XV6 CPU ALARM 作业报告

软工2班 3017218092 侯雨茜

一、作业要求

在本练习中，向xv6添加一项功能，该功能会在使用CPU时间的情况下定期向进程发出警报。这对于想要限制消耗多少CPU时间的受计算限制的进程，或者对于想要进行计算但还希望采取一些定期操作的进程，可能很有用。更一般而言，您将实现用户级中断/故障处理程序的原始形式。例如，您可以使用类似的方法来处理应用程序中的页面错误。

我们应该添加一个新的 `alarm(interval, handler)` 系统调用。如果应用程序调用 `alarm(n, fn)` ，则在程序消耗的CPU时间每n次 “ticks”之后，内核将导致应用程序函数 `fn` 被调用。当 `fn` 返回时，应用程序将从上次中断的地方继续。Tick是xv6中相当随意的时间单位，由硬件计时器产生中断的频率决定。

二、实现过程

2.1 添加测试命令

新建文件 `alarmtest.c` ，并放入以下代码：

```
#include "types.h"
#include "stat.h"
#include "user.h"

void periodic();

int
main(int argc, char *argv[])
{
  int i;
  printf(1, "alarmtest starting\n");
  alarm(10, periodic);
  for(i = 0; i < 25*500000; i++){
```

```
    if((i % 250000) == 0)
      write(2, ".", 1);
  }
  exit();
}

void
periodic()
{
  printf(1, "alarm!\n");
}
```

为了将 `alarmtest.c` 编译成Xv6的用户程序，需要修改 `Makefile` ，在 `UPROGS` 中添加对应命令的定义 `_alarmtest.c\` :

```
UPROGS=\
  _cat\
  _echo\
  _forktest\
  _grep\
  _init\
  _kill\
  _ln\
  _ls\
  _mkdir\
  _rm\
  _sh\
  _stressfs\
  _usertests\
  _wc\
  _zombie\
  _big\
  _date\
  _alarmtest\
```

2.2 添加系统调用

在 `user.h` 中添加用户态函数的定义:

```
int alarm(int ticks, void(*hander)());
```

在 `usys.S` 中添加用户态函数的实现:

```
SYSCALL(alarm)
```

在 `syscall.h` 中添加alarm的系统调用编号：

```
#define SYS_alarm  23
```

在 `syscall.c` 中添加系统调用函数的外部声明：

```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_date(void);
extern int sys_alarm(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
```

```
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_date]    sys_date,
[SYS_alarm]   sys_alarm,
};
```

在 `proc.h` 的结构体 `proc` 中添加：

```
int alarmticks;
int curalarmticks;
void (*alarmhandler)();
```

在 `sysproc.c` 中添加系统调用函数 `sys_date` 的实现：

```
// cpu alarm
int
sys_alarm(void)
{
  int ticks;
  void (*handler)();

  if(argint(0, &ticks) < 0)
    return -1;
  if(argptr(1, (char**)&handler, 1) < 0)
    return -1;
  myproc()->alarmticks = ticks;
  myproc()->alarmhandler = handler;
  return 0;
}
```

至此，系统调用 `alarm` 添加完成。

2.3 中断处理

修改文件 `trap.c`，在函数 `void trap(struct trapframe *tf)` 中修改 `case T_IRQ0 + IRQ_TIMER`：

```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
```

```
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
          if (myproc() && (tf->cs & 3) == 3) {
              myproc()->curalarmticks++;
              // 当到达周期
              if (myproc()->alarmticks == myproc()->curalarmticks) {
                  myproc()->curalarmticks = 0;
                  // 将eip压栈
                  tf->esp -= 4;
                  *((uint *)(tf->esp)) = tf->esp;
                  // 将alarmhandler复制给eip，准备执行
                  tf->eip = (uint) myproc()->alarmhandler;
              }
          }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
    uartintr();
    lapiceoi();
    break;
  case T_IRQ0 + 7:
  case T_IRQ0 + IRQ_SPURIOUS:
    cprintf("cpu%d: spurious interrupt at %x:%x\n",
```

```
            cpuid(), tf->cs, tf->eip);
    lapiceoi();
    break;

  //PAGEBREAK: 13
  default:
    if(myproc() == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpuid(), tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
  }

  // Force process exit if it has been killed and is in user space.
  // (If it is still executing in the kernel, let it keep running
  // until it gets to the regular system call return.)
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check
nlock.
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();

  // Check if the process has been killed since we yielded
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();
}
```

三、测试结果

使用以下命令在Ubuntu下编译运行：

```
make CPUS=1 qemu-nox
```

其中， `CPUS=1` 是为了将执行速度变慢，以观察系统中断的执行

运行Xv6后，输入命令：

```
1  alarmtest
```

输出结果如下：