Homework: xv6 CPU alarm

Submit your solutions before the beginning of the next lecture to the submission web site.

We encourage you to collaborate with others on these in-class exercises.

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, for example.

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.

Challenges: 1) Save and restore the caller-saved user registers around the call to handler. 2) Prevent re-entrant calls to the handler -- if a handler hasn't returned yet, don't call it again. 3) Assuming your code doesn't check that `tf->esp` is valid, implement a security attack on the kernel that exploits your alarm handler calling code.

Submit: The code that you added to trap.c.