OS MMXX

MIT ;)

https://pdos.csail.mit.edu/6.828/2020

Prechod user ←→kernel pomocou systemovych volani

Tema

Tema

- Ako funguje prechod do/z jadra pri systemovych volaniach

Tema

- Ako funguje prechod do/z jadra pri systemovych volaniach
- Ponorime sa do jadra OS

write(fd, buf, n)

write(fd, buf, n)

- Mozeme pouzit v uzivatelskom priestore funkciu (z pohladu jazyka C) na [za|vy] volanie (uskutocnenie) sluzby jadra?

write(fd, buf, n)

- Mozeme pouzit v uzivatelskom priestore funkciu (z pohladu jazyka C) na [za|vy] volanie (uskutocnenie) sluzby jadra?

- Bolo by to rychle a efektivne
- Taktiez flexibilne na prenos komplexnejsich udajovych typov
- A islo by o programatorom znamy mechanizmus

write(fd, buf, n)

- Zial, nemozeme pouzit takyto mechanizmus!

write(fd, buf, n)

- Zial, nemozeme pouzit takyto mechanizmus!
- Dovodom je IZOLACIA procesov

write(fd, buf, n)

- Zial, nemozeme pouzit takyto mechanizmus!

- Dovodom je IZOLACIA procesov
- Izolacia ovplyvnuje vacsinu navrhu jadra OS

Co to je izolacia

Co to je izolacia

- Vynutena separacia z dovodu zapuzdrenia dosledkov zlyhani

Co to je izolacia

- Vynutena separacia z dovodu zapuzdrenia dosledkov zlyhani
- Predmetom izolacie je zvycajne proces

Co to je izolacia

- Vynutena separacia z dovodu zapuzdrenia dosledkov zlyhani
- Predmetom izolacie je zvycajne proces
- Zabranuje
  - Procesu X modifikovat/citat udaje o procese Y (napr. modifikovat tabulku deskriptorov, r/w pristupy do pamate…)
  - Procesu miesat sa do uloh a sluzieb OS

Hlavne nastroje izolacie

Hlavne nastroje izolacie

- Virtualny adresny priestor procesu

Hlavne nastroje izolacie

- Virtualny adresny priestor procesu
- Privilegovany rezim cinnosti CPU (proces nemoze robit I/O operacie a menit dolezite systemove registre CPU)

Hlavne nastroje izolacie

- Virtualny adresny priestor procesu
- Privilegovany rezim cinnosti CPU (proces nemoze robit I/O operacie a menit dolezite systemove registre CPU)
- Vynutena kontrola toku riadenia pomocou systemovych volani

Hlavne nastroje izolacie

- Virtualny adresny priestor procesu
- Privilegovany rezim cinnosti CPU (proces nemoze robit I/O operacie a menit dolezite systemove registre CPU)
- Vynutena kontrola toku riadenia pomocou systemovych volani; **prechod user→ kernel!!!**

Privilegovany rezim CPU

Privilegovany rezim CPU

- RISC-V CPU 2 rezimy: supervisor, user

Privilegovany rezim CPU

- RISC-V CPU 2 rezimy: supervisor, user
- Kernel (jadro OS) bezi v rezime supervisor
- Bezny uzivatelsky program v rezime user

Privilegovany rezim CPU

- RISC-V CPU 2 rezimy: supervisor, user
- Kernel (jadro OS) bezi v rezime supervisor
- Bezny uzivatelsky program v rezime user
- Co moze supervisor oproti user naviac

Privilegovany rezim CPU

- RISC-V CPU 2 rezimy: supervisor, user
- Kernel (jadro OS) bezi v rezime supervisor
- Bezny uzivatelsky program v rezime user
- Co moze supervisor oproti user naviac:
  - I/O operacie (pristup k zariadeniam)
  - Nastavovanie adresneho priestoru (virtualna pamat)
  - R/W pristup k specialnym registrom CPU

Privilegovany rezim CPU

- RISC-V CPU 2 rezimy: supervisor, user
- Kernel (jadro OS) bezi v rezime supervisor
- Bezny uzivatelsky program v rezime user
- Co moze supervisor oproti user naviac:
  - I/O operacie (pristup k zariadeniam)
  - Nastavovanie adresneho priestoru (virtualna pamat)
  - R/W pristup k specialnym registrom CPU
- Temer kazdy CPU vyuziva tento princip

Co sa deje pri systemovom volani?

Co sa deje pri systemovom volani?

- Procesor je nastaveny na beh user programu

Co sa deje pri systemovom volani?

- Procesor je nastaveny na beh user programu
- Co je potrebne urobit?

Co sa deje pri systemovom volani?

- Procesor je nastaveny na beh user programu
- Co je potrebne urobit?
  - Ulozit 32 user registrov a PC
  - Zmenit mod CPU z User na Supervisor (Kernel)
  - Zmenit tabulku stranok z User na Kernel
  - Zmenit zasobnik z User na Kernel
  - Skocit na vykonavanie kodu v C

Ciele riadeneho prechodu

1.

2.

Ciele riadeneho prechodu

1. nedovolit zasahovat user programu do prechodu user → kernel

2. prechod musi byt pre user program transparentny (bez akehokolvek zasahu)

Preco prepinat cpu mod?

- Preco umoznit prechod do supervisor modu?

Preco prepinat cpu mod?

- Preco umoznit prechod do supervisor modu?
- Pristup k specialnym registrom CPU
  - satp – tabulka stranok
  - stvec – instrukcia ecall skoci na adresu ulozenu v tomto registri (v xv6 ukazuje na kod trampoliny)
  - sepc – instrukcia ecall sem ulozi hodnotu user pc
  - sscratch – pomocny register; xv6 ho vyuziva na ulozenie adresy pamatovej oblasti trapframe

Preco prepinat cpu mod?

- Preco umoznit prechod do supervisor modu?
- Pristup k specialnym registrom CPU
  - satp – tabulka stranok
  - stvec – instrukcia ecall skoci na adresu ulozenu v tomto registri (v xv6 ukazuje na kod trampoliny)
  - sepc – instrukcia ecall sem ulozi hodnotu user pc
  - sscratch – pomocny register; xv6 ho vyuziva na ulozenie adresy pamatovej oblasti trapframe
- Supervisor ma pristup ku strankam, ktore nemaju nastaveny priznak PTE_U

Preco prepinat cpu mod?

- Preco umoznit prechod do supervisor modu?
- Supervisor ma priamy pristup k hw

Preco prepinat cpu mod?

- Preco umoznit prechod do supervisor modu?
- Supervisor ma priamy pristup k hw

- Nic ine nema naviac…
- To znamena, ze ani v Supervisor mode nie je mozne pristupovat k adresam, pre ktore nie je platne mapovanie v tabulke stranok

Kod xv6 prechodu user→kernel

Kod xv6 prechodu user→kernel

```
User space

-----------------------------------------------------

Kernel space
```

Kod xv6 prechodu user→kernel

```
User space    write()
--------------------------------------------------------
Kernel space
```

Kod xv6 prechodu user→kernel

```
User space     write()
-------------------------------------------------------
Kernel space   uservec()
               usertrap()
               syscall()
               sys_write()
```

Kod xv6 prechodu user→kernel

```
User space     write()
-------------------------------------------------------
Kernel space   uservec()       userret()
               usertrap()      usertrapret()
               syscall()           ^
               sys_write() -------|
```

Kod xv6 prechodu user→kernel

```
User space    write()         navrat z write()
-------------------------------------------------
Kernel space  uservec()       userret()
              usertrap()      usertrapret()
              syscall()           ^
              sys_write() ------|
```

Kod xv6 prechodu user→kernel

```
User space     write()          navrat z write()
-------------------------------------------------------
Kernel space   uservec()        userret()        trampoline.S
               usertrap()       usertrapret()    trap.c
               syscall()           ^             syscall.c
               sys_write() ------->|             sysfile.c
```

Tok riadenia

Tok riadenia

write()

Tok riadenia

write()

trampoline.S / uservec

Tok riadenia

write()

trampoline.S / uservec

trap.c / usertrap()

Tok riadenia

write()

trampoline.S / uservec

trap.c / usertrap()

syscall.c / syscall()

Tok riadenia

write()

trampoline.S / uservec

trap.c / usertrap()

syscall.c / syscall()

sysfile.c / sys_write()

Tok riadenia

write()

trampoline.S / uservec

trap.c / usertrap()

syscall.c / syscall()

sysfile.c / sys_write()

syscall.c / syscall()

Tok riadenia

write()

trampoline.S / uservec

trap.c / usertrap()

syscall.c / syscall()

sysfile.c / sys_write()

syscall.c / syscall()

trap.c / usertrapret()

Tok riadenia

write()

trampoline.S / uservec

trap.c / usertrap()

syscall.c / syscall()

sysfile.c / sys_write()

syscall.c / syscall()

trap.c / usertrapret()

trampoline.S / userret

Tok riadenia

write()

    trampoline.S / uservec

        trap.c / usertrap()

            syscall.c / syscall()

                sysfile.c / sys_write()

            syscall.c / syscall()

        trap.c / usertrapret()

    trampoline.S / userret

write()

Ako syscall vstupi do jadra

Ako syscall vstupi do jadra

- napriklad xv6 shell vypisujuci prompt (‘$’)

Ako syscall vstupi do jadra

- napriklad xv6 shell vypisujuci prompt (‘$’)
  - user/sh.c:136 fprintf(2, “$ “);
  - user/printf.c:103 vprintf(…)
  - user/printf.c:64 putc(fd, c)
  - user/printf.c:12 write(fd, &c, 1);

Ako syscall vstupi do jadra

- napriklad xv6 shell vypisujuci prompt (‘$’)
  - user/sh.c:136 fprintf(2, “$ “);
  - user/printf.c:103 vprintf(…)
  - user/printf.c:64 putc(fd, c)
  - user/printf.c:12 write(fd, &c, 1);

  - user/usys.S:29
    - li a7, SYS_write (→kernel/syscall.h:17)
    - ecall (vyvola RIADENY prechod user→kernel)

Ako syscall vstupi do jadra

- napriklad xv6 shell vypisujuci prompt (‘$’)
  - user/sh.c:136 fprintf(2, “$ “);
  - user/printf.c:103 vprintf(…)
  - user/printf.c:64 putc(fd, c)
  - user/printf.c:12 write(fd, &c, 1);

  - user/usys.S:29
    - li a7, SYS_write (→kernel/syscall.h:17)
    - ecall (vyvola RIADENY prechod user→kernel)
  - user/sh.asm:1916 adresa ecall 0xde4

Ako syscall vstupi do jadra

$ make CPUS=1 qemu-gdb …

(gdb) c

Ctrl+C

(gdb) b *0xde2

(gdb) c

V konzole qemu-gdb zadame Enter a vratime sa do okna s gdb

Nachadzame sa pred systemovym volanim write()

Ako syscall vstupi do jadra

(gdb) symbol-file user/sh.o

(gdb) ctrl-x 2

(gdb) ctrl-x 2

V hornom okne mame zobrazene registre CPU

V strednom vidime asm kod

V dolnom zadavame prikazy gdb

Ako syscall vstupi do jadra

(gdb) b *0xde8

→ pozri okno behu xv6!

(gdb) c

→ pozri okno behu xv6!

Znak sa na konzolu vypise…

Ako syscall vstupi do jadra

(gdb) si

→ user/sh.asm: adresa 0xe7c

(gdb) si 4

→ user/sh.asm:2247 adresa 0xf88

(gdb) si 11

→ user/sh.asm: adresa 0xe62 (fnc putc(‘ ‘))

(gdb) c

→ sme na adrese 0xde2 (syscall write())

→ vid okno behu xv6

Ako syscall vstupi do jadra

(gdb) c

→ sme spat z kernelu na konci write()

→ vid okno behu xv6 (pribudla medzera)

(gdb) si 5

→ user/sh.asm:2247 adresa 0xf88

(gdb) si 4

→ user/sh.asm: adresa 0x10ee (vprintf())

(gdb) si 15

→ user/sh.asm: adresa 0x1132 (fprintf())

Ako syscall vstupi do jadra

(gdb) si

(gdb) si

(gdb) si

(gdb) si

→ naspat v getcmd()

Ctrl-x 2 ……zobrazi nam C kod funkcie getcmd()

Podme znovu pred syscall write

(gdb) c

Ako syscall vstupi do jadra

- Registre $pc a $sp su na nizkych adresach
- Argumenty funkcii su v registroch a0, a1, a2…
- Argumenty systemoveho volania write()
  - a0 je fd
  - a1 je buf
  - a2 je n

(gdb) x/2c $a1

sh vypisuje prompt ‘$ ‘

Ako syscall vstupi do jadra

- Aka tabulka stranok sa aktualne pouziva?

(gdb) p/x $satp

- Informacia o adrese nie je velmi uzitocna…

Ako syscall vstupi do jadra

- Aka tabulka stranok sa aktualne pouziva?

(gdb) p/x $satp

- Informacia o adrese nie je velmi uzitocna…
- Podme na okno qemu so spustenym xv6
  - ctrl-a c
  - info mem – iba 6 stranok namapovanych!
  - info mtree

Ako syscall vstupi do jadra

- Vykonajme instrukciu ecall

(gdb) si 2

- Kde sa nachadzame?

(gdb) print $pc → velmi vysoka adresa!

(gdb) x/6i 0x3ffffff000

  - 6 instrucii, ktore sa budu vykonavat
  - Vid uservec v kernel/trampoline.S
  - Ide o zaciatok kodu trampoliny do kernelu

Ako syscall vstupi do jadra

- Vsetky registre maju hodnoty z user programu (okrem $pc)

(gdb) info reg

- Este stale sa pouziva tabulka stranok user procesu

(gdb) p/x $satp

Ako syscall vstupi do jadra

- Co sa udialo instrukciou ecall?

- Procesor sa prepol do supervisor modu
- Ako to vieme? Vykonava sa instrukcia na adrese danej registrom $stvec (ecall zoberie hodnotu v $stvec a nastavi $pc)
- Trampolina nie je pristupna pre user program (nie je nastaveny priznak PTE_U)!

(gdb) p/x $stvec

Ako syscall vstupi do jadra

- Co sa udialo instrukciou ecall?

1. Zmeni sa mod procesora z user na supervisor
2. Ulozi sa hodnota $pc do $sepc

   (gdb) p/x $sepc
3. Vykonavanie skoci na adresu ulozenu v $stvec (register $pc sa nastavi na hodnotu, ktora je v $stvec)

Ako syscall vstupi do jadra

- Kernel musel nastavit $stvec pred prechodom do user priestoru!!!

- ecall umoznuje zmenit mod CPU z user na superuser
- Ale iba kernel ma kontrolu nad tym, CO sa zacne v tomto rezime vykonavat, a to na zaklade obsahu registra $stvec
- Program uzivatela NEDOKAZE zmenit $stvec

Ako syscall vstupi do jadra

- Co dalsie sa musi urobit? Zatial mame “iba” zmeneny mod CPU

Ako syscall vstupi do jadra

- Co dalsie sa musi urobit? Zatial mame “iba” zmeneny mod CPU

- Ulozit registre CPU, ktore vyuziva user (aby sa mohli neskor obnovit pri prechode z kernelu spat k vykonavaniu kodu uzivatelskeho programu)
- Zmenit tabulku stranok z user na kernel
- Nastavit zasobnik pre C-ckovsky kod kernelu
- Skocit na vykonavanie C kodu

Ako syscall vstupi do jadra

- Preco je ecall tak navrhnuty, ze nerobi tieto ostatne cinnosti?

Ako syscall vstupi do jadra

- Preco je ecall tak navrhnuty, ze nerobi tieto ostatne cinnosti? Aby bolo mozne implementovat velmi rychlu obsluhu vynimiek:

- Obsluzit niektore vynimky bez nutnosti zmeny tabulky stranok
- Ak by kernel a user pouzivali to iste mapovanie, netreba menit tabulku stranok
- Ak je obsluha pisana v asm, nie je nutne pouzivat zasobnik

Ako syscall vstupi do jadra

- Ake moznosti mame na RISC-V pre ulozenie stavu CPU (registrov uzivatela)?

Ako syscall vstupi do jadra

- Ake moznosti mame na RISC-V pre ulozenie stavu CPU (registrov uzivatela)?

- Mozeme ich len tak zapisat niekam do fyzickej pamate?

Ako syscall vstupi do jadra

- Ake moznosti mame na RISC-V pre ulozenie stavu CPU (registrov uzivatela)?

- Mozeme ich len tak zapisat niekam do fyzickej pamate? Nie, lebo strankovanie!

Ako syscall vstupi do jadra

- Ake moznosti mame na RISC-V pre ulozenie stavu CPU (registrov uzivatela)?

- Mozeme ich len tak zapisat niekam do fyzickej pamate? Nie, lebo strankovanie!
- Mozeme najprv zmenit $satp na kernel stranky?

Ako syscall vstupi do jadra

- Ake moznosti mame na RISC-V pre ulozenie stavu CPU (registrov uzivatela)?

- Mozeme ich len tak zapisat niekam do fyzickej pamate? Nie, lebo strankovanie!
- Mozeme najprv zmenit $satp na kernel stranky?
  - Supervisor mod to umoznuje
  - Ale nepozname ADRESU stranky urovne L2!
  - A $satp pred prepisanim musi byt tiez odlozeny!

Ako syscall vstupi do jadra

- Ulozenie registrov pozostava z 2 casti
  1. Z dostupneho pamatoveho miesta, kam sa ulozia
  2. Z adresy tohto pamatoveho miesta

Ako syscall vstupi do jadra

- Ulozenie registrov pozostava z 2 casti
  1. Z dostupneho pamatoveho miesta, kam sa ulozia
  2. Z adresy tohto pamatoveho miesta

- Xv6 mapuje do user stranok 2 stranky jadra, ku ktorym user nema pristup: trampolinu (uplne posledna stranka virtualneho priestoru) a pod nou tzv. trapframe
- RISC-V poskytuje 1 pracovny register, ku ktoremu nema user pristup, sscratch

Ako syscall vstupi do jadra

- Trapframe
  - 1 stranka je dostatocne velka na ulozenie 32 registrov CPU
  - Kazdy proces ma mapovany trapframe na tu istu virtualnu adresu, ale vzdy do ineho ramca v RAM
  - Adresa je 0x3ffffe000
  - Vid strukturu trapframe v kernel/proc.h

Ako syscall vstupi do jadra

- Trapframe
  - 1 stranka je dostatocne velka na ulozenie 32 registrov CPU
  - Kazdy proces ma mapovany trapframe na tu istu virtualnu adresu, ale vzdy do ineho ramca v RAM
  - Adresa je 0x3ffffe000
  - Vid strukturu trapframe v kernel/proc.h

- Pamatove miesto mame, ale potrebujeme mat niekde ulozenu adresu tohto miesta!

Ako syscall vstupi do jadra

- sscratch
  - Pred prechodom do user priestoru kernel nastavi nielen $stvec, ale aj $sscratch register
  - Ulozi do neho adresu na trapframe strukturu
  - RISC-V poskytuje instrukciu na vymenu lubovolneho registra s $sscratch (prehodenie hodnot medzi registrom a $sscratch)

- Prva instrukcia uservec() v trapframe.S:

  csrrw a0, sscratch, a0

Ako syscall vstupi do jadra

(gdb) p/x $sscratch

(gdb) p/x $a0

(gdb) si

(gdb) p/x $sscratch

(gdb) p/x $a0

Dalej kod uklada 32 registrov CPU do struktury trapframe pomocou nepriamej adresacie cez register $a0

Ako syscall vstupi do jadra

az po instrukciu csrr t0, sscratch (30x si)

(gdb) si

- este je potrebne ulozit povodnu hodnotu $a0 (ta sa nachadza v $sscratch)

(gdb) si

(gdb) si

- po ulozeni registrov je potrebne pripravit zasobnik a skocit na vykonavanie C kodu

Ako syscall vstupi do jadra

- Pozrime sa na strukturu trapframe do kernel/proc.h
- Kde sa v nej nachadza adresa zasobnika jadra?

Ako syscall vstupi do jadra

- Pozrime sa na strukturu trapframe do kernel/proc.h
- Kde sa v nej nachadza adresa zasobnika jadra? V strukture trapframe od 8. bajtu
  - ld sp, 8(a0)
  - Nezabudajme, ze v a0 je adresa zaciatku trapframe; takze po pripocitani hodnoty 8 k hodnote v registri a8 dostaneme adresu, z ktorej sa nacita hodnota do registra $sp
  - Vsetky udaje potrebne pre obnovenie behu kodu jadra su k dispozicii v strukture trapframe

Ako syscall vstupi do jadra

- Vid komentare v kernel/trampoline.S pre dalsie instrukcie pri krokovani pomocou “si”
  - Obnovi sa ID jadra CPU
  - Nacita sa adresa funkcie v C, ktorou bude pokracovat spracovanie po ukonceni asm kodu
  - Nacita sa adresa tabuliek jadra do $satp, vycisti sa TLB – preco neprislo ku padu jadra po vymene tabuliek stranok?

Ako syscall vstupi do jadra

- Vid komentare v kernel/trampoline.S pre dalsie instrukcie pri krokovani pomocou “si”
  - Obnovi sa ID jadra CPU
  - Nacita sa adresa funkcie v C, ktorou bude pokracovat spracovanie po ukonceni asm kodu
  - Nacita sa adresa tabuliek jadra do $satp, vycisti sa TLB – preco neprislo ku padu jadra po vymene tabuliek stranok?
    - Lebo na rovnaku virtualnu adresu je mapovana trampolina aj v tabulkach jadra!!!
    - (gdb) p/x $satp
    - Qemu: info mem

Ako syscall vstupi do jadra

- Napokon mozno skocit do funkcie usertrap()
  - Jej adresa sa nachadza v registri $t0

(gdb) p/x $t0

(gdb) x/4i $t0

(gdb) si

(gdb) tui enable

- A konecne sa nachadzame v C kode
- usertrap() v kernel/trap.c

Ako syscall vstupi do jadra

- Uloha usertrap()
  - Dispecing roznych typov preruseni, chybovych stavov a systemovych volani prichadzajucich z user modu CPU
  - Zistenie priciny vynimocneho stavu pomocou hodnoty registra $scause
  - Vid obrazok 10.3 na strane 102 manualu “The RISC-V Reader”
  - $scause rovne hodnote 8 je systemove volanie (environment call z U-mode, ecall)

Ako syscall vstupi do jadra

- Az po funkciu syscall() (gdb) n

  (gdb) s

  (gdb) n

- Nachadzame sa vo funkcii syscall() v subore kernel/syscall.c

Ako syscall vstupi do jadra

- Funkcia syscall()
  - p→trapframe ukazuje na pamatovu oblast, v ktorej su uchovane VSETKY registre user programu

- p→trapframe→a7 uchovava cislo systemoveho volania (v nasom pripade hodnotu 16, SYS_write)
- p→trapframe→a0 uchovava 1. argument (fd)
- p→trapframe→a1 uchovava 2. argument (buf)
- p→trapframe→a2 uchovava 3. argument (n)

Ako syscall vstupi do jadra

(gdb) n

(gdb) p num

(gdb) n

(gdb) s

- A sme vo funkcii sys_write()

Ako syscall vstupi do jadra

(gdb) n

(gdb) p num

(gdb) n

(gdb) s

- A sme vo funkcii sys_write()

- Dalsi kod je nezaujimavy – standardne C
- Pozrime sa, ako sa vratime spat do user modu

Ako syscall vstupi do jadra

(gdb) finish

- Vratime sa spat do funkcie syscall()
- Funkcia sys_write() vykonala vypis na monitor

- Dolezite: PO vykonani systemoveho volania sa vysledok ulozi do p→trapframe→a0, aby sa pri obnoveni obsahu registrov dostala navratova hodnota do registra $a0
- Preco $a0? Konvencia volani C na RISC-V!

Ako syscall vstupi do jadra

- Z pohladu uzivatelskeho programu je vyvolanie systemoveho volania volanim “obycajnej funkcie”
- Konvencia volani C definuje, ze navratova hodnota volanej funkcie sa u volajuceho musi objavit v registri $a0
- Preto sa navratova hodnota systemoveho volania v jadre “injektazou” cez trapframe dostane az k uzivatelskemu kodu tiez v registri $a0

Prechod kernel→user

(gdb) p/x $a0

(gdb) p/x $sscratch

- $a0 obsahuje navratovu hodnotu systemoveho volania sys_write()
- $sscratch obsahuje adresu trapframe

Prechod kernel→user

(gdb) n

(gdb) n

(gdb) s

- Nachadzame sa vo funkcii usertrapret(), ktora riesi navrat spat do uzivatelskeho programu

Prechod kernel→user

(gdb) n

(gdb) n

(gdb) s

- Nachadzame sa vo funkcii usertrapret(), ktora riesi navrat spat do uzivatelskeho programu

- V prvom rade treba prichystat vsetky udaje potrebne pre nasledujuci prechod z user→kernel

Prechod kernel→user

- Priprave pre dalsi prechod user→kernel

- $stvec ← uservec() kvoli dalsiemu ecall
- tf satp ←kernel page table kvoli dalsiemu volaniu uservec()
- tf sp ←vrchol zasobnika kernelu
- tf trap ←usertrap()
- tf hartid ←hartid (nachadza sa v $tp)

Prechod kernel→user

- Prechod do user modu sposobuje instrukcia **sret** (na architekture RISC-V)

Prechod kernel→user

- Prechod do user modu sposobuje instrukcia **sret** (na architekture RISC-V)
- Tato instrukcia vyuziva na svoju cinnost viacere registre (podobne ako ecall)
  - $sstatus (pomocou neho sa nastavi bit “predosleho modu” na user)
  - $sepc (adresa instrukcie user programu, ktorou sa bude pokracovat vykonavanie kodu – tu mame ulozenu p→trapframe→epc)
- Pomocou ‘n’ prejdime na posledny riadok (c.128) funkcie

Prechod kernel→user

- Teraz by bolo vhodne zmenit tabulku stranok spat na uzivatelsku
- To sa vsak NEDA v usertrapret(), pretoze nie je mapovana do uzivatelskeho priestoru!

Prechod kernel→user

- Teraz by bolo vhodne zmenit tabulku stranok spat na uzivatelsku
- To sa vsak NEDA v usertrapret(), pretoze nie je mapovana do uzivatelskeho priestoru!
- Preto sa v usertrapret() na konci vypocita adresa funkcie userret() v stranke trampoliny, a tam sa odovzda riadenie

(gdb) tui disable

Pomocou ‘si’ sa v gdb presunme na 0x3ffffff090

(gdb) x/8i 0x3ffffff090

Prechod kernel→user

- $a0 obsahuje adresu trapframe
- $a1 obsahuje adresu stranok uzivatelskeho priestoru
- Instrukciou csrw satp sa nastavi tabulka stranok uzivatelskeho procesu
- Vid v okne qemu info mem
- Preco pri vykonavani kodu nebude vynimka?

(gdb) si

(gdb) si

Prechod kernel→user

- Uzivatelsky $a0 sa umiestni do $sscratch a tesne pred navratom do user priestoru sa prehodi $sscratch s $a0 (vtedy bude ukazovat na trapframe, takze pri dalsom vyvolani ecall bude v $sscratch hodnota trapframe)

Prechod kernel→user

- Uzivatelsky $a0 sa umiestni do $sscratch a tesne pred navratom do user priestoru sa prehodi $sscratch s $a0 (vtedy bude ukazovat na trapframe, takze pri dalsom vyvolani ecall bude v $sscratch hodnota trapframe)

(gdb) si

(gdb) si

Prechod kernel→user

- Nasleduje 30 instrukcii obnovy hodnot registrov z trapframe

- Preskocme na instrukciu prehodenia $a0 a $sscratch (csrw a0, sscratch, a0) pomocou `si`

Prechod kernel→user

(gdb) p/x $a0

(gdb) p/x $sscratch

(gdb) si

(gdb) p/x $a0

(gdb) p/x $sscratch

- $a0 obsahuje navratovu hodnotu systemoveho volania sys_write()
- $sscratch obsahuje adresu trapframe

Prechod kernel→user

- Nachadzame sa pred instrukciou sret, ktora podla registra $sstatus nastavi mod CPU a podla $sepc obnovi hodnotu PC

Prechod kernel→user

- Nachadzame sa pred instrukciou sret, ktora podla registra $sstatus nastavi mod CPU a podla $sepc obnovi hodnotu PC

(gdb) p/x $pc

(gdb) si

(gdb) p/x $pc

- A sme spat v uzivatelskom programe, po vyvolani systemoveho volania write()

Prechod kernel→user

- Nachadzame sa pred instrukciou sret, ktora podla registra $sstatus nastavi mod CPU a podla $sepc obnovi hodnotu PC

(gdb) p/x $pc

(gdb) si

(gdb) p/x $pc

- A sme spat v uzivatelskom programe, po vyvolani systemoveho volania write()
- Navratovu hodnotu mame v registri $a0

Zhrnutie

- Systemove volanie cez entry/exit je ovela komplexnejsie ako obycajne volanie fnc…
- Takato komplikovana vec je v dosledku poziadavky izolacie procesov
- Natiska sa legitimna otazka: neda sa to nejako jednoduchsie?

Zhrnutie

- Systemove volanie cez entry/exit je ovela komplexnejsie ako obycajne volanie fnc…
- Takato komplikovana vec je v dosledku poziadavky izolacie procesov
- Natiska sa legitimna otazka: neda sa to nejako jednoduchsie?

- Odpoved… treba hladat ;)

TF

- Sluzi na uchovavanie stavu CPU pri prechode user→ kernel a naopak

TF

- Sluzi na uchovavanie stavu CPU pri prechode user→ kernel a naopak
- Vid kernel/proc.h

TF

- Sluzi na uchovavanie stavu CPU pri prechode user→ kernel a naopak
- Vid kernel/proc.h

  - Kernel page table
  - Kernel stack pointer
  - Address of usertrap() fnc in kernel
  - User pc CPU register
  - User 32 CPU registers

Niekolko RISC-V registrov

- Iba niekolko najdolezitejsich

- Dalsie vid kapitola 10 v https://github.com/Lingrui98/RISC-V-book/blob/master/rvbook.pdf

Niekolko RISC-V registrov

- stvec – supervisor trap-vector base addr reg
  - addr v jadre, kam skace ‘ecall’; addr trampoliny

Niekolko RISC-V registrov

- stvec – supervisor trap-vector base addr reg
  - addr v jadre, kam skace ‘ecall’; addr trampoliny
- sepc – supervisor exceptional instruction pc
  - Ecall vyvola vynimku; v tomto reg sa uchova adresa instrukcie, ktora vyvolala vynimku; v nasom pripade tu bude user pc (ecall je instrukcia v uzivatelskom programe)

Niekolko RISC-V registrov

- stvec – supervisor trap-vector base addr reg
  - addr v jadre, kam skace ‘ecall’; addr trampoliny
- sepc – supervisor exceptional instruction pc
  - Ecall vyvola vynimku; v tomto reg sa uchova adresa instrukcie, ktora vyvolala vynimku; v nasom pripade tu bude user pc (ecall je instrukcia v uzivatelskom programe)
- scause – supervisor cause
  - Ulozeny kod priciny vyvolanej vynimky; v pripade xv6 tam bude hodnota 8 (system call)

Niekolko RISC-V registrov

- sscratch – supervisor scratch
  - Jedno slovo (word) udajov na docasne pouzitie; v pripade xv6 tam je adresa trapframe

Niekolko RISC-V registrov

- sscratch – supervisor scratch
  - Jedno slovo (word) udajov na docasne pouzitie; v pripade xv6 tam je adresa trapframe
- satp – supervisor address translation and protection (riadi strankovanie)
  - Aktualna tabulka stranok

Domace citanie

- Chapter 4: Operating system organization knizky “xv6: a simple, Unix-like teaching operating system”

- Okrem casti 4.6; tej sa budeme zvlast venovat buduci tyzden…