3. Методы управления памятью

**Распределение памяти фиксированными разделами**

Самым простым способом управления оперативной памятью является разделение её на несколько разделов (сегментов) фиксированной величины (статическое распределение). Очередная задача, поступающая на выполнение, помещается либо в общую очередь, либо в очередь к некоторому разделу. Подсистема управления памятью в этом случае выполняет следующие задачи: сравнивает размер программы, поступившей на выполнение, и свободных разделов па­мяти; выбирает подходящий раздел; осуществляет загрузку программы и на­стройку адресов.

Данный метод имеет существенный недостаток — жесткость. Так как в каж­дом разделе может выполняться только одна программа, то уровень мульти­программирования заранее ограничен числом разделов.

Даже если программа имеет небольшой объем, она будет занимать весь раздел, что приводит к неэффективному использованию памяти.

**Распределение памяти разделами переменной величины**

В этом случае память машины не делится заранее на разделы. Сначала вся память свободна. Каждой вновь поступающей задаче выделяется необхо­димая ей память. Если достаточный объем памяти отсутствует, то задача не принимается на выполнение и стоит в очереди. После завершения задачи па­мять освобождается, и на это место может быть загружена другая задача. Та­ким образом, в произвольный момент времени оперативная память представ­ляет собой случайную последовательность занятых и свободных участков (разделов) произвольного размера. Задачами операционной системы при реализации данного метода управ­ления памятью являются: ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти; анализ запроса (при поступлении новой задачи), просмотр таблицы свободных об­ластей и выбор раздела, размер которого достаточен для размещения посту­пившей задачи; загрузка задачи в выделенный ей раздел и корректировка таблиц свободных и занятых областей;

**Распределение памяти динамическими разделами**

Программный код не перемещается во время выполнения, т.е. может быть проведена единовременная настройка адресов посредством использова­ния перемещающего загрузчика.

По сравнению с методом распределения памяти фиксированными разде­лами данный метод обладает гораздо большей гибкостью, но ему присущ очень серьезный недостаток — фрагментация памяти. Фрагментация — это наличие большого числа несмежных участков свободной памяти очень ма­ленького размера (фрагментов). Настолько маленького, что ни одна из вновь поступающих программ не может поместиться ни в одном из участков, хотя суммарный объем фрагментов может составить значительную величину, на­много превышающую требуемый объем памяти.

6. Синхронизация потоков с помощью критических секций

Критические секции

Объект-критическая секция помогает программисту выделить участок кода, где нить получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса. Перед использованием ресурса нить входит в критическую секцию (вызывает функцию EnterCriticalSection). Если после этого какая-либо другая нить попытается войти в ту же самую критическую секцию, ее выполнение приостановится, пока первая нить не покинет секцию с помощью вызова LeaveCriticalSection. Используется только для нитей одного процесса. Порядок входа в критическую секцию не определен.

Существует также функция TryEnterCriticalSection, которая проверяет, занята ли критическая секция в данный момент. С ее помощью нить в процессе ожидания доступа к ресурсу может не блокироваться, а выполнять какие-то полезные действия.

Пример. Синхронизация нитей с помощью критических секций.

9. Семафоры

Объект-семафор - это фактически объект-взаимоисключение со счетчиком. Данный объект позволяет "захватить" себя определенному количеству нитей. После этого "захват" будет невозможен, пока одна из ранее "захвативших" семафор нитей не освободит его. Семафоры применяются для ограничения количества нитей, одновременно работающих с ресурсом. Объекту при инициализации передается максимальное число нитей, после каждого "захвата" счетчик семафора уменьшается. Сигнальному состоянию соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается не установленным (сброшенным).

Функция CreateSemaphore создает объект-семафор с указанием и максимально возможного начального его значения, OpenSemaphore – возвращает дескриптор существующего семафора, захват семафора производится с помощью ожидающих функций, при этом значение семафора уменьшается на единицу, ReleaseSemaphore - освобождение семафора с увеличением значения семафора на указанное в параметре число.

**Читатель-писатель:**

1. HANDLE   s\_write,   // семафор, допускающий только одного писателя  
           s\_acess,   // семафор для учета и управления читателями   
       eOnRead;   // событие - разрешение на подключение новых   
                  // читателей

int am\_readers=4;//макс. кол читателей

vector<int> vect;//разделяемый ресурс  
...  
s\_acess=CreateSemaphore(NULL,am\_readers,am\_readers,NULL);

s\_write=CreateSemaphore(NULL,1,1,NULL);

eOnRead = CreateEvent(NULL,true,false,NULL);

Семафор s\_acess визначає максимально можливу кількість одночасно читаючих читачів. Спочатку його лічильник вільних ресурсів дорівнює максимальному значенню лічильника, тобто ніхто не читає - все ресурси вільні.

1. **Потік "письменник"**

У циклі, поки не надійшов сигнал про необхідність завершення:

- Чекаємо завершення роботи попереднього письменника (семафор s\_write).

- Дочекалися, значить це єдиний письменник в цей момент часу. Забороняємо підключення нових читачів, займаючи подія eOnRead.

- Монопольно пишемо.

- Дозволяємо підключення нових читачів, звільняючи подія eOnRead.

- Дозволяємо писати іншим письменникам.//Поток «писатель»

DWORD WINAPI Writer(PVOID pParam)

{

int ind = \*(int\*)pParam;

for (int i=0; i<25; i++)

{

// ждем пока другой писатель закончит

WaitForSingleObject(s\_write,INFINITE);

// запрещаем подключение новых читателей

ResetEvent(eOnRead);

// пишем

vect.push\_back(i);

cout<<"! "<<ind<<" writer, zapisano: "<<i<<"\n";

//разрешаем новые чтения

SetEvent(eOnRead);

// пусть новый писатель появляется не сразу, а спустя

// какое-то время. Это даст возможность поработать читателям.

// Если эту задержку не ставить, то всегда будет следующая

// картина: писатель отработав, дает возможность запуститься

// читателям - они все и запускаются, однако, почти

// сразу после этого в очереди появляется новый писатель, и

// запрещает появление новых читателей. Уже читающие читатели

// дорабатывают, а новые не создаются. Когда очередь читателей

// иссякает - запускается новый писатель. Таким образом, имеем

// всегда одну и ту же картину: почти одновременно запускаются

// все читатели, читают, все завершаются, запускается писатель,

// читает, завершается, и далее в цикле.

Sleep(500);

// разрешаем другим писать

ReleaseSemaphore(s\_write,1,NULL);

}

}

1. **Потік "читач"**  
   У циклі, поки не надійшов сигнал про необхідність завершення:

- Чекаємо дозволу на підключення нового читача (для цього подія eOnRead має бути вільно) і вільну вакансію читача, тому що кількість одночасно читаючих читачів у нас обмежена величиною am\_readers. Ми повинні дочекатися одночасного виконання цих двох умов, а тому використовуємо функцію WaitForMultipleObjects замість двох викликів WaitForSingleObjects.

- Читати дозволили, вакансія була (вільне володіння ресурс семафора, який після успішного виконання функції очікування був зайнятий) - читаємо.

- Читання завершено - звільняємо вакансію читача (звільняємо один ресурс семафора s\_acess).

DWORD WINAPI Reader(PVOID pParam)

{

HANDLE h[2];

h[0] = eOnRead;

h[1] = s\_acess;

int ind = \*(int\*)pParam;

for(int i=0;i<vect.size();i++)

{

// ждем 1) разрешения на чтение (событие eOnRead) и

//2) свободную вакансию читателя (семафор s\_acess)

WaitForMultipleObjects(2,h,true,INFINITE);

cout<<"@ "<<ind<<" reader, schitano: "<<vect[i]<<"\n";

ReleaseSemaphore(s\_acess,1,NULL);

// освобождаем вакансию читателя

// пускай этот читатель немного задержится для устранения эффекта

// "закончил чтение - начал чтение" читать можно, однако, пусть

// новый читатель появляется не сразу, а спустя некоторое время.

// Это устранит эффект почти одновременного появления читателей.

Sleep(500);

}

}

1. **Запуск потоків**Читач і письменник реалізовані у вигляді потоків одного процесу. Кількість потоків нічим не обмежена.

int \_tmain(int argc, \_TCHAR\* argv[])

{

bool ex=false;

//список ид-ров потоков

vector<HANDLE> han;

//номера текущих читателей и писателей

int r\_count=0;

int w\_count=0;

//создаем семафоры и событие

s\_acess=CreateSemaphore(NULL,am\_readers,am\_readers,NULL);

s\_write=CreateSemaphore(NULL,1,1,NULL);

eOnRead = CreateEvent(NULL,true,false,NULL);

while(!ex)

{

switch(menu())

{

case 2:

r\_count++;

//запуск потока читателя

han.push\_back(CreateThread(NULL,0,Reader,&r\_count,0,NULL));

break;

case 1:

w\_count++;

//запуск потока писателя

han.push\_back(CreateThread(NULL,0,Writer,&w\_count,0,NULL));

break;

case 0:

ex=true;

break;

}

//Sleep(500);

}

//закрываем потоки перед выходом

for(vector<HANDLE>::iterator it=han.begin();it<han.end();it++)

TerminateThread(\*it,0);

return 0;

}

**Функции для работы с:**

#include <thread>

1. **Семафоры:**

HANDLE Semaphore;

Semaphore = CreateSemaphore(NULL, 1, 5, NULL);

WaitForSingleObject(Semaphore, INFINITE);

ReleaseSemaphore(Semaphore, 1, NULL);

CloseHandle(Semaphore);

1. **Мютексы и крит секции**

Есть еще вариант с критическими секциями. Принцип тот же.  
Создаем критическую секцию

|  | |
| --- | --- |
| |  |  | | --- | --- | | 1 | CRITICAL\_SECTION cs; | |  | HANDLE mutex;  Mutex = CreateMutex(NULL, FALSE, NULL) | |  |

1. глобально, в main инициализируем

|  | |
| --- | --- |
| |  |  | | --- | --- | | 1 | InitializeCriticalSection( &cs ); | |  |

1. , а в потоках, вместо WaitForSingleObject

|  | |
| --- | --- |
| |  |  | | --- | --- | | 1 | EnterCriticalSection( &cs ); | |  |

1. , а вместо ReleaseMutex

|  | |
| --- | --- |
| |  |  | | --- | --- | | 1 | LeaveCriticalSection( &cs ); | |  |