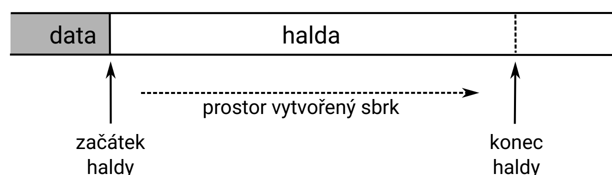
Alokace paměti na haldě (Linux) 9

Programy ke své práci obvykle využívají dvě oblasti paměti – *zásobník* (stack) a *haldu* (heap). Zatímco zásobník obvykle slouží k uložení lokálních proměnných, jejichž platnost je pouze po dobu provádění dané funkce, halda slouží k dynamické alokaci objektů (úseků paměti), které můžeme použít i po skončení funkce, která je alokovala. V tomto cvičení si ukážeme základní principy, které se používají při dynamické alokaci paměti.

1 Získání oblasti paměti

Práci s haldou zajišťuje buď standardní knihovna (např. jazyk C) nebo běhové prostředí (např. jazyk Java, C#). V obou případech běžící proces získá od operačního systému oblast paměti, kterou podle potřeby dělí na menší úseky (objekty). V jazyce C k získání těchto úseků slouží funkce `malloc` a k uvolnění již nepoužívaný úseků paměti slouží funkce `free`.

K získání oblasti paměti, kterou budeme dělit na menší části, můžeme v unixových operačních systémech použít systémové volání `sbrk`. Toto systémové volání rozšíří datový segment procesu[1] o zadaný počet bytů. Návratovou hodnotou je ukazatel na předchozí konec datového segmentu. Použití `sbrk` ilustruje Obrázek 1.



Obrázek 1: Halda

Pro správu této oblasti paměti si zavedeme několik globálních proměnných.

```
unsigned char *data_area = NULL;        // ukazatel na volnou oblast haldy
unsigned char *data_area_start = NULL; // ukazatel na zacatek haldy
size_t data_area_capacity = 0;          // velikost haldy v bytech
```

A funkci, která v případě potřeby inicializuje oblast, kde bude halda, případně ji rozšíří.

```
/** rozsiri haldu o MIN_ACQ_SIZE */
static void acquire_mem()
```

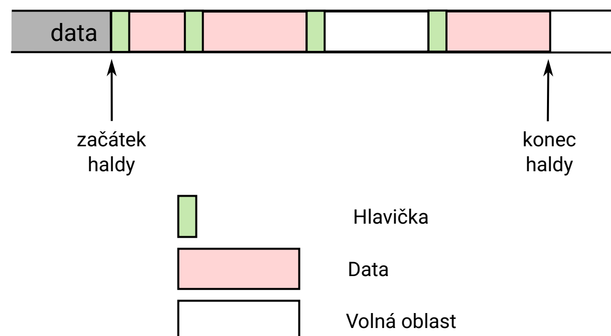
[1] Ve smyslu oblasti paměti, kterou je možné použít pro práci s daty, nemusí se jednat o datový segment ve smyslu procesorů x86.

```
{
        unsigned char *last_pos = sbrk(MIN_ACQ_SIZE);
        if (!data_area_start) {
                data_area_start = last_pos;
                data_area = data_area_start;
        }
        data_area_capacity += MIN_ACQ_SIZE;
}
```

Poznámka: Alternativně můžeme k získání oblasti paměti, kterou je možné dále dělit, použít mapování souboru do paměti. Buď můžeme namapovat do paměti soubor `/dev/zero` nebo použít mapování anonymní oblasti paměti.

2 Struktura haldy

Máme-li k dispozici souvislou oblast paměti, můžeme ji dělit na menší bloky. Každý takový blok se skládá z hlavičky nesoucí informace o jednotlivých blocích (např. jeho velikost) a z části obsahující data, viz Obrázek 2. Bloky dále budeme dělit na *používané*, obsahující platná data, a *uvolněné*, které je možné opětovně použít. V našem případě budeme předpokládat, že každý objekt má hlavičku složenou ze dvou slov a pro data je alokována paměť minimálně o velikosti dvou slov.[2]



Obrázek 2: Ilustrace rozdělení haldy na menší bloky

K popisu obou typů objektů si zavedeme strukturovaný datový typ:

```
struct chunk {
    size_t prev_size;    // velikost predchoziho bloku (v bytech, vcetne hlavicky)
    size_t size;         // velikost bloku (v bytech, vcetne hlavicky)
    struct chunk *prev;  // odkaz na predchozi volny blok
    struct chunk *next;  // odkaz na dalsi volny blok
};
```

V této struktuře první dva atributy představují hlavičku bloku paměti, kdy `size` udává velikost celého bloku v bytech a to včetně hlavičky, a `prev_size` udává velikost předchozího bloku[3] a v nejnižším bitu

[2]Tj. při každé alokaci je použita oblast minimálně o velikosti čtyř slov.
[3]Tuto informaci můžeme využít při slučování sousedních volných bloků.

tohoto atributu si udržujeme informaci o tom, zda je aktuální blok používaný (bit je nastaven na 0) nebo je uvolněný (bit je nastaven na 1).[4] Atributy `prev` a `next` jsou již v datové části bloku a slouží k vytvoření spojového seznamu volných bloků.

2.1 Práce s jednotlivými bloky

Pro manipulaci s jednotlivými bloky dat si zavedeme několik pomocných funkcí.

Následující funkce pro zadaný blok vrací odkaz na jeho datovou část (oblast za hlavičkou). V tomto kódu se používá přetypování na typ `unsigned char *`, díky čemuž můžeme (společně s pointerovou aritmetikou) dohledat příslušný byte v paměti, tj. posunout se za hlavičku daného bloku.

```
static inline void *chunk_to_ptr(struct chunk *mem) {
    return ((unsigned char *) mem) + HDR_SIZE;
}
```

Analogicky můžeme z ukazatele na datovou oblast získat odkaz na celý blok včetně hlavičky.[5]

```
static inline struct chunk *ptr_to_chunk(void *ptr) {
    return (struct chunk *) (((unsigned char *) ptr) - HDR_SIZE);
}
```

Dále si zavedeme funkce, které umožní zjistit a nastavit příznak, zda je blok používaný nebo uvolněný.

```
static inline int chunk_is_free(struct chunk *mem) {
    return mem->prev_size & 0x01;
}
static inline void chunk_set_free(struct chunk *mem, int free) {
    mem->prev_size = (mem->prev_size & ~0x01) | free;
}
```

K procházení bloků paměti na haldě si vytvoříme další dvě pomocné funkce, které vrátí odkaz na předchozí nebo následující blok na haldě, pokud takové bloky existují.

```
static inline struct chunk *chunk_preceding(struct chunk *mem) {
    unsigned char *pos = (unsigned char *) mem;
    if (pos == data_area_start) return NULL;
    return (struct chunk *) (pos - (mem->prev_size & ~0x01));
}
static inline struct chunk *chunk_succeeding(struct chunk *mem) {
    unsigned char *pos = (unsigned char *) mem;
```

[4] To můžeme bezpečně použít, protože velikost bloků paměti jsou zaokrouhleny na dvojnásobky slov, tudíž jsou nejnižší bity vždy 0.

[5] Protože jsou obě funkce relativně malé, jsou deklarovány jako `inline`. Překladač pak nebude volat funkce tradičním způsobem, ale vloží jejich tělo na místo v programu, kde jsou použity. V případě funkce `chunk_to_ptr` nemusíme explicitně uvádět převod na typ `void *`, ten se v jazyce C provádí implicitně.

```
    if ((pos + mem->size) == data_area) return NULL;
    return (struct chunk *) (pos + mem->size);
}
```

2.2 Alokace bloků

Při alokaci bloků paměti máme dvě možnosti, jak postupovat. Buď můžeme „recyklovat" již nepoužívaný blok nebo alokovat blok nový. Tento princip se dá přímo popsat následujícím kódem.

```
void *tmalloc(size_t size) {
        size = ROUND_UP(size + HDR_SIZE);
        struct chunk *mem = chunk_find(size);
        if (mem) chunk_set_free(mem, 0);
        else mem = chunk_allocate(size);
        return chunk_to_ptr(mem);
}
```

Nejdříve spočítáme velikost bloku, který potřebujeme (přičteme velikost hlavičky a zaokrouhlíme na dvojnásobek velikosti slova). Funkcí `chunk_find` zkusíme najít vhodný uvolněný blok, a pokud existuje, označíme jej jako používaný a použijeme jej. V opačném případě alokujeme blok nový. Funkce `tmalloc`, která je ekvivalentem `malloc`, vrátí odkaz na datovou část bloku bez ohledu na to, jestli se jedná o nový nebo „recyklovaný" blok.

Podívejme se nejprve na to, jak alokovat nový blok.

```
1  static struct chunk *chunk_allocate(size_t size) {
2      while (size > data_area_capacity)
3          acquire_mem();
4      struct chunk *mem = (struct chunk *) data_area;
5      data_area += size;
6      data_area_capacity -= size;
7      mem->size = size;
8      if (last_chunk) mem->prev_size = last_chunk->size;
9      else mem->prev_size = 0;
10     last_chunk = mem;
11     return mem;
12 }
```

Nejdříve zajistíme, že máme na haldě dostatek místa, které můžeme přidělit (řádky 2 a 3). Alokujeme příslušný blok (řádky 4 až 6) nastavíme hodnoty v hlavičce (řádky 7 až 9). Abychom věděli, jak velký byl předchozí blok, použijeme globální proměnnou `last_chunk`, která obsahuje ukazatel na poslední alokovaný blok na haldě (pokud takový existuje).

2.3 Uvolnění bloku

Abychom mohli opětovně použít uvolněné bloky, musíme mít o nich přehled. Pro jednoduchost budeme všechny uvolněné bloky shromažd'ovat v oboustranném spojovém seznamu, který je určen globální proměnnou `free_blocks`. Práci s tímto seznamem zajištuje dvojice funkcí `chunk_enqueue` a `chunk_dequeue`, které daný blok zařadí na seznam nebo z něj vyjmou, viz přiložené zdrojové kódy.

Při uvolňování bloku postupujeme tak, že se podíváme na sousední bloky, a pokud jsou volné, sloučíme je s uvolňovaným blokem, jak ilustruje následující kód.

```
1  static void chunk_enqueue_merged(struct chunk *mem) {
2      struct chunk *prec = chunk_preceding(mem);
3      struct chunk *succ = chunk_succeeding(mem);
4      if (prec && chunk_is_free(prec)) {
5          chunk_dequeue(prec);
6          prec->size += mem->size;
7          mem = prec;
8      }
9      if (succ && chunk_is_free(succ)) {
10         chunk_dequeue(succ);
11         mem->size += succ->size;
12         succ = chunk_succeeding(mem);
13     }
14     if (succ) {
15         succ->prev_size = mem->size;
16         chunk_enqueue(mem);
17     } else {
18         data_area -= mem->size;
19         data_area_capacity += mem->size;
20         last_chunk = chunk_preceding(mem);
21     }
22 }
```

Pokud je předchozí blok volný (řádky 4 až 8), je tento blok zvětšen o námi uvolněný blok a dále pracujeme s tímto (předchozím) blokem. Pokud je následující blok volný, je uvolňovaný blok rozšířen a jeho velikost. V obou případech jsou sousední volné bloky odstraněny ze seznamu uvolněných bloků. Díky tomuto přístupu, nejsou v paměti nikdy dva volné bloky vedle sebe.

V případě, že existuje za uvolněným blokem blok paměti, který je používaný, je mu upravena hlavička a uvolněný blok je zařazen na seznam uvolněných bloků (řádky 14 až 16), v případě, že jsme uvolnili poslední blok na haldě, můžeme oblast haldy zmenšit (řádky 18 až 20).

S pomocnou funkcí `chunk_enqueue_merged` je implementace ekvivalentu funkce `free` přímočará.

```
void tfree(void *ptr) {
    struct chunk *mem = ptr_to_chunk(ptr);
```

```
    chunk_enqueue_merged(mem);
}
```

2.4 Vyhledání uvolněného bloku

U funkce `tmalloc` jsme předpokládali, že existuje funkce `chunk_find`, která najde uvolněný blok vhodné velikosti. Nyní, když víme, jak jsou uvolněné bloky organizovány, můžeme tuto funkci doplnit. Následující funkce je jedna z možných a implementuje strategii *first-fit*, tj. je použit první vhodný blok.

```
static struct chunk *chunk_find(size_t size) {
    if (!free_blocks) return NULL;
    struct chunk *c = free_blocks;
    while (c) {
        if (c->size >= size) {
            chunk_dequeue(c);
            return c;
        }
        c = c->next;
    }
    return NULL;
}
```

Poznámka: V tomto bodě je již dynamická alokace kompletní, i když má četné nedostatky. Zejména správa volných bloků a jejich vyhledávání není efektivní. Důležité je i zmínit, že takto naprogramovanou alokaci paměti nemůžeme použít ve vícevláknových aplikacích. Pro ně by bylo nutné doplnit zamykaní.

3 Úkoly

1. Struktura haldy je zvolena tak, aby ji šlo snadno analyzovat, k tomu slouží funkce `tmalloc_debug`. Podívejte se, jak tato funkce funguje, a vyzkoušejte si alokovat/uvolňovat různé objekty.
2. Vytvořte (jednoduchý) spojový seznam a do něj načtěte různě dlouhé řádky ze souboru. Podívejte se, jak budou v paměti uloženy a jak pro ně bude alokován prostor.
3. Dvojici funkcí `tmalloc` a `tfree` doplňte o funkci `trealloc`, která se bude chovat jako funkce `realloc` ze standardní knihovny.
4. Vyzkoušejte funkci `trealloc` a podívejte se, jak je alokován a uvolňován prostor.
5. Upravte funkci `chunk_find`, aby používala strategii best-fit.
6. (**bonusový úkol**) Upravte práci s volnými bloky tak, aby bloky do určité velikosti, např. 512 B byly rozděleny do samostatných seznamů a nemusely být vyhledávany mezi ostatními (velkými) bloky.