**ЛАБОРАТОРНА РОБОТА №1-4**

З дисципліни «Комп’ютерні системи»

Тема: «Планування задач в багатопроцесорних системах»

|  |  |  |
| --- | --- | --- |
| **Виконали**  Студенти групи КВ-21  Комарницький О.Б.  Савченко В.Г.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | **Перевірила**  Доцент кефедри СКС  Потапова К.Р.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |

Опис алгоритмів

* **Алгоритм FIFO (First In First Out)**

Код планування задач:

private async void FIFO\_Mode()

{

while (\_fileOutput.d\_MODE == 1)

{

if (AllTasks.Count < 1)

{

await Task.Delay(1);

continue;

}

int possibleProcSize = AllTasks[0].getRequiredProcessors().Count;

for(int i=0; i < possibleProcSize; i++)

{

int i\_th\_processor = AllTasks[0].getRequiredProcessors()[i];

if (!Processors[i\_th\_processor].isBusy())

{

Processors[i\_th\_processor].setTask(AllTasks[0]);

AddedTasksBox.AppendText(" Processor " + Convert.ToInt16(i\_th\_processor+1) + " taken complexity=" + AllTasks[0].getOperationAmount() + " at " + DateTime.Now.ToString("hh:mm:ss:ms") + Environment.NewLine);

AllTasks.RemoveAt(0);

Processors[i\_th\_processor].executeTask();

break;

}

}

await Task.Delay(1);

}

}

Суть алгоритму:

FIFO (First In First Out) – найпростіший алгоритм розподілу задач. Його недолік – низька продуктивність через те, що дві задачі (або більше), котрі можуть бути реалізовані тільки на одному процесорі, можуть змусити чекати решту процесорів

Пояснення до коду:

*if (AllTasks.Count < 1)*Спочатку перевіряємо буфер задач на пустоту, якщо немає задач – то нічого не робимо. Якщо знаходимо – розпочинаємо планування:

*possibleProcSize = AllTasks[0].getRequiredProcessors().Count* ***–*** визначаємо кількість процесорів, що можуть виконати задачу, або потім циклічно по кожному пройтися.

*i\_th\_processor = AllTasks[0].getRequiredProcessors()[i]* ***-*** циклічно проходимося по всім процесорам, що можуть виконати це завдання

*if (!Processors[i\_th\_processor].isBusy())*– якщо даний процесор не зайнятий, то

віддаємо йому завдання, видаляємо завдання із списку і завершуємо планування:

*Processors[i\_th\_processor].setTask(AllTasks[0]);*

*AllTasks.RemoveAt(0);*

*Processors[i\_th\_processor].executeTask();*

*break;*

Блок-схема алгоритму FIFO

Start

await Task.Delay(1)

i++

!Processors[i\_th\_processor].isBusy()

i\_th\_processor = AllTasks[0].getRequiredProcessors()[i]

possibleProcSize = AllTasks[0].getRequiredProcessors().Count

i = 0;

AllTasks.Count < 1

\_fileOutput.d\_MODE == 1

End

Processors[i\_th\_processor].setTask(AllTasks[0]);

AllTasks.RemoveAt(0); Processors[i\_th\_processor].executeTask();

0

0

1

1

0

0

1

1

1

0

i < possibleProcSize

Результати роботи алгоритму FIFO

1. Проаналізуємо випадки рівносильних процесорів. Зробимо кілька вимірів за однакових умов, адже цей спосіб планування дуже залежить від списку можливих процесорів кожної задачі, що приходить у чергу та є суто випадковою величиною.

Візьмемо продуктивність Pi=20. Тобто кожна задача виконується на будь-якому процесорі протягом 4000 / 20 = 200 (мс), що відповідає максимально допустимому часу виконання однієї задачі.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,648 | 0,648 | 20 | 20 | 20 | 20 | 20 | 162 | 648000 |
| 4000 | 4000 | 0,628 | 0,628 | 20 | 20 | 20 | 20 | 20 | 157 | 628000 |
| 4000 | 4000 | 0,656 | 0,656 | 20 | 20 | 20 | 20 | 20 | 164 | 656000 |

Візьмемо продуктивність Pi=40. Тепер кожна задача виконується на будь-якому процесорі протягом 4000 / 40 = 100 (мс).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,524 | 0,524 | 40 | 40 | 40 | 40 | 40 | 262 | 1048000 |
| 4000 | 4000 | 0,534 | 0,534 | 40 | 40 | 40 | 40 | 40 | 267 | 1068000 |
| 4000 | 4000 | 0,566 | 0,566 | 40 | 40 | 40 | 40 | 40 | 283 | 1132000 |

Висновок: при різній продуктивності системи ми отримали різні ККД : чим більша продуктивність системи – тим менша ККД виконання, а саме:

при Pi=20 середнє арифметичне ККД (η) дорівнює 0.644, при Pi=40 середнє арифметичне ККД (η) становить 0.541.

Це пояснюється тим, що чим довше процесори виконують задачі, тим менше часу виділяється на алгоритмічну частину розподілу задач (циклічний прохід, пошук вільних процесорів, присвоєння завдання конкретному процесору). Під час розподілу процесори знаходяться в режимі очікування та не виконують задачі, які могли б виконати за цей час.

З іншого боку, важливо відзначити моменти застою процесорів, коли приходять дві задачі, які можуть бути виконаними одним процесором. У цьому випадку, чим довше процесори працюють над задачами, тим більше часу вони будуть стояти. Тому доречно підрахувати кількість таких випадків у роботі системи.

Імовірність появи двох задач із одним можливим процесором становить

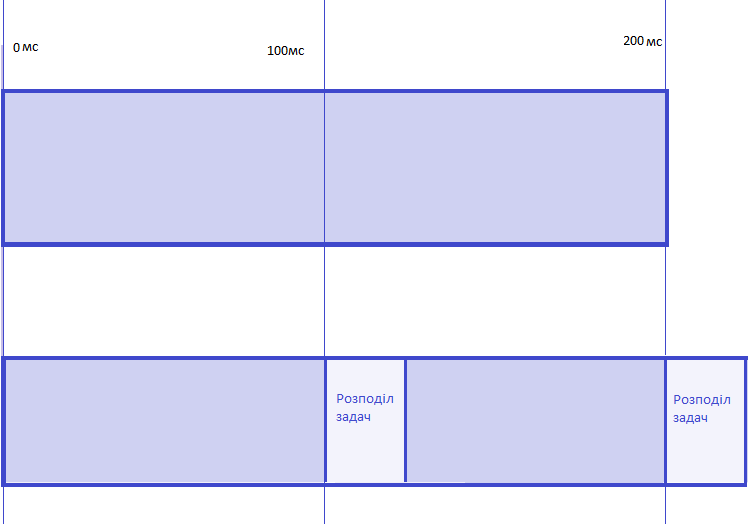
0.2 \* 0.2 \* 0.2 = 0.008

Тобто кожна 125-а задача дасть затримку при Pi=40 більшу на 100мс у порівнянні із Pi=20

Враховуючи те, що всього за 10с виконуються близько 250 задач (як показують досліди), то в середньому відбуваються лише два застої процесори за цей час, які не сильно впливають на загальний коефіцієнт корисної дії.

Вище наведені результати значно більше залежать від іншого фактору – розподілу задач в алгоритмі, адже цей процес відбувається значно частіше, аніж застій процесора.

Умовно можна зобразити дану ситуацію ось так:

На верхній частині рисунку зображене виконання процесором задачі протягом 200мс без переключення на розподіл чи присвоєння задачі.

В нижній частині рисунку зображене виконання процесором задач протягом 100мс.

Враховуючи високу частоту появи таких зон, очевидно, що при Pi=40 ККД буде нижчим за ККД при Pi=20

1. Проаналізуємо випадки роботи системи, в якій процесори мають різні продуктивності, але системи рівносильні (сума продуктивності процесорів однакова).

Зробимо кілька вимірів за однакових умов, адже цей спосіб планування дуже залежить від списку можливих процесорів кожної задачі, що приходить у чергу та є суто випадковою величиною

Розглянемо 2 випадки, в кожному досліді загальна сума продуктивності однакова = 250.

1-ий випадок

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,473 | 0,473 | 30 | 40 | 40 | 40 | 100 | 296 | 1184000 |
| 4000 | 4000 | 0,454 | 0,454 | 30 | 40 | 40 | 40 | 100 | 284 | 1136000 |
| 4000 | 4000 | 0,467 | 0,467 | 30 | 40 | 40 | 40 | 100 | 292 | 1168000 |

2-ий випадок

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,488 | 0,488 | 50 | 50 | 50 | 50 | 50 | 305 | 1220000 |
| 4000 | 4000 | 0,500 | 0,500 | 50 | 50 | 50 | 50 | 50 | 313 | 1252000 |
| 4000 | 4000 | 0,497 | 0,497 | 50 | 50 | 50 | 50 | 50 | 311 | 1244000 |

Висновок: при однаковій сумі продуктивності процесорів ми отримали такі показники :

в першому досліді середнє арифметичне ККД (η) дорівнює 0.464, в другому - 0.495.

Результати виправдані, адже системи мають однакову сумарну продуктивність, алгоритм виконання не залежить від величини окремого процесора – всі процесори працюють в одному режимі, на відміну від алгоритму із окремим процесором-планувальником.

Різниця в 3% пояснюється випадковим підбором списку можливих процесорів у випадку застою . Як було вияснено раніше, застій в середньому відбувається раз в 125 задач, і в другому досліді принципово важливо на якому процесорі відбувається застій. Саме це і пояснює доволі невелику різницю продуктивності систем.

* **Алгоритм із слабким процесором-планувальником**

Код планування задач:

private async void WeakPlanner\_Mode()

{

int index = \_fileOutput.d\_weakestId; // Знаходимо найслабший

while (\_fileOutput.d\_MODE == 2)

{

if (AllTasks.Count < 1) // Перевірка чи є задачі в черзі

{

await Task.Delay(1);

continue;

}

for (int i = 0; i < 5; i++)

{

if (i == index || Processors[i].isBusy()) // Якщо це найслабший процесор, або даний процесор зайнятий,

continue; // то пропускаємо планування задач для нього

int j = 0;

while (AllTasks.Count > j && !AllTasks[j].getRequiredProcessors().Contains(i)) // Шукаємо завдання для кожного процесора

j++;

if(AllTasks.Count > j) // Якщо j в правильних межах

{

Processors[i].setTask(AllTasks[j]);

AddedTasksBox.AppendText(" Processor " + Convert.ToInt16(i+1) + " taken complexity=" +

AllTasks[0].getOperationAmount() + " at " + DateTime.Now.ToString("hh:mm:ss:ms") + Environment.NewLine);

AllTasks.RemoveAt(j);

Processors[i].executeTask();

}

}

await Task.Delay(1);

}

}

Суть алгоритму:

*Алгоритм із окремим планувальником*. В цьому випадку розподілом ресурсів у системі займається окремий процесор, що не бере участі в обчисленні. Так як алгоритм планування не надто складний, для цього доцільно обрати найслабший (із найменшою продуктивністю) процесор. Але в цьому випадку система втрачає один процесор із точки зору продуктивності. Щоб цього уникнути, необхідно додати до системи додатковий процесор-планувальник. Але це негативно вплине на надійність системи, так як його відмова призведе до відмови цілої системи.

Пояснення до коду:

*int index = \_fileOutput.d\_weakestId* – спочатку знаходимо найслабший процесор.

*\_fileOutput –* структура, де зберігаються основні дані для аналізу

*if (AllTasks.Count < 1)* – дивимось чи є хоч одна задача у черзі.

*for (int i = 0; i < 5; i++)* – циклом проходимось для кожного процесора.

*if (i == index || Processors[i].isBusy())* – якщо це планувальник, або даний процесор є зайнятим, то переходимо на наступну ітерацію циклу

у черзі задач знаходимо ту, яку може виконати даний процесор. Тобто j – індекс задачі в черзі, яку може виконати цей процесор:

*int j = 0;*

*while (AllTasks.Count > j && !AllTasks[j].getRequiredProcessors().Contains(i))*

*j++;*

Якщо j – існуючий індекс, то призначаємо даному процесору задачу із цим номером у списку задач, видаляємо цю задачу із списку, та запускаємо процесор на виконання заданої йому задачі:

*if(AllTasks.Count > j) {*

*Processors[i].setTask(AllTasks[j]);*

*AllTasks.RemoveAt(j);*

*Processors[i].executeTask();*

*}*

Блок-схема алгоритму із слабким планувальником

index = \_fileOutput.d\_weakestId

1

1

1

1

1

1

0

0

0

0

0

0

i++;

i < 5

Processors[i].setTask(AllTasks[j]);

AllTasks.RemoveAt(j);

Processors[i].executeTask();

AllTasks.Count > j

j++;

AllTasks.Count > j && !AllTasks[j].getRequiredProcessors().Contains(i)

j = 0;

i == index || Processors[i].isBusy())

Start

await Task.Delay(1)

i = 0;

AllTasks.Count < 1

\_fileOutput.d\_MODE == 3

End

Результати роботи алгоритму із слабким планувальником

1. Проаналізуємо випадки рівносильних процесорів. Проведемо 3 виміри для кожного випадку щоб обчислити їх середнє арифметичне ККД . Візьмемо в двох експериментах різну сумарну продуктивність

В першому випадку візьмемо продуктивності Pi=20

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,752 | 0,94 | 20 | 20 | 20 | 20 | 20 | 188 | 752000 |
| 4000 | 4000 | 0,748 | 0,935 | 20 | 20 | 20 | 20 | 20 | 187 | 748000 |
| 4000 | 4000 | 0,752 | 0,94 | 20 | 20 | 20 | 20 | 20 | 188 | 752000 |

В другому випадку візьмемо продуктивності Pi=50

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,604 | 0,756 | 50 | 50 | 50 | 50 | 50 | 378 | 1512000 |
| 4000 | 4000 | 0,609 | 0,762 | 50 | 50 | 50 | 50 | 50 | 381 | 1524000 |
| 4000 | 4000 | 0,609 | 0,762 | 50 | 50 | 50 | 50 | 50 | 381 | 1524000 |

Висновок: при однаковій продуктивності процесорів ми отримали такі показники :

|  |  |
| --- | --- |
| Теор. (η) | Реал. (η) |
| 1-ий випадок | 0.75 | 0.938 |
| 2-ий випадок | 0.607 | 0.76 |

Як і в алгоритмі FIFO ми отримали різні показники ККД у випадку рівносильних процесорів. Причина та ж сама – при більшій продуктивності процесори частіше задіяні в організації розподілу роботи, який забирає час, що процесор міг би використати для обчислення операцій.

До того ж, тут процесори не простоюють. Це значить, що при більшому часу виконання операцій час простою не збільшиться.

1. Проаналізуємо випадки роботи системи, в якій процесори мають різні продуктивності, але системи рівносильні (сума продуктивності процесорів однакова).

Зробимо кілька вимірів за однакових умов, адже цей спосіб планування дуже залежить від списку можливих процесорів кожної задачі, що приходить у чергу та є суто випадковою величиною

Розглянемо випадки із продуктивністю систем по 200 од.

1-ий випадок

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,712 | 0,791 | 20 | 30 | 40 | 50 | 60 | 356 | 1424000 |
| 4000 | 4000 | 0,706 | 0,784 | 20 | 30 | 40 | 50 | 60 | 353 | 1412000 |
| 4000 | 4000 | 0,71 | 0,788 | 20 | 30 | 40 | 50 | 60 | 355 | 1420000 |

2-ий випадок

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,662 | 0,827 | 40 | 40 | 40 | 40 | 40 | 331 | 1324000 |
| 4000 | 4000 | 0,656 | 0,82 | 40 | 40 | 40 | 40 | 40 | 328 | 1312000 |
| 4000 | 4000 | 0,656 | 0,82 | 40 | 40 | 40 | 40 | 40 | 0,82 | 1312000 |

Висновок: при однаковій продуктивності процесорів ми отримали такі показники :

|  |  |
| --- | --- |
| Теор. (η) | Реал. (η) |
| 1-ий випадок | 0.709 | 0.787 |
| 2-ий випадок | 0.658 | 0.822 |

Ми отримали відмінність 5% . На відміну від алгоритму FIFO, тут значення продуктивності кожного окремого процесора має значення, а саме значення мінімальної продуктивності в системі. В першому випадку ми отримали кращі показники тому, що на планування ми віддали слабший процесор (P1=20), в другому – слабший процесор P1=40. Тобто, в теорії процесор P1=40 зміг би виконати значно більше операцій ніж P1=20, тому і втрата теоретичного ККД доволі помітна.

А реальний ККД у другому випадку навіть кращий ніж в першому. Це пояснюється врахуванням у формулі реального ККД факту того, що процесор-планувальник не виконує обчислень

* **Алгоритм із сильним процесором-планувальником**

Код планування задач:

private async void PowerFulPlanner\_Mode(int plannerTimeForTasks)

{

int index = \_fileOutput.d\_strongestId; // Індекс найсильнішого процесора

while (\_fileOutput.d\_MODE == 3 || \_fileOutput.d\_MODE == 4)

{

if (AllTasks.Count < 1 || Processors[index].isBusy()) // Якщо є задачі і якщо планувальник звільнився після виконання

{

await Task.Delay(1);

continue;

}

for (int i = 0; i < 5; i++)

{

if (Processors[i].isBusy()) // Якщо даний процесор зайнятий, то йому нічого не плануємо

continue;

int j = 0;

while (AllTasks.Count > j && !AllTasks[j].getRequiredProcessors().Contains(i))

j++;

if(AllTasks.Count > j) // Якщо знайшли

{

if (i == index) // якщо задача адресована планувальнику

{

if (Processors[index].hasNoFinishedTasks()) // Якщо процесор-планувальник має незавершене своє завдання

Processors[index].executeTask(plannerTimeForTasks);

else

{

Processors[index].setTask(AllTasks[j]);

AllTasks.RemoveAt(j);

Processors[index].executeTask(plannerTimeForTasks);

}

}

else // якщо задача адресована НЕ планувальнику

{

Processors[i].setTask(AllTasks[j]);

AddedTasksBox.AppendText(" Processor " + Convert.ToInt16(i+1) + " taken complexity=" + AllTasks[0].getOperationAmount() +

" at " + DateTime.Now.ToString("hh:mm:ss:ms") + Environment.NewLine);

AllTasks.RemoveAt(j);

Processors[i].executeTask();

}

}

}

await Task.Delay(4); // 4мс відбувається планування

}

}

Суть алгоритму:

*Алгоритм із сильним планувальником*. Найбільш правильним є присвоєння функції планування на найсильніший (найбільш продуктивний) процесор, що періодично здійснює переривання для керування чергою задач. В цьому випадку можна вважати, що час для керування чергою становить 4мс.

Пояснення до коду:

*int index = \_fileOutput.d\_strongestId* – спочатку знаходимо найсильніший процесор.

*\_fileOutput –* структура, де зберігаються основні дані для аналізу

*if (AllTasks.Count < 1 || Processors[index].isBusy())* – умова виходу: якщо немає задач у черзі або процесор-планувальник зайнятий

*for (int i = 0; i < 5; i++)* – циклом проходимось для кожного процесора.

*if (Processors[i].isBusy())* - якщо даний процесор зайнятий, то нічого йому не плануємо

у черзі задач знаходимо ту, яку може виконати даний процесор. Тобто j – індекс задачі в черзі, яку може виконати цей процесор:

*int j = 0;*

*while (AllTasks.Count > j && !AllTasks[j].getRequiredProcessors().Contains(i))*

*j++;*

Якщо j – існуючий індекс, то далі дивимось чи задача адресована планувальнику:

*if (i == index)*

Якщо планувальник має незавершену задачу, то нехай дороблює його:

*if (Processors[index].hasNoFinishedTasks()) Processors[index].executeTask(plannerTimeForTasks);*

В іншому випадку (всі попередні задачі він завершив) призначаємо задачу, запускаємо процесор на виконання впродовж певного проміжку часу, видаляємо задачу із черги задач:

*Processors[index].setTask(AllTasks[j]);*

*AllTasks.RemoveAt(j);*

*Processors[index].executeTask(plannerTimeForTasks);*

Якщо задача адресована не планувальнику, призначаємо задачу, запускаємо процесор, видаляємо задачу із черги задач:

*Processors[i].setTask(AllTasks[j]);*

*AllTasks.RemoveAt(j);*

*Processors[i].executeTask();*

Блок-схема алгоритму із сильним планувальником

1

0

0

1

Processors[index].hasNoFinishedTasks()

Start

Processors[i].setTask(AllTasks[j]); AllTasks.RemoveAt(j);

Processors[i].executeTask();

i == index

1

0

0

AllTasks.Count > j

j++;

AllTasks.Count > j && !AllTasks[j].getRequiredProcessors().Contains(i)

1

j = 0;

0

1

Processors[i].isBusy()

Processors[index].setTask(AllTasks[j]);

AllTasks.RemoveAt(j); Processors[index].executeTask(

plannerTimeForTasks);

Processors[index].executeTask(

plannerTimeForTasks);

1

0

i++;

i < 5

i = 0;

index = \_fileOutput.d\_strongestId

await Task.Delay(4)

\_fileOutput.d\_MODE == 3

0

End

1

0

1

AllTasks.Count < 1 || Processors[index].isBusy()

Результати роботи алгоритму із сильним планувальником

1. Проаналізуємо випадки рівносильних процесорів. Проведемо 3 виміри для кожного випадку щоб обчислити їх середнє арифметичне ККД . Візьмемо в двох експериментах різну сумарну продуктивність

В першому випадку візьмемо продуктивності Pi=20

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,815 | 0,849 | 20 | 20 | 20 | 20 | 20 | 203 | 815600 |
| 4000 | 4000 | 0,804 | 0,837 | 20 | 20 | 20 | 20 | 20 | 201 | 804000 |
| 4000 | 4000 | 0,813 | 0,847 | 20 | 20 | 20 | 20 | 20 | 203 | 813200 |

В другому випадку візьмемо продуктивності Pi=40

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,714 | 0,739 | 40 | 40 | 40 | 40 | 40 | 357 | 1428000 |
| 4000 | 4000 | 0,704 | 0,729 | 40 | 40 | 40 | 40 | 40 | 352 | 1408000 |
| 4000 | 4000 | 0,702 | 0,727 | 40 | 40 | 40 | 40 | 40 | 351 | 1404800 |

Висновок: при однаковій продуктивності процесорів (різній продуктивності систем) ми отримали такі показники :

|  |  |
| --- | --- |
| Теор. (η) | Реал. (η) |
| 1-ий випадок | 0.810 | 0.843 |
| 2-ий випадок | 0.706 | 0.732 |

Як і в алгоритмі FIFO і в алгоритмі із слабким планувальником, ми отримали помітну відмінність 10% між першим і другим випадком. Це знову пояснюється частотою появи періодів розподілу задач. Алгоритм розподілу задач в цьому випадку значно складніший, аніж в попередніх способах.

Крім того, варто відзначити доволі низький як для найсильнішого алгоритму ККД. Це пояснюється втратою ККД у перший момент виконання досліду – адже є момент, коли задачі ще не встигли прийти в буфер, і відповідно, планувальник не розподіляє задачі в цей момент. Для аргументації цього приведемо ще одну таблицю, в якій подані результати послідовно після 10с, 20с, 30с, адже з часом ККД повинен вирівнятися після втрати швидкодії на перших етапах:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,808 | 0,842 | 20 | 20 | 20 | 20 | 20 | 202 | 808400 |
| 4000 | 4000 | 0,810 | 0,843 | 20 | 20 | 20 | 20 | 20 | 405 | 1620400 |
| 4000 | 4000 | 0,815 | 0,849 | 20 | 20 | 20 | 20 | 20 | 611 | 2447600 |

1. Проаналізуємо випадки роботи системи, в якій процесори мають різні продуктивності, але системи рівносильні (сума продуктивності процесорів однакова).

Проведемо 3 виміри для кожного випадку щоб обчислити їх середнє арифметичне ККД

Розглянемо випадки із продуктивністю систем по 150 од.

1-ий випадок

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,739 | 0,764 | 30 | 30 | 30 | 30 | 30 | 277 | 1109200 |
| 4000 | 4000 | 0,759 | 0,786 | 30 | 30 | 30 | 30 | 30 | 284 | 1139000 |
| 4000 | 4000 | 0,743 | 0,769 | 30 | 30 | 30 | 30 | 30 | 278 | 1115600 |

2-ий випадок

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,727 | 0,763 | 20 | 25 | 30 | 35 | 40 | 272 | 1091200 |
| 4000 | 4000 | 0,728 | 0,764 | 20 | 25 | 30 | 35 | 40 | 273 | 1092000 |
| 4000 | 4000 | 0,741 | 0,777 | 20 | 25 | 30 | 35 | 40 | 277 | 1111200 |

Висновок: при однаковій продуктивності систем ми отримали такі показники :

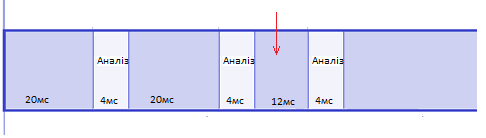
|  |  |
| --- | --- |
| Теор. (η) | Реал. (η) |
| 1-ий випадок | 0.747 | 0.773 |
| 2-ий випадок | 0.732 | 0.768 |

Ми отримали відмінність ККД між першим і другим випадком 1.5% .

Схожість ККД пояснюється тим, що неважливо, яке значення продуктивності процесора планувальника, адже він працює рівно 20мс незалежно від своєї продуктивності, тобто чим повільніше в режимі роботи працює планувальник, тим швидше працює звичайний процесор (згідно до критеріїв аналізу даної ситуації – загальна продуктивність має бути однаковою).

Відмінність 1.5% можна пояснити лише нашою реалізацією способу (ми моделюємо планування – await Task.Delay(time)). Згідно із алгоритмом, 4мс виділяється на планування. В першому випадку цим займається процесор P1=30, в другому P1=40. Тобто в другому випадку процесор із більшою продуктивністю не виконує обчислення, що дає більш помітну втрату теоретичного (η). В реальних комп’ютерних системах такого не спостерігатиметься! Оскільки ми працюємо із моделюванням, тому у нас відмінність помітна.

Також, важливою особливістю реалізації алгоритму із сильним процесором є така ситуація, показана на рисунку: стрілкою позначений момент, коли процесор виконав своє завдання раніше та переходить до планування. Оскільки в обох дослідах у нас різні планувальники, то імовірність появи таких ситуацій буде різною. Підрахуємо приблизну кількість таких появ за 10с в першому випадку:



Час виконання однієї задачі в першому випадку: 4000 / 30 = 166.6 (мс)

Кількість підходів процесора-планувальника: 166.6 / 20 = 8.33 ~ 9

Час виконання планувальником однієї задачі із врахуванням переривань:

166.6 + 9 \* 4 = 202.6

Тобто кожні 202.6мс спостерігаємо дану ситуацію (10 000/202.6) ~ 50 разів за 10с

Результати роботи алгоритму із сильним планувальником (свій час планування)

Оскільки особливості алгоритму із сильним планувальником вже розглянуті, тому лише покажемо значення ККД , які вдалося отримати тестами та пояснимо отримані результати.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,846 | 0,872 | 20 | 20 | 20 | 20 | 20 | 211 | 846580 |
| 4000 | 4000 | 0,763 | 0,784 | 30 | 30 | 30 | 30 | 30 | 286 | 1145290 |

Час роботи = 43 мс:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,807 | 0,847 | 20 | 20 | 20 | 20 | 20 | 201 | 807700 |
| 4000 | 4000 | 0,780 | 0,801 | 30 | 30 | 30 | 30 | 30 | 292 | 1170220 |

Час роботи = 37 мс:

Час роботи = 50 мс:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Найменша складність | Найбільша складність | Теор.  (η) | Реал.  (η) | P1 | P2 | P3 | P4 | P5 | Викон.  задачі | Викон.  операції |
| 4000 | 4000 | 0,838 | 0,863 | 20 | 20 | 20 | 20 | 20 | 209 | 838000 |
| 4000 | 4000 | 0,701 | 0,720 | 30 | 30 | 30 | 30 | 30 | 263 | 1052000 |

Висновок: в результаті ми отримали найбільш оптимальну роботу системи, коли час роботи планувальника становить 43мс.

З більшим часом роботи відбувається застій решти процесорів, які виконавши завдання чекають, допоки планувальник звільниться.

З меншим часом роботи ККД зменшується, бо інтенсивність переключення процесора між режимами зумовлює переважання часу непотрібного розподілу над, власне, виконанням обчислень.

Порівняння алгоритмів

|  |  |  |  |
| --- | --- | --- | --- |
| Критерії | **FIFO** | **Алгоритм із слабким планувальником** | **Алгоритм із сильним планувальником** |
| ККД при однаковій продуктивності процесорів | При зростанні значення продуктивності системи ККД спадає | При зростанні значення продуктивності системи ККД спадає | При зростанні значення продуктивності системи ККД спадає |
| ККД при однаковій продуктивності системи, різні процесори | Залежить від застою процесорів. Якщо найсильніший процесор має значно більше значення за решту – то імовірність застою на слабших збільшується. | Залежить від величини значення продуктивності найслабшого процесора.  Чим більше значення – тим більша втрата ККД. | В реальних комп’ютерних системах – чим більша продуктивність сильнішого процесора – тим більша ККД, на моделі відмінність не істотна. |
| ККД залежно від часу роботи планувальника | Всі процесори тільки працюють | Всі процесори тільки працюють | Залежить.  Має оптимальне значення, яке можна отримати тестуванням системи |

Коди реалізації С#

Код класу “Processor”:

class Processor

{

private Label info;

private int performance = 0, performedTasks = 0, performedOperations = 0;

private bool busy = false, has\_not\_finished\_tasks = false;

private TasK executedTask;

public Processor(int \_productivity, Label \_info)

{

performance = \_productivity;

info = \_info;

}

public async void executeTask()

{

busy = true;

performedTasks++;

performedOperations += executedTask.getOperationAmount();

transferText(Convert.ToString(performedTasks));

await Task.Delay(Convert.ToInt32(executedTask.getOperationAmount() / performance));

busy = false;

}

public async void executeTask(int msecForTask) // Робота процесора-планувальника за проміжок часу

{

busy = true;

if (executedTask.getOperationAmount() <= msecForTask \* performance)//Якщо встигне виконати завдання

{

performedOperations += executedTask.getOperationAmount();

performedTasks++;

transferText(Convert.ToString(performedTasks)); // Обновлюємо лічильник виконаних завдань

await Task.Delay(Convert.ToInt32(executedTask.getOperationAmount() / performance));

busy = false;

has\_not\_finished\_tasks = false;

return;

}

// Якщо за стільки-то мсекунд НЕ встигне виконати завдання, а лише певну к-сть операцій

performedOperations += msecForTask \* performance;

executedTask.setOperationAmount(executedTask.getOperationAmount() - msecForTask \* performance);

await Task.Delay(msecForTask); // Працює стільки-то мсекунд

has\_not\_finished\_tasks = true; // Мусить продовжити виконання завдання після чергового планування

busy = false; // Вільний до планування завдань

}

public int getPerformedTasksAmount() { return performedTasks; }

public int getPerformedOperationsAmount() { return this.performedOperations; }

public int getProductivity() { return performance; }

public void transferText(String text)

{

if (info.InvokeRequired)

info.Invoke((MethodInvoker)delegate()

{

transferText(text);

});

else info.Text = text;

}

public void setTask(TasK aTask) { executedTask = aTask; }

public bool isBusy() { return busy; }

public bool hasNoFinishedTasks() { return has\_not\_finished\_tasks; }

}

Код класу “TasK”:

class TasK

{

private int operationAmount;

private List<int> requiredProcessors = new List<int>();

public TasK(List<int> \_possibleProcessors, int \_LowComplexity, int \_HighComplexity)

{

operationAmount = new Random().Next(\_LowComplexity, \_HighComplexity);

requiredProcessors = \_possibleProcessors;

}

public int getOperationAmount() { return operationAmount; }

public void setOperationAmount(int \_operationAmount) { operationAmount = \_operationAmount; }

public List<int> getRequiredProcessors() { return requiredProcessors; }

}

Код всіх структур в програмі:

struct \_DataType

{

public int d\_MODE;

public int d\_tenSecondsCount;

public DateTime d\_Start;

public int d\_weakestId, d\_strongestId;

}

struct \_UserInput

{

public double d\_percent;

public int d\_LowComplexity;

public int d\_HighComplexity;

}

Код функції, що додає задачу в кожну 1мс:

private void t\_Int(object sender, EventArgs e)

{

// Додаємо задачу із певною імовірністю появи в дану мілісекунду

if (new Random().Next(0, 100) < \_userInput.d\_percent)

{

List<int> proc\_list = new List<int> {0, 1, 2, 3, 4};

Random rand\_digit = new Random();

for (int i = 0; i < 4; i++)

{

int new\_value = rand\_digit.Next(0, 5);

if (proc\_list.Contains(new\_value))

proc\_list.Remove(new\_value);

}

AllTasks.Add(new TasK(proc\_list,\_userInput.d\_LowComplexity,\_userInput.d\_HighComplexity));

}

TasksAmount.Text = Convert.ToString(AllTasks.Count);

}

Код функцій, що знаходять найсильніші і найслабші процесори:

private int getWeakestProcessorID()

{

int index = 0;

int MinProductivity = Processors[0].getProductivity();

for (int i = 1; i < 5; i++) // Обираємо найслабший процесор

{

if (Processors[i].getProductivity() < MinProductivity)

{

MinProductivity = Processors[i].getProductivity();

index = i; // і-ий процесор - планувальник

}

}

return index;

}

private int getStrongestProcessorID()

{

int index = 0;

int MaxProductivity = Processors[0].getProductivity();

for (int i = 1; i < 5; i++) // Обираємо сильніший процесор

{

if (Processors[i].getProductivity() > MaxProductivity)

{

MaxProductivity = Processors[i].getProductivity();

index = i; // i-ий процесор є планувальником

}

}

return index;

}

Код функцій, що знаходять к-сть обчислених задач і операцій:

private int getPerformedTasksAmount()

{

int sum = 0; // Read amount of performed tasks

for (int i = 0; i < 5; i++)

sum += Processors[i].getPerformedTasksAmount();

return sum;

}

private int getPerformedOperationsAmount()

{

int sum = 0;

for (int i = 0; i < 5; i++)

{

sum += Processors[i].getPerformedOperationsAmount();

}

return sum;

}

Код функцій, що обчислюють теоретичний та реальний ККД:

private float getTheoryEfficiency()

{

int N\_max = 0;

for (int i = 0; i < 5; i++)

{

N\_max += Processors[i].getProductivity();

}

N\_max \*= 10000 \* \_fileOutput.d\_tenSecondsCount; // 10 seconds = 10000 msec

return (float) getPerformedOperationsAmount() / N\_max;

}

private float getRealEfficiency()

{

int N\_max = 0;

switch (\_fileOutput.d\_MODE)

{

case 1:

for (int i = 0; i < 5; i++)

N\_max += Processors[i].getProductivity() \* 10000 \* \_fileOutput.d\_tenSecondsCount;

break;

case 2:

int weakIndex = \_fileOutput.d\_weakestId; // getWeakestProcessorID();

for (int i = 0; i < 5; i++)

{

if (i == weakIndex)

continue;

N\_max += Processors[i].getProductivity() \* 10000 \* \_fileOutput.d\_tenSecondsCount;

}

break;

case 3:

int strongIndex = \_fileOutput.d\_strongestId;// getStrongestProcessorID();

for (int i = 0; i < 5; i++)

{

if (i == strongIndex)

{

N\_max += Convert.ToInt32(Processors[strongIndex].getProductivity() \* 20/24 \* 10000 \* \_fileOutput.d\_tenSecondsCount);

continue;

}

N\_max += Processors[i].getProductivity() \* 10000 \* \_fileOutput.d\_tenSecondsCount;

}

break;

case 4:

int strongIndex2 = \_fileOutput.d\_strongestId; // getStrongestProcessorID();

for (int i = 0; i < 5; i++)

{

if (i == strongIndex2)

{

N\_max += Convert.ToInt32(Processors[strongIndex2].getProductivity() \* 43 / 48 \* 10000 \* \_fileOutput.d\_tenSecondsCount);

continue;

}

N\_max += Processors[i].getProductivity() \* 10000 \* \_fileOutput.d\_tenSecondsCount;

}

break;

}

return (float)getPerformedOperationsAmount() / N\_max;

}