4.traps

Tags

```
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  memmove(p->trapframe, p->old_frame, sizeof(struct trapframe));
  kfree(p->old_frame);
  p->alarm_lock = 1;
  p->alarm_count = 0;
  return p->trapframe->a0;
}
```

preparation

read initCode.S

1. exec(init, argv[])
2. for(;;) exit();

init : .string “/init\0”

argv[] = long init, long 0

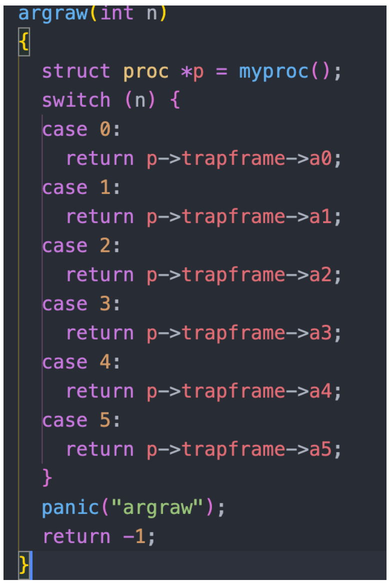
read syscall.c

get myproc()

num = p→trapframe→a7;

a0 = syscalls[num]();

argint, argaddr, argstr all use argraw(n)



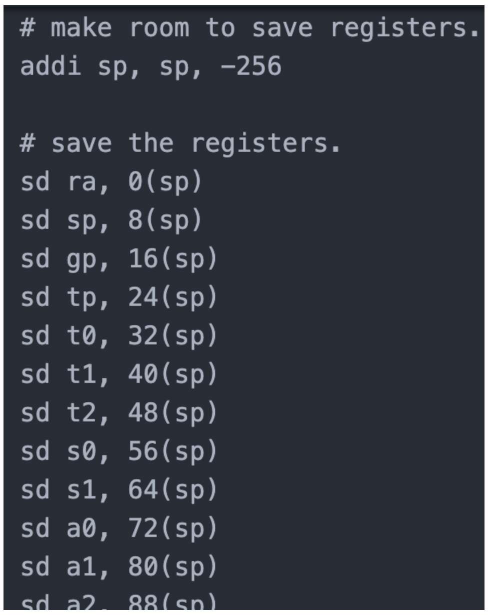
```
argraw(int n)
{
  struct proc *p = myproc();
  switch (n) {
  case 0:
    return p->trapframe->a0;
  case 1:
    return p->trapframe->a1;
  case 2:
    return p->trapframe->a2;
  case 3:
    return p->trapframe->a3;
  case 4:
    return p->trapframe->a4;
  case 5:
    return p->trapframe->a5;
  }
  panic("argraw");
  return -1;
}
```

according to the argument n , and return regs in trapframe (a0→a5)

fetchstr() get a max-len string.

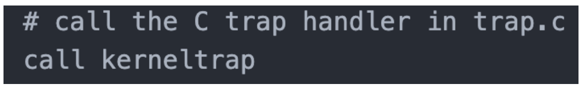
read kernelvec.S

deal with interrupts and exceptions in supervisor mode current stack is a kernel stack. store all reg and call kerneltrap()



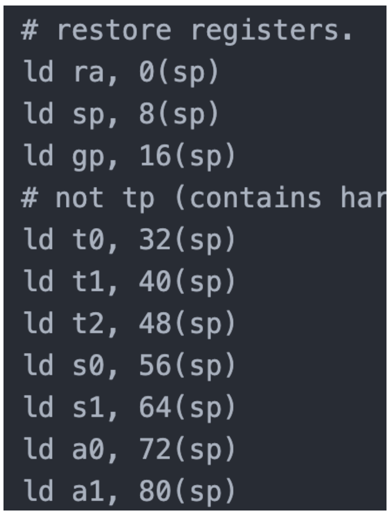
```
# make room to save registers.
addi sp, sp, -256

# save the registers.
sd ra, 0(sp)
sd sp, 8(sp)
sd gp, 16(sp)
sd tp, 24(sp)
sd t0, 32(sp)
sd t1, 40(sp)
sd t2, 48(sp)
sd s0, 56(sp)
sd s1, 64(sp)
sd a0, 72(sp)
sd a1, 80(sp)
sd a2, 88(sp)
```

call



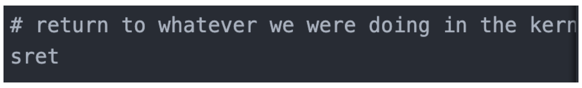
```
# call the C trap handler in trap.c
call kerneltrap
```

when return load reg



```
# restore registers.
ld ra, 0(sp)
ld sp, 8(sp)
ld gp, 16(sp)
# not tp (contains har
ld t0, 32(sp)
ld t1, 40(sp)
ld t2, 48(sp)
ld s0, 56(sp)
ld s1, 64(sp)
ld a0, 72(sp)
ld a1, 80(sp)
```

return (sret)



```
# return to whatever we were doing in the kern
sret
```

- each kernel thread's regs safely stored in their own kernel stack.

when come acrsoss machine-timer interrupt

- **mtime:** a real-time counter, exposed as a memory-mapped machine-mode read-write registe.
- **mtimecmp:** a 64-bit memory-mapped machine-mode timer compare register, which causes a timer interrupt to be posted when the mtime register contains a value ≥ to the value in the mtimecmp register.

- use CSRRW read mscrch reg to find the space in machine mode to store some info :

1. [0,8,16]: reg save (only need 3 regs)
2. [24] : address of MTIMECMP reg
3. [32]: interval between interrupts

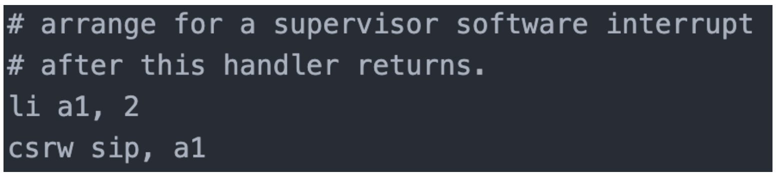
- schedule next timer interrupt by adding interval to "mtimecmp" reg

a1 = MTIMECMP

a1 = a1 + (interval)

store a1

- arrange a supervisor software interrupt after this handler returns.



```
# arrange for a supervisor software interrupt
# after this handler returns.
li a1, 2
csrw sip, a1
```

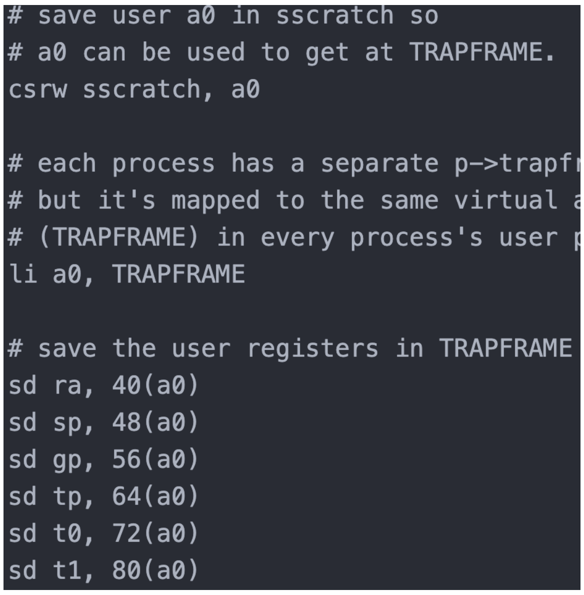
read trampoline.S

uservec

trap.c set staved to point here, **trap** from user space **start here**

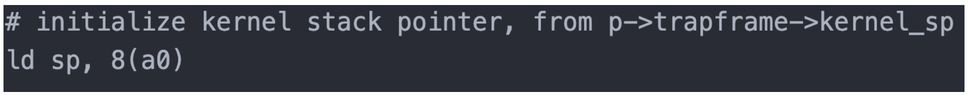
sscratch and trapframe

each process has a separate p→trapframe in memory area, but it's mapped to the same virtual address(TRAPFRAME ,always stored in a0 reg, so we put this into sscratch, so that we can sd reg into this place).



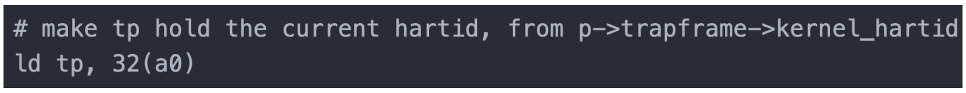
```
# save user a0 in sscratch so
# a0 can be used to get at TRAPFRAME.
csrw sscratch, a0

# each process has a separate p->trapfr
# but it's mapped to the same virtual a
# (TRAPFRAME) in every process's user p
li a0, TRAPFRAME

# save the user registers in TRAPFRAME
sd ra, 40(a0)
sd sp, 48(a0)
sd gp, 56(a0)
sd tp, 64(a0)
sd t0, 72(a0)
sd t1, 80(a0)
```

initialize the kernel stack by setting sp



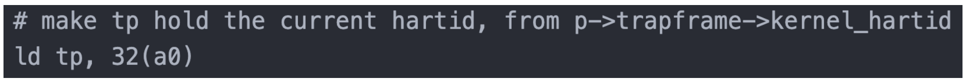
```
# initialize kernel stack pointer, from p->trapframe->kernel_sp
ld sp, 8(a0)
```

hold the hartid



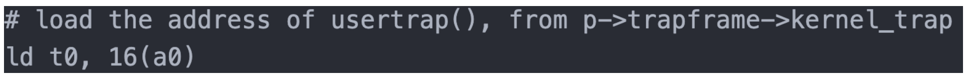
```
# make tp hold the current hartid, from p->trapframe->kernel_hartid
ld tp, 32(a0)
```

this is hart id : often used in multi_CPU os, it can read the current CPU id



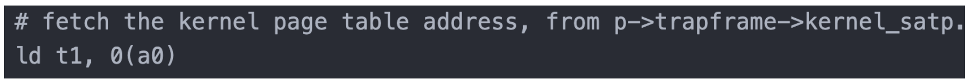
```
# make tp hold the current hartid, from p->trapframe->kernel_hartid
ld tp, 32(a0)
```

load the address of usertrap() at t0 reg



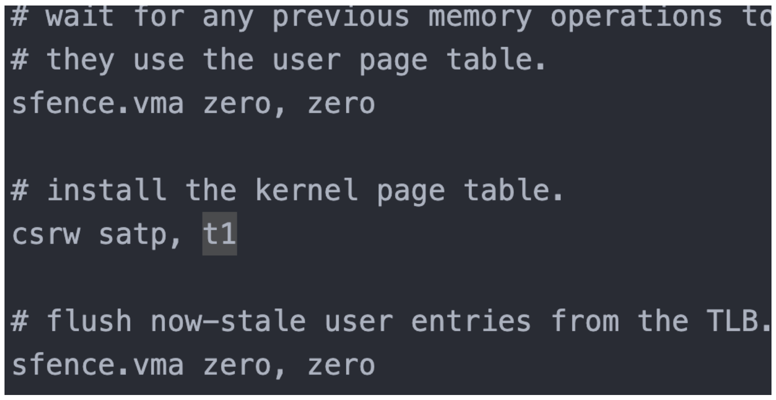
```
# load the address of usertrap(), from p->trapframe->kernel_trap
ld t0, 16(a0)
```

fetch the kernel pagetable address



```
# fetch the kernel page table address, from p->trapframe->kernel_satp.
ld t1, 0(a0)
```

wait for remains for user page table and then set satp



```
# wait for any previous memory operations t
# they use the user page table.
sfence.vma zero, zero

# install the kernel page table.
csrw satp, t1

# flush now-stale user entries from the TLB.
sfence.vma zero, zero
```

why the first fence can have this function: when running

1. flush all TLB
2. clear the writing buffer
3. stop all sequential memory operations.(include read next inst), until previous operations have been done.

kust like a synchronization point .

so ,we need to run it again to make sure that all entries in TLB is from new satp.

now ,we have completely switch into supervisor mode

we, can jump to user trap

userret

called by usertrapret() in trap.c. with one argument : a0 = page table

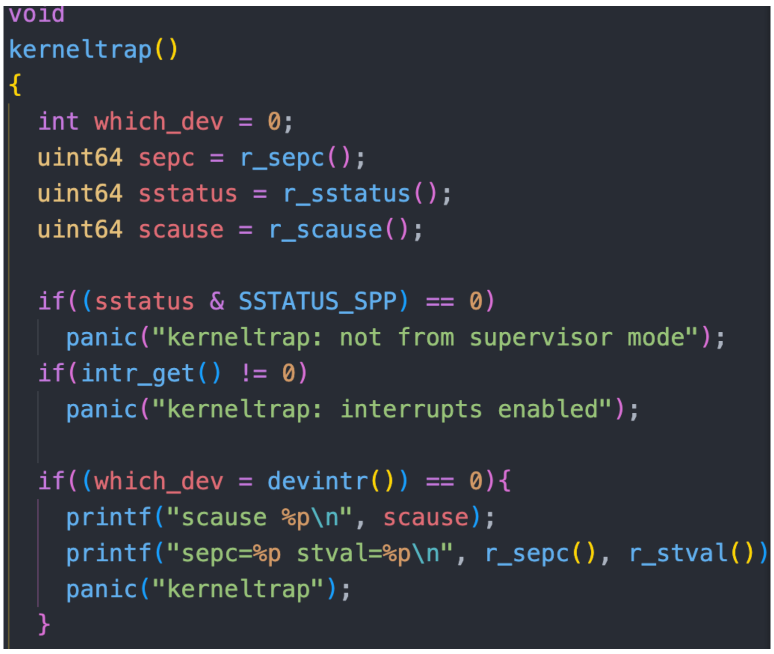
switch from kernel to user

1. switch to user page
2. reload value from TRAPFRAME to regs
3. sret (sstatus has been set in usertrapret())

read trap.c

kerneltrap()

must be device interrupt or exception.



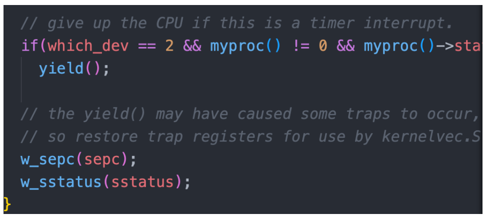
```
void
kerneltrap()
{
  int which_dev = 0;
  uint64 sepc = r_sepc();
  uint64 sstatus = r_sstatus();
  uint64 scause = r_scause();

  if((sstatus & SSTATUS_SPP) == 0)
    panic("kerneltrap: not from supervisor mode");
  if(intr_get() != 0)
    panic("kerneltrap: interrupts enabled");

  if((which_dev = devintr()) == 0){
    printf("scause %p\n", scause);
    printf("sepc=%p stval=%p\n", r_sepc(), r_stval())
    panic("kerneltrap");
  }
```

read sepc ,sstatus, scause

first, it need to be supervisor mode (SSTATUS_SPP=1)

second, check for SIE. (whether intr enabled)

third, try to handle as a hardware interrupt. if fail, panic and output the info.(1 for other device, 2 for timer interrupt)



```
// give up the CPU if this is a timer interrupt.
if(which_dev == 2 && myproc() != 0 && myproc()->sta
  yield();

// the yield() may have caused some traps to occur,
// so restore trap registers for use by kernelvec.S
w_sepc(sepc);
w_sstatus(sstatus);
}
```

then , after hardware interrupt, test this interrupt whether a timer interrupt. (will give up CPU using yield())

finally, restore spec, sstatus, in case that timer interrupt handle occur failure.

return to kernelvec.S

Luckily the RISC-V always disables interrupts when it starts to take a trap, and xv6 doesn't enable them again until after it sets stvec.

usertrap()

called by trampoline.S

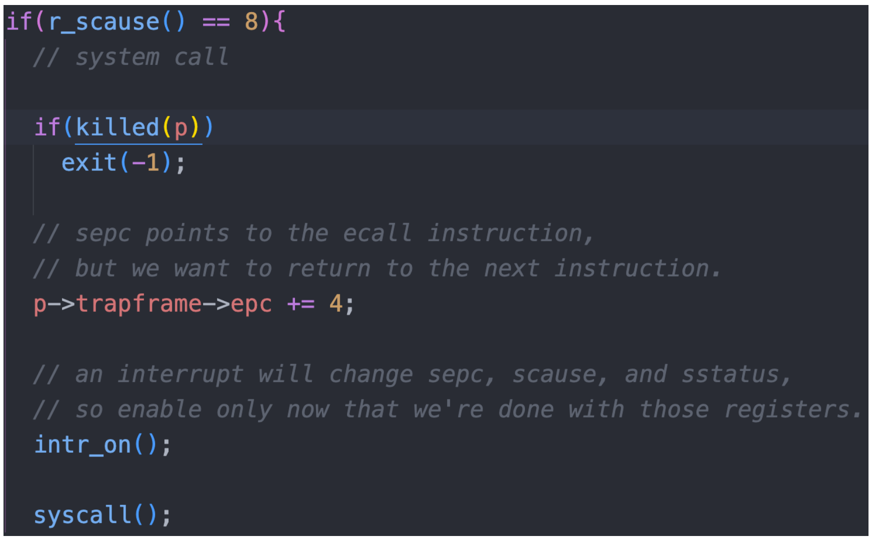
user's trap → trampoline.S → usertrap

now we have been in kernel mode , so we need to change our stvec to kernelvec

handle the trap

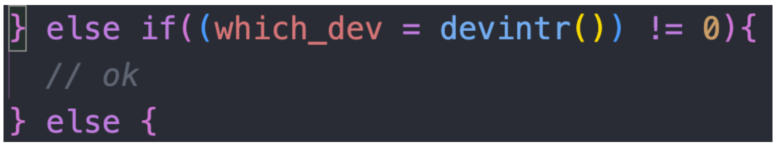
(0. save user pc (now spec) in trapframe→epc)

1. cause = syscall: if not kill, then add epc, open_interrupt, then do syscall



```
if(r_scause() == 8){
  // system call

  if(killed(p))
    exit(-1);

  // sepc points to the ecall instruction,
  // but we want to return to the next instruction.
  p->trapframe->epc += 4;

  // an interrupt will change sepc, scause, and sstatus,
  // so enable only now that we're done with those registers.
  intr_on();

  syscall();
```

2. cause = deviant()



```
} else if((which_dev = devintr()) != 0){
  // ok
} else {
```

3. error

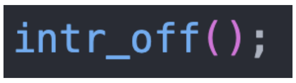
finally, if(device == timer) yield()

ret use the usertraprret()

usertrapret

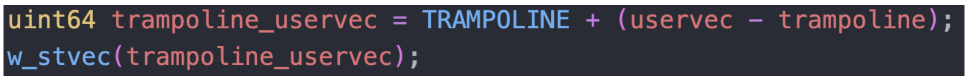
return to user space

1. turn off the interrupt, because trap handle function need to be change



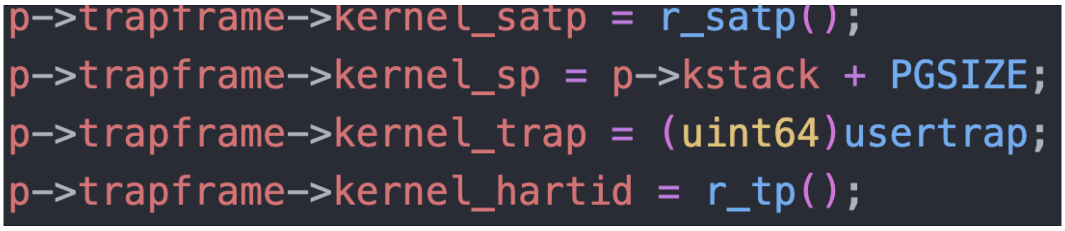
```
intr_off();
```

2. set stvec = usertrap in TRAMPOLINE



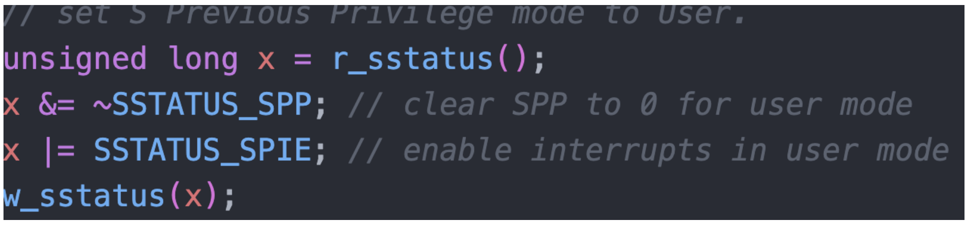
```
uint64 trampoline_uservec = TRAMPOLINE + (uservec - trampoline);
w_stvec(trampoline_uservec);
```

3. set up some value about kernel ,for next trap to use(especially kernel_trap = usertrap)



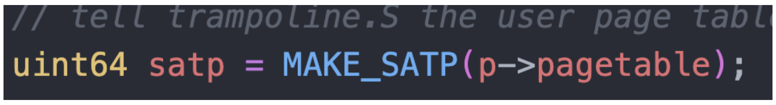
```
p->trapframe->kernel_satp = r_satp();
p->trapframe->kernel_sp = p->kstack + PGSIZE;
p->trapframe->kernel_trap = (uint64)usertrap;
p->trapframe->kernel_hartid = r_tp();
```

4. set sstatus



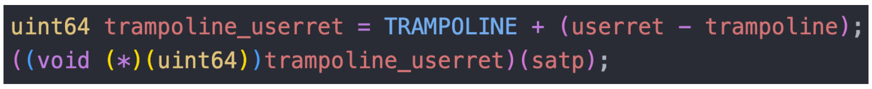
```
// set S Previous Privilege mode to User.
unsigned long x = r_sstatus();
x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
x |= SSTATUS_SPIE; // enable interrupts in user mode
w_sstatus(x);
```

5. get and make page table for user



```
// tell trampoline.S the user page tabl
uint64 satp = MAKE_SATP(p->pagetable);
```

6. get address of userrep() and call it



```
uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
((void (*)(uint64))trampoline_userret)(satp);
```

page fault

what to do when raising exceptions

many kernels use page faults to implementment copy-on-write(COW) fork

xv6’s fork use umvcopy: allocates physical memory and copies parent’s memory

umvcopy will copy all vaild physical space to new space allocated by kalloc()

- RISCV distinguishes 3 kinds of page faults: load~~~,store~~~,instructions~~~ (the scause reg will log this info).
- COW works by set all entry at read-only, if anyone plan to write, RISCV raises a page-fault exception, kernel will responds by allocating a new physical page and map to it.Then resumes the faulting instruction.
- this machinism need to implement the book-keeping, because each page was referred by various number of processes.
- one optimization: if only referred by one precess , exception’s handle will not allocate new one.

lazy allocation

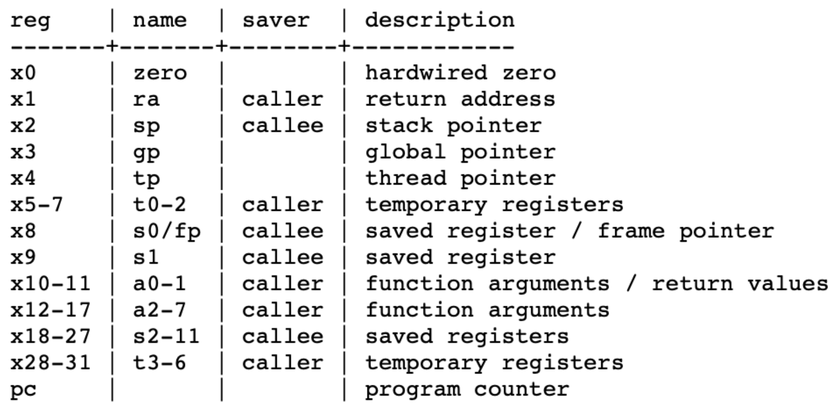
1. app need more space by sark(),kernel just change size but not allocate the new pages.
2. page fault on these new address, kernel corresponds by allocating new pages.

demanding paging

for large application, load all data and text from startup is unwisely. so Morden kernel create page table but set some entry not valid. and read from disk when page fault raises.

lab4-1 Backtrace

The GCC compiler stores the frame pointer of the currently executing function in the register s0



reg	name	saver	description
x0	zero		hardwired zero
x1	ra	caller	return address
x2	sp	callee	stack pointer
x3	gp		global pointer
x4	tp		thread pointer
x5-7	t0-2	caller	temporary registers
x8	s0/fp	callee	saved register / frame pointer
x9	s1	callee	saved register
x10-11	a0-1	caller	function arguments / return values
x12-17	a2-7	caller	function arguments
x18-27	s2-11	callee	saved registers
x28-31	t3-6	caller	temporary registers
pc			program counter

```
Stack
                   .
                   .
      +->          .
      |   +-----------------+   |
      |   | return address  |   |
      |   |   previous fp ------+
      |   | saved registers |
      |   | local variables |
      |   |       ...       | <-+
      |   +-----------------+   |
      |   | return address  |   |
      +------ previous fp   |   |
          | saved registers |   |
          | local variables |   |
      +-> |       ...       |   |
      |   +-----------------+   |
      |   | return address  |   |
      |   |   previous fp ------+
      |   | saved registers |
      |   | local variables |
      |   |       ...       | <-+
      |   +-----------------+   |
      |   | return address  |   |
      +------ previous fp   |   |
          | saved registers |   |
          | local variables |   |
  $fp --> |       ...       |   |
          +-----------------+   |
          | return address  |   |
          |   previous fp ------+
          | saved registers |
  $sp --> | local variables |
          +-----------------+
```

return address lives at a fixed offset (-8) from the frame pointer of a stackframe,

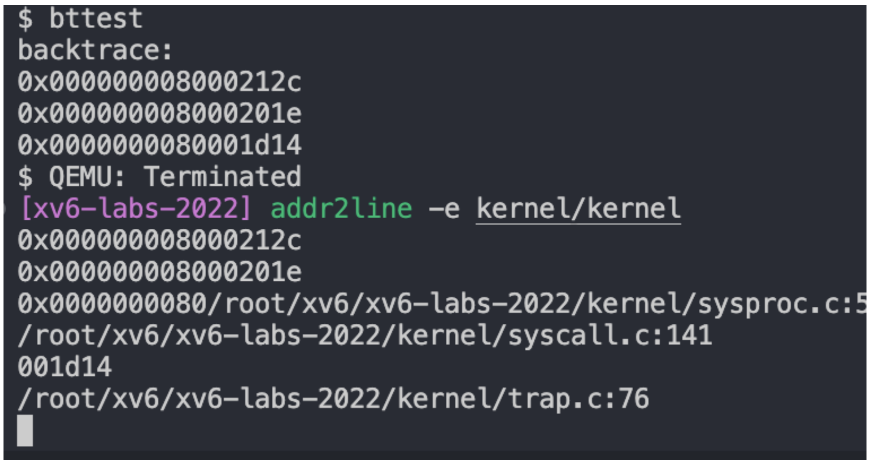
the frame pointer lives at fixed offset (-16) from the frame pointer.

will need a way to recognize that it has seen the last stack frame:

fact is that the each kernel stack consists of a single page-aligned page,

so that all the stack frames are on the same page.

You can use PGROUNDDOWN(fp)
 (see kernel/riscv.h) to identify the page that a frame pointer refers to.

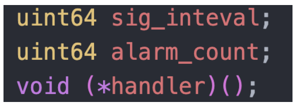


```
$ bttest
backtrace:
0x000000008000212c
0x000000008000201e
0x0000000080001d14
$ QEMU: Terminated
[xv6-labs-2022] addr2line -e kernel/kernel
0x000000008000212c
0x000000008000201e
0x0000000080/root/xv6/xv6-labs-2022/kernel/sysproc.c:5
/root/xv6/xv6-labs-2022/kernel/syscall.c:141
001d14
/root/xv6/xv6-labs-2022/kernel/trap.c:76
```

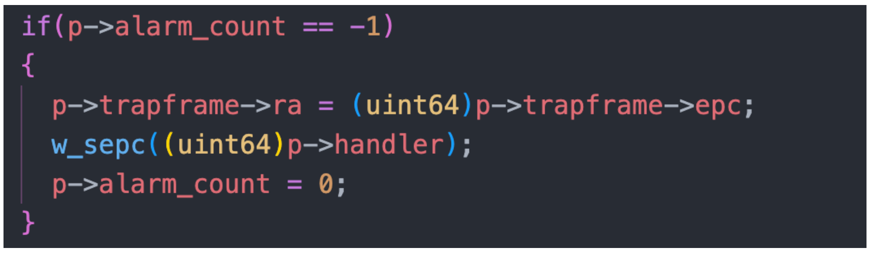
just use int64* to pointer the stack space;

```
void backtrace() {
  uint64* frame_p = (uint64 *)r_fp();
  printf("backtrace:\n");
  struct proc *p = myproc();
  while(PGROUNDDOWN((uint64)frame_p) < (uint64)p->kstack + PGSIZE)
  {
    printf("%p\n", *(frame_p -1));
    frame_p = (uint64 *)*(frame_p - 2);
  }
}
```

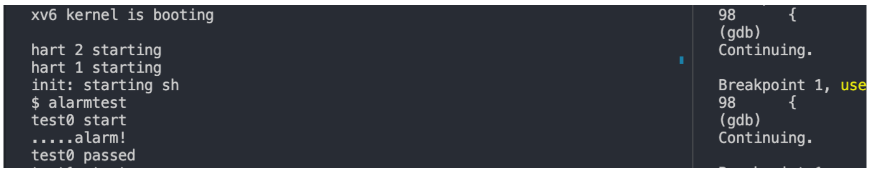
lab4-2



```
uint64 sig_inteval;
uint64 alarm_count;
void (*handler)();
```



```
if(p->alarm_count == -1)
{
  p->trapframe->ra = (uint64)p->trapframe->epc;
  w_sepc((uint64)p->handler);
  p->alarm_count = 0;
}
```



```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ alarmtest
test0 start
.....alarm!
test0 passed
```

```
98      {
(gdb)
Continuing.

Breakpoint 1, use
98      {
(gdb)
Continuing.
```

- when a process's alarm interval expires, the user process executes the handler function. When a trap on the RISC-V returns to user space, what determines the instruction address at which user-space code resumes execution?

  usertrap()

- 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

lab. 4-3

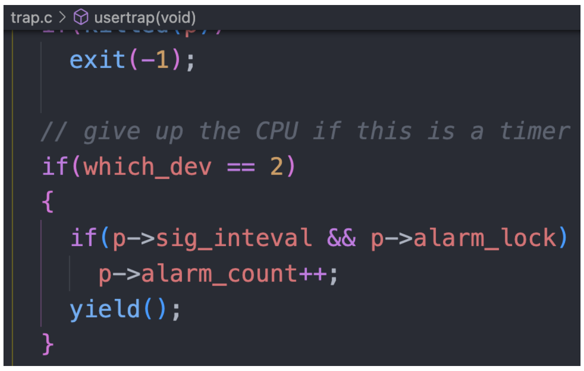
alarm interrupt → unservec() (saved ra in to framework) → usertrap() (clk++ count++) → usertrapret() (lock , a0 = trapframe→a0, ra = trapframe→epc, epc = p→handler) set sepc = p→handler → userret() → handler() → trap again

unservec() (saved ra in to framework) → usertrap() → syscall() →sigreturn() → usertrapret() →

real return

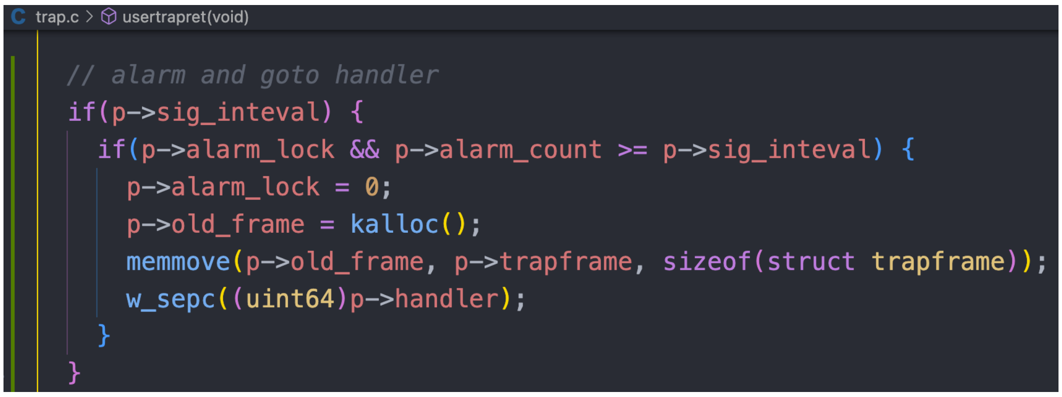
	ra	pc	handler	
before alarm	reg	reg:pc		
uservec	trapframe	reg:pc		
usertrap	trapframe→ra	trapframe→epc		
usertrapret(1)	trapframe→ra	reg:sepc		

usertrapret2)	non	trapframe→ra	reg:sepc	
userret	non	reg:ra	reg:sepc	not use ra
sret	non	reg:ra	reg:pc	
handler		user_stack	reg:pc	
sigreturn()				not just return to pc
ret		reg:pc		



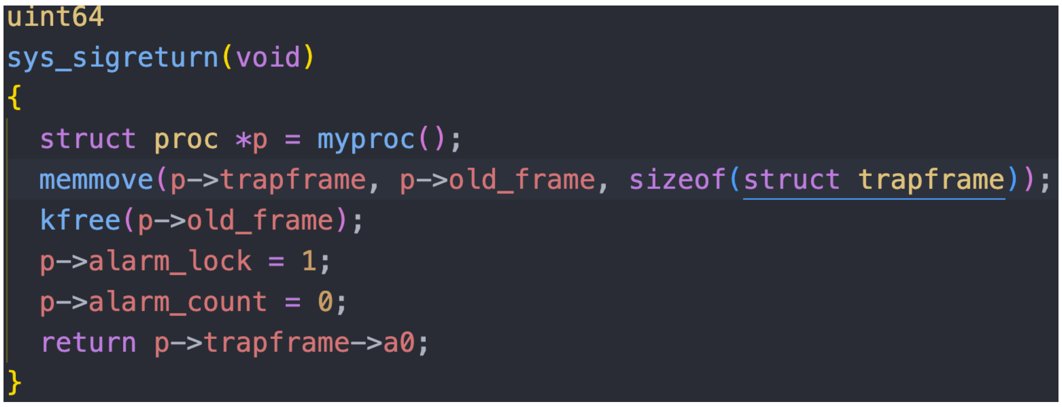
trap.c > usertrap(void)

```c
    exit(-1);

  // give up the CPU if this is a timer
  if(which_dev == 2)
  {
    if(p->sig_inteval && p->alarm_lock)
      p->alarm_count++;
    yield();
  }
```

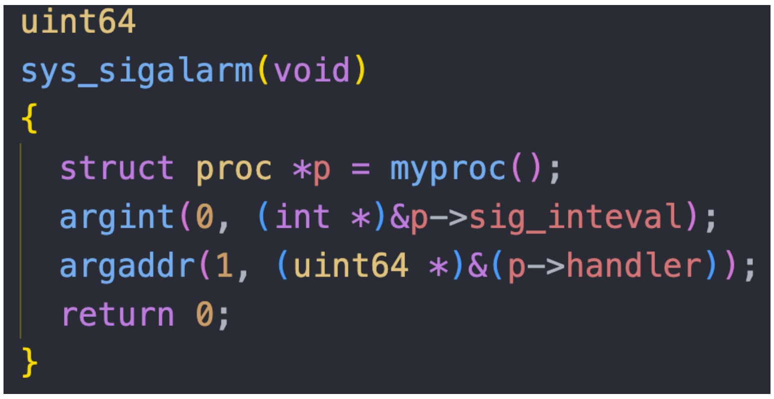


trap.c > usertrapret(void)

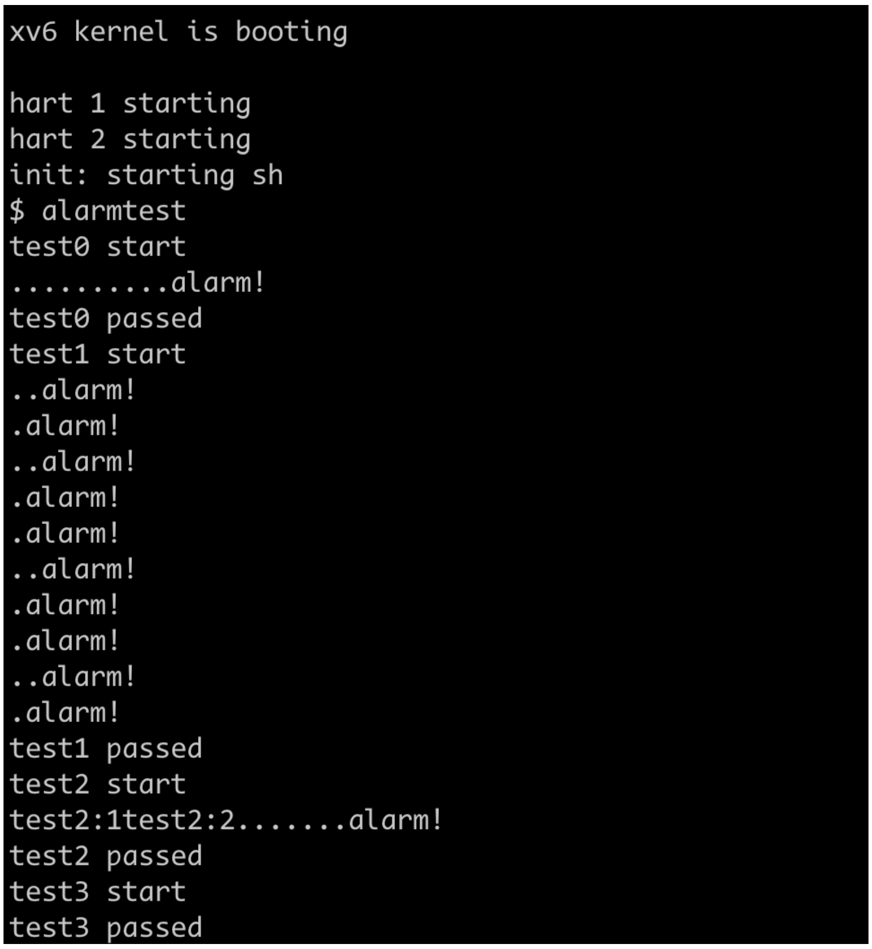
```c
  // alarm and goto handler
  if(p->sig_inteval) {
    if(p->alarm_lock && p->alarm_count >= p->sig_inteval) {
      p->alarm_lock = 0;
      p->old_frame = kalloc();
      memmove(p->old_frame, p->trapframe, sizeof(struct trapframe));
      w_sepc((uint64)p->handler);
    }
  }
```



```c
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  memmove(p->trapframe, p->old_frame, sizeof(struct trapframe));
  kfree(p->old_frame);
  p->alarm_lock = 1;
  p->alarm_count = 0;
  return p->trapframe->a0;
}
```



```c
uint64
sys_sigalarm(void)
{
  struct proc *p = myproc();
  argint(0, (int *)&p->sig_inteval);
  argaddr(1, (uint64 *)&(p->handler));
  return 0;
}
```



```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ alarmtest
test0 start
..........alarm!
test0 passed
test1 start
..alarm!
.alarm!
..alarm!
.alarm!
.alarm!
..alarm!
.alarm!
.alarm!
..alarm!
.alarm!
test1 passed
test2 start
test2:1test2:2.......alarm!
test2 passed
test3 start
test3 passed
```