Homework: xv6 CPU alarm

刘恒星 2022229044

March 24, 2023

1 实验目的

在这个练习中，要给 xv6 添加一个功能，在进程使用 CPU 时间时定期发出警告。这对于想要限制其占用的 CPU 时间的计算型进程，或者对于想要计算但又想采取一些定期行动的进程来说，可能很有用。更广泛地说，你将实现一个原始形式的用户级中断/故障处理程序；例如，你可以使用类似的东西来处理应用程序中的页面故障。应该添加一个新的 alarm(interval, handler) 系统调用。如果一个应用程序调用 alarm(n, fn)，那么在程序每消耗 n 个 CPU 时间后，内核将使应用程序的函数 fn 被调用。当 fn 返回时，应用程序将恢复到它离开的地方。在 xv6 中，tick 是一个相当随意的时间单位，由硬件定时器产生中断的频率决定。

2 实验步骤

这次实验步骤严格按照实验指导书进行即可

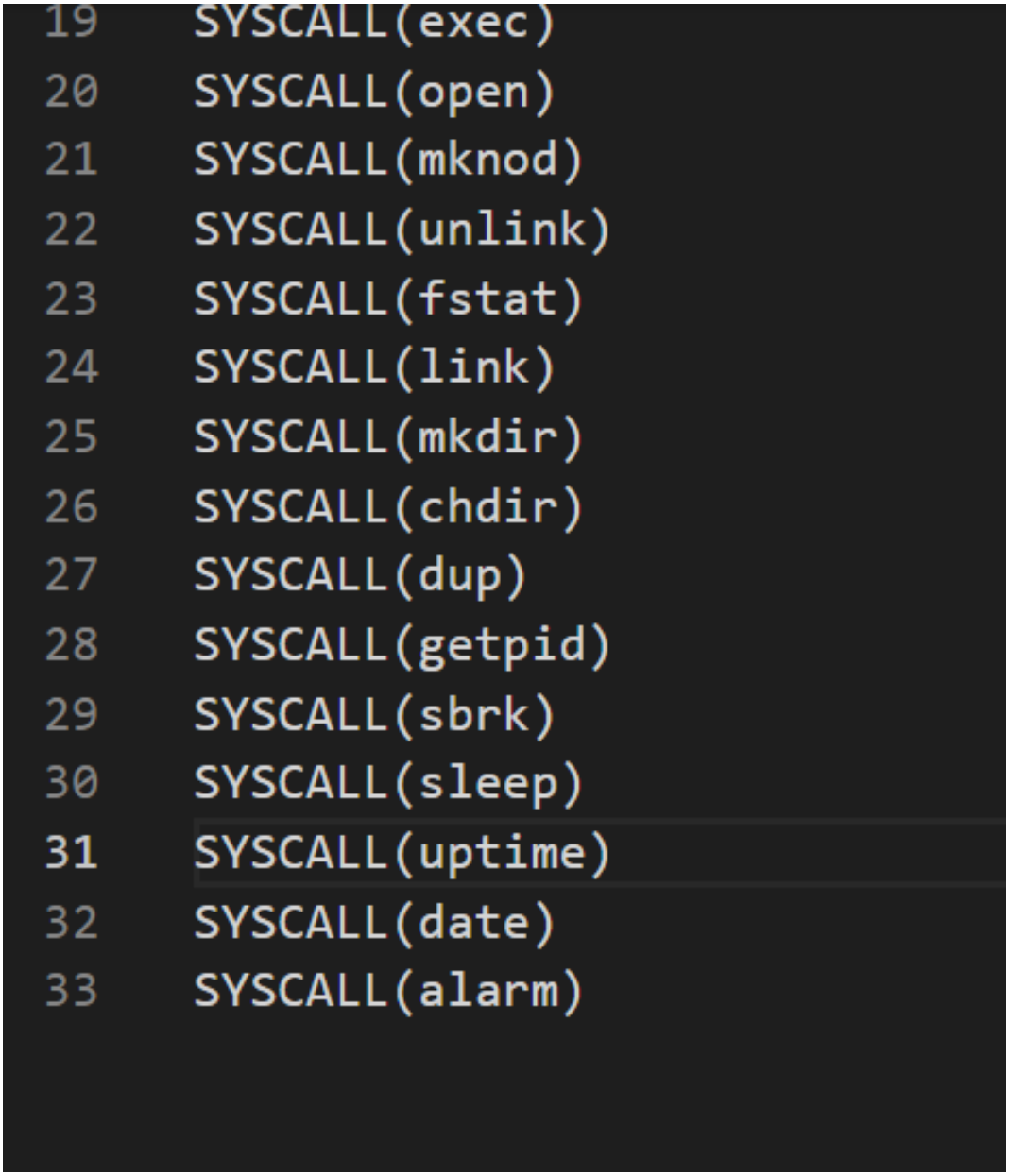
1. 需要修改 Makefile，使 alarmtest.c 被编译为 xv6 用户程序。

2. 放在 user.h 中的正确声明是：int alarm(int ticks, void (*handler)())

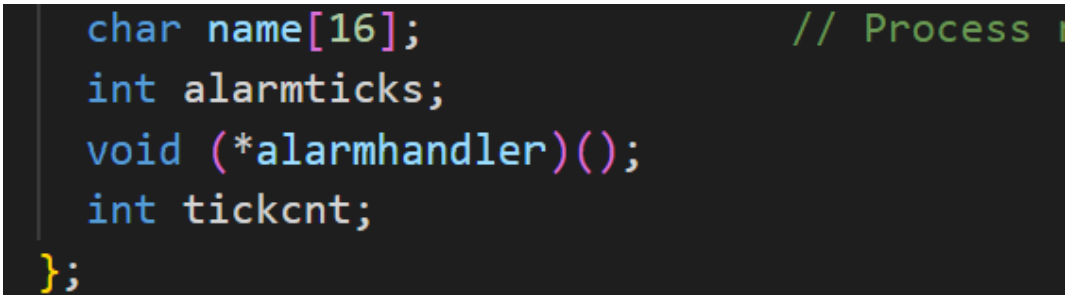
```
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int date(struct rtcdate*);
int alarm(int ticks,void(*handler)());
```

3. 必须更新 syscall.h 和 usys.S 以允许 alarmtest 调用 alarm 系统调用。

```
7   #define SYS_kill    6
8   #define SYS_exec    7
9   #define SYS_fstat   8
0   #define SYS_chdir   9
1   #define SYS_dup    10
2   #define SYS_getpid 11
3   #define SYS_sbrk   12
4   #define SYS_sleep  13
5   #define SYS_uptime 14
6   #define SYS_open   15
7   #define SYS_write  16
8   #define SYS_mknod  17
9   #define SYS_unlink 18
0   #define SYS_link   19
1   #define SYS_mkdir  20
2   #define SYS_close  21
3   #define SYS_date   22
4   #define SYS_alarm  23
```



4. sys_alarm() 应该在 proc 结构的新字段中存储报警间隔和指向处理函数的指针; 见 proc.h。



5. 把 sys_alarm 添加到 syscall.c 中，并在该文件的 syscalls 数组中添加 SYS_ALARM 的条目。

6. 需要跟踪从最后一次调用（或直到下一次调用）一个进程的报警处理程序以来已经过了多

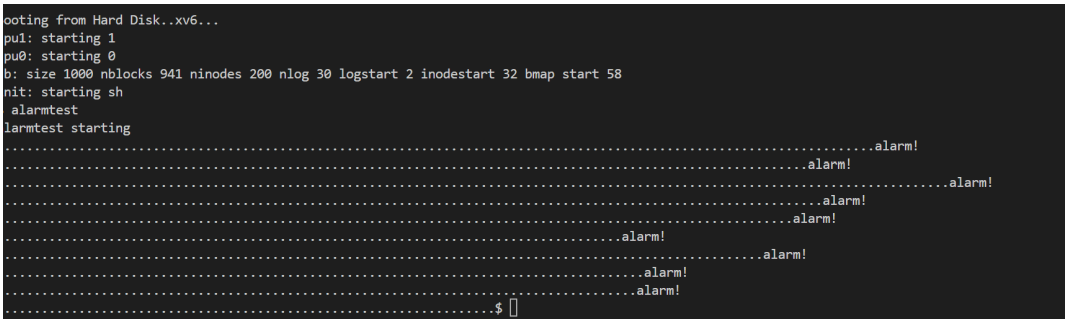
少个时间点；你也需要在 proc 结构中加入一个新字段。你可以在 proc.c 的 allocproc() 中初始化 proc 字段。

```
static struct proc *
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->state == UNUSED)
      goto found;
  p->alarmticks = 0;
  p->alarmhandler = 0;
  p->tickcnt = 0;
  release(&ptable.lock);
  return 0;
```

7. 每一次 tick，硬件时钟都会强制中断，在 trap() 中通过 case T_IRQ0 + IRQ_TIMER 来处理；你应该在这里添加一些代码。只有当有一个进程在运行，并且定时器中断来自用户空间时，才操纵一个进程的 tick alarm。

```
1   case T_IRQ0 + IRQ_TIMER:
2   if (cpuid() == 0)
3   {
4     acquire(&tickslock);
5     ticks++;
6     wakeup(&ticks);
7     release(&tickslock);
8   }
9   if (myproc() != 0 && (tf->cs & 3) == 3 && myproc()->alarmhandler)
10  {
11    myproc()->tickcnt++;
12    // cprintf("DEBUG: %d %d\n", myproc()->tickcnt, myproc()->alarmticks);
13
14    if (myproc()->tickcnt == myproc()->alarmticks)
15    {
16      myproc()->tickcnt = 0;
17      tf->esp -= 4;
18      *((uint *)tf->esp) = tf->eip;
19      tf->eip = (uint)(myproc()->alarmhandler);
20    }
21  }
```

实验效果如下：



感受

相比于上次的简单添加系统中断，这次的改动更加的复杂和深入，包括对进程的修改以及一些寄存器的处理，让我更加深入的了解这方面的知识