**ЛАБОРАТОРНА РОБОТА #3**

З дисципліни «Операційні системи»

На тему «Управління пам’яттю»

|  |  |
| --- | --- |
| **Виконав**  Студент групи КВ-21  Комарницький О.Б. | **Перевірив**  Проф. СП і СКС  Зайцев В.Г. |

|  |  |  |
| --- | --- | --- |
| **Оцінка** | **Дата** | **Підпис** |
|  |  |  |

**Варіант #8**

**Завдання:**

Розробити модель алгоритму управління пам’яті (без використання зовнішньої пам’яті) переміщуваними розділами. Кількість розділів повинна бути меншою за кількість процесів. Процедура “стиск” виконується вбік молодших розрядів в тому випадку, якщо даний розділ не може бути розміщеним в пам’яті.

Процеси утворюють спільну чергу до розділів пам’яті. Структуризація адресного простору – сегментована. Розміри процесів задаються випадково.

Продемонструвати процес перетворення заданої віртуальної адреси у фізичну.

**Код програми мовою C++**

***Main.cpp***

#include "Interface.h"

#include <thread>

#include <iostream>

using namespace std;

void read\_user();

int main()

{

thread inp(read\_user), start(Run);

inp.join();

start.join();

return 0;

}

void read\_user()

{

char ch;

while ((ch = getchar()) != 'q')

{

switch (ch)

{

case 'a':

add\_process();

break;

case 'd':

del\_process();

break;

case 'o':

output();

break;

}

}

}

***Interface.h***

struct Segm

{

size\_t addr, size;

};

struct Settings

{

size\_t max\_segm\_size = 100;

size\_t mem\_size = 1024;

const unsigned int delay = 300;

int i = 5; // every i-th process uses existed segment

};

void add\_process();

void del\_process();

void previous\_load();

void Run();

void output();

***Functions.cpp***

#include "Interface.h"

#include <map>

#include <list>

#include <vector>

#include <mutex> // For synchronysing user input and queue workers

#include <Windows.h>

using namespace std;

void defragment();

Settings settings;

mutex m;

map<size\_t, size\_t> Table;

list<size\_t> Queue;

list<pair<size\_t, size\_t>> CurProcesses;

vector<Segm> Memory;

void add\_process()

{

size\_t size = rand() % settings.max\_segm\_size;

Queue.push\_back(size);

}

void del\_process()

{

if (!CurProcesses.size()) return;

while (m.try\_lock() == false) Sleep(settings.delay);

// Select random process

int i = rand() % CurProcesses.size();

auto p = CurProcesses.begin();

advance(p, i);

printf("Deleted addr=%d\n", Table[p->first]);

// If two or more processes use same segment, then only process must be deleted

int count = 0;

for (auto & proc : CurProcesses)

if (proc.first == p->first)

count++;

if (count > 1)

{

CurProcesses.erase(p);

m.unlock();

return;

}

// Only one process uses this segment. Finding this segment

for (auto it = Memory.begin(); it != Memory.end(); it++)

if ((\*it).addr == Table[p->first])

{

Memory.erase(it);

Table.erase(p->first);

CurProcesses.erase(p);

break;

}

m.unlock();

}

void Run()

{

while (true)

{

if (!Queue.size()) continue;

size\_t size = Queue.front();

int seg\_number = 0;

size\_t shift = 0;

if (rand() % 5 == 0 && settings.i-- > 0) // Segment exists in RAM

{

auto n = Table.begin();

advance(n, rand() % Table.size());

seg\_number = n->first;

shift = n->second;

}

else // Create new segment for process

{

// Find segment number

while (Table.find(seg\_number) != Table.end()) seg\_number++;

// Find free address

int i = -1;

size\_t addr0 = 0;

while (++i < (int)Memory.size())

{

if (Memory[i].addr >= addr0 + size)

break;

else

addr0 = Memory[i].addr + Memory[i].size;

}

// Check if space is found

if (addr0 + size > settings.mem\_size || !addr0 && Memory.size() && !Memory[0].addr)

{

if (m.try\_lock())

{

defragment();

m.unlock();

}

Sleep(settings.delay);

continue;

}

Table[seg\_number] = addr0;

// Put in right place of memory

Segm s;

s.size = size;

s.addr = addr0;

auto it = Memory.begin();

advance(it, i);

Memory.insert(it, s);

}

CurProcesses.push\_back(pair<size\_t, size\_t>(seg\_number, shift));

Queue.pop\_front();

output();

Sleep(settings.delay);

}

}

void defr(unsigned int i)

{

if (i >= Memory.size()) return;

Segm \* segm = &Memory[i];

if (i == 0)

{

if (segm->addr != 0)

{

// Find segment id

int seg\_number;

for (auto & rec : Table)

if (rec.second == segm->addr)

seg\_number = rec.first;

Table[seg\_number] = segm->addr = 0;

}

}

else

{

size\_t new\_addr = Memory[i - 1].addr + Memory[i - 1].size;

if (segm->addr != new\_addr)

{

// Find segment id

size\_t seg\_number;

for (auto & rec : Table)

if (rec.second == segm->addr)

{

seg\_number = rec.first;

break;

}

Table[seg\_number] = segm->addr = new\_addr;

}

}

return defr(i + 1);

}