Jozef Varga, ID: 86060

Správca pamäti

Autor: Jozef Varga

Riešenie

V mojom riešení som využil štruktúry a jednosmerne spájaný zoznam. Dôvodom využitia štruktúry je väčšia prehľadnosť.

```
struct bloky{
   unsigned int velkost;
   struct bloky *po;
};
```

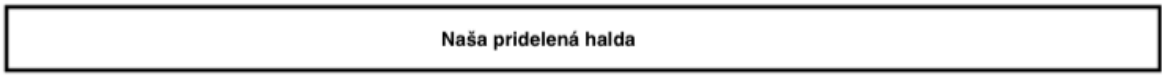
Moja štruktúra obsahuje informáciu o veľkosti bloku. Táto štruktúra sa vytvára aj pre voľné aj pre plné bloky s tým, že prepojené sú len voľné bloky. Pre plné bloky táto štruktúra slúži len ako informácia o veľkosti daného plného bloku. Ďalej obsahuje pointer na nasledujúcu štruktúru (ako bolo vyššie spomenuté na nasledujúcu voľnú)

Pomocné funkcie

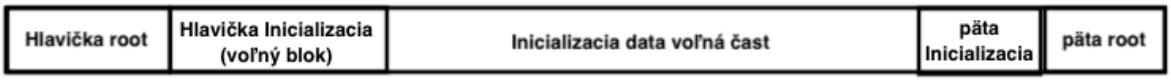
```
void set_pata_velkost(unsigned int *cesta, unsigned int velkost){
   *cesta = velkost;
}
void set_pata_obsadenost(char *cesta, char obsadenost){
   *cesta = obsadenost;
}
```

Tieto dve funkcie slúžia na priradenie päty. Päta slúži na rozpoznávanie bloku, je tam zaznamenaná hodnota či je daná bunka obsadená/ voľná a aká je veľkosť možnej pamäte na alokovanie. Vytvorené sú na sprehľadnenie kódu.

Funkcia memory_init()



Obr. 1. Pamät pred vykonaním memory_init



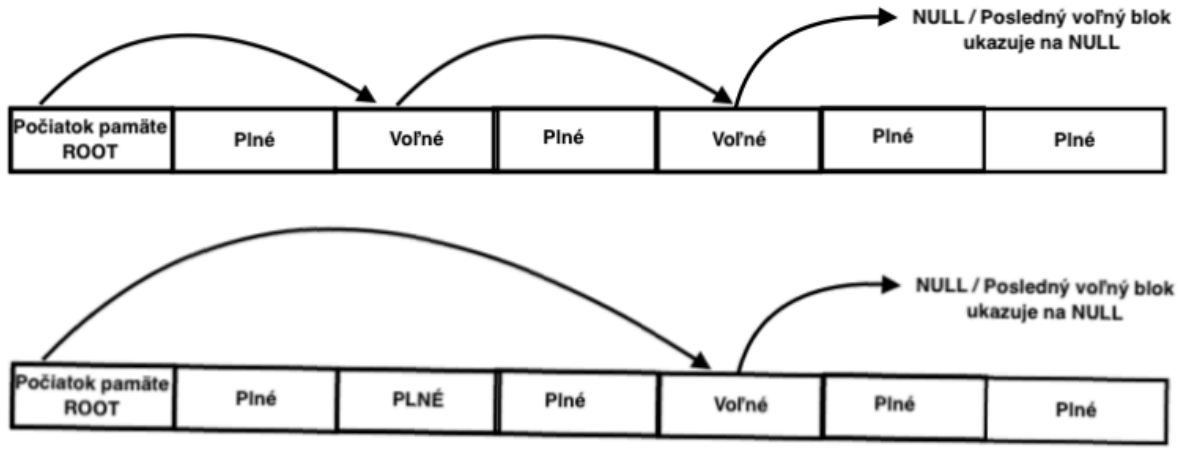
Obr. 2. Pamäť po vykonaní memory_init

Memory_init nám pripravuje našu "haldu" na následné spracovanie. Ako prvé si vytvoríme našu root hlavičku ktorá je reprezentovaná štruktúrou a má veľkosť 0. Jej primárnou úlohou je ukazovať na voľný blok. Ďalej si inicializujeme štruktúru Inicializacia ktorá je v pamäti hned za root a obsahuje veľkosť s hodnotou SIZE - 2*veľkosť päty - 2*veľkosť štruktúry. Táto veľkosť nám hovorý o množstve bajtov aké sme schopný v tomto voľnom bloku alokovať tým pádom musíme odpočítať funkčné bloky. Následne sa pomocou veľkostí vieme posunúť

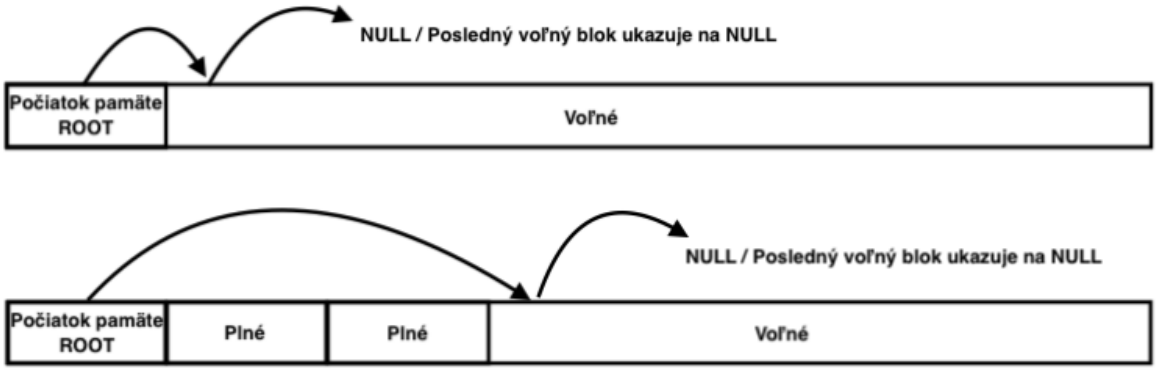
v pamäti na miesto päty Inicializacie a zapísať tam zase danú veľkosť a informáciu či je daný blok obsadený (1) alebo voľný (0). V našom prípade to bude 0 pretože daný blok je voľný. Posledný funkčný blok ktorý sa vytvorý pri memory_init je päta root ktorá je odlišná od všetkých pät ktoré v programe vytvárame. Narozdiel od ostatných obsahuje len jednu hodnotu a to 1. Táto hodnota slúži na kontrolu presahu pamäte (využíva sa pri memory_free). Štruktúre/ hlavičke root nastavíme pointer na inicializacia blok a inicializacia blok nastavíme pointer na NULL čo znamená že je to posledný voľný blok v zozname.

Funkcia memory_alloc()

Funkcia memory_alloc() nám ako prvé vytvorí dva ukazovatele na štruktúry (prve_volne, pred_prve_volne) a nasmeruje ich na počiatok pamäte. Následne ich pošle funkcii najvhodnejsi_blok(), ktorá ich nastaví (prve_volne bude ukazovať na najvhodnejší blok a funkcia pred_prve_volne bude ukazovať na blok predtým). Ďalej máme if, ktorý nám zisťuje či po alokovaní nám ostane ešte nejaké voľné miesto a následne zistujeme či toho miesta je dosť na vytvorenie voľného bloku. Ak nám voľné miesto na vytvorenie voľného bloku neostane, hlavičku a pätu voľného bloku zmeníme na hlavičku a pätu plného bloku, pred_prve_volne nastavíme aby ukazovalo na štruktúru prve_volne->po. Tým vynecháme tento blok v zozname voľných blokov (Obr. 3). Ak nám ostane dosť voľného miesta (t.j. ak sa nám tam zmestí aj veľkosť voľnej hlavičky a päty), vytvoríme novú hlavičku aj s pätou plného bloku a o to miesto posunieme voľný blok (jeho hlavičku) ktorú nastavíme (zmeníme veľkosť) a nakoniec upravíme pätu (Obr. 4).



Obr. 3. Alokácia pamäta bez vytvorenia voľného miesta



Obr. 4. Alokácia pamäta s vytvorením voľného miesta

memory_alloc nám vracia smerník na alokovanú pamäť. Pri tejto funkcii je najpodstatnejšie uvedomiť si posuny smerníkov a zapísať to na správne miesto v pamäti.

Funkcia najvhodnejsi_blok ()

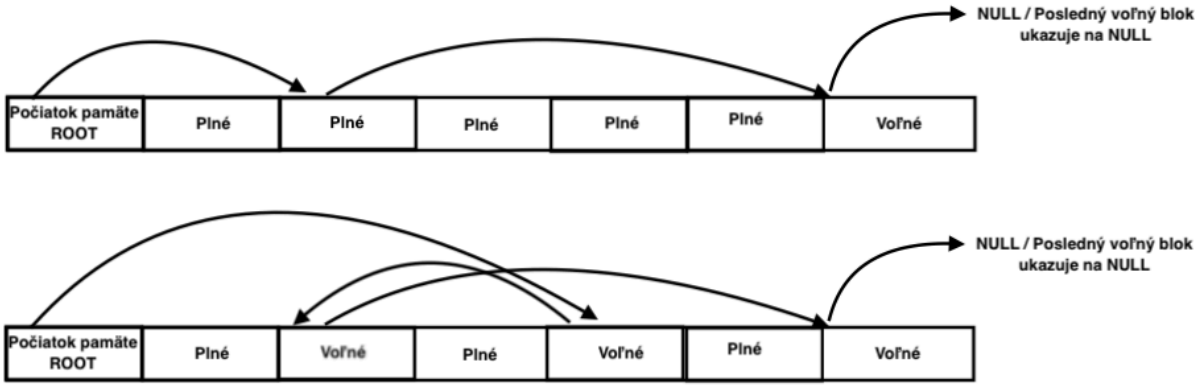
Daná funkcia vyhľadáva najvhodnejší blok pre alokovanie pamäte. Skladá sa z dvoch while. Prvý while vyhľadáva prvú vyhovujúcu hodnotu. Nasledujúci while porovnáva nami nájdenu hodnotu s ostanými a ukladá tú najvhodnejsiu. Je to síce pre program časovo zložitejšie, ale vďaka vyhľadávaniu najvhodnejšej bunky dokážeme alokovať väčšiu pamäť (menšia fragmentácia).

Funkcia memory_check ()

Memory_check nám len zistuje či je platný smerník. To porovnávame s NULL a podľa toho sa rozhodujeme

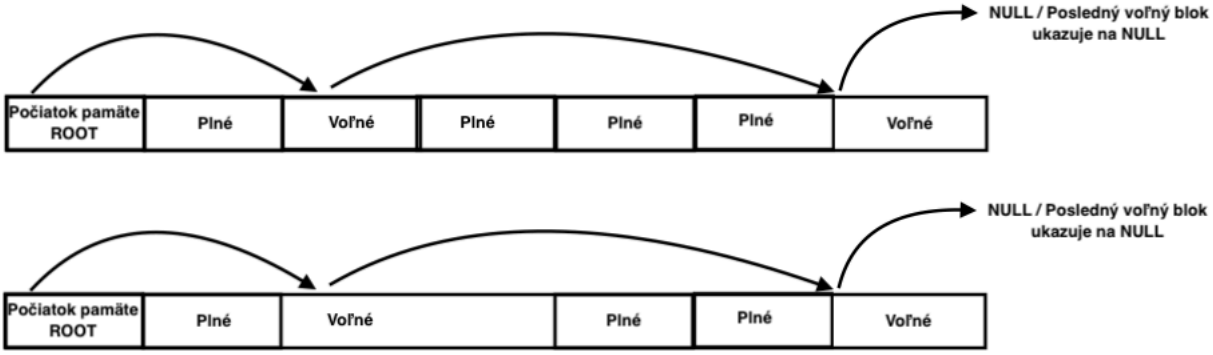
Funkcia memory_free ()

Memory_free nám uvoľnuje alokovanú pamäť no ako prvé nám skontroluje ukazovateľ pomocou memory_check. Pri uvolnovaní máme 4 možnosti. Prvá možnosť je že pamäť ktorú chceme uvoľniť vedľa seba má plné bloky alebo stoji na začiatku/ konci našej pamäte. Vtedy nám stačí vložiť do zoznamu daný voľný blok medzi root štruktúru a nasledujúcu štruktúru.



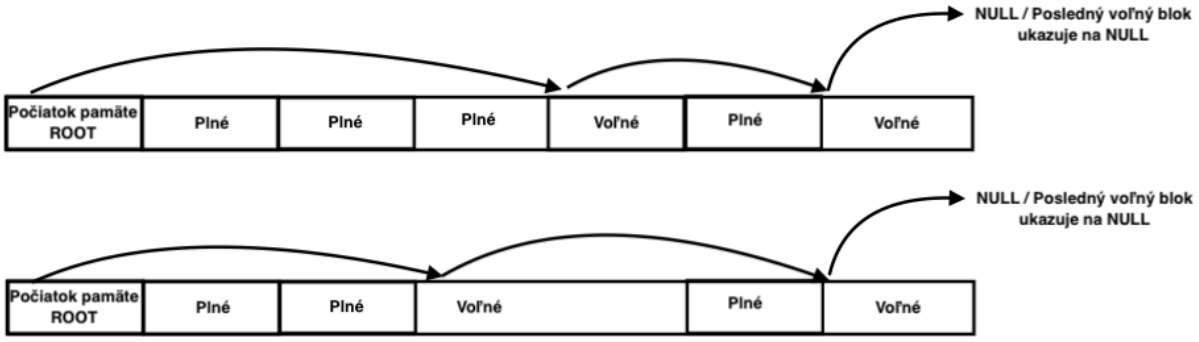
obr. 5. uvolnenie pamäte bez spájania voľného miesta

Druhá možnosť je, že pamäť ktorú chceme uvolniť má naľavo od seba voľný blok, vtedy sa posunieme ukazovateľom na ľavý voľný blok, ku tomu nám dopomôže päta (ukazovateľ na plnú pamäť - veľkosť ktorá je zapísaná v päte - veľkosť štruktúry - veľkosť päty)) a zmeníme veľkosť danej štruktury (pripočítame ku nej veľkosť plnej štruktúry ktorú uvolnujeme + jej hlavičku a pätu) a na koniec nastavíme konečnú pätu. Tým pádom nemusíme nastavovať ukazovatele, pretože sme len upravili už existujúci voľný blok (zmenili sme jeho veľkosť a pätu).



Obr. 6. uvolnovanie pamäte ak je voľný blok len na ľavo od uvolňovaného miesta

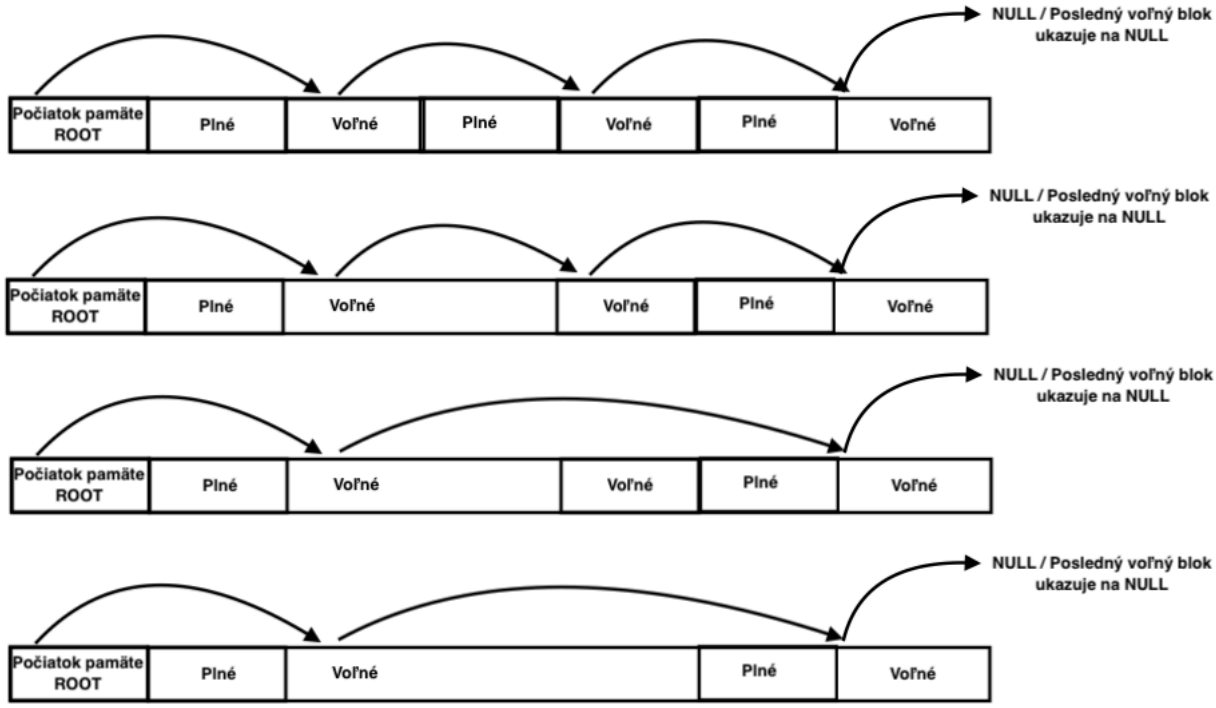
Tretia možnosť je, ak je od neho napravo voľný blok, vtedy to je podobné ako pri druhej možnosti len s tým že teraz už musíme danú štruktúru voľného bloku nie len zmeniť, ale aj posunúť na začiatok uvolnovanej pamäte.



Obr. 7. uvolnovanie pamäte ak je voľný blok na pravo od uvolňovaného miesta

Štvrtá možnosť je, ak sú voľné bloky po oboch stranách uvolňovanej pamäte , vtedy

využijeme funkcionalitu možnosti dva a jedna. Funkcionalitu možnosti dva využijeme tak, že ľavej voľnej pamäti nastavíme veľkosť, ktorá sa bude skladať z jej veľkosti, veľkosti celého uvolňovaného bloku a aj veľkost pravého voľného bloku (samozrejme počítame aj hlavičky a päty). A funkcionalitu možnosti jedna využijeme tak že vynecháme pravý voľny blok zo zoznamu voľných blokov.



Obr. 8. uvolnovanie pamäte ak sú na oboch stranách od uvolnovanej pamäte voľné bloky

To či je na ľavo alebo na pravo voľný blok zistujeme pomocou päty bloky. Každá päta (okrem päty root) obsahuje dva údaje. Prvý údaj hovorý o veľkosti dát daného bloku. Druhý údaj hovorý o obsadenosti t. j. či je obsadený daný blok (1) alebo je voľný (0).
Ako prvé sa vždy kontroluje ľavá strana. Pri nej stačí nazrieť na prvý bajt za našou hlavičkou (aktuálna hlavička bloku ktorý uvolnujeme) a zistiť či je predošlí blok 1 alebo 0 (obsadený alebo voľný blok). Ak je voľný spojíme ho, ak nie začneme riešiť pravú stranu. Pri pravej strane existuje prípad kedy sme na konci pamäte a pozeráme sa do prava či je tam voľný blok na spojenie. Pre tento prípad je vytvorená päta root. Vždy sa pozrieme na prvý bajt za našou pamäťou (myslíme tým za pätou bloku ktorý chceme uvolniť) a kontrolujeme či je 1 ak je ten prvý bajt 1 tak vieme že sme na konci našej "haldy" a ďalej nesmieme ísť, ak tam 1 nie je načítame odtial štruktúru, prečitame jej veľkosť, posunieme sa o jej veľkosť, posunieme sa o veľkosť unsigned int a tým sa dostaneme na časť päty ktorá nám hovorý o obsadenosti. Ak je blok voľný spojíme ho ak nie tak uvolníme len aktuálny blok. Náš program teda ako prvé rieši ľavú stranu následne pravú a nakoniec svoj blok.

Zložitosť

Memory_check() má výpočtovú aj priestorovú zložitosť O(1). Čo sa týka memory_init je to rovnaké. Pri memory_alloc() sa to už mení nakoľko prehľadávame celý zoznam voľných blokov, naša časová zložitosť bude O(n). Čo sa týka alokácie tak hlavička berie konštantne 16 bajtov v Turingu a päta nakoľko to je 1 unsignet int a 1 char dokopy 5 bajtov. Tým že tam máme v našej halde vždy vytvorenú počiatočnú hlavičku a pätu (táto päta ako jediná je len char takže 1 bajt) pri prvej alokáci napr. ak alokujeme 1 bajt naša pamäť bude zaberať už 39

bajtov ((root hlavička + päta = 17) + (plný blok hlavička + päta = 21) + náš jeden bajt). Každý další alokovaný priestor bude zaberať 21 bajtov plus alokované miesto preto napríklad pri alokovaní 8 bajtov dookola zasebou alokujeme len 2 krad. No čo sa týka vyžších čísel (ak alokujeme viac ako 100 bajtov) je náš algoritmu vyhodujúci a rýchly vďaka týmto hlavičkám a pätám. Čo sa týka memory_free jeho zložitosť je O(n) ak by sme chceli menšiu zložitosť museli by sme napríklad zväčšiť hlavičku o pointer nasäť (obojsmerne spájaný zoznam) tým by sme v algoritme nevyhľadávali predošlú štruktúru voľných blokov v zozname no táto voľba by nás pripravila o ďalšie miesto v pamät (hlavička by nebola 16 bajtov ale 24 bajtov). Ja som sa rozhodol pre toto riešenie v rámci úspory pamäte.

Zhodnotenie

Program memory alloc je veľmi podstatným a silným nástrojom ktorý programátor využíva. Táto funkcia má množstvo implementácií a ja som si vybral implementáciu explicitným zoznamom pomocou štruktúr s využitím nájdenia najvhodnejšieho bloku. Táto možnosť je rýchla no zaberá viac pamäte. To je problém pri menších alokáciach ale pri väčších je táto implementácia výhodou.