

Segmentace paměti

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

04.12.2025

CC BY-NC-SA 4.0

Segmentace paměti

Organizace paměti programu

Každý program má paměť rozdělenou do několika segmentů

Proč?

- Organizace a ochrana dat
- Efektivní využití paměti
- Bezpečnost (některé oblasti read-only)

Organizace paměti programu

1. Text (Code)
 - kód programu
2. Data (initialized)
 - inicializované globální a statické proměnné
3. BSS (Block Started by Symbol) (uninitialized)
 - neinicializované globální a statické proměnné
4. Heap
 - dynamická paměť (`malloc`)
5. Stack
 - lokální proměnné a parametry funkcí

Segmentace paměti






Stack (zásobník)

- Lokální proměnné
- Parametry funkcí
- Návrátové adresy

Heap (halda)

- Dynamická paměť (malloc)

BSS 
Data 
Text 

BSS (Block Started by Symbol)

- Neinicializované globální/static

Data

- Inicializované globální/static

Text

- Strojový kód programu

Segmentace paměti - Praktický příklad

```
int globalInit = 42;    // Data segment
int globalUninit;      // BSS segment

int sqr(int num) {      // Text segment (code)
    return num * num;
}

int main() {
    int x = 5;           // Stack
    static int s = 10;   // Data segment

    int *ptr = malloc(sizeof(int)); // Heap
    *ptr = 20;
```

Stack (zásobník)

Stack (zásobník) je speciální oblast paměti pro **dočasná data**

Klíčové vlastnosti:

- **LIFO** (Last In, First Out) - jako zásobník talířů
- **Fixní velikost** - určená při startu programu (typicky 1-8 MB)
(`ulimit -s` - v kB)
- **Rychlý přístup** - velmi efektivní
- **Automatická správa** - není třeba `free()`

Stack (zásobník)

Co se ukládá na Stack?

- Lokální proměnné funkcí
- Parametry funkcí
- Návrátové adresy (kam se vrátit po funkci)
- **Stack frame** každé funkce

Stack frame

```
void func2(int b) {    // Stack frame for func2
    int local2 = b * 2;
} // ← Stack frame is freed
```

```
void func1(int a) {    // Stack frame for func1
    int local1 = a + 1;
    func2(local1);
} // ← Stack frame is freed
```

```
int main() {           // Stack frame for main
    int x = 5;
    func1(x);
} // ← Stack frame is freed
```

Stack frame

Při každém volání funkce:

1. Vytvoří se nový **stack frame**
2. Po návratu z funkce se stack frame **automaticky** uvolní

Stack - vizualizace

```
void func(int x) {  
    int y = x * 2;  
}
```

```
int main() {  
    int a = 10;  
    func(a);  
}
```

Stack - vizualizace

Začátek

main: a=10

Volání func

func: y=20
func: x=10
func: ret = 0x....
main: a=10

Návrat

main: a=10

Konec

prázdný

Stack - Proč je důležitý?

Výhody:

- Automatická správa paměti
- Velmi rychlý (cache-friendly)
- Žádné memory leaks z lokálních proměnných

Nevýhody / Omezení:

- Omezená velikost (stack overflow!)
- Proměnné „zmizí“ po ukončení funkce
- Nelze předat velká data efektivně

Stack - Proč je důležitý?

Otázka: Je dobrý nápad vrátit pointer na lokální proměnnou?

Odpověď: NE! Proměnná přestane existovat po návratu z funkce!

Pointer na lokální proměnnou

```
struct Person {  
    char name[20];  
    int age;  
};  
struct Person *createJohn() {  
    struct Person p = {"John", 20};  
    return &p;  
}  
int main() {  
    struct Person *john = createJohn();  
    printf("%s \n", john->name);  
    return 0;  
}
```

Pointer na lokální proměnnou

Výstup:

warning: address of stack memory associated with local variable 'c' returned [-Wreturn-stack-address]

Stack overflow

```
int main() {  
    int arr[1000000000000];  
    printf("%d", arr[0]);  
  
    return 0;  
}
```

Výstup:

```
[1]      97844 segmentation fault  ./a.out
```