操 作 系 统 原 理

实

验

报

告

学　　院	智能与计算学部
年　　级	**2019** 级
班　　级	留学生班
学　　号	**6319000359**
姓　　名	张明君

2020 年　**12** 月　**5** 日

天津大學

操作系统原理实验报告

题目：xv6 CPU alarm

学院名称	智能与计算学部
专　　业	计算机科学与技术
学生姓名	张明君
学　　号	6319000359
年　　级	2019 级
班　　级	留学生班
时　　间	2020.12.05

目　录

Homework: xv6 CPU alarm

1. 实验目的

-In this exercise you'll add a feature to xv6 that periodically alerts a process as it uses CPU time. This might be useful for compute-bound processes that want to limit how much CPU time they chew up, or for processes that want to compute but also want to take some periodic action. More generally, you'll be implementing a primitive form of user-level interrupt/fault handlers; you could use something similar to handle page faults in the application,

2. 实验内容

You should add a new alarm(interval, handler) system call. If an application calls alarm(n, fn), then after every n "ticks" of CPU time that the program consumes, the kernel will cause application function fn to be called. When fn returns, the application will resume where it left off. A tick is a fairly arbitrary unit of time in xv6, determined by how often a hardware timer generates interrupts.

You should put the following example program in alarmtest.c:

```
#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 < 50*500000; i++){
    if((i++ % 500000) == 0)
      write(2, ".", 1);
  }
  exit();
}
void
periodic()
{
  printf(1, "alarm!\n");
}
```

The program calls alarm(10, periodic) to ask the kernel to force a call to periodic() every 10 ticks, and then spins for a while. After you have implemented alarm(), alarmtest should produce output like this:

```
$ alarmtest
alarmtest starting
.....alarm!
....alarm!
.....alarm!
......alarm!
.....alarm!
....alarm!
....alarm!
......alarm!
.....alarm!
...alarm!
...$
```

Hint: the right declaration to put in user.h is:

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

Hint: Your sys_alarm() should store the alarm interval and the pointer to the handler function in new fields in the proc structure; see proc.h.

Hint: here's a sys_alarm() for free:

```
    int
    sys_alarm(void)
    {
      int ticks;
      void (*handler)();

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

Hint: You'll need to keep track of how many ticks have passed since the last call (or are left until the next call) to a process's alarm handler; you'll need a new field in struct proc for this too. You can initialize proc fields in allocproc() in proc.c.

Hint: Every tick, the hardware clock forces an interrupt, which is handled in trap() by case T_IRQ0 + IRQ_TIMER; you should add some code here.

Hint: You only want to manipulate a process's alarm ticks if there's a process running and if the timer interrupt came from user space; you want something like

```
if(proc && (tf->cs & 3) == 3) ...
```

Hint: In your IRQ_TIMER code, when a process's alarm interval expires, you'll want to cause it to execute its handler. How can you do that?

Hint: You need to arrange things so that, when the handler returns, the process resumes executing where it left off. How can you do that?

Hint: You can see the assembly code for the alarmtest program in alarmtest.asm.

Hint: It will be easier to look at traps with gdb if you tell qemu to use only one CPU, which you can do by running

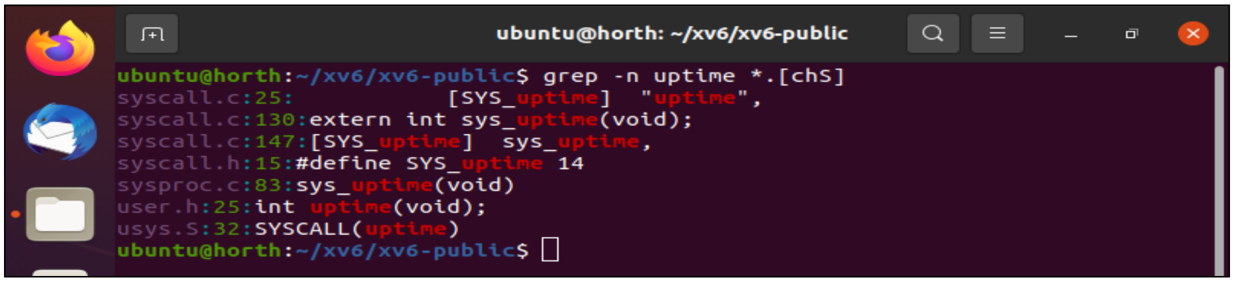
```
make CPUS=1 qemu
```

It's OK if your solution doesn't save the caller-saved user registers when calling the handler.

3. 实验步骤和分析（要细化如何实现的思路或流程图）

The requirement of this job is that we add an alarm (interval, handler) system call to make the CPU remind the process regularly. This is useful for applications that compute binding to monitor how much CPU time they are consuming, or to do something else in the process of computing.

1/ Before modifying, use the command `grep -n uptime *.[chS]` to search out all the files containing uptime system call, so that we can implement alarm system call like uptime system.



```
ubuntu@horth:~/xv6/xv6-public$ grep -n uptime *.[chS]
syscall.c:25:           [SYS_uptime]  "uptime",
syscall.c:130:extern int sys_uptime(void);
syscall.c:147:[SYS_uptime]  sys_uptime,
syscall.h:15:#define SYS_uptime 14
sysproc.c:83:sys_uptime(void)
user.h:25:int uptime(void);
usys.S:32:SYSCALL(uptime)
ubuntu@horth:~/xv6/xv6-public$
```

Figure 1: Find the location of files

We already did it in the date section so I don’ t need to explain more about that. So now let’ s continue to change to files in system below:

2/ Then add the number of the system call to **syscall.h**.

```
#define SYS_alarm  23
```

3/ Then add the declaration of the system call function in **syscall.c**.
Three statements need to be added in three places. The code is as follows:

```
[SYS_alarm]  "alarm",
```

```
extern int sys_alarm(void);
```

```
[SYS_alarm]    sys_alarm,
```

4/ Add the definition of the date function in **user.h**.

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

5/In the file **usys.S** we have to add the function in the last of the line.

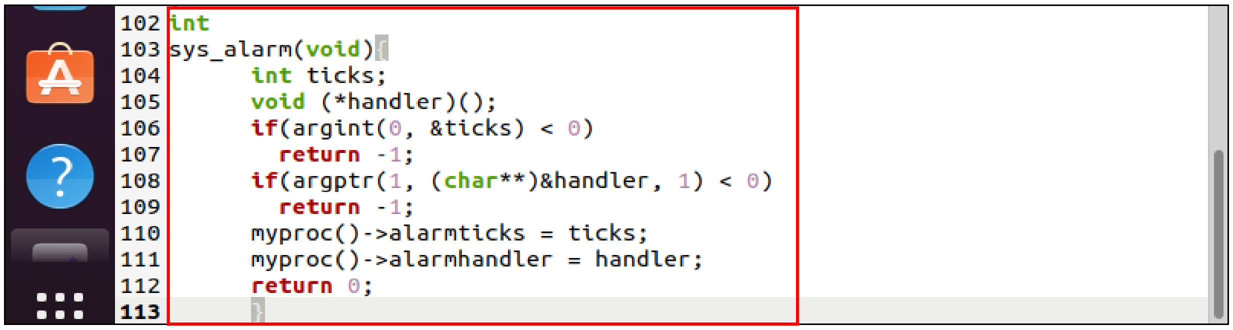
```
SYSCALL(alarm)
```

6/ Add the definition of the command corresponding to UPROGS in the **makefile** file.

```
_zombie\
_date\
//we have to add it here
_alarmtest\
```

*Note: In the makefile file we add _alarmtest\ to the file not _alarm\ .

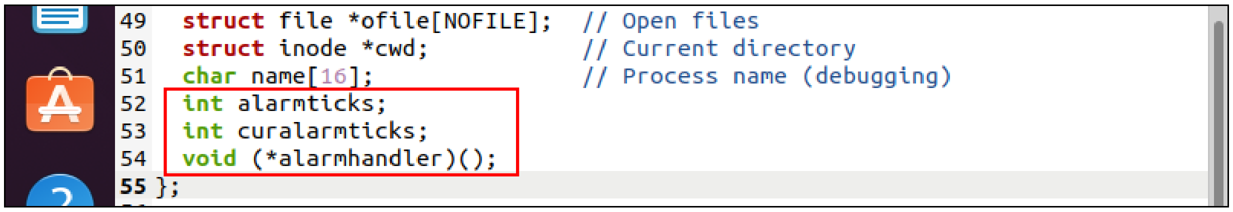
7/Adding function in sysproc.c Implementation of alarm()



```
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;
	}
```

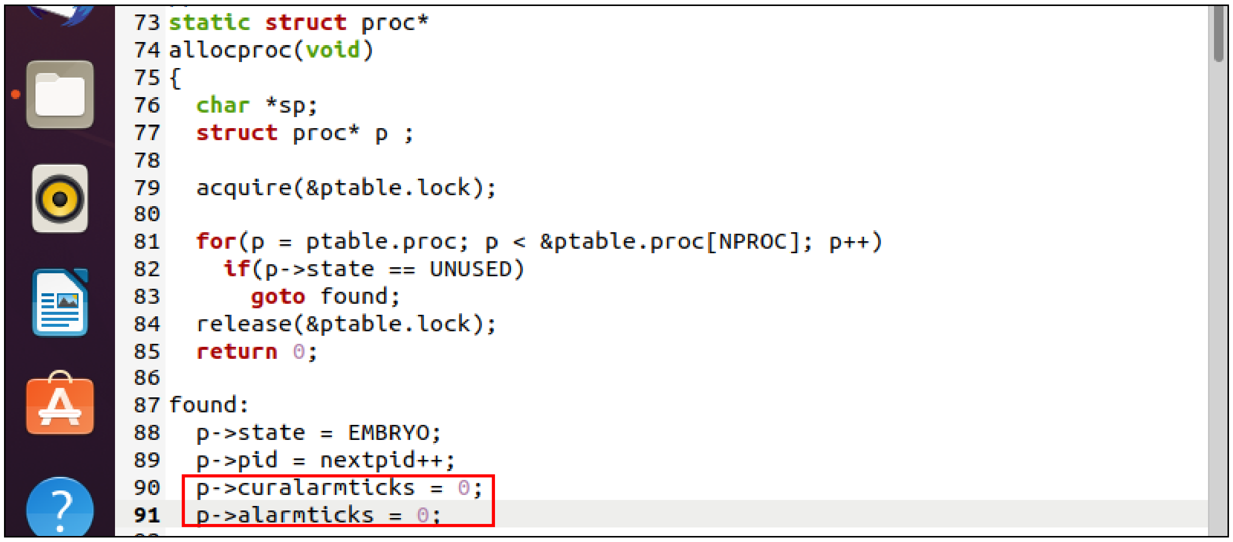
Figure 2:Add code in sysproc.c

8/We can see that there are two more members in the structure of Pro: alarmticks and alarmhandler, so we need to extend the structure accordingly. So we have to go through proc.h file to add some functions there.



```
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int alarmticks;
  int curalarmticks;
  void (*alarmhandler)();
};
```

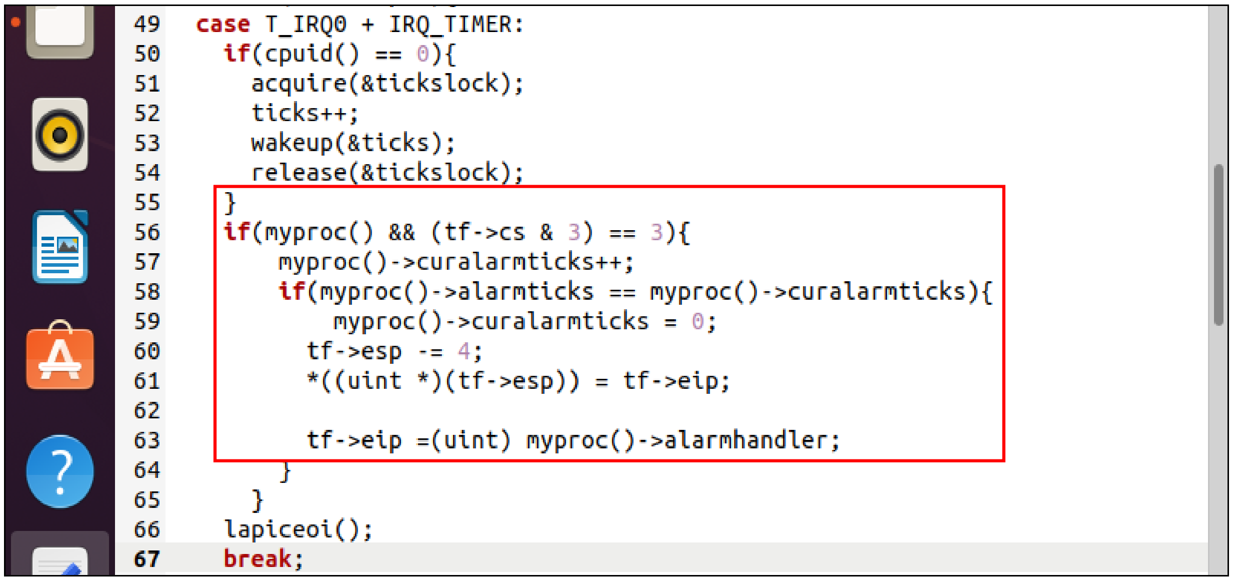
We also have to add some functions in the static struct proc * allocproc(void) of proc.c



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

  acquire(&ptable.lock);

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

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->curalarmticks = 0;
  p->alarmticks = 0;
```

The thing why we added alarmticks to 0 is to never trigger an alarm when no system calls it.

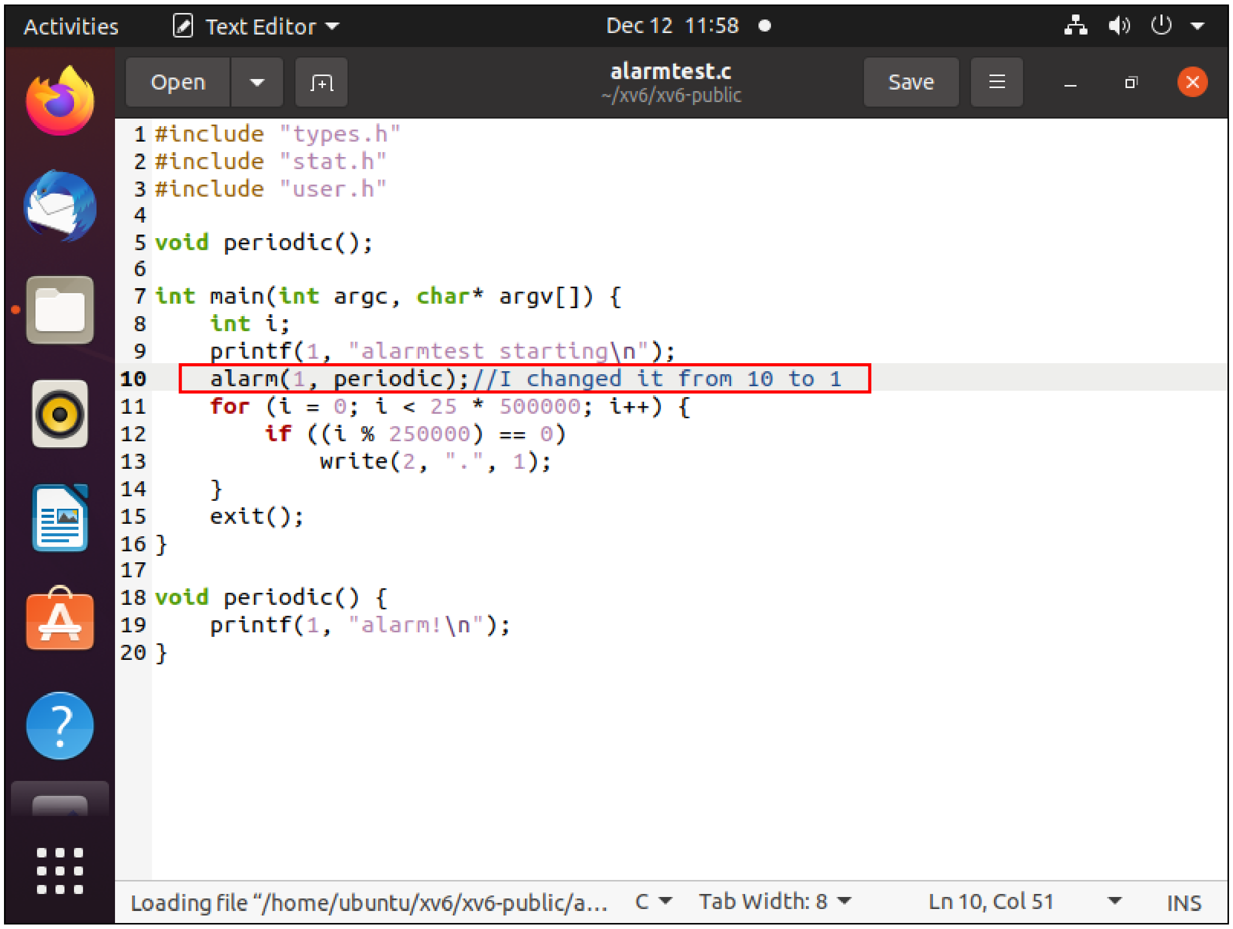
9/In the trap() function in trap.c when the ticks meet the number we set, we call the alarmhandler function. We have to add some codes in case T_IRQ0 + IRQ_TIMER.



```
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    if(myproc() && (tf->cs & 3) == 3){
        myproc()->curalarmticks++;
        if(myproc()->alarmticks == myproc()->curalarmticks){
            myproc()->curalarmticks = 0;
          tf->esp -= 4;
          *((uint *)(tf->esp)) = tf->eip;

          tf->eip =(uint) myproc()->alarmhandler;
        }
      }
    lapiceoi();
    break;
```

After the code myproc()->current_ticks = 0; We can’ t execute the command directly So we have to modify the trapframe, make sure the user program run the ″handler″ program. After iret, In order to resume previous eip, we need to push the eip to stack first.

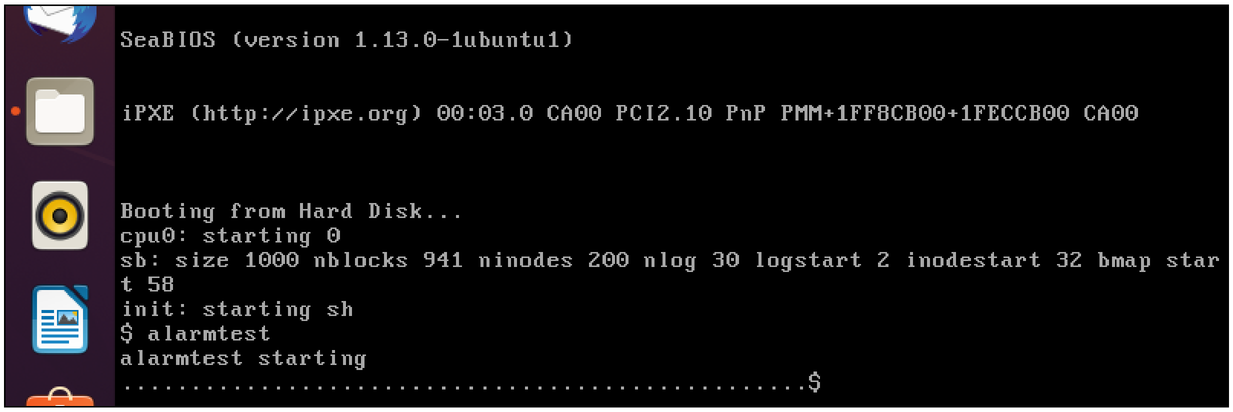
10/ Create a new file named alarmtest.c The code is as follows:



```
 1 #include "types.h"
 2 #include "stat.h"
 3 #include "user.h"
 4
 5 void periodic();
 6
 7 int main(int argc, char* argv[]) {
 8     int i;
 9     printf(1, "alarmtest starting\n");
10     alarm(1, periodic);//I changed it from 10 to 1
11     for (i = 0; i < 25 * 500000; i++) {
12         if ((i % 250000) == 0)
13             write(2, ".", 1);
14     }
15     exit();
16 }
17
18 void periodic() {
19     printf(1, "alarm!\n");
20 }
```

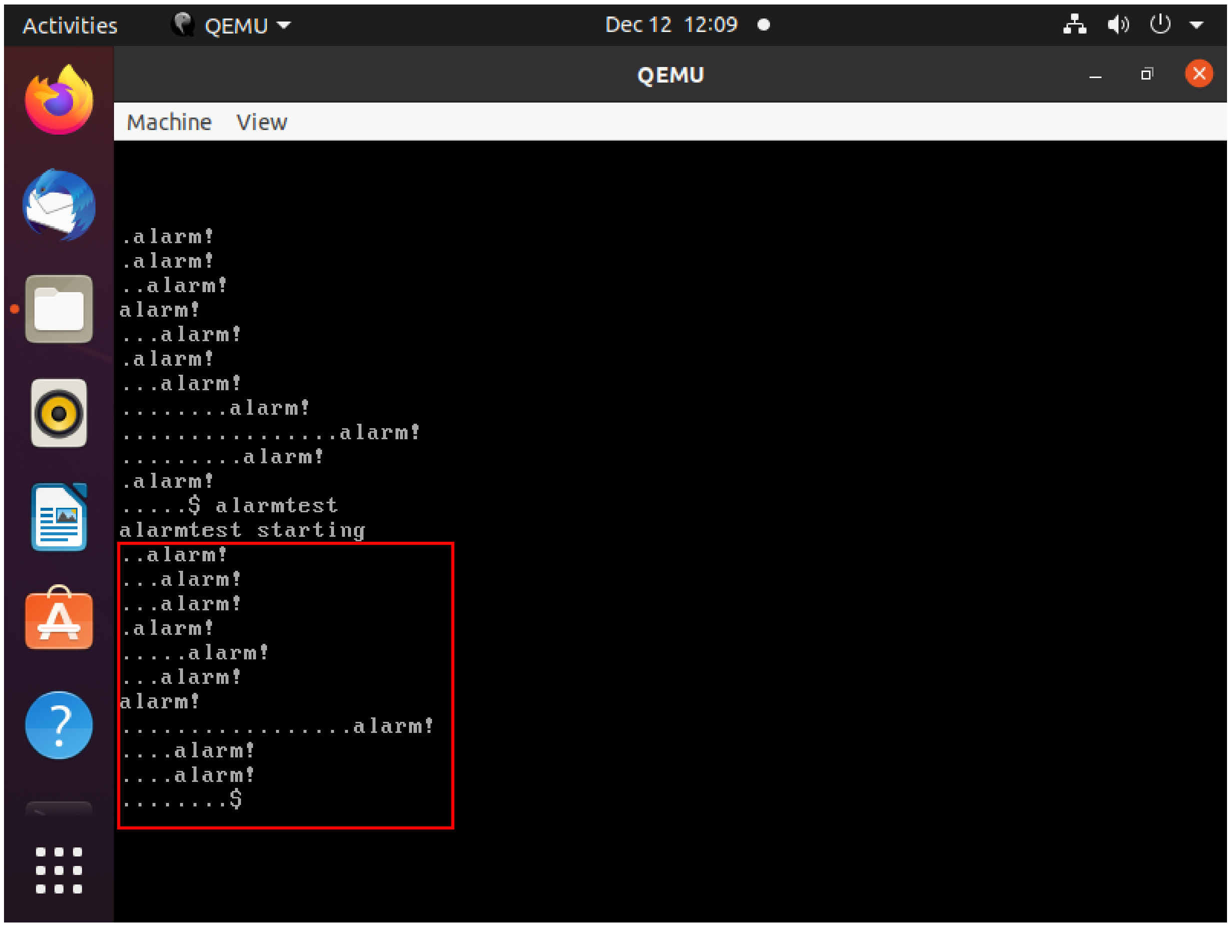
Figure 3: Alarmtest file

In the line 10 I have changed it from 10 to 1 because when I run it on qemu It’ s always show only one line alarm like this:



```
SeaBIOS (version 1.13.0-1ubuntu1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ alarmtest
alarmtest starting
..................................................$
```

11/After all the changes have been made, run it with make CPUS=4 qemu and then we have to put alarmtest in the terminal to see the results. As shown in the figure:



```
.alarm!
.alarm!
..alarm!
alarm!
...alarm!
.alarm!
...alarm!
........alarm!
................alarm!
.........alarm!
.alarm!
.....$ alarmtest
alarmtest starting
..alarm!
...alarm!
...alarm!
.alarm!
.....alarm!
...alarm!
alarm!
.................alarm!
....alarm!
....alarm!
........$
```

Figure 4: Final answer

Before I got this answer I have changed the CPU to 4 to see if it can show alarm 10 times. And yeah, it exactly show 10 times and I have to run it 4 or 5 times to get that 10 alarm! Result.

4. 实验结论与心得体会

I think that Alarmtest is the hardest one in this experiments Because they require that we have to print 10 alarms in the qemu so it’ s not that easy as we thought .So we have to find many solutions to solve it .And after the hard work we finally got the answer that they wanted us to do .