B35APO: Architektury počítačů

Lekce 10. Překlad jazyka C

Pavel Píša
pisa@fel.cvut.cz

Petr Štěpán
stepan@fel.cvut.cz

4. května, 2023

Obsah

Cíl dnešní přednášky

- Zjistit, jak se program v jazyce C přeloží na strojové instrukce například procesoru RISC-V
- Zaměříme se jak se překládá volání funkce
- Podíváme se kam se ukládají lokální a globální proměnné
- Ukážeme si, jak se liší volání funkcí operačního systému od volání funkcí

Jak se překládá program v C

Jednoduchý příklad – překlad přiřazení `a = b + c;`

1. Přiřadit proměnným registry, např. b - t0, c - t1, a - t0
2. Nahrát do registrů hodnoty proměnných:
   ```
   lw t0, &b(gp);
   lw t1, &c(gp)
   ```
3. Provést výpočet:
   ```
   add t0, t0, t1
   ```
4. Uložit hodnotu do proměnné a:
   ```
   sw t0, &a(gp)
   ```

- Obecné výrazy se analyzují pomocí bezkontextových gramatik - definuje všechny výrazy možné v jazyce C.
- Výše uvedený překlad je bez optimalizace, uloží hodnotu do proměnné a, v dalším kroku ji může opět nahrávat z paměti třeba i do stejného registru.
- Pokud by adresy proměnných nebyly dosažitelné registrem gp, je nutné před instrukcí `lw` nahrát adresu proměnné.

Překlad konstrukce while

Složitější příklad – překlad konstrukce `while (cond) body;`

- analýza bezkontextové gramatiky jazyka detekuje while konstrukci
- rekurzivně se zajistí překlad podmínky `cond` na množinu instrukcí `COND` a překlad těla cyklu `body` na množinu instrukcí `BODY`.
- vygeneruje se instrukce `j cond_1`, skok na návěstí `cond_1`
- vloží se návěstí `body_1`
- vloží se všechny instrukce těla cyklu `BODY`
- vloží se návěstí `cond_1`
- vloží se instrukce pro generování podmínky `COND`, výsledek bude např. v registru t0
- vygeneruje se instrukce `bne t0, zero, body_1`

```
    j   cond_1
body_1:
    BODY
cond_1:
    COND
    bne t0, zero, body_1
```

Proč se zabývat překladem?

- Buď používáte interpretované jazyky, které jsou ze své podstaty pomalejší
  - rychlejší jsou pouze, pokud využívají knihovny přeložené do strojových instrukcí a většinou velmi optimalizované např. OpenCV, NumPy, TensorFlow, PyTorch apod.
- nebo používáte překládané jazyky, jako je C/C++, Rust, Fortran, Pascal a pak neběží program v C, ale program přeložený do strojových instrukcí.
- Pokud nevíte, jak se program přeloží:
  - plně spoléháte na překladač a jeho optimalizace
  - nadefinujete si ve funkci pole o velikostech 100MiB
  - bude používat rekurzi, i když neznáte hloubku této rekurze
- V domácím úkolu 4 si procvičíte analýzu strojového kódu a zkusíte si napsat C program, který se bude chovat obdobně (ideálním cílem by bylo, že se přeloží do zadaného kódu strojových instrukcí).

Jak se přeložit funkci?

Pro překlad funkce např. `int secti(int a, int b);` je nutné vymyslet:

- Jak předat parametry a,b?
- Jak předat výsledek secti?
- Jakou instrukci dát na konec funkce, kam skočit?

Volající (caller) a volaný (callee) se musí dohodnout na těchto otázkách, aby si rozuměli.

- Překlad volaného může být na jiném počítači, jiným překladačem (typicky knihovny) než překlad volajícího - Vaším překladačem na Vašem počítači.
- Je nutné definovat konvenci, typ konvence volání funkce musí být v hlavičce objektových souborů (a tím i knohoven) aby zajistila správnost předání dat a získání návratové hodnoty funkce.

Obsah

Konvence volání funkce pro RISC-V

- Parametry funkce uloží volající do registrů a0, ... , a7
  - Co když bude parametrů více? Probereme v této přednášce později.
- Výsledek funkce bude v registrech a0 a a1.
  - Co když se výsledek nevejde do těchto dvou registrů?
  - Volající musí připravit místo, kam se výsledek uloží.
  - Ukazatel na adresu výsledku předá jako skrytý parametr do funkce.
  - Volaný uloží výsledek funkce rovnou na připravenou adresu.

```
struct a {
  int a, b, c, d;
};

struct a permut(int x, int y);

struct a t;

t = permut(2, 3);
```

```
struct a {
  int a, b, c, d;
};

void permut(struct a *r, int x, int y);

struct a t;

permut(&t, 2, 3);
```

Jak přeložit návrat z funkce?

```
[0x100]   j   soucet

          ...

[0x254]   j   soucet


soucet:

     ...

     j ?
0x104 nebo 0x258?
nebo jinam
```

- funkce soucet může být volána z mnoha různých míst programu
- nelze vygenerovat adresu skoku návrat z funkce v době překladu
- návratová adresa musí být nastavena volajícím
- konvence volání funkce – návratová adresa je uložena v registru ra (return address) – registr x1
- potřebujeme instrukci, která skočí na adresu uloženou v registru
- hodila by se i instrukce, která uloží do registru ra adresu následující instrukce

Instrukce jal, jalr (ret, jr)

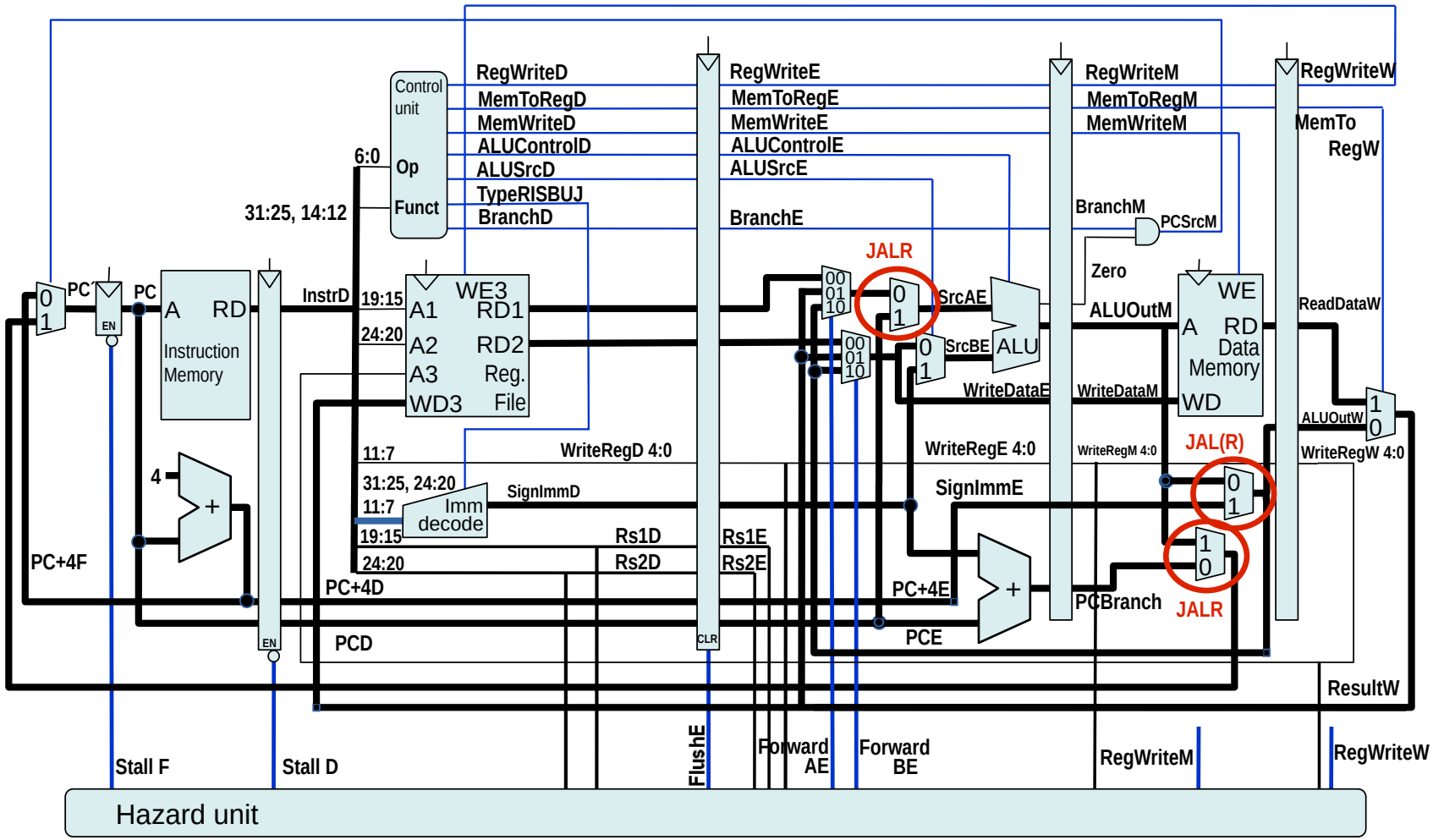
Instrukce jal "rd,"address

- skoč na adresu address (21 bitů, nejnižší bit defaultně 0) a do registru rd ulož hodnotu PC+4
- pokud neuvedete registr rd, tak se standardně doplní x1
- pokud chceme jen skok (instrukce j), tak se PC+4 uloží do registru x0

Instrukce jalr rd, rs1, imm12

- skoč na adresu rs1 + imm12 a do registru rd ulož hodnotu PC+4
- pokud neuvedete registr rd, tak se standardně doplní x1
- pokud chceme jen návrat z funkce, lze použít pro rd registr x0

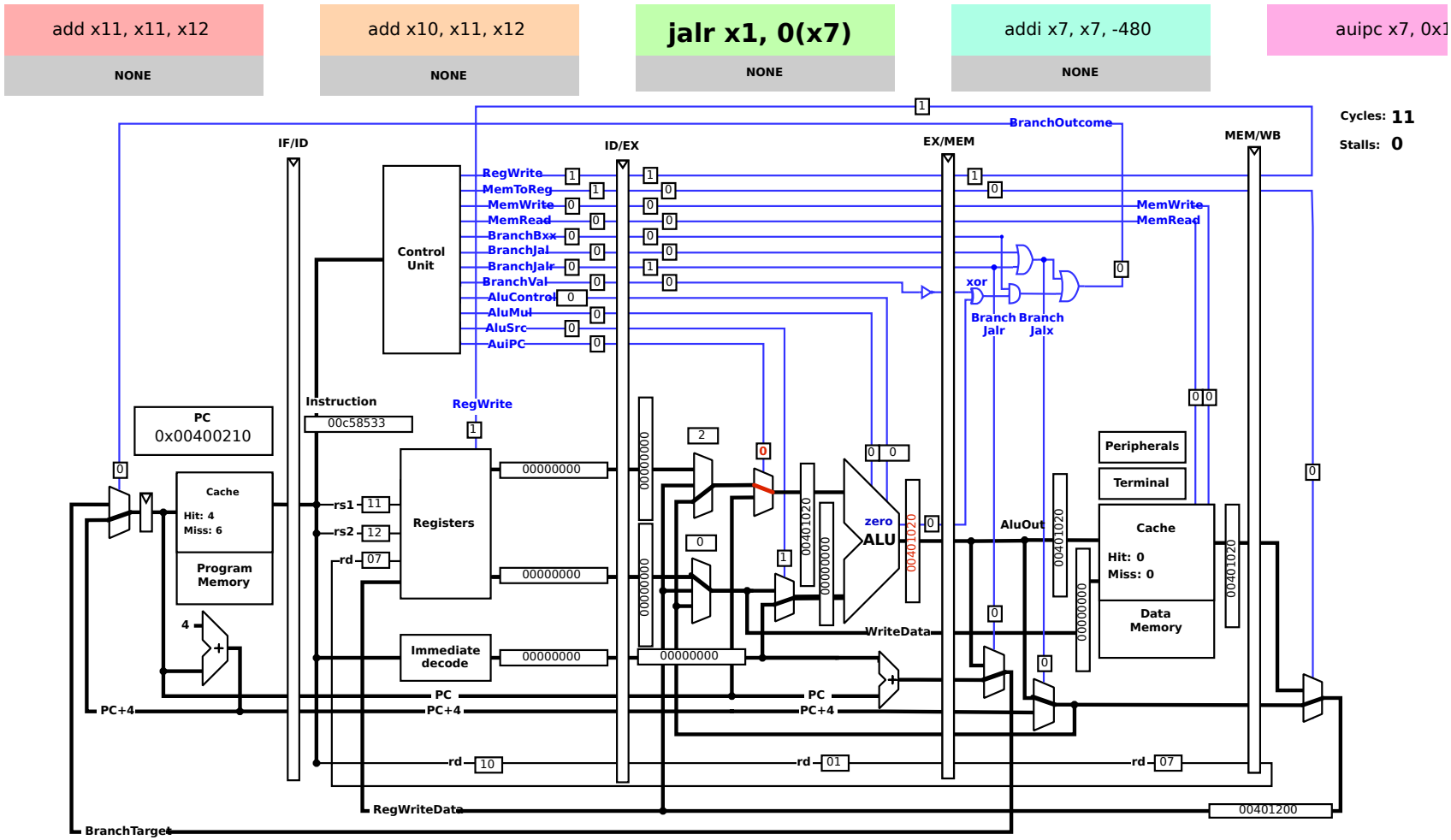
Implemetace jal a jalr



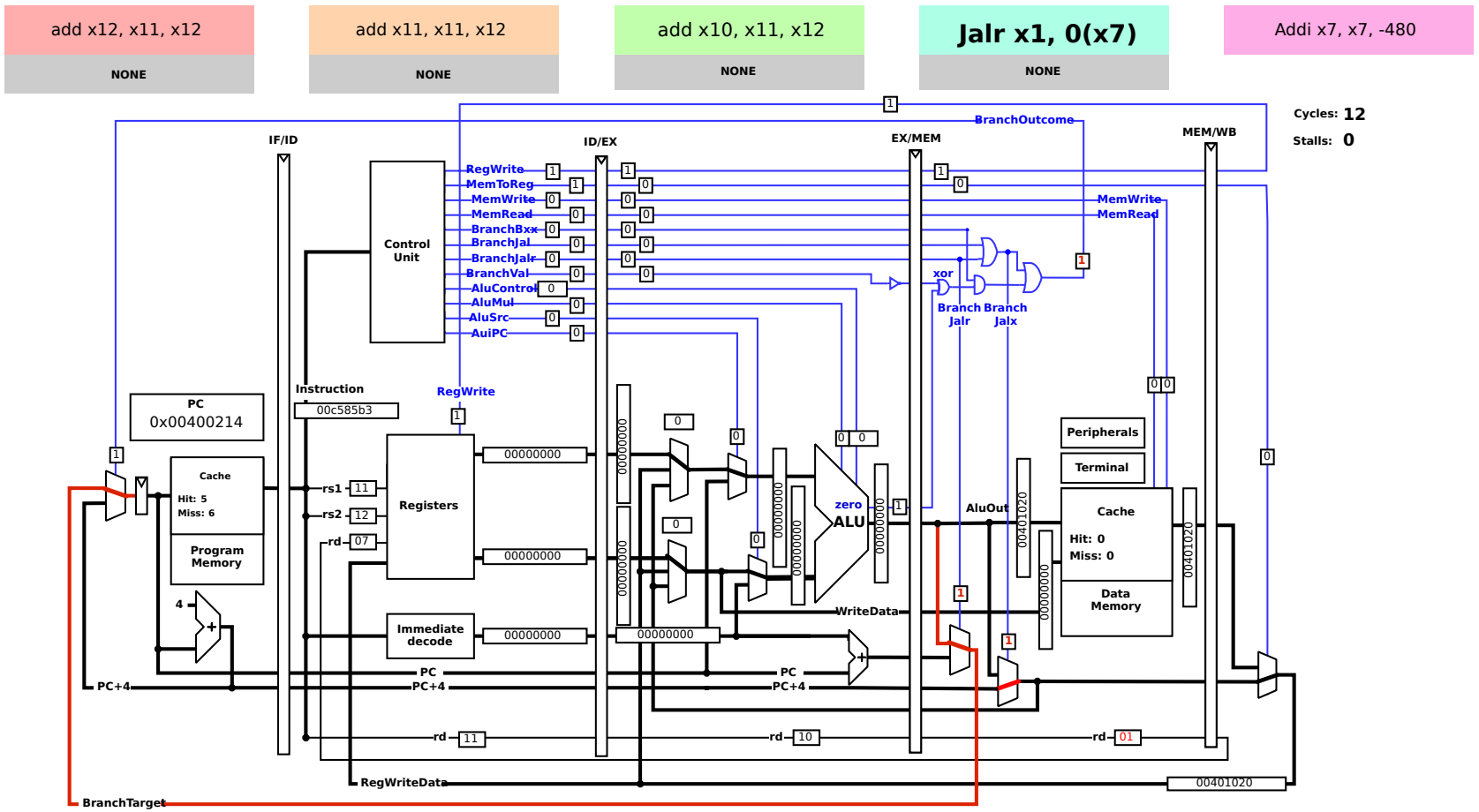
Kvíz: Kolik různých čar byste uměli vysvětlit

A – žádnou B – asi třetinu C – asi polovinu D – téměř všechny.

Simulace Jalr



Řídicí hazard při Implemetaci jalr



Co když funkce ve svém těle volá jinou funkci?

Volající připraví argumenty do registrů a0 až a7 a do registru ra návratovou adresu.

Jak ale vyřešit, když funkce ve svém těle volá jinou funkci?

Je nutné někam uložit registr ra, registry a0-a7, ale kam?

- Pro uložení dočasných proměnných slouží: Activation record nebo také activation frame.
- Tento záznam, nebo rámec se uloží na zásobník (anglicky call stack nebo stack frame).

Zásobník

Zásobník je datová struktura LIFO (Last In First Out) – neboli poslední vložená data jdou první ven

- push – vlož data do zásobníku
- pop – vyber data ze zásobníku

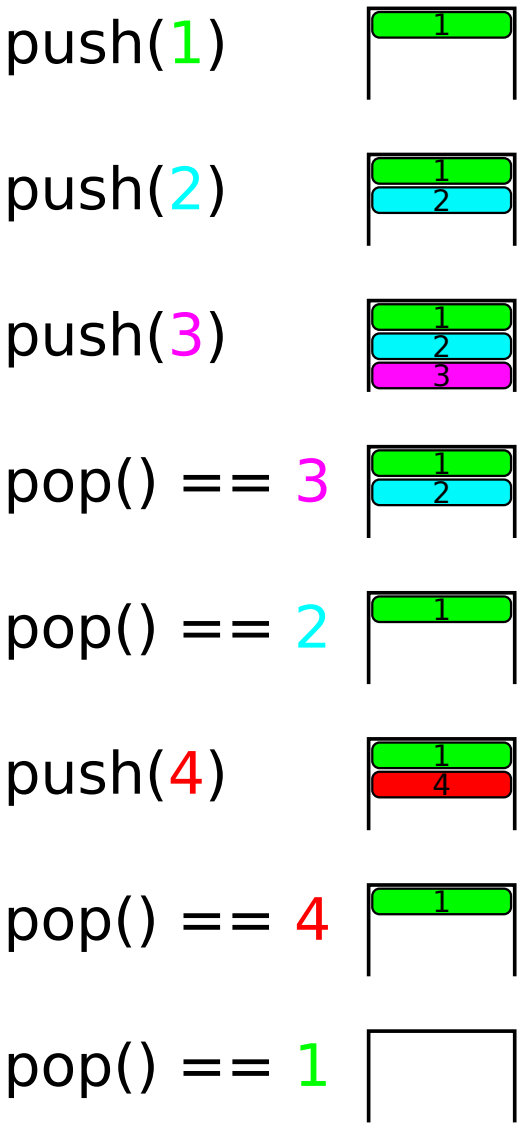
Implemetace zásobníku registrem sp (x2):

- push

```
addi sp, sp, -4     – alokace místa
sw   x10, 0(sp)     – uložení hodnoty
```

- pop

```
lw   x10, 0(sp)     – načtení hodnoty
addi sp, sp, 4      – dealokace místa
```



Co se ukládá na zásobník?

Activation frame obsahuje:

- Parametry funkce a návratová adresa, pokud se v těle funkce volá jiná funkce.
- Lokální proměnné funkce, které existují jen v průběhu vykonávání funkce.
- Podle konvence překladu funkce se nesmí po návratu z funkce změnit hodnota registrů s0-s11
  - Registr s0 se někdy používá jako ukazatel na Activation frame a nazývá se fp – frame pointer
  - Registr fp se využívá jako pevný ukazatel na předem dané místo rámce a tím ho lze použít k odkazu na lokální proměnné funkce a parametry funkcí.

Konvence volání funkce

Každý register buď nastavuje a je za něj zodpovědný volající (caller), nebo volaný (callee).

Symbolické jméno	Registr	Popis	Ukládá
a0 - a7	x10 - x17	vstupní parametry funkce	volající
a0, a1	x10, x11	výstupní hodnota funkce	volaný
ra	x1	návratová adresa	volající
t0 - t6	x5-7, x28-x31	dočasné registry	volající
s0 - s11	x8-9, x18-x27	ukládané registry	volaný
sp	x2	ukazatel zásobníku	volaný
gp	x3	globální ukazatel	– –
tp	x4	ukazatel na vlákno (thread)	– –

Konvence volání funkce 32 bitového systému

- char, short, int, long, float, pointer – každý parametr jeden registr
- long long int, double – každý parametr dva registry (první registr má sudé číslo)
- struct předávaná hodnotou se kopíruje do tolika registrů, kolik je potřeba
- pokud je překládáno pro RISC-V s rozšířením pro práci s reálnými čísly, pak se pro předávání float, double i struktur využijí registry f10 – f17 označovaných také fa0 – fa7
- pokud se parametry nevejdou do registrů a0-a7, tak se přebývající parametry umístí na zásobník do rámce nově volané funkce

Kvíz

Kdy se parametry funkce, nebo registry s0-s11 uloží na zásobník?

A nikdy.

B když se v těle funkce uloží do parametru jiná hodnota.

C když bude v těle funkce volání jiné funkce se stejným počtem parametrů.

D když bude v těle funkce použit vstupní parametr po volání funkce.

Kvíz

Kdy se lokální proměnné, nebo registry s0-s11 uloží na zásobník?

A nikdy.

B když se v těle funkce bude využívat tolik lokálních proměnných, že se jejich hodnoty nevejdou do registrů.

C když bude v těle funkce volání jiné funkce.

D pouze když bude v těle funkce rekurzivní volání té samé funkce.

Volání funkce

Volání funkce s maximálně 8 parametry:

```
t = secti(1, 2, 3, 4);
```

Překlad do RISC-V

```
li  a3,4
li  a2,3
li  a1,2
li  a0,1
jal ra,10054 <secti>
```

Volání funkce s 10 parametry:

```
t = secti(1, 2, 3, 4, 5,
          6, 7, 8, 9, 10);
```

Překlad do RISC-V

```
li  a5,10
sw  a5,4(sp)
li  a5,9
sw  a5,0(sp)
li  a7,8
li  a6,7
li  a5,6
li  a4,5
li  a3,4
li  a2,3
li  a1,2
li  a0,1
jal ra,10054 <secti>
```

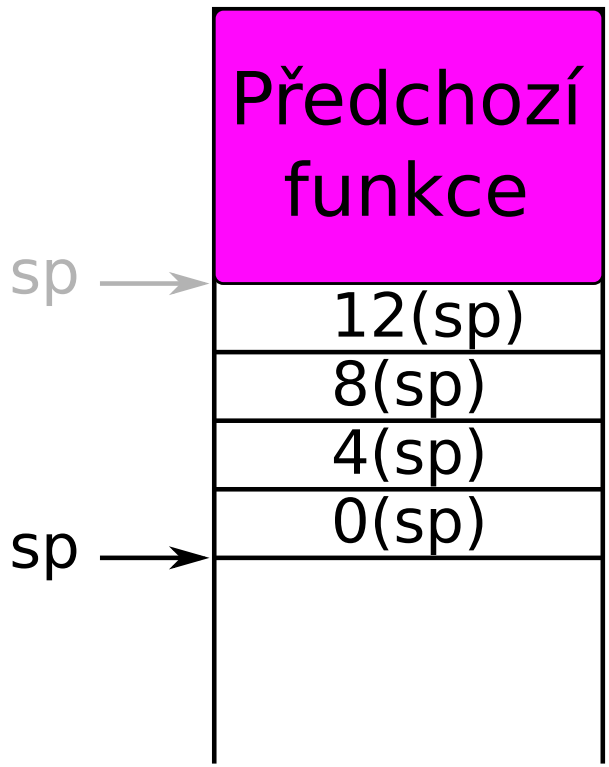
Překlad funkce - jednoduchá funkce

Příklad s málo parametry bez vnitřního volání funkce se čtyřmi lokálními proměnými.

- Alokace místa pro lokální proměnné
  `addi sp,sp,-16`
- Lokální proměnné na adresách:
  0(sp), 4(sp), 8(sp), 12(sp)
  - Pokud to lze, jsou proměnné umístěny v registrech a stack by se vůbec nepoužil

Ukončení funkce:

- Dealokace místa pro lokální proměnné
  `addi sp,sp,16`
- Návrat z funkce
  `jalr 0(x1)` – `ret`



Překlad funkce

Příklad s 10 parametry, volání funkce uvnitř funkce, se dvěma lokálními proměnými.

Začátek funkce:

```
addi  sp,sp,-48
sw  ra,44(sp)
sw  s0,40(sp)
sw  s1,36(sp)
sw  s2,32(sp)
sw  s3,28(sp)
sw  s4,24(sp)
```

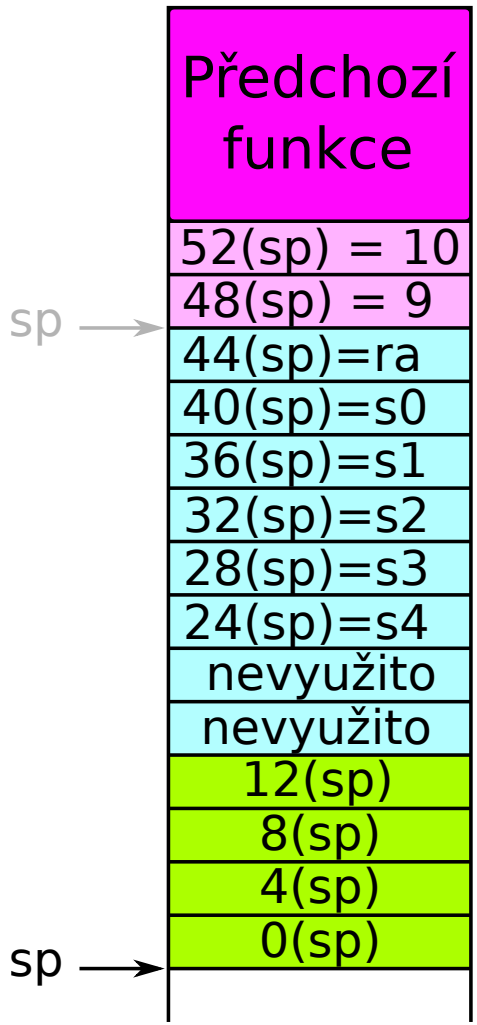
Lokální proměnné:

```
lw  t0,8(sp)
lbu t1,15(sp)
```

`int i;` na adrese 8(sp)
`char c;` na adrese 15(sp)
0(sp) – 7(sp) nevyužito

Ukončení funkce:

```
lw   ra,44(sp)
lw   s0,40(sp)
lw   s1,36(sp)
lw   s2,32(sp)
lw   s3,28(sp)
lw   s4,24(sp)
addi sp,sp,48
ret
```



Ukazatel na rámec funkce

- Ukazatel na rámec funkce obsahuje v RISC-V hodnotu sp při vstupu do funkce
- Pro fp (frame pointer) je využit registr s0, je to tedy registr, který musí volaný uložit a při návratu z funkce obnovit
- Výhody použití fp
  - Pevně nastavené ukazatele na argumenty a lokální proměnné funkce, pokud se sp v těle mění (např. argumenty pro volné funkce), pak se mění i posunutí vzhledem k sp pro přístup k argumentům a lokálním funkcím.
  - Přehlednější zásobník pro debugery a při tzv. odvinutí zásobníku.
    - Odvinutí zásobníku se provádí v C++ při výskytu výjimky ve funkci, kdy je ale výjimka zpracována až ve funkci nadřazené, která tuto funkci volala.
    - Pro zpracování výjimky je nutné nastavit zásobník do stavu, ve které nadřazená funkce volala funkci, ve které výjimka nastala.
- Nevýhody použití fp:
  - Zpomaluje program, jedná se sice jen o několik instrukcí při vstupu a ukončení funkce, ale pokud se funkce volá často a její tělo je krátké, může to být i značné zpomalení.
  - Fp obsadí registr s0, který by se dal využít pro uchování hodnoty. Pokud by došli volné registry, pak se hodnota musí uložit na zásobník do paměti RAM (přes cache) což je pomalejší než využití registru.

Překlad funkce s ukazatelem na rámec funkce

Pro překlad gcc pro RISC-V je nutné využít přepínač `-fno-omit-frame-pointer`

Začátek funkce:

```
addi sp,sp,-48
sw   ra,44(sp)
sw   s0,40(sp)
sw   s1,36(sp)
sw   s2,32(sp)
sw   s3,28(sp)
sw   s4,24(sp)
sw   s5,20(sp)
addi s0,sp,48
```

Odkaz na lokální proměnnou:

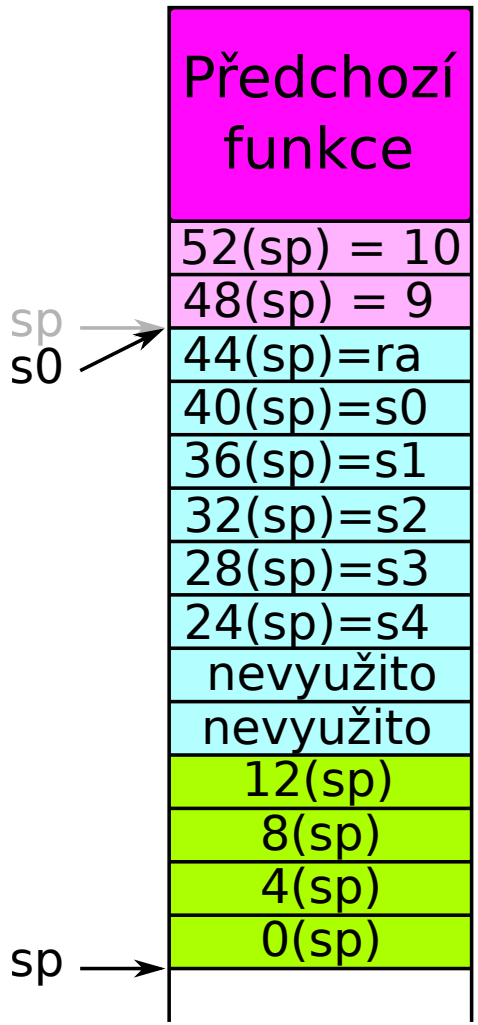
```
sw a0,-36(s0)
```

dříve:

```
sw a0, 12(sp)
```

Ukončení funkce:

```
lw   ra,44(sp)
lw   s0,40(sp)
lw   s1,36(sp)
lw   s2,32(sp)
lw   s3,28(sp)
lw   s4,24(sp)
lw   s5,20(sp)
addi sp,sp,48
ret
```



Řešení domácího úkolu 4

Analyzovat funkci subrutine_fnc:

- zjistit, jaké registry a0 – a7 se ve funkci používají a jaké se před volání funkce naplní
  - POZOR překladač využívá registry aX také pro pomocné výpočty stejně jako registry t0 – t6, protože jejich hodnoty může měnit
- zjistit, co je v registru a0 před zavoláním instrukce `ret` (`jalr 0(x1)`)
- prozkoumat začátek funkce:
  - pokud funkce začíná sekvencí instrukcí podobné:

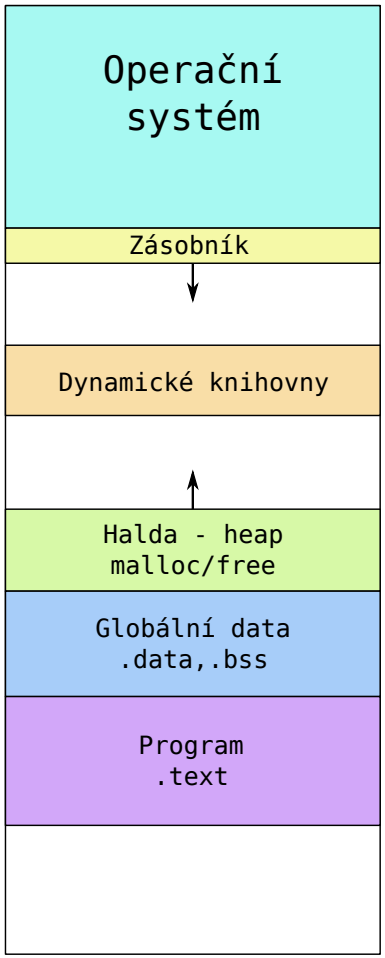
```
addi  sp,sp,-48
sw  ra,44(sp)
sw  s0,40(sp)
sw  s1,36(sp)
sw  s2,32(sp)
sw  s3,28(sp)
sw  s4,24(sp)
```

    - pak funkce využívá zásobník pro uložení návratové adresy a lokálních dat
    - adresy od 0(sp) do 23(sp) jsou využity pro lokální proměnné funkce
    - registry s0 – s4 (v tomto případě) jsou využity pro uchování dočasných hodnot, pravděpodobně argumentů funkce
  - pokud funkce neobsahuje instrukci `addi sp,sp,-X` pak se jedná o "jednoduchoou"funkci, která nepoužívá zásobník
    - všechny lokální proměnné a mezivýpočty jsou uloženy v nepoužitých registrech a0 – a7 a registrech t0 – t6

Rozložení běžícího programu v paměti

Každý proces má pro sebe 4GiB virtuálního prostoru:

- Nejvyšší 1GiB má pro sebe operační systém
- Následuje zásobník (pokud má proces více vláken, pak jsou zásobníky vláken řazeny za sebou)
- Níže jsou dynamické knihovny
- Od globálních dat roste dynamicky alokovaná paměť – heap (malloc,new/free,delete)
- Globální data (inicializovaná i neinicializovaná)
- Program (.text)
- Adresy od 0 se nechávají neobsazené kvůli odchytávání chyby dereference NULL ukazatele.



Řešení domácího úkolu 4

Využití globálních proměnných:

- globální proměnná je proměnná definovaná mimo funkci, nebo uvnitř funkce s použitím klíčového slova `static`
- globální inicializované proměnné jsou umístěné v sekci .data, neinicializované proměnné v sekci .bss
- Jak zjistit, zda v mém programu jsou globální proměnné?
  - podívejte se na konec výpisu riscv assembleru a najděte sekci my_data. Tato sekce může vypadat takto:

```
Contents of section my_data:
 11008 00000000                    ....
```

    - 11008 je adresa dat ve virtuálním prostoru procesu
    - 00000000 je hexadecimální výpis uložených dat
    - .... je textový výpis uložených dat, pokud znak nelze zobrazit je místo něj použita tečka
    - využití dat v programu vypadá takto:

```
lui  a5, 0x11
addi a5, a5, 8 # 11008 <nazev_promenne>
lw   a4, 0(a5)
```

Kvíz

Uvažujte tuto funkci

```
int fce (int a) {
  int i;

  // telo funkce

  return i+a;
}
```

Kde budou umístěny proměnné `a` a `i`:

A obě na zásobníku nebo v registrech

B obě v datové sekci

C `a` na zásobníku nebo v registru, `i` v datové sekci

D `a` v datové sekci, `i` na zásobníku nebo v registru

Útok na program přes zásobník

Uvažujme tento program:

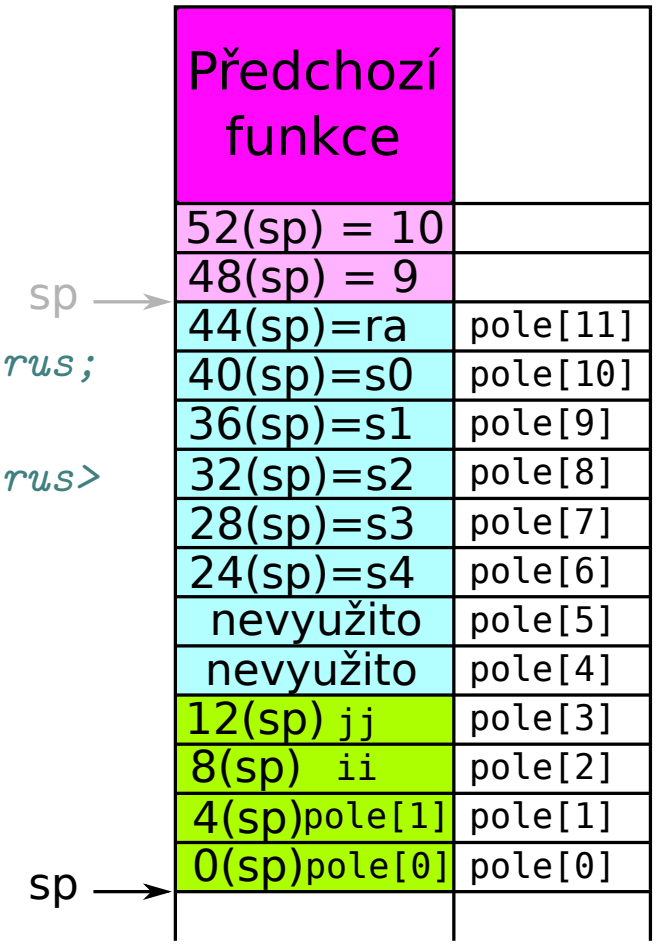
```
int virus() {
  // skodi
  return 0;
}

int secti(int a, int b, int c,
  int d, int e, int f, int g,
  int h, int i, int j) {

  volatile int ii,jj=i+j;
  volatile int pole[2];

  // neco pocita a
  // vola nejakou funkci

  pole[11] = (int)&virus;

  return pole[0]+pole[1];
}
```

Úvod funkce:

```
addi sp,sp,-48
sw   ra,44(sp)
sw   s0,40(sp)
sw   s1,36(sp)
sw   s2,32(sp)
sw   s3,28(sp)
# pole[11]=(int)&virus;
lui  a5,0x10
addi a5,a5,84 # <virus>
sw   a5,44(sp)
```

Ukončení funkce:

```
lw   ra,44(sp)
lw   s0,40(sp)
lw   s1,36(sp)
lw   s2,32(sp)
lw   s3,28(sp)
addi sp,sp,48
ret
```



Proměnný počet parametrů va_list

Definice funkce s proměnným počtem parametrů:

```
#include <stdarg.h>

int secti(int n, ...) {
  int souc = 0;
  int i;
  va_list ap;

  va_start(ap, n);
  for (i=0; i<n; i++) {
    souc+=va_arg(ap,int);
  }
  va_end(ap);
  return souc;
}
```

Volání funkce:

```
int  main() {
  printf("Soucet %d\n", secti(10, 1,2,3,
                              4,5,6,7,8,9,10);
  printf("Soucet %d\n", secti(2, 1,2);
  printf("Soucet %d\n", secti(8, 1,2,3,
                                 4,5,6,7,8);
}
```

Překlad funkce secti s proměnným počtem parametrů

Makro `va_start` a `va_arg` potřebuje mít uloženy všechny parametry v paměti:

Začátek funkce:

```
addi sp,sp,-64
sw   ra,28(sp)
sw   s0,24(sp)
sw   s1,20(sp)
sw   a1,36(sp)
sw   a2,40(sp)
sw   a3,44(sp)
sw   a4,48(sp)
sw   a5,52(sp)
sw   a6,56(sp)
sw   a7,60(sp)
```

`va_start(ap, n)`:

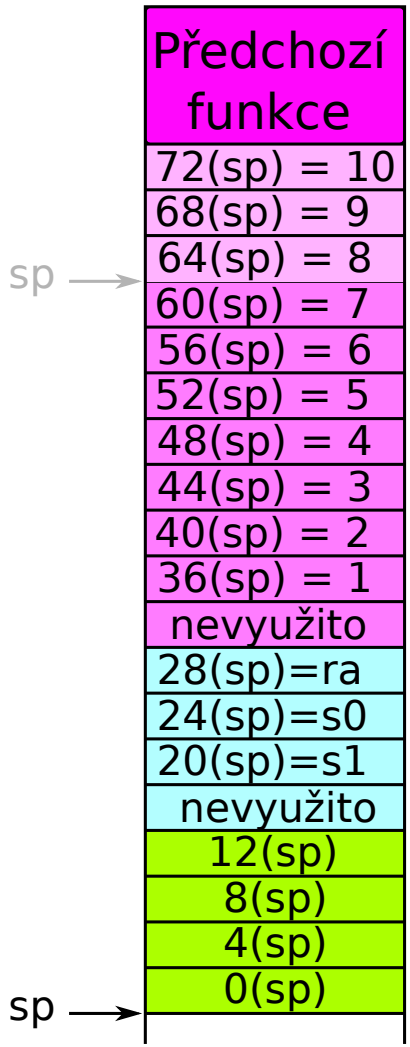
```
addi a5,sp,36
sw   a5,8(sp)
```

`va_arg(ap, int)`:

```
lw   a4,8(sp)
addi a3,a4,4
sw   a3,8(sp)
lw   s0,0(a4)
```

Ukončení funkce:

```
lw   ra,28(sp)
lw   s0,24(sp)
lw   s1,20(sp)
addi sp,sp,64
ret
```



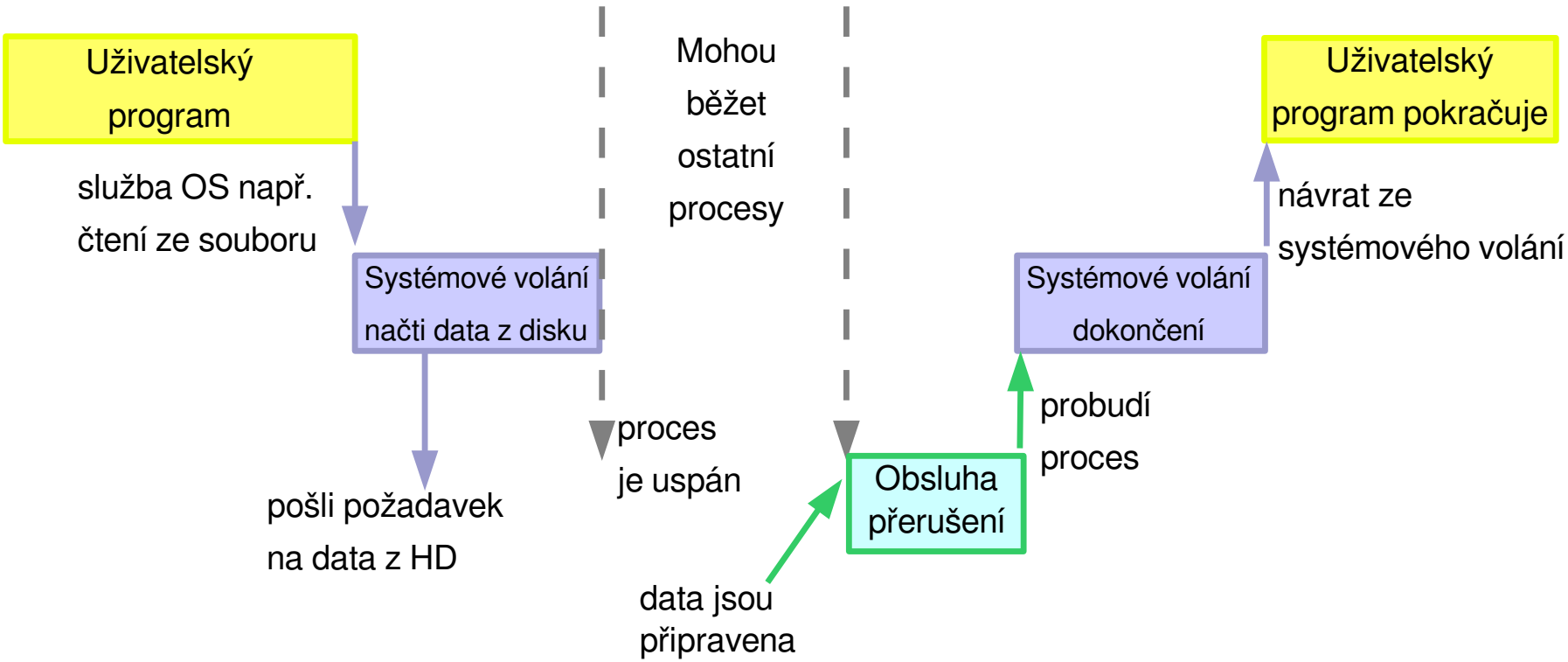
Obsah

Základy ochrany OS

- Uživatelský proces nemůže přímo přistupovat k HW počítače.
- Uživatelský proces musí požádat OS o zpřístupnění HW, nebo vyřízení požadavku.
- To co OS umožňuje procesům jsou pouze systémová volání (system calls).
- Na rozdíl od volání funkce neznáme adresy funkcí v jádře.
- Systémové volání se vybírá číslem systémového volání.
- Vlastní zavolání funkce je vyvolání přerušení nebo výjimky specializovanou instrukcí:
  - x86 – používá přímo int 0x80 – vyvolej přerušení číslo 0x80.
  - x86 – nověji specializované instrukce sysenter/syscall – podobný mechanizmus, ale nevyužívá uživatelský zásobník a méně přistupuje k paměti $\rightarrow$ je rychlejší.
  - RISC-V – specializovaná instrukce ecall – obdobné chování jako při přerušení.
- Při přerušení/výjimce se volá program z adresy, která je nastavená OS, uživatel ji nemůže měnit.

Ochrana OS

Uživatelský program nastaví parametry volání do registrů vyvolá speciální přerušení.
OS podle čísla systémového volání zavolá odpovídající funkci.



Návrat ze systémového volání odpovídá návratu z přerušení.

ABI systémového volání

- API – application programming interface – popis, jaké funkce může Váš program zavolat z knihoven.
  - API je definováno pro jazyk C hlavičkovými soubory.
  - API definuje i co dané funkce dělají, jaké vrací hodnoty, jak se chovají při chybě
  - zkuste např. `man 2 read`
- ABI – application binary interface – popis jaké registry a jaké instrukce použít
- ABI pro systémová volání na architektuře RISC-V
  - a7 – obsahuje číslo systémového volání (přehled např. `https://jborza.com/post/2021-05-11-riscv-linux-syscalls/` nebo `https://marcin.juszkiewicz.com.pl/download/tables/syscalls.html`)
  - a0 až a5 – parametry systémového volání (Linuxové systémové volání má maximálně 6 parametrů)
  - systémové volání se provede instrukcí `ecall`
  - a0 obsahuje návratovou hodnotu systémového volání
    - pokud by mělo vracet více dat (např. čtení ze souboru) musí uživatel zadat ukazetel na buffer a velikost bufferu, kam OS zapíše data.

Příklad Hello world!

```
.global _start
.text
_start:
#   write(1, "Hello world!\n", 13);
    addi  a7, zero, 64
    addi  a0, zero, 1
    la    a1, zero, text_1
    addi  a2, zero, 13
    ecall
final:
#   exit(0);
    addi  a7, zero, 93
    addi  a0, zero, 0
    ecall
    ebreak
    j final
.data
# store ASCII text, no termination
text_1: .ascii "Hello world!\n"
```

Řešení domácího úkolu 4

Co dělat se systémovým voláním?

- najděte instrukci `ecall`.
- podle hodnotu registru a7, z ní zjistěte o jaké systémové volání se jedná.
- zjistěte hodnoty registrů a0, a1, a2 (případně a3 pokud je použit).
- do C programu zařaďte volání funkce z API, která provede dané systémové volání.
- Příklad: zjistíte, že registr a7 má hodnotu 63 – read, vygenerujete volání funkce read

  ```
  read(a0, a1, a2);
  ```

  - jediný problém je s funkcí open/openat, jejíž parametry mají jiné hodnoty pro systém x86 a pro riscv
  - protože programy kontroluje Brute na systému x86, je lepší ověřit hodnoty parametrů ve výpisu program-x86.list

Systémové volání x86

- systémové volání provádí instrukce `int 0x80`.
- číslo systémového volání je v registru eax.
  - POZOR čísla systémových volání x86 a riscv se liší.
- parametry systémového volání jsou uloženy postupně v registrech:
  - ebx
  - ecx
  - edx
  - esi
  - edi
  - ebp
- příští přednášku si probereme assembler x86, abyste mohli využít pro řešení domácího úkolu 4 i výpis instrukcí pro procesor x86.

Kvíz za bonusový bod

Uvažujte tuto funkci

```
int fce (int a) {
  static int s;
  int i;

  // telo funkce

  return i+s;
}
```

Kde budou umístěny proměnné `s` a `i`:

A obě na zásobníku

B obě v datové sekci

C `s` na zásobníku, `i` v datové sekci

D `s` v datové sekci, `i` na zásobníku