LU-LSP-b:L07 - PD_Heap

Agris Pudāns, st. apl. nr. ap08426

date

1. Ievads

Šajā dokumentā tiek analizēta vienkārša dinamiskās atmiņas izdalīšanas sistēma, kas izmanto NextFit algoritmu. Mērķis ir noteikt tās veiktspēju dažādos scenārijos.

Piedāvātais kods implementē vienkāršu atmiņas izdalīšanas mehānismu, izmantojot NextFit algoritmu. Tas darbojas ar fiksēta izmēra buferi, kura lielums ir 4096 baiti.

1.1. Algoritma pamatelementi

```
#define MY_BUFFER_SIZE 4096

unsigned char mybuffer[MY_BUFFER_SIZE];
size_t next_fit_index = 0;
```

1.2. Atmiņas izdalīšanas funkcija

```
void *myalloc(size_t size) {
        size_t start_index;
        size_t free_space;
        size_t alloc_start;
        size_t i;

        if (size == 0 || size > MY_BUFFER_SIZE) {
                return NULL;
        }

        start_index = next_fit_index;
        free_space = 0;

        while (free_space < size) {
                if (mybuffer[next_fit_index] == 0) {
                        free_space++;
                } else {
                        free_space = 0;
                }

                next_fit_index = (next_fit_index + 1) % MY_BUFFER_SIZE;

                if (next_fit_index == start_index) {
                        return NULL;
                }
        }

        alloc_start = (next_fit_index + MY_BUFFER_SIZE - free_space) % MY_BUFFER_SIZE;
        for (i = 0; i < size; i++) {
                mybuffer[alloc_start + i] = 1;
        }

        next_fit_index = (alloc_start + size) % MY_BUFFER_SIZE;
        return &mybuffer[alloc_start];
}
```

Atmiņas izdalīšanas funkcija `myalloc` darbojas šādi:

1. Pārbauda, vai pieprasītais izmērs ir derīgs, t.i., nav 0 un nepārsniedz bufera lielumu.
2. Sāk meklēšanu no pašreizējās `next_fit_index` pozīcijas.
3. Meklē pietiekami lielu brīvu atmiņas bloku.
4. Ja atrasts pietiekami liels bloks, tad atzīmē to kā aizņemtu un atgriež norādi uz tā sākumu.
5. Ja visa atmiņa ir pārmeklēta un pietiekami liels bloks nav atrasts, atgriež NULL.

1.3. Atmiņas atbrīvošanas funkcija

```
int myfree(void *ptr) {
        unsigned char *byte_ptr;

        if (ptr == NULL || ptr < (void *)mybuffer || ptr >= (void *)(mybuffer + MY_BUFFER_SIZE))
        ↪  {
                return -1;
        }

        byte_ptr = (unsigned char *)ptr;
        while (byte_ptr < mybuffer + MY_BUFFER_SIZE && *byte_ptr == 1) {
                *byte_ptr = 0;
                byte_ptr++;
        }

        return 0;
}
```

Atmiņas atbrīvošanas funkcija `myfree` darbojas šādi:

1. Pārbauda, vai ievades norāde ir derīga, t.i., nav NULL un atrodas bufera robežās.
2. Atbrīvo visus aizņemtos baitus sākot no norādītās adreses atzīmējot tos kā brīvus.
3. Atgriež 0, ja atbrīvošana bija veiksmīga, vai -1 kļūdas gadījumā.

2. Veiktspējas novērtējums

2.1. Testa gadījumi

Koda veiktspēja ir testēta ar šādiem testa gadījumiem, kur kā argumentu nodod atmiņas lielumu baitos:

```
int main() {
        test_case(64);
        test_case(128);
        test_case(512);
        test_case(1024);
        test_case(2048);
        test_case(3072);
        test_case(4000);
        test_case(4096);
        return 0;
}
```

Katrs testa gadījums mēra laiku, kas nepieciešams, lai izdalītu un atbrīvotu noteikta izmēra atmiņas bloku. Katram gadījumam atmiņas atbrīvošana aizņem vienādu laiku. Tāpēc tās uzskaite neietekmē ātrdarbības atšķirību novērtēšanu.

2.2. Izmantotās metrikas

Veiktspējas novērtēšanai tiek izmantoti šādi mērījumi:

- Sistēmas laiks (system time) - laiks, ko sistēma patērē I/O operācijām un citām sistēmas darbībām.
- Lietotāja laiks (user time) - laiks, ko procesors patērē, izpildot lietotāja funkcijas (mūsu kodu).
- Kopējais laiks - sistēmas un lietotāja laika summa.

Sistēmas laiks visur ir vienāds, 0s, tāpēc var uzskaitīt tikai lietotāja laiku, kas ir vienāds arī ar kopējo laiku.

2.3. Rezultāti

Iegūtie veiktspējas rezultāti:

Izmērs (baiti)	**Lietotāja laiks (s)**	**Sistēmas laiks (s)**	**Kopējais laiks (s)**
64	0,000503	0,000000	0,000503
128	0,000543	0,000000	0,000543
512	0,000580	0,000000	0,000580
1024	0,000620	0,000000	0,000620
2048	0,000666	0,000000	0,000666
3072	0,000713	0,000000	0,000713
4000	0,000766	0,000000	0,000766
4096	0,000807	0,000000	0,000807

1. tabula Atmiņas izdalīšanas un atbrīvošanas operāciju izpildes laiki

Var redzēt, ka jo lielāks atmiņas daudzums ir jārezervē, jo vairāk tiek patērēts laiks šāda bloka meklēšanai. Saskaroties ar konstantes `MY_BUFFER_SIZE` noteikto limitu, iegūstam kļūdu. Kļūda nozīmē, ka nevar izdalīt atmiņu, kas ir vienāda vai lielāka par doto konstanti.

2.4. Papildu testa gadījumi

Lai pilnvērtīgāk novērtētu atmiņas izdalītāja veiktspēju, var veikt arī šādus papildu testus:

- Fragmentācijas tests, lai novērtētu, kā algoritms tiek galā ar atmiņas fragmentāciju.
- Slodzes tests, kas novērtē algoritma veiktspēju intensīvos lietošanas apstākļos.
- Kopējais laiks - sistēmas un lietotāja laika summa.

3. Secinājumi

Analizētais dinamiskās atmiņas izdalītājs ar NextFit algoritmu nodrošina vienkāršu un saprotamu atmiņas pārvaldību. Tomēr tā veiktspēja un efektivitāte ir atkarīga no atmiņas lietošanas scenārija. Tas ir vienkāršs un viegli saprotams, bet atmiņas izmantošana ir neefektīva, jo katru baitu atzīmē atsevišķi. Tāpat teorētiskā laika sarežģītība ir atkarīga no atmiņas izmēra (n) un operāciju skaita (m):

- `myalloc`: $O(n)$ sliktākajā gadījumā (kad jāmeklē cauri visam buferim).
- `myfree`: $O(k)$, kur k ir atbrīvojamā bloka izmērs.