Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičová 3, 842 19 Bratislava 4

Michal Slovík

Vlastná implementácia alokácia a uvoľňovanie pamäte

Dátové štruktúry a algoritmy

Autor: **Michal Slovík**
Predmet: **Dátové štruktúry a algoritmy**
Vedúci projektu: **Ing. Ivan Kapustník**

Reprezentácia blokov pamäti

Na reprezentáciu blokov pamäte som využíval jednu hlavičku na začiatku každého bloku pamäte, ktorá obsahovala veľkosť daného bloku a tiež či je blok obsadený alebo voľný pre potreby užívateľa. Hlavičky sú reprezentované údajovým typom „int“ ,to znamená štyrmi bajtami(byte). Informáciu o využívaní bloku poskytuje najvyšší bit (posledný bit), takže

(1 – plný, alebo aj nepárny, 0 – voľný, teda párny). To znamená, že ak používateľ požiada o *size* veľkosť pamäte a ak ide o nepárnu veľkosť tak zapíšeme *size* aj do hlavičky, ako náhle by požiadal užívateľ o párnu veľkosť do hlavičky by sme zapísali *size+1*.

Memory init()

Táto funkcia dostáva ako argumenty veľkosť celej pamäte a ukazovateľ, ktorý ukazuje na jej začiatok. V mojom prípade som tento prvý „blok“ pamäte upravoval na párny, teda hlavička má zapísanú parnú hodnotu, aj keď blok obsahuje inú hodnotu. Pracuje teda podobne ako funkcia malloc opísaná nižšie.

Memory malloc()

V tejto funkcii priradzujem samotnú pamäť. Používateľ ma požiada o veľkosť, ktorú potrebuje a ktorú mu mám vyhradiť v pamäti. Najskôr zvýšim jeho požiadavku aj o veľkosť hlavičky, ktorú tento blok potrebuje. A túto celkovú veľkosť porovnám či sa vojde do pamäti.

Ak našiel dostatočne miesto teda použijem **First Fit**, zmením hlavičku, v prípade že je nepárna posledný bit ostane nezmení v opačnom prípade sa prepíše.

**BlockOfMemory = *BlockOfMemory | 1;*

Alternatívny *zápis *BlockOfMemory = *BlockOfMemory | 0x1;*

Po úspešnom pridelení pamäte si zapamätám ukazovateľ, ktorý ukazuje na začiatok prideleného bloku pamäte a ten vraciam užívateľovi.

Zvyšok pamäte ktorý ostal, vyplním prázdnym blokom. Takže v konečnom dôsledku sa pri vykonávaní funkcie malloc vytvárajú dva bloky pamäte, jeden blok ako ten o ktorý ma požiadal používateľ plus ďalší zvyčajne väčší blok pamäte označení ako voľný.

V prípade neúspešného alokovania pamäte vracia pointer null.

Memory free()

Funkcia free() uvoľňuje miesto v pamäti. Najskôr kontrolujeme či ukazovateľ, ktorý som dostal od užívateľa je správny. To znamená či je z intervalu od začiatku do konca pamäte. Ak nie tak funkcia sa ukončí a vracia 1.

Ak patrí do intervalu, postupne prechádza pamäťou (lineárne) kým nenájde ten správny ukazovateľ. Uvoľnený blok pamäte sa porovnáva s nasledujúcim blokom pamäte(ktorý je vzdialený presne o veľkosť, ktorú ideme uvoľniť) ak by aj tento blok bol uvoľnený (jeho posledný bit by obsahoval 0) pripojil by ho. Takýmto postupom sa vytvára fragmentácia pamäte, teda spájanie voľných blokov pamäte.

Pri takomto spájaní blokov pamäte sa iba upraví iba hlavička prvého bloku do ktorej sa pridá hodnota nasledujúceho bloku aj s jeho hlavičkou, a posledný bit sa nastaví na voľný(0)

Memory check()

Posledná funkcia Memory check, dostáva ako argumenty už alokované pointre, ktoré neboli zatiaľ uvoľnené funkciou Memory_free().

Najskôr sa overí či pointer, ktorý je na vstupe je platný a či je z rozsahu aktuálnej pamäte.

Potom postupne prechádza pamäťou (preskakuje po hlavičkách , kde si prečíta o koľko má vykonať nasledujúci skok) a kontroluje kedy sa pointer zo vstupu rovná pointru v pamäti. Ak takýto pointer v pamäti nájde, skontroluje jeho posledný bit, keďže by mal byť ešte aktívny, teda alokovaný mal by byť nastavený na 1. Ak by nebol alebo by nastala nezhoda ešte skôr vracia neúspech.

V prípade že prejde celý cyklus a nenájde vyhovujúci pointer taktiež vracia neúspech.

```
for ( start_Char = start_Char + HEAD_SIZE ; start_Char < size_of_memory; )
            if ( next_p_char == start_Char )
                    return 1;
```

Príklad pridelenia pamäte

Takto to vyzerá pamäť po funkciách *init() a memory_alloc()*

Funkcia init(), ktorá je vždy spustená ako prvá, nastavuje globálny ukazovateľ

*void *start_of_memory = NULL;*

aby ukazoval na začiatok celej pamäte, v tomto prípade je to adresa 0x24fde0(decimálne ...288). Tento ukazovateľ nám zároveň poskytuje veľkosť celej pamäte s ktorou môžeme pracovať.

Ako náhle si používateľ popýta alokovať blok pamäte o veľkosti 10 byte: pracujeme s týmito údajmi. Vieme aká je celková veľkosť, na základe globálnej premennej. Ďalej vieme, že náš blok pamäte musí mať hlavičku o veľkosti 4 byte, takže celkovo potrebujeme 14 bajtov. Zároveň potrebujeme označiť tento blok ako alokovaný, takže do hlavičky, ktorá bude nasledovať hneď po prvej globálnej hlavičke zapíšeme údaj 15(keďže nepárnymi hodnotami reprezentujeme obsadený blok)

Používateľovi vraciam pointer, ktorý ukazuje na začiatok tohto bloku pamäte, teda na adresu 0x24fde8 (... 296).

Blok pamäte končí na adrese 0x24fdf2, tým vieme, že presne na ďalšom bloku pamäte bude existovať hlavička s určitou hodnotou. Aby sme sa vyhli problém, tak táto druhá hlavička sa tiež vytvára pri funkcii alok, a to tým spôsobom, že sa vypočíta zvyšok pamäte, ktorý mi ostané po alokovaní. Tento zvyšný blok pamäte je voľný, takže do hlavičky zapisujem reálnu hodnotu (párnu).

Ak by som chcel uvoľniť pamäť, tak hodnota, ktorá je v hlavičke na adrese 0x24fde4 by som zmenil na párnu, v našom prípade by to znamenalo 14.

Pri každom uvoľňovaní pamäť sa zisťuje, či nasledujúci blok pamäte je voľný alebo obsadený. Ak by bol voľný dochádza ku fragmentácií, teda spájaniu voľných blokov pamäte. Do prvej hlavičky sa zapíše veľkosť jej bloku + veľkosť nasledujúceho bloku aj s jeho hlavičkou. A tým nám vznikne jeden veľký blok.

```
 while (( (size_of_memory) > heavy_next ) && !(IS_ODD(*heavy_next)))
{
            *next_p_int +=  *heavy_next + HEAD_SIZE;
            next_p_char += *heavy_next + HEAD_SIZE;
            heavy_next = next_p_char;
}
```

Asymptotická zložitosť

Väčšina operácií prebieha v O(N) čase, keď musíme prehľadávať všetky bloky pamäte.

Funkcia init() má zložitosť O(1), keďže prebehne iba raz a to buď úspešne alebo neúspešne. Memory_alloc(), záleží od počtu hlavičiek v pamäti, cez ktoré musí prechádzať ,takže O(N).

V prípade Memory_free() je podobné, kde tiež musí preskakovať cez hlavičky blokov pamäte, a hľadať kontrolovaný blok. S tým, že ešte musí pozerať aj na nasledujúci blok a spájať ho s predošlým, čo ale nemusí nastať vo všetkých prípadoch.

Funkcia Memory_check pracuje veľmi podobne, ako memory_free(), takže zložitosť bude tiež O(N).

Testovanie

Vytvoril som aj vlastné testovacie scenáre, test_case1, ktorý pracuje s blokom o veľkosti 100 a zaplní ho 7 blokmi o 8 byte (56 + 7*4 = 84 + 4 = 88) a tie sú následne uvoľnené.

V druhom teste ide o pamäť 1204, a 50 blokov o veľkosti 20 (1200 +4) a tie sú uvoľnené a skontrolované funkciou memory_check.

V poslednom teste pracuje podobne ako pri teste 2, ale bloky sú alokované ešte raz, a znova kontrolované.

```
(test_case1()) ? printf("test 1 OK\n") : printf("test 1 BAD\n");
```

```
int test_case2()
{
	char region[1300];	// cela pamat ma velkost 1000

	char *ptr[50];
	int i;

	memory_init(region, 1204);

	for (i=0; i < 50; i++)
	{
		ptr[i] = memory_alloc(20); // by sa ich tam nemalo vojst
		if (!ptr[i])
			return 0;
...
```