```
6.S081 2020 Lecture 6: System Call Entry/Exit

Today: user -> kernel transition
  system calls, faults, interrupts enter the kernel in the same way
  important for isolation and performance
  lots of careful design and important detail

What needs to happen when a program makes a system call, e.g. write()?
  [CPU | user/kernel diagram]
  CPU resources are set up for user execution (not kernel)
    32 registers, sp, pc, privilege mode, satp, stvec, sepc, ...
  what needs to happen?
    save 32 user registers and pc
    switch to supervisor mode
    switch to kernel page table
    switch to kernel stack
    jump to kernel C code
  high-level goals
    don't let user code interfere with user->kernel transition
      e.g. don't execute user code in supervisor mode!
    transparent to user code -- resume without disturbing

Today we're focusing on the user/kernel transition
  and ignoring what the system call implemenation does once in the kernel
  but the sys call impl has to be careful and secure also!

What does the CPU's "mode" protect?
  i.e. what does switching mode from user to supervisor allow?
  supervisor can use CPU control registers:
    satp -- page table physical address
    stvec -- ecall jumps here in kernel; points to trampoline
    sepc -- ecall saves user pc here
    sscratch -- address of trapframe
  supervisor can use PTEs that have no PTE_U flag
  but supervisor has no other powers!
    e.g. can't use addresses that aren't the in page table
    so kernel has to carefully set things up so it can work

preview:
  write()                        write() returns             User
  -----------------------------------------------------------------
  uservec() in trampoline.S      userret() in trampoline.S   Kernel
  usertrap() in trap.c           usertrapret() in trap.c
  syscall() in syscall.c           ^
  sys_write() in sysfile.c      ---|

let's watch an xv6 system call entering/leaving the kernel
  xv6 shell writing its $ prompt
  sh.c line 137: write(2, "$ ", 2);
  user/usys.S line 29
    this is the write() function, still in user space
  a7 tells the kernel what system call we want -- SYS_write = 16
  ecall -- triggers the user/kernel transition

let's start by putting a breakpoint on the ecall
  user/sh.asm says write()'s ecall is at address 0xde6

$ make qemu-gdb
(gdb) b *0xde6
(gdb) c
(gdb) delete 1
(gdb) x/3i 0xde4

let's look at the registers
```

```
(gdb) print $pc
(gdb) info reg

$pc and $sp are at low addresses -- user memory starts at zero
C on RISC-V puts function arguments in a0, a1, a2, &c
write() arguments: a0 is fd, a1 is buf, a2 is n

(gdb) x/2c $a1

the shell is printing the $ prompt

what page table is in use?
  (gdb) print/x $satp
        not very useful
  qemu: control-a c, info mem
    there are mappings for six pages
    instructions, data, stack guard (no PTE_U), stack,
    then two high mystery pages: trapframe and trampoline
    there are no mappings for kernel memory, devices, physical mem

let's execute the ecall

(gdb) stepi

where are we?
  (gdb) print $pc
        we're executing at a very high virtual address
  (gdb) x/6i 0x3ffffff000
        these are the instructions we're about to execute
        see uservec in kernel/trampoline.S
        it's the start of the kernel's trap handling code
  (gdb) info reg
        the registers hold user values (except $pc)
  qemu: info mem
        we're still using the user page table
        note that $pc is in the trampoline page, the very last page

we're executing in the "trampoline" page, which contains the start of
the kernel's trap handling code. ecall doesn't switch page tables, so
these kernel instructions have to exist somewhere in the user page
table. the trampoline page is the answer: the kernel maps it at the
top of every user page table. the kernel sets $stvec to the trampoline
page's virtual address. the trampoline is protected: no PTE_U flag.

(gdb) print/x $stvec

can we tell that we're in supervisor mode?
  I don't know a way to find the mode directly
  but observe $pc is executing in a page with no PTE_U flag
    lack of crash implies we are in supervisor mode

how did we get here?
  ecall did three things:
    change mode from user to supervisor
    save $pc in $sepc
      (gdb) print/x $sepc
    jump to $stvec (i.e. set $pc to $stvec)
      the kernel previously set $stvec, before jumping to user space

note: ecall lets user code switch to supervisor mode
  but the kernel immediately gains control via $stvec
  so the user program itself can't execute as supervisor

what needs to happen now?
  save the 32 user register values (for later transparent resume)
```

```
    switch to kernel page table
    set up stack for kernel C code
    jump to kernel C code

  why didn't the RISC-V designers have ecall do these things for us?
    ecall does as little as possible:
    to give O/S designers scope for very fast syscalls / faults / intrs
      maybe O/S can handle some traps w/o switching page tables
      maybe we can map BOTH user and kernel into a single page table
         so no page table switch required
      maybe some registers do not have to be saved
      maybe no stack is required for simple system calls

  there have been many clever schemes invented for kernel entry!
    different amounts of work by CPU
    different strategies for handler s/w
    performance here is often super important

  what are our options at this point for saving user registers?
    can we just write them somewhere convenient in physical memory?
      no, even supervisor mode is constrained to use the page table
    can we first set satp to the kernel page table?
      supervisor mode is allowed to set satp...
      but we don't know the address of the kernel page table at this point!
      and we need a free register to even execute csrw satp, $xx

  two parts to the solution for where to save the 32 user registers:
    1) xv6 maps a 2nd kernel page, the trapframe, into every user page table
       it has space to hold the saved registers
       the kernel gives each process a different trapframe page
       the page at 0x3fffffe000 is the trapframe page
       see struct trapframe in kernel/proc.h
       (but we still need a register holding the trapframe's address...)
    2) RISC-V provides the sscratch register
       the kernel puts a pointer to the trapframe in sscratch
         before entering user space
       supervisor code can swap any register with sscratch
         thus both getting hold of the value in sscratch,
         and simultaneously saving the register's user value

  see this at the start of uservec in trapframe.S:
    csrrw a0, sscratch, a0

  the csrrw has already been executed due to some gdb quirk...

  (gdb) print/x $a0
        address of the trapframe
  (gdb> print/x $sscratch
        0x2, the old first argument (fd)

  now uservec() has 32 saves of user registers to the trapframe, via a0
    so they can be restored later, when the system call returns
    let's skip them

  (gdb) b *0x3ffffff076
  (gdb) c

  now we're setting up to be able to run C code in the kernel
  first a stack
    previously, kernel put a pointer to top of this process's
      kernel stack in trapframe
    look at struct trapframe in kernel/proc.h
    "ld sp, 8(a0)" fetches the kernel stack pointer
    remember a0 points to the trapframe
    at this point the only kernel data the code can
```

```
    get at is the trapframe, so everything has to be loaded from there.

(gdb) stepi

retrieve hart ID into tp

(gdb) stepi

we want to jump to the kernel C function usertrap(), which
  the kernel previously saved in the trapframe.
  "ld t0, 16(a0)" fetches it into t0, we'll use it in a moment,
    after switching to the kernel page table

(gdb) stepi

load a pointer to the kernel pagetable from the trapframe,
and load it into satp, and issue an sfence to clear the TLB.

(gdb) stepi
(gdb) stepi
(gdb) stepi

why isn't there a crash at this point?
  after all we just switched page tables while executing!
  answer: the trampoline page is mapped at the same virtual address
    in the kernel page table as well as every user page table

(gdb) print $pc
qemu: info mem

with the kernel page table we can now use kernel functions and data

the jr t0 is a jump to usertrap() (using t0 retrieved from trapframe)

(gdb) print/x $t0
(gdb) x/4i $t0
(gdb) stepi
(gdb) tui enable

we're now in usertrap() in kernel/trap.c
  various traps come here, e.g. errors, device interrupts, and system calls
  usertrap() looks in the scause register to see the trap cause
    see Figure 10.3 on page 102 of The RISC-V Reader
  scause = 8 is a system call

(gdb) next ... until syscall()
(gdb) step
(gdb) next

now we're in syscall() kernel/syscall.c
myproc() uses tp to retrieve current struct proc *
p->xxx is usually a slot in the current process's struct proc

syscall() retrieves the system call number from saved register a7
  p->trapframe points to the trapframe, with saved registers
  p->trapframe->a7 holds 16, SYS_write
  p->trapframe->a0 holds write() first argument -- fd
  p->trapframe->a1 holds buf
  p->trapframe->a2 holds n

(gdb) next ...
(gdb) print num

then dispatches through syscall[num], a table of functions
```

```
(gdb) next ...
(gdb) step

aha, we're in sys_write.
at this point system call implementations are fairly ordinary C code.
let's skip to the end, to see how a system call returns to user space.

(gdb) finish

notice that write() produced console output (the shell's $ prompt)
back to syscall()
the p->tf->a0 assignment causes (eventually) a0 to hold the return value
  the C calling convention on RISC-V puts return values in a0

(gdb) next

back to usertrap()

(gdb) print p->trapframe->a0

write() returned 2 -- two characters -- $ and space

(gdb) next
(gdb) step

now we're in usertrapret(), which starts the process of returning
  to the user program

we need to prepare for the next user->kernel transition
  stvec = uservec (the trampoline), for the next ecall
  traframe satp = kernel page table, for next uservec
  traframe sp = top of kernel stack
  trapframe trap = usertrap
  trapframe hartid = hartid (in tp)

at the end, we'll use the RISC-V sret instruction
  we need to prepare a few registers that sret uses
  sstatus -- set the "previous mode" bit to user
  sepc -- the saved user program counter (from trap entry)

we're going to switch to the user page table while executing
  not OK in usertrapret(), since it's not mapped in the user page table.
  need a page that's mapped in both user and kernel page table -- the trampoline.
  jump to userret in trampoline.S

(gdb) tui disable
(gdb) step
(gdb) x/8i 0x3ffffff090

a0 holds TRAPFRAME
a1 holds user page table address
the csrw satp switches to the user address space

(gdb) stepi
(gdb) stepi
(gdb) stepi

the csrw scratch puts the user a0 into sscratch
  just before sret we'll do a swap,
  so that a0 holds the user a0 and sscratch holds trapframe pointer.
  which is what uservec expects.

now 32 loads from the trapframe into registers
  these restore the user registers
  let's skip over them
```

```
(gdb) b *0x3ffffff10a
(gdb) c

here's the csrw that swaps a0 with sscratch

(gdb) stepi
(gdb) print/x $a0 -- the return value from write()
(gdb) print/x $sscratch -- trapframe address for uservec

now we're at the sret instruction

(gdb) print $pc
(gdb) stepi
(gdb) print $pc

now we're back in the user program ($pc = 0x0xdea)
  returning 2 from the write() function

(gdb) print/x $a0

and we're done with a system call!

summary
  system call entry/exit is far more complex than function call
  much of the complexity is due to the requirement for isolation
    and the desire for simple and fast hardware mechanisms
  a few design questions to ponder:
    can an evil program abuse the entry mechanism?
    can you think of ways to make the hardware or software simpler?
    can you think of ways to make traps faster?
```