
	PCB[1].state=STATE_RUNNING;
	PCB[1].tf.cs=USEL(SEG_UCODE(1));
	PCB[1].tf.eip=0x200000;
	PCB[1].tf.error=0;
	PCB[1].tf.ss=USEL(SEG_UDATA(1));
	PCB[1].tf.esp=0x200000+4*MAX_STACK_SIZE;
	PCB[1].tf.eflags=0x202;
	PCB[1].tf.ds=USEL(SEG_UDATA(1));
	PCB[1].tf.es=USEL(SEG_UDATA(1));
	PCB[1].tf.fs=USEL(SEG_UDATA(1));
	PCB[1].tf.gs=USEL(SEG_UDATA(1));

	//init_idle
	PCB[0].pid=0;
	PCB[0].sleepTime=-1;
	PCB[0].timeCount=-1;
	PCB[0].state=STATE_RUNNABLE;
	PCB[0].tf.cs=KSEL(SEG_KCODE);
	PCB[0].tf.eip=(uint32_t)idle;
	PCB[0].tf.error=0;
	PCB[0].tf.ss=KSEL(SEG_KDATA);
	PCB[0].tf.eflags=0x202;
	PCB[0].tf.ds=KSEL(SEG_KDATA);
	PCB[0].tf.es=KSEL(SEG_KDATA);
	PCB[0].tf.fs=KSEL(SEG_KDATA);
	PCB[0].tf.gs=KSEL(SEG_KDATA);
}
```

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

```
void enterUserSpace(uint32_t entry) {
	clr_screen();
	uint32_t tmp = (uint32_t)&PCB[1].tf.gs;
	asm volatile("movl %0, %%esp"::"r"(tmp));
	asm volatile("popl %gs");
	asm volatile("popl %fs");
	asm volatile("popl %es");
	asm volatile("popl %ds");
	asm volatile("popal");
	asm volatile("addl $4, %esp");
	asm volatile("addl $4, %esp");
	asm volatile("iret");
}
```

7、程序执行过程中，每个时钟中断到来，陷入内核执行 TimerInterrupt（）函数

TimerInterrupt()函数中，更新处于 BLOCK 状态进程的 sleepTime，调用 schedule 函数

```
void TimerInterrupt(struct TrapFrame *tf){
        for(int i=1;i<MAX_PCB_NUM;i++)
                if(PCB[i].state==STATE_BLOCKED)
                        PCB[i].sleepTime--;
        schedule();
}
```