

OS MMXXIII

MIT ;)

<https://pdos.csail.mit.edu/6.828>

Prechod user \leftrightarrow kernel pomocou
systémových volaní

Téma

- Ako funguje prechod do/z jadra pri systémových volaniach
- Ponoríme sa do jadra OS

write(fd, buf, n)

- Môžeme použiť v používateľskom priestore funkciu (z pohľadu jazyka C) na [za|vy] volanie (uskutočnenie) služby jadra?
- Bolo by to rýchle a efektívne
- Taktiež flexibilné na prenos komplexnejších údajových typov
- A išlo by o programátorom známy mechanizmus

write(fd, buf, n)

- Žiaľ, nemôžeme použiť takýto mechanizmus!
- Dôvodom je IZOLÁCIA procesov
- Izolácia ovplyvňuje väčšinu návrhu jadra OS

Čo to je izolácia

- **Vynútená separácia z dôvodu zapúzdenia dôsledkov zlyhaní**

Čo to je izolácia

- **Vynútená separácia z dôvodu zapúzdrenia dôsledkov zlyhaní**
- Predmetom izolácie je zvyčajne proces
- Zabraňuje
 - procesu X modifikovať/čítať údaje o procese Y (napr. modifikovať tabuľku deskriptorov, r/w prístupy do pamäte...)
 - procesu miešať sa do úloh a služieb OS

Hlavné nástroje izolácie

1) Virtuálny adresný priestor procesu

2) Privilegovaný režim činnosti CPU (používateľský proces nemôže robiť I/O operácie a meniť dôležité systémové registre CPU)

- Vynútená **kontrola toku riadenia** pomocou systémových volaní; prechod user → kernel!!!

Privilegovaný režim CPU

- RISC-V CPU **3** režimy: *supervisor, user, machine*
- Kernel (jadro OS) běží v režime *supervisor*
- Bežný používateľský program v režime *user*

Privilegovaný režim CPU

- RISC-V CPU **3** režimy: *supervisor*, *user*, *machine*
- Kernel (jadro OS) beží v režime *supervisor*
- Bežný používateľský program v režime *user*
- Čo môže *supervisor* oproti *user* naviac:
 - I/O operácie (prístup k zariadeniam)
 - Nastavovanie adresného priestoru (virtuálna pamäť pomocou zmeny registra *satp*)
 - R/W prístup k špeciálnym registrom CPU
- Takmer každý CPU využíva tento princíp

Čo sa deje pri systémovom volaní?

- Procesor je nastavený na beh *user* programu
- Čo je potrebné na procesore RISC-V urobiť?

Čo sa deje pri systémovom volaní?

- Procesor je nastavený na beh *user* programu
- Čo je potrebné na procesore RISC-V urobiť:
 - Zmeniť mód CPU z *user* na *supervisor* (pre jadro)
 - Uložiť 32 *user* registrov a register PC
 - Zmeniť tabuľku stránok z *user* na *kernel*
 - Zmeniť zásobník z *user* na *kernel*
 - Skočiť na vykonávanie kódu jadra v C

Ciele riadeného prechodu

1. nedovoliť zasahovať *user* programu do prechodu *user* → *kernel*
2. prechod musí byť pre *user* program transparentný (bez akéhokoľvek zásahu používateľa)

Prečo prepínať cpu mód?

- Prečo umožniť prechod do *supervisor* módu?

Prečo prepínať cpu mód?

- Prečo umožniť prechod do *supervisor* módu?
 - 1) Prístup k špeciálnym registrom CPU

Prečo prepínať cpu mód?

- Prečo umožniť prechod do *supervisor* módu?

1) Prístup k špeciálnym registrom CPU

- `satp` – tabuľka stránok
- `stvec` – inštrukcia `ecall` skočí na `addr` uloženú v tomto registri (v `xv6` ukazuje na kód trampolíny)
- `sepc` – inštrukcia `ecall` sem uloží user PC
- `scratch` – pomocný register; `xv6` ho využíva na uloženie adresy pamäťovej oblasti trapframe

Prečo prepínať cpu mód?

- Prečo umožniť prechod do *supervisor* módu?

1) Prístup k špeciálnym registrom CPU

- `satp` – tabuľka stránok
- `stvec` – inštrukcia `ecall` skočí na `addr` uloženú v tomto registri (v `xv6` ukazuje na kód trampolíny)
- `sepc` – inštrukcia `ecall` sem uloží user PC
- `scratch` – pomocný register; `xv6` ho využíva na uloženie adresy pamäťovej oblasti trapframe

2) *Supervisor* má prístup ku stránkam, ktoré nemajú nastavený príznak `PTE_U`

Prečo prepínať cpu mód?

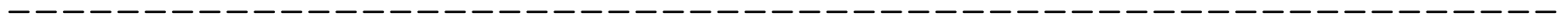
- Prečo umožniť prechod do *supervisor* módu?
- 3) *Supervisor* má priamy prístup k hw

Prečo prepínať cpu mód?

- Prečo umožniť prechod do *supervisor* módu?
- 3) *Supervisor* má priamy prístup k hw
- Nič iné nemá navyše...
 - To znamená, že ani v *supervisor* móde nie je možné pristupovať k adresám, pre ktoré nie je platné mapovanie v tabuľke stránok

Kód xv6 prechodu user→kernel

User space



Kernel space

Kód xv6 prechodu user→kernel

User space write()



Kernel space

Kód xv6 prechodu user→kernel

User space `write()`



Kernel space `uservec()`
 `usertrap()`
 `syscall()`
 `sys_write()`

Kód xv6 prechodu user→kernel

User space

`write()`

Kernel space

`uservec()`

`userret()`

`usertrap()`

`usertrapret()`

`syscall()`

^

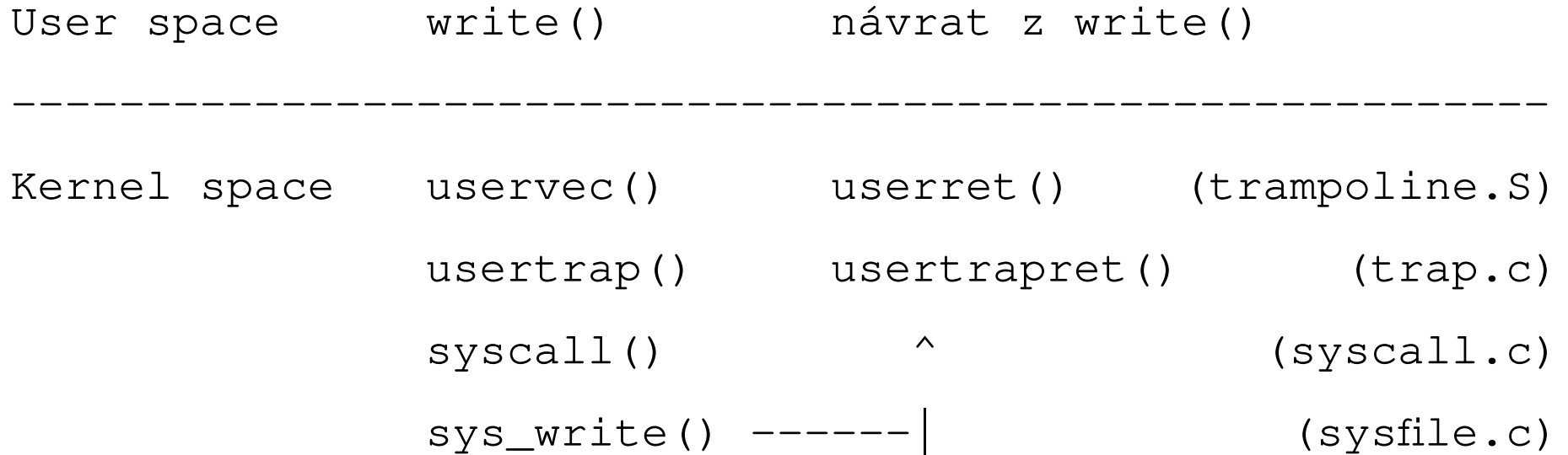
`sys_write()` -----|

Kód xv6 prechodu user→kernel

User space write() návrat z write()

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

Kód xv6 prechodu user→kernel



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* vstúpi do jadra

- napríklad *xv6 shell* vypisujúci *prompt* ('\$')

Ako *syscall* vstúpi do jadra

- napríklad *xv6 shell* vypisujúci *prompt* ('\$')
 - `user/sh.c:137 write(2, "$ ", 2)`
 - `user/usys.S:29`
 - `li a7, SYS_write (→kernel/syscall.h:17)`
 - `ecall` (vyvolá **RIADENÝ** prechod user→kernel)
 - `user/sh.asm:22`, adresa `write` je `0xe16`
 - `user/sh.asm:1939`, adresa `ecall` je `0xe18`

Ako *syscall* vstúpi do jadra

```
$ make CPUS=1 qemu-gdb ...
```

```
(gdb) c
```

```
Ctrl+C
```

```
(gdb) b *0xe16
```

```
(gdb) c
```

V konzole `qemu-gdb` zadáme Enter a vrátíme sa do okna s `gdb`

Nachádzame sa pred systémovým volaním `write()`

Ako *syscall* vstúpi do jadra

```
(gdb) add-symbol-file user/sh.o
```

```
(gdb) ctrl-x 2
```

```
(gdb) ctrl-x 2
```

V hornom okne máme zobrazené registre CPU

V strednom vidíme asm kód

V dolnom zadávame príkazy gdb

Ako *syscall* vstúpi do jadra

(gdb) b *0xe1c (adresa inštrukcie ret v sh.asm)

→ pozri okno behu xv6!

(gdb) c

→ pozri okno behu xv6!

Na konzolu sa vypíše *prompt*.

Ako *syscall* vstúpi do jadra

(gdb) si (posun o 1 inštrukciu ďalej)

→ user/sh.asm:30 adresa 0x24 (`memset()`)

(gdb) c

→ nič sa nedeje... vstup z klávesnice!

→ sme na adrese 0xe16 (`syscall write()`)

→ vid' okno behu xv6

Ako *syscall* vstúpi do jadra

(gdb) c

→ sme späť z kernelu na konci write()

→ vid' okno behu xv6

→ podme znovu pred syscall write

(gdb) c (v okne Qemu nezabudnime dať Enter)

Ako *syscall* vstúpi do jadra

- Registre \$pc a \$sp sú na nízkych adresách
- Argumenty funkcií sú v registroch a0, a1, a2...
- Argumenty systémového volania write()
 - a0 je fd
 - a1 je buf
 - a2 je n

(gdb) x/2c \$a1 (vypíš 2 znaky od addr v reg a1)

sh vypisuje prompt '\$ '

Ako *syscall* vstúpi do jadra

- Vykonajme inštrukciu `eca ll`

(gdb) si 2

- Kde sa nachádzame?

(gdb) print \$pc → veľmi vysoká adresa!

(gdb) x/6i \$pc

- 6 inštrukcií, ktoré sa budú vykonávať
- Vid' `uservec` v `kernel/trampoline.S`
- Ide o začiatok kódu trampolíny do jadra

Ako *syscall* vstúpi do jadra

- Všetky registre majú hodnoty z *user* programu (okrem \$pc)

(gdb) info reg

- Ešte stále sa používa tabuľka stránok *user* procesu

(gdb) p/x \$satp

Ako *syscall* vstúpi do jadra

- Čo sa udialo inštrukciou `eca ll`?
- Procesor sa prepol do *supervisor* módu
- Ako to vieme? Vykonáva sa inštrukcia na adrese danej registrom `$stvec` (`eca ll` zoberie hodnotu v `$stvec` a nastaví podľa nej `$pc`)
- Trampolína **nie je** prístupná pre *user* program (nie je nastavený príznak `PTE_U`)!

(gdb) p/x \$stvec

Ako *syscall* vstúpi do jadra

- Čo sa udeje inštrukciou `ecall`?
 1. Zmení sa mód procesora z *user* na *supervisor*
 2. Uloží sa hodnota `$pc` do `$sepc`
(gdb) p/x \$sepc
 3. Vykonávanie skočí na adresu uloženú v `$stvec`
(register `$pc` sa nastaví na hodnotu, ktorá je v `$stvec`)

Ako *syscall* vstúpi do jadra

- Kernel musel nastaviť \$stvec pred prechodom do *user* priestoru!!!
- `syscall` umožňuje zmeniť mód CPU z *user* na *superuser*
- Ale iba *kernel* má kontrolu nad tým, ČO sa začne v tomto režime vykonávať, a to na základe obsahu registra \$stvec
- Program používateľa NEDOKÁŽE zmeniť \$stvec

Ako *syscall* vstúpi do jadra

- Čo ďalšie sa musí urobiť? Zatiaľ máme „iba“ zmenený mód CPU

Ako *syscall* vstúpi do jadra

- Čo ďalšie sa musí urobiť? Zatiaľ máme „iba” zmenený mód CPU
- Uložiť registre CPU, ktoré využíva *user* (aby sa mohli neskôr obnoviť pri prechode z jadra späť k vykonávaniu kódu používateľského programu)
- Zmeniť tabuľku stránok z *user* na *kernel*
- Nastaviť zásobník pre C-čkovský kód jadra
- Skočiť na vykonávanie C kódu jadra

Ako *syscall* vstúpi do jadra

- Prečo je `eca` tak navrhnutý, že nerobí tieto ostatné činnosti?

Ako *syscall* vstúpi do jadra

- Prečo je `syscall` tak navrhnutý, že nerobí tieto ostatné činnosti? Aby bolo možné implementovať veľmi rýchlu obsluhu výnimiek.

Ako *syscall* vstúpi do jadra

- Prečo je `eca` tak navrhnutý, že nerobí tieto ostatné činnosti? Aby bolo možné implementovať veľmi rýchlu obsluhu výnimiek:
 - Obslúžiť niektoré výnimky bez nutnosti zmeny tabuľky stránok
 - Ak by *kernel* a *user* používali to isté mapovanie, netreba meniť tabuľku stránok
 - Ak je obsluha písaná v `asm`, nie je nutné používať zásobník

Ako *syscall* vstúpi do jadra

- Aké možnosti máme na RISC-V pre uloženie stavu CPU (registrov používateľa)?

Ako *syscall* vstúpi do jadra

- Aké možnosti máme na RISC-V pre uloženie stavu CPU (registrov používateľa)?
- Môžeme ich len tak zapísať niekam do fyzickej pamäte?

Ako *syscall* vstúpi do jadra

- Aké možnosti máme na RISC-V pre uloženie stavu CPU (registrov používateľa)?
- Môžeme ich len tak zapísať niekam do fyzickej pamäte? Nie, lebo je zapnuté stránkovanie!

Ako *syscall* vstúpi do jadra

- Aké možnosti máme na RISC-V pre uloženie stavu CPU (registrov používateľa)?
- Môžeme ich len tak zapísať niekam do fyzickej pamäte? Nie, lebo je zapnuté stránkovanie!
- Môžeme najprv zmeniť `$satp` na *kernel* stránky?

Ako *syscall* vstúpi do jadra

- Aké možnosti máme na RISC-V pre uloženie stavu CPU (registrov používateľa)?
- Môžeme ich len tak zapísať niekam do fyzickej pamäte? Nie, lebo je zapnuté stránkovanie!
- Môžeme najprv zmeniť `$satp` na *kernel* stránky?
 - *Supervisor* mód to umožňuje
 - Ale nepoznáme ADRESU tabuľky jadra úrovne L2!
 - A `$satp` pred prepísaním musí byť tiež odložený!

Ako *syscall* vstúpi do jadra

- Uloženie registrov pozostáva z 2 častí
 1. Z dostupného pamäťového miesta, kam sa uložia
 2. Z adresy tohto pamäťového miesta

Ako *syscall* vstúpi do jadra

- Uloženie registrov pozostáva z 2 častí
 1. Z dostupného pamäťového miesta, kam sa uložia
 2. Z adresy tohto pamäťového miesta
- Xv6 mapuje do *user* stránok 2 stránky jadra, ku ktorým *user* nemá prístup: trampolínu (úplne posledná stránka virtuálneho priestoru) a podňou tzv. *trapframe*
- RISC-V poskytuje 1 pracovný register, ku ktorému nemá *user* prístup: `$scratch`

Ako *syscall* vstúpi do jadra

- *Trapframe*
 - 1 stránka je dostatočne veľká na uloženie 32 registrov RISC-V CPU
 - Každý proces má mapovaný *trapframe* na tú istú virtuálnu adresu, ale vždy do iného rámca v RAM
 - Virtuálna adresa je 0x3fffffe000
 - Vid' štruktúru *trapframe* v kernel/proc.h

Ako *syscall* vstúpi do jadra

- *Trapframe*
 - 1 stránka je dostatočne veľká na uloženie 32 registrov RISC-V CPU
 - Každý proces má mapovaný *trapframe* na tú istú virtuálnu adresu, ale vždy do iného rámca v RAM
 - Virtuálna adresa je 0x3fffffe000
 - Vid' štruktúru *trapframe* v kernel/proc.h
- Pamäťové miesto máme, ale potrebujeme mať niekde uloženú adresu tohto miesta!

Ako *syscall* vstúpi do jadra

- Register `sscratch`
 - Pred prechodom do *user* priestoru *kernel* nastaví nielen `$stvec`, ale aj `$sscratch` register
 - Uloží do neho odkaz na *trapframe* štruktúru
 - RISC-V poskytuje inštrukciu na výmenu ľubovoľného registra s `$sscratch` (prehodenie hodnôt medzi registrom a `$sscratch`)
- Prvá inštrukcia `uservec()` v `trapframe.S`:
`csrrw sscratch, a0`

Ako *syscall* vstúpi do jadra

```
(gdb) p/x $sscratch
```

```
(gdb) p/x $a0
```

```
(gdb) si
```

```
(gdb) p/x $sscratch
```

```
(gdb) p/x $a0
```

Ďalej kód ukladá 32 registrov CPU do štruktúry *trapframe* pomocou nepriamej adresácie cez register \$a0

Ako *syscall* vstúpi do jadra

až po inštrukciu `csrr t0, sscratch (30x si)`

(gdb) si

- ešte je potrebné uložiť pôvodnú hodnotu `$a0` (tá sa nachádza v `$sscratch`)

(gdb) si

(gdb) si

- po uložení registrov je potrebné pripraviť zásobník a skočiť na vykonávanie C kódu

Ako *syscall* vstúpi do jadra

- Pozrime sa na štruktúru *trapframe* do `kernel/proc.h`
- Kde sa v nej nachádza adresa zásobníka jadra?

Ako *syscall* vstúpi do jadra

- Pozrime sa na štruktúru *trapframe* do `kernel/proc.h`
- Kde sa v nej nachádza adresa zásobníka jadra? V štruktúre *trapframe* od 8. bajtu
 - `ld sp, 8(a0)`
 - Nezabúdajme, že v `$a0` je adresa začiatku *trapframe*; takže po pripočítaní hodnoty 8 k hodnote v registri `$a0` dostaneme adresu, z ktorej sa načíta hodnota do registra `$sp`
 - Všetky údaje potrebné pre obnovenie behu kódu jadra sú k dispozícii v štruktúre *trapframe*

Ako *syscall* vstúpi do jadra

- Vid' komentáre v kernel/trampoline.S pre ďalšie inštrukcie pri krokovaní pomocou „si”
 - Obnoví sa hw ID vlákna
 - Načíta sa adresa funkcie v C, ktorou bude pokračovať spracovanie po ukončení asm kódu
 - Načíta sa adresa tabuliek jadra do \$satp, vyčistí sa TLB – prečo neprišlo ku výnimke po výmene tabuliek stránok?

Ako *syscall* vstúpi do jadra

- Vid' komentáre v kernel/trampoline.S pre ďalšie inštrukcie pri krokovaní pomocou „si”
 - Obnoví sa hw ID vlákna
 - Načíta sa adresa funkcie v C, ktorou bude pokračovať spracovanie po ukončení asm kódu
 - Načíta sa adresa tabuliek jadra do \$satp, vyčistí sa TLB – prečo neprišlo ku výnimke po výmene tabuliek stránok?
 - Lebo na rovnakú virtuálnu adresu je mapovaná trampolína aj v tabuľkách jadra!!!
 - (gdb) p/x \$satp

Ako *syscall* vstúpi do jadra

- Napokon možno skočiť do funkcie `usertrap()`
 - Jej adresa sa nachádza v registri `$t0`
 - (gdb) p/x \$t0
 - (gdb) x/4i \$t0
 - (gdb) si
 - (gdb) tui enable
- A konečne sa nachádzame v C kóde
- `usertrap()` v `kernel/trap.c`

Ako *syscall* vstúpi do jadra

- Úloha `usertrap()`
 - Dispečing rôznych typov prerušení, chybových stavov a systémových volaní prichádzajúcich z *user* módu CPU
 - Zistenie príčiny výnimočného stavu pomocou hodnoty registra `$scause`
 - Vid' obrázok 10.3 na strane 102 manuálu „The RISC-V Reader”
 - `$scause` rovné hodnote 8 je systémové volanie (*environment call from U-mode*, čiže `ecall`)

0	Instruction address misaligned
1	Instruction access fault
2	Illegal instruction
3	Breakpoint
4	Load address misaligned
5	Load access fault
6	Store address misaligned
7	Store access fault
8	Environment call from U-mode
9	Environment call from S-mode
11	Environment call from M-mode
12	Instruction page fault
13	Load page fault
15	Store page fault

Prevzaté a upravené z: Patterson, D.; Waterman, A. The RISC-V Reader: An Open Architecture Atlas. Strawberry Canyon LLC: San Francisco, CA, USA, 2017. Obrázok 10.3 na strane 102.

Ako *syscall* vstúpi do jadra

- Až po funkciu `syscall()`: (gdb) n
(gdb) s
(gdb) n
- Nachádzame sa vo funkcii `syscall()`
v súbore `kernel/syscall.c`

Ako *syscall* vstúpi do jadra

- Funkcia `syscall()`
 - `p` → `trapframe` ukazuje na pamäťovú oblasť, v ktorej sú uchované VŠETKY registre user programu
- `p` → `trapframe` → `a7` uchováva číslo systémového volania (v našom prípade hodnotu 16, čiže `SYS_write`)
- `p` → `trapframe` → `a0` uchováva 1. argument (`fd`)
- `p` → `trapframe` → `a1` uchováva 2. argument (`buf`)
- `p` → `trapframe` → `a2` uchováva 3. argument (`n`)

Ako *syscall* vstúpi do jadra

(gdb) n

(gdb) p num

(gdb) n

(gdb) s

- A sme vo funkcii `sys_write()`

Ako *syscall* vstúpi do jadra

(gdb) n

(gdb) p num

(gdb) n

(gdb) s

- A sme vo funkcii `sys_write()`
- Ďalší kód je nezaujímavý – štandardné C
- Pozrime sa, ako sa vrátíme späť do *user* módu

Ako *syscall* vystúpi z jadra

(gdb) finish

- Vrátime sa späť do funkcie `syscall()`
- Funkcia `sys_write()` vykonala výpis na monitor
- Dôležité: PO vykonaní systémového volania sa výsledok uloží do `p→trapframe→a0`, aby sa pri obnovení obsahu registrov dostala návratová hodnota volania do registra `$a0` !!!!!!!!!!!
- Prečo `$a0`? Konvencia volaní C na RISC-V

Ako *syscall* vystúpi z jadra

- Z pohľadu používateľského programu je vyvolanie systémového volania volaním „obyčajnej funkcie“
- Konvencia volaní C definuje, že návratová hodnota volanej funkcie sa u volajúceho musí objaviť v registri \$a0
- Preto sa návratová hodnota systémového volania v jadre „injektážou“ cez `trapframe` dostane až k používateľskému kódu tiež v registri \$a0

Prechod kernel→user

(gdb) fin — vrátíme sa do `usertrap()`

(gdb) n

(gdb) s

Nachádzame sa vo funkcii `usertrapret()`, ktorá rieši návrat späť do používateľského programu

V prvom rade treba prichystať všetky údaje potrebné pre ďalší prechod z `user`→`kernel`

Prechod kernel→user

- Príprava pre ďalší prechod user→kernel
- $\$stvec \leftarrow \text{uservec}()$ kvôli ďalšiemu `call`
- $\text{tf->satp} \leftarrow \text{kernel page table}$ kvôli ďalšiemu volaniu `uservec()`
- $\text{tf->sp} \leftarrow \text{vrchol zásobníka kernelu}$
- $\text{tf->trap} \leftarrow \text{adresa funkcie usertrap}()$
- $\text{tf->hartid} \leftarrow \text{id vlákna (nachádza sa v \$tp)}$

Prechod kernel→user

- Prechod do *user* módu spôsobuje inštrukcia **sret** (na architektúre RISC-V)

Prechod kernel→user

- Prechod do *user* módu spôsobuje inštrukcia **sret** (na architektúre RISC-V)
- Táto inštrukcia využíva na svoju činnosť viaceré registre (podobne ako `eca ll`)
 - `$sstatus` (pomocou neho sa nastaví bit „predošlého módu CPU“ na *user*)
 - `$sepc` (adresa inštrukcie *user* programu, ktorou sa bude pokračovať vykonávanie kódu – tú máme uloženú v `p→trapframe→epc`)
- Pomocou ‘`n`’ prejdeme na posledný riadok (č. 129) funkcie

Prechod kernel→user

- Teraz by bolo vhodné zmeniť tabuľku stránok späť na používateľskú
- To sa však NEDÁ v `usertrapret()`, pretože nie je mapovaná do používateľského priestoru!

Prechod kernel→user

- Teraz by bolo vhodné zmeniť tabuľku stránok späť na používateľskú
- To sa však NEDÁ v `usertrapret()`, pretože nie je mapovaná do používateľského priestoru!
- Preto sa v `usertrapret()` na konci vypočíta adresa funkcie `userret()` v stránke trampolíny, a tam sa odovzdá riadenie

(gdb) `tui disable`

Pomocou `'si'` sa v gdb presuňme na `0x3fffffff09c`

(gdb) `x/8i 0x3fffffff09c`

Prechod kernel→user

- \$a0 obsahuje adresu stránok používateľského priestoru
- Inštrukciou `csrw satp` sa nastaví tabuľka stránok používateľského procesu
- Prečo pri vykonávaní ďalšieho kódu nenastane výnimka?

(gdb) si

(gdb) si

Prechod kernel→user

- Do registra \$a0 sa nastaví adresa *trapframe* a do *trapframe* sa uložia všetky registre okrem \$a0
(gdb) si
(gdb) si
- Nasleduje 30 inštrukcií obnovy hodnôt registrov z *trapframe*

Prechod kernel→user

- Nasleduje 30 inštrukcií obnovy hodnôt registrov z *trapframe*
- Nakoniec sa obnoví hodnota registra \$a0 a pomocou inštrukcie `sret a0` sa obnoví beh používateľského procesu v režime *user*:
 - \$sstatus má nastavený *user* mód
 - \$sepc obsahuje adresu inštrukcie, ktorá sa má v používateľskom procese vykonávať

Prechod kernel→user

- Nakoniec sa nachádzame pred inštrukciou `sret`, ktorá podľa registra `$sstatus` nastaví mód CPU a podľa `$sepc` obnoví hodnotu PC

```
(gdb) p/x $pc
```

```
(gdb) si
```

```
(gdb) p/x $pc
```


Prechod kernel→user

- Nakoniec sa nachádzame pred inštrukciou `sret`, ktorá podľa registra `$sstatus` nastaví mód CPU a podľa `$sepc` obnoví hodnotu PC

```
(gdb) p/x $pc
```

```
(gdb) si
```

```
(gdb) p/x $pc
```

- A sme späť v používateľskom programe, po vyvolaní systémového volania `write()`
- Návratovú hodnotu volania máme v registri `$a0`

Zhrnutie

- Systémové volanie cez entry/exit je oveľa komplexnejšie ako obyčajné volanie fnc...
- Takáto komplikovaná vec je v dôsledku požiadavky izolácie procesov
- Natíska sa legitímna otázka: nedá sa to nejako jednoduchšie?

Zhrnutie

- Systémové volanie cez entry/exit je oveľa komplexnejšie ako obyčajné volanie fnc...
- Takáto komplikovaná vec je v dôsledku požiadavky izolácie procesov
- Natíska sa legitímna otázka: nedá sa to nejako jednoduchšie?
- Odpoveď... treba hľadať ;)

TF

- Slúži 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

Niekoľko RISC-V registrov

- Iba niekoľko najdôležitejších
- Ďalšie vid' kapitola 10 v <https://github.com/Lingrui98/RISC-V-book/blob/master/rvbook.pdf>

Niekoľko RISC-V registrov

- *stvec* – *supervisor trap-vector base addr reg*
 - *addr* v jadre, kam skáče *eca* \ll ; *addr* trampolíny
- *sepc* – *supervisor exceptional instruction pc*
 - *eca* \ll vyvolá výnimku; v tomto reg. sa uchová adresa inštrukcie, ktorá vyvolala výnimku; v našom prípade tu bude hodnota *user pc* (*eca* \ll je inštrukcia v používateľskom programe)
- *scause* – *supervisor cause*
 - Uložený kód príčiny vyvolanej výnimky; v prípade xv6 tam bude hodnota 8 (*system call*)

Niekoľko RISC-V registrov

- *sscratch* – *supervisor scratch*
 - Jedno slovo (*word*) údajov na dočasné použitie; v prípade xv6 tam je adresa *trapframe*
- *satp* – *supervisor address translation and protection* (riadi stránkovanie)
 - Aktuálna tabuľka stránok

Domáce čítanie

- Chapter 4: Operating system organization
knižky „xv6: a simple, Unix-like teaching operating system”
- Okrem časti 4.6; tej sa budeme zvlášť venovať
budúci týždeň