Traps

Design

0x01 backtrace

> Implement a `backtrace()` function in `kernel/printf.c`. Insert a call to this function in `sys_sleep`, and then run bttest, which calls `sys_sleep`. Your output should be as follows:
>
> ```
> backtrace:
> 0x0000000080002cda
> 0x0000000080002bb6
> 0x0000000080002898
> ```

The key point is to use frame stack pointer register `fp` to track function call

1. read fp register use asm code:

```
static inline uint64
r_fp()
{
  uint64 x;
  asm volatile("mv %0, s0" : "=r" (x) );
  return x;
}
```

2. track all function address in stack using stack frame index, the stack frame is like a jump-table, we can track the frame pointer one by one and find all function entry point.

```
  while (fp < pageEdge)
  {
    rAddr = *(uint64*)(fp - 8);
    fp = *(uint64*)(fp - 16);
    printf("%p\n", rAddr);
  }
```

0x02 alarm

> 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. Your solution is correct if it passes alarmtest and usertests.

This excercise implement a simple unix-like signal handler.

The key point is that we can use pcb to save to the signal which passed to a certain process.

basic data stored in pcb:

```
struct proc {
    ...
  // for tick handler
  struct spinlock sigLock;
  int tickNum;                //tick number to trigger the handler
  int ticked;                 //tick number since last signalarm call
  uint64 tickHandler ;         //tick handler entry in user's vm page table
  struct trapframe backup;
  int handling;
};

```

1. send signal, just save the data in target process's pcb

   ```
   uint64 sys_sigalarm(void){
   ...
     struct proc *p = myproc();
       p->tickNum = interval;
       p->tickHandler = handler;
     return 0;
   }
   ```

2. check if tick, if a time iterrupt occurs, a process will trigger `usertrap()` . In this function, we will add the process's timer counter and check if this process reach the interval. if reach the interval, bakup current registers and go to signal handler by change the `epc` register

```
  if(which_dev == 2){
    p->ticked++;
    if(p->tickNum >0 && p->ticked>=p->tickNum && !p->handling){
      p->ticked =0;
      // active handler
      p->backup =*p->trapframe;
      p->handling =1;
      p->trapframe->epc = p->tickHandler;
    }
  }
```

3. `alarm return`, this syscall will make the signal handler go back to pointer where the signal sent by recover all bakup data and change epc back.

```
uint sys_sigreturn(void){
  struct proc *p= myproc();
  //recover from handler
  p->trapframe->epc = p->backup.epc;
  p->trapframe->ra = p->backup.ra;
...
  p->handling = 0;
  return 0;
}
```

Result

Pass all code related test:

```
== Test answers-traps.txt == answers-traps.txt: FAIL
    Cannot read answers-traps.txt
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (2.7s)
== Test running alarmtest ==
$ make qemu-gdb
(4.0s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (254.9s)
== Test time ==
time: FAIL
    Cannot read time.txt
Score: 79/85
make: *** [Makefile:318: grade] Error 1
```