МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КПІ»

Кафедра

обчислювальної техніки

**ЛАБОРАТОРНА РОБОТА 3**

з дисципліни

“Операційні Системи”

Дослідження дисциплін обслуговування заявок при обмежених ресурсах

Студента 3 курсу

групи ІП-64

Вінницький В.А.

Перевірив: Сімоненко А.В.

Київ 2018

**Завдання:**

1. Алгоритм FIFO без пріоритетів

FIFO є найбільш простою стратегією планування процесів і полягає в тому, що ресурс передається тому процесу, котрий раніше всіх інших звернувся до нього. Коли процес потрапляє в чергу готових процесів, process control block приєднується до хвоста черги. Середній час очікування для стратегії FIFO є часто досить великим і залежить від порядку надходження процесів в чергу готових процесів.

Суть данного алгоритма заключается в том, что каждая заявка проходит в i-ой очереди несколько кругов и только потом переходит в очередь i+1. Этот алгоритм является производным алгоритма с обратной связью и смешанного алгоритма.

Квант времени, выделяемый каждой работе, выбирается исходя из конкретной ситуации: число работ, параллельно занимающих процессор, интервал Т1, дисциплины обслуживания.

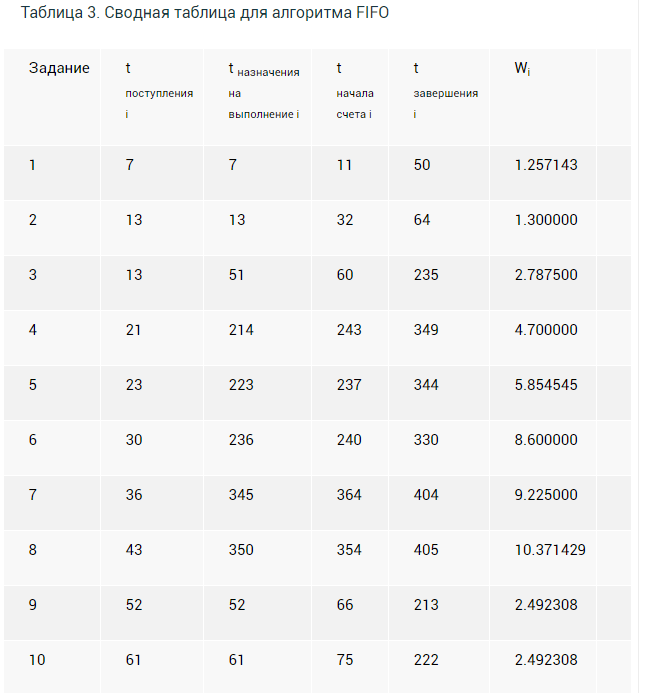
n\_replay - параметр диспетчера, задающий количество квантов работы задачи перед ее переходом в следующую очередь.

rep - счетчик, связанный с каждой задачей, индицирующий количество квантов ее исполнения в текущей очереди.

i - номер текущей очереди.

t - текущее значение времени

?t - квант времени, предоставляемый задаче



**Лістинг:**

lab3.cpp

#include <iostream>

#include "disp.h"

using namespace std;

int main()

{

cout << "1 - FIFO" << endl;

int c;

cin >> c;

double kv, sv;

cout << "Enter CPU kvant: ";

cin >> kv;

cout << "Enter SV kvant: ";

cin >> sv;

Dispather\* d;

if (c == 1)

{

d = new DispatherNoPrt(kv, sv, 2);

d->Add(2, 2);

d->Add(3, 3);

d->Add(4, 1);

d->Add(5, 4);

d->Add(8, 2);

d->Do(0, 42);

cin >> c;

delete d;

}

else

return -1;

}

disp.cpp

#include <iostream>

#include "disp.h"

using namespace std;

Queue::Queue() : first(NULL), last(NULL), cnt(0) { }

void Queue::Enqueue(int num, int pr)// = -1) // включение

{

if (!first)

first = last = new Task(num, pr);

else {

last->next = new Task(num, pr);

last = last->next;

}

cnt++;

}

void Queue::Enqueue(Task\* t) // включение

{

if (!first)

first = last = t;

else {

last->next = t;

last = last->next;

}

cnt++;

}

Task\* Queue::Dequeue() // исключение

{

if (first)

{

Task\* t = first;

first = first->next;

cnt--;

return t;

}

}

int Queue::Empty() // пусто?

{

return !cnt;

}

Dispather::Dispather(double kvant, int sv\_kvant) : kvant(kvant), n\_tasks(0), sv\_kvant(sv\_kvant){ }

DispatherNoPrt::DispatherNoPrt(double kvant, int sv\_kvant, int n\_replay) : Dispather(kvant, sv\_kvant), n\_replay(n\_replay) { }

void DispatherNoPrt::Add(int n, int prt)

{

Q[0].Enqueue(n);

n\_tasks++;

}

void DispatherNoPrt::Do(double t, double T)

{

int cur\_q = 0;

Task\* cur\_t;

double tmAbs = t;

for (double tm = t; tm <= T;) {

for (int rep = 0; rep < n\_replay; rep++) {

for (int i = 0; i < n\_tasks && tm <= T; i++) {

cur\_t = Q[cur\_q].Dequeue();

cout << "SV " << ' ' << " Time = " << tmAbs << endl;

tmAbs += sv\_kvant;

cout << "Task #" << cur\_t->num << ' ' << "Time = " << tmAbs << endl;

if (rep == n\_replay - 1)

Q[cur\_q + 1].Enqueue(cur\_t);

else

Q[cur\_q].Enqueue(cur\_t);

tm += kvant;

tmAbs += kvant;

}

}

cur\_q++;

}

}

disp.h

#pragma once

#ifndef DISP\_H

#define DISP\_H

#include <iostream>

using namespace std;

struct Task // Структура, определяющая задачу

{

int pr; // приоритет

int num; // номер

Task\* next; // следующая задача

Task(int num, int pr = -1){};

};

class Queue // класс очереди

{

Task \*first, \*last;

public:

int cnt;

Queue();

void Enqueue(int num, int pr = -1);

void Enqueue(Task\* t) ;// включение

Task\* Dequeue();// исключение

int Empty(); // пусто?

};

class Dispather { // базовый класс диспетчера

protected:

double kvant; // квант CPU, выделяемый задаче

int n\_tasks; // количество задач

int sv\_kvant; // квант SV

Queue Q[10]; // массив очередей

public:

Dispather(double kvant, int sv\_kvant);

virtual void Add(int n, int prt) {}; // длбавить задачу

virtual void Do(double t, double T) {}; // выполнять

};

class DispatherNoPrt : public Dispather // класс диспетчера со смешанным алгоритмом

{

int n\_replay; // количество повторений

public:

DispatherNoPrt(double kvant, int sv\_kvant, int n\_replay);

void Add(int n, int prt = -1);

void Do(double t, double T);

};

#endif