Mateusz Pater

Zarządzanie pamięcią - strategie przydziału pamięci

Spis treści

1. Wstęp

Gdy system operacyjny przydziela procesowi obszar pamięci, to zwykle może go przydzielić z jednego z wielu obszarów wolnej pamięci. Oczywiście obszar wolnej pamięci, z którego przydzielamy pamięć musi być wystarczająco duży. Istnieje kilka strategii wybierania, z którego obszaru przydzielić pamięć [1].

2. Algorytm First-fit

Wybierany jest pierwszy wolny obszar, który jest wystarczająco duży.

```
94  void firstFit(){
95      int aktualny_proces;
96      for( int i = 0; i < liczba_procesow; ++i ){
97          aktualny_proces = procesy[i];
98          for( int k = 0; k < liczba_blokow; ++k){
99              if(pamiec[k] == -1 ){
100                 if( bloki[k] >= aktualny_proces )
101                 {
102                     pamiec[k] = i;
103                     break;
104                 }
105             }
106         }
107     }
108 }
109
```

Rysunek 1. Implementacja algorytmu First-fit

Jest to najprostszy spośród wszystkich algorytm jeżeli chodzi o jego implementację. Dla każdego procesu (wiersz 96) przeglądamy wszystkie bloki (wiersz 98), a następnie wybieramy pierwszy wolny, którego wielkość spełnia potrzeby naszego procesu (wiersze 99-102).

```
------------------------------------------------------------
                FIRST FIT
BLOK    PROCES  PAMIEC PROCESU  PAMIEC BLOKU    FRAGMENTACJA
0       P1      3               5               2
1       P2      1               7               6
2       P3      7               9               2
3       P4      10              11              1
4       null    0               13              13

        Utracona pamiec: 24
```

Rysunek 2. Przykładowy wynik działania algorytmu First-fit

3. Algorytm Best-fit

Wybieramy najmniejszy wolny obszar, który jest wystarczająco duży. Pomysł, który się kryje za tą strategią jest następujący: Wolny obszar jest zwykle większy niż przydzielany obszar, więc po przydzieleniu część wolnego obszaru pozostaje wolna. W przypadku strategii best-fit, taka "końcówka"jest możliwie najmniejsza. Jeśli w przyszłości nie wykorzystamy jej, to straty z powodu fragmentacji zewnętrznej będą możliwie małe.

```
110  void bestFit(){
111      int aktualny_proces;
112      for( int i = 0; i < liczba_procesow; ++i ){
113          aktualny_proces = procesy[i];
114          int max;
115
116          for( int k = 0; k < liczba_blokow; ++k){
117              if(pamiec[k] == -1 && bloki[k] >= aktualny_proces){
118                  max = bloki[k];
119                  break;
120              }
121          }
122
123          for( int k = 0; k < liczba_blokow; ++k){
124              if(pamiec[k] == -1 && bloki[k] >= aktualny_proces){
125                  if( max > bloki[k])
126                      max = bloki[k];
127              }
128          }
129
130          for( int k = 0; k < liczba_blokow; ++k){
131              if(pamiec[k] == -1 ){
132                  if( bloki[k] == max && bloki[k] - aktualny_proces >= 0)
133                  {
134                      pamiec[k] = i;
135                      break;
136                  }
137              }
138          }
139      }
140  }
```

Rysunek 3. Implementacja algorytmu Best-fit

Dla każdego procesu (wiersz 112) znajdujemy najmniejszy wolny obszar, którego wielkość jest wystarczająca (wiersze 116-128). Następnie przeglądając wszystkie bloki (wiersz 130) znajdujemy wolne miejsce, które możemy zagospodarować na nasz proces (wiersze 131-134).

```
-------------------------------------------------------------
                BEST FIT
BLOK    PROCES  PAMIEC PROCESU  PAMIEC BLOKU    FRAGMENTACJA
0       P1      3               5               2
1       P2      1               7               6
2       P3      7               9               2
3       P4      10              11              1
4       null    0               13              13

        Utracona pamiec: 24
```

Rysunek 4. Przykładowy wynik działania algorytmu Best-fit

4. Algorytm Worst-fit

Przydzielamy pamięć zawsze z największego wolnego obszaru (oczywiście, o ile jest on wystarczająco duży). W przypadku tej strategii, część obszaru, która pozostaje wolna jest możliwie jak największa. Jest więc szansa, że będzie można ją jeszcze wykorzystać bez konieczności uzwracenia.

```
142  void worstFit(){
143      int aktualny_proces;
144
145      for( int i = 0; i < liczba_procesow; ++i ){
146          aktualny_proces = procesy[i];
147
148          int max;
149
150          for( int k = 0; k < liczba_blokow; ++k){
151              if(pamiec[k] == -1 && bloki[k] >= aktualny_proces){
152                  max = bloki[k];
153                  break;
154              }
155          }
156
157          for( int k = 0; k < liczba_blokow; ++k){
158              if(pamiec[k] == -1 && bloki[k] >= aktualny_proces){
159                  if( max < bloki[k] )
160                      max = bloki[k];
161              }
162          }
163
164          for( int k = 0; k < liczba_blokow; ++k){
165              if(pamiec[k] == -1 ){
166                  if( bloki[k] == max && bloki[k] - aktualny_proces >= 0)
167                  {
168                      pamiec[k] = i;
169                      break;
170                  }
171              }
```

Rysunek 5. Implementacja algorytmu Worst-fit

Odwrotnie do algorytmu Best-fit. Dla każdego procesu (wiersz 145) znajdujemy blok z *największą* ilością wolnego obszaru. (wiersze 150-162). Następnie przeglądając wszystkie bloki (wiersz 164) znajdujemy wolne miejsce, które możemy zagospodarować na nasz proces (wiersze 165-169).

```
-------------------------------------------------------------
                WORST FIT
BLOK    PROCES  PAMIEC PROCESU  PAMIEC BLOKU    FRAGMENTACJA
0       null    0               5               5
1       null    0               7               7
2       P3      7               9               2
3       P2      1               11              10
4       P1      3               13              10

        Utracona pamiec: 34
```

Rysunek 6. Przykładowy wynik działania algorytmu Worst-fit

5. Kompilowanie i testowanie

Przed uruchomieniem programu należy go skompilować poleceniem:

```
$ gcc main.c
```

Uruchomienie:

```
$ .\a.out
```

Na wejściu powinniśmy wpisać liczbę bloków, liczbę procesów oraz pamięć kolejnych bloków i procesów biorących udział w symulacji przydziału pamięci.

```
Liczba bloków   5

Liczba procesow 5

Pamiec kolejnych blokow:
Block [1]:      5
Block [2]:      7
Block [3]:      9
Block [4]:      11
Block [5]:      13
Pamiec kolejnych procesow:
Proces [1]:     3
Proces [2]:     1
Proces [3]:     7
Proces [4]:     10
Proces [5]:     15
```

Rysunek 7. Wymagane parametry startowe

```
------------------------------------------------------------
                BEST FIT
BLOK    PROCES  PAMIEC PROCESU  PAMIEC BLOKU    FRAGMENTACJA
0       P1      3               5               2
1       P2      1               7               6
2       P3      7               9               2
3       P4      10              11              1
4       null    0               13              13

        Utracona pamiec: 24
------------------------------------------------------------
                WORST FIT
BLOK    PROCES  PAMIEC PROCESU  PAMIEC BLOKU    FRAGMENTACJA
0       null    0               5               5
1       null    0               7               7
2       P3      7               9               2
3       P2      1               11              10
4       P1      3               13              10

        Utracona pamiec: 34
------------------------------------------------------------
                FIRST FIT
BLOK    PROCES  PAMIEC PROCESU  PAMIEC BLOKU    FRAGMENTACJA
0       P1      3               5               2
1       P2      1               7               6
2       P3      7               9               2
3       P4      10              11              1
4       null    0               13              13

        Utracona pamiec: 24
```

Rysunek 8. Przykładowy wynik

5.1. Analiza wyników

Wynikiem działania programu jest tabela ze spisem wszystkich bloków pamięci. Jeżeli blok pamięci jest pusty to kolumna **PROCES** ma wartość *null*.

Opis nagłówka tabeli:

- **BLOK** - liczba porządkowa bloku pamięci,
- **PROCES** - identyfikator przydzielonego procesu,
- **PAMIĘĆ PROCESU** - wymagana pamięć dla danego procesu,
- **PAMIĘĆ BLOKU** - dostępna pamięć bloku,
- **FRAGMENTACJA** - wynik fragmentacji wewnętrznej.

Utracona pamięć to suma pamięci zajętej w wyniku fragmentacji wewnętrznej.

Literatura

[1] Zarządzanie pamięcią, edu.pjwstk.edu.pl *http://edu.pjwstk.edu.pl/wyklady/sop/scb/wyklad7/*