Лабораторна робота № 3

Тема: Керування оперативною пам’яттю.

Мета: Навчитися керувати оперативною пам’яттю в середовищі операційної системи, розробляти програми керування пам’яттю.

Обладнання та програмне забезпечення: ПК, операційна система Windows, середовище розробки Visual Studio.

Вказівки для самостійної підготовки

Під час підготовки необхідно повторити теоретичний матеріал:

1. Технологія віртуальної пам'яті.
2. Сегментація пам'яті.
3. Сторінкова організація пам'яті.
4. Сторінково-сегментна організація пам'яті.
5. Керування основною пам'яттю в ОС Windows.

Теоретичні відомості

1 Технологія віртуальної пам'яті

1.1 Поняття віртуальної пам’яті

Віртуальна пам'ять —технологія, в якій вводиться рівень додаткових перетворень між адресами пам'яті, використовуваних процесом, і адресами фізичної пам'яті комп'ютера. Перетворення забезпечують захист пам'яті та відсутність прив'язки процесу до адрес фізичної пам'яті.

Завдяки віртуальній пам'яті фізична пам'ять адресного простору процесу може бути фрагментованою, оскільки основний обсяг пам'яті, яку займає процес, більшу частину часу залишається вільним. Адреси можна переміщати так, щоб основній пам'яті відповідали тільки ті розділи адресного простору процесу, які використовуються у конкретний момент.

Невикористовувані розділи адресного простору ставляться у відповідність повільнішій пам'яті (простору на жорсткому диску), а в цей час інші процеси використовують основну пам'ять. Коли ж розділ знадобиться, його дані завантажують з диска в основну пам'ять. Дані зчитуються з диска в основну пам'ять під час звертання до них.

У такий спосіб можна збільшити розмір адресного простору процесу і забезпечити виконання процесів, що за розміром перевищують основну пам'ять.

1.2 Логічна і фізична адресація пам'яті

Логічна або ***віртуальна адреса*** — адреса, яку генерує програма, запущена на деякому процесорі. Адреси, що використовують інструкції конкретного процесора, є логічними адресами. Сукупність логічних адрес становить логічний адресний простір.

Фізична адреса - адреса, якою оперує мікросхема пам'яті. Прикладна програма в сучасних комп'ютерах не має справи з фізичними адресами. Спеціальний апаратний пристрій MMU (memory management unit - пристрій керування пам'яттю) відповідає за перетворення логічних адрес у фізичні. Сукупність усіх доступних фізичних адрес становить фізичний адресний простір.

2 Сегментація пам'яті

2.1 Особливості сегментації пам'яті

Сегментація пам'яті зображає логічний адресний простір як сукупність незалежних блоків змінної довжини, які називають ***сегментами.*** Кожен сегмент містить дані одного призначення.

Кожен сегмент має ім'я і довжину. Логічна адреса складається з номера сегмента і зсуву всередині сегмента; з такими адресами працює прикладна програма. Компілятори створюють окремі сегменти для різних даних програми (сегмент коду, сегмент даних, сегмент стека). Під час завантаження програми у пам'ять створюють таблицю дескрипторів сегментів процесу, кожний елемент якої відповідає одному сегменту і складається із базової адреси, значення межі та прав доступу.

Під час формування адреси її сегментна частина вказує на відповідний елемент таблиці дескрипторів сегментів процесу.

Якщо зсув більший, ніж задане значення межі (або якщо права доступу процесу не відповідають правам, заданим для сегмента), то апаратне забезпечення генерує помилку. Коли ж усе гаразд, сума бази і зсуву в разі чистої сегментації дасть у результаті фізичну адресу в основній пам'яті. Якщо сегмент вивантажений на диск, спроба доступу до нього спричиняє його завантаження з диска в основну пам'ять. Кожному сегменту відповідає неперервний блок пам'яті такої самої довжини, що перебуває в довільному місці фізичної пам'яті або на диску.

Переваги сегментації пам'яті:

1. Є можливість організувати кілька незалежних сегментів пам'яті для процесу і використати їх для зберігання даних різної природи. Права доступу до кожного такого сегмента можуть бути задані по-різному.

2. Окремі сегменти можуть спільно використовуватися різними процесами, їхні таблиці дескрипторів сегментів повинні містити однакові елементи, що описують такий сегмент.

3. Фізична пам'ять процесу не обов'язково має бути неперервною. Сегментація дає змогу окремим частинам адресного простору процесу відображатися не в основну пам'ять, а на диск, і довантажуватися з нього за потребою, забезпечуючи виконання процесів будь-якого розміру.

Недоліки:

1. Необхідність введення додаткового рівня перетворення пам'яті знижує продуктивності. Для ефективної реалізації сегментації потрібна апаратна підтримка.

2. Керування блоками пам'яті змінної довжини з урахуванням необхідності їхнього збереження на диску може бути складним.

3. Вимога, щоб кожному сегменту відповідав неперервний блок фізичної пам'яті відповідного розміру, спричиняє зовнішню фрагментацію пам'яті. Внутрішньої фрагментації у цьому разі не виникає, оскільки сегменти мають змінну довжину і завжди можна виділити сегмент довжини, необхідної для виконання програми.

Сегментацію застосовують обмежено через фрагментацію і складність реалізації ефективного звільнення пам'яті та обміну з диском.

2.2 Реалізація сегментації

Логічні адреси в програмі формуються з використанням сегментації і мають такий вигляд:“***селектор-зсув***”. Значення селектора завантажуютьу спеціальний регістр процесора (сегментний регістр) і використовують як індекс у таблиці дескрипторів сегментів, що перебуває в пам'яті та є аналогом таблиці сегментів.

Селектор містить індекс дескриптора в таблиці, біт індикатора локальної або глобальної таблиці та необхідний рівень привілеїв.

Для системи задають спільну ***глобальну таблицю дескрипторів*** (Global Descriptor Table, ***GDT***), а для кожної задачі - ***локальну таблицю дескрипторів*** (Local Descriptor Table, ***LDT***).

Дескриптори мають довжину 64 біти. Вони визначають властивостіпрограмних об'єктів (наприклад, сегментів пам'яті або таблиць дескрипторів).

Дескриптор містить значення бази, яке відповідає адресі об'єкта (початок сегмента); значення межі; тип об'єкта (сегмент, таблиця дескрипторів тощо); характеристики захисту.

Звертання до таблиць дескрипторів підтримується апаратно. Якщо задані в дескрипторі характеристики захисту не відповідають рівню привілеїв, визначеному селектором, отримати доступ до пам'яті за його допомогою буде неможливо. Так забезпечують захист пам'яті.

Для архітектури внаслідок перетворення логічної адреси отримують нефізичну адресу, а вид адреси, який називають лінійною адресою.

3 Сторінкова організація пам'яті

До основних технологій реалізації віртуальної пам'яті належить ***сторінкова організація пам'яті***. Її головна ідея — розподіл пам'яті блоками фіксованої довжини, що називають ***сторінками.***

3.1 Базові принципи сторінкової організації пам'яті

Фізичну пам'ять розбивають на блоки фіксованої довжини — ***фрейми***, або сторінкові блоки. Логічну пам'ять, у свою чергу, розбивають на блоки такої самої довжини — ***сторінки***. Коли процес починає виконуватися, його сторінки завантажуються в доступні фрейми фізичної пам'яті з диска або іншого носія.

Сторінкова організація пам'яті має апаратну підтримку. Кожна адреса, яку генерує процесор, ділиться на дві частини: номер сторінки і зсув сторінки. Номер сторінки використовують як індекс у таблиці сторінок.

Таблиця сторінок — це структура даних, що містить набір елементів, кожен із яких містить інформацію про номер сторінки, номер відповідного їй фрейму фізичної пам'яті (або його базову адресу) та права доступу. Номер сторінки використовують для пошуку елемента в таблиці. Після його знаходження до базової адреси відповідного фрейму додають зсув сторінки, чим і визначають фізичну адресу.

Розмір сторінки є ступенем числа 2, у сучасних ОС використовують сторінки розміром від 2 до 8 Кбайт. У спеціальних режимах адресації можна працювати зі сторінками більшого розміру.

Для кожного процесу створюють його власну таблицю сторінок. Коли процес починає своє виконання, ОС розраховує його розмір у сторінках і кількість фреймів у фізичній пам'яті. Кожну сторінку завантажують у відповідний фрейм, після чого його номер записують у таблицю сторінок процесу.

На логічному рівні для процесу вся пам'ять зображується неперервним блоком і належить тільки цьому процесові, а фізично вона розосереджена по адресному простору мікросхеми пам'яті, чергуючись із пам'яттю інших процесів. Процес не може звернутися до пам'яті, адреса якої не вказана в його таблиці сторінок (так реалізований захист пам'яті).

ОС повинна мати інформацію про поточний стан фізичної пам'я-ті (про зайнятість і незайнятість фреймів, їхню кількість тощо). Цю інформацію звичайно зберігають у ***таблиці фреймів***. Кожен її елемент відповідає фрейму і містить всі відомості про нього.

3.2 Порівняльний аналіз сторінкової організації пам'яті та сегментації

Основні переваги сторінкової організації пам'яті:

1. Реалізація розподілу і звільнення пам'яті спрощується. Усі сторінки з погляду процесу рівноправні, тому можна підтримувати список вільних сторінок і при потребі виділяти першу сторінку із цього списку, а після звільнення повертати сторінку в список. Із сегментами так чинити не можна, оскільки кожен сегмент використовується лише за його призначенням.

2. Реалізація обміну даними з диском спрощується. Для організації обміну ділянка на диску, яку використовують для зберігання інформації про сторінки, вивантажені з пам'яті може бути теж розбита на блоки фіксованого розміру, рівного розмірові фрейму.

Сторінкова організація пам'яті має недоліки:

1. Цей підхід спричиняє внутрішню фрагментацію, пов'язану з тим, що розмір сторінки завжди фіксований, і при потребі виділення блоку пам'яті конкретної довжини його розмір буде кратним розміру сторінки. У середньому розмір невикористовуваної пам'яті становить приблизно половину сторінки для кожного виділеного блоку пам'яті . Така фрагментація може бути знижена зменшенням кількості та збільшенням розміру блоків, що виділяються.

2. Таблиці сторінок мають бути більші за розміром, ніж таблиці сегментів.

3.3 Багаторівневі таблиці сторінок

Щоб адресувати логічний адресний простір значного обсягу за допомогою однієї таблиці сторінок, її роблять великою.

Щоб уникнути великих таблиць, запропоновано технологію багаторівневих таблиць сторінок. Таблиці сторінок самі розбиваються на сторінки, інформацію про які зберігають в таблиці сторінок верхнього рівня. Кількість рівнів рідко перевищує 2, але може доходити й до 4.

Логічну адресу розбивають на індекс у таблиці верхнього рівня, індекс у таблиці нижнього рівня і зсув.

Дві основні переваги. **1)**Таблиці сторінок стають менші за розміром, пошук у них можна робити швидше. **2)**Не всі таблиці сторінок мають перебувати в пам'яті у

конкретний момент часу. Якщо процес не використовує якийсь блок пам'яті, то вміст усіх сторінок нижнього рівня невикористовуваного блоку може бути тимчасово збережений на диску.

3.4 Реалізація таблиць сторінок

Таблицю верхнього рівня називають ***каталогом сторінок,*** для кожної задачі має бути заданий окремий каталог сторінок, фізичну адресу якого зберігають у спеціальному керуючому регістрі **cr3** і куди він автоматично завантажується апаратним забезпеченням при перемиканні контексту. Таблицю нижнього рівня називають просто ***таблицею сторінок***. Лінійна адреса поділяється на три поля:

1. ***каталогу*** (Directory) — визначає елемент каталогу сторінок, що вказує на потрібну таблицю сторінок;

2. ***таблиці*** (Table) — визначає елемент таблиці сторінок, що вказує на потрібний фрейм пам'яті;

3. ***зсуву*** (Offset) — визначає зсув у межах фрейму, що у поєднанні з адресою фрейму формує фізичну адресу.

Розмір полів каталогу і таблиці становить 10 біт, що дає таблиці сторінок, які містять 1024 елементи, розмір поля зсуву — 12 біт, що дає сторінки і фрейми розміром 4 Кбайт.

Одна таблиця сторінок нижнього рівня адресує 4 Мбайт пам'яті (1 Мбайт фреймів), а весь каталог сторінок - 4 Гбайт.

Елементи таблиць сторінок всіх рівнів мають однакову структуру. Поля елемента:

1. ***прапорець присутності*** (Present), дорівнює одиниці, якщо сторінка перебуває у фізичній пам'яті (їй відповідає фрейм); рівність цього прапорця нулю означає, що сторінки у фізичній пам'яті немає;

2. ***20 найбільш значущих бітів***, які задають початкову адресу фрейму, кратну 4 Кбайт;

3. ***прапорець доступу*** (Accessed), який рівний одиниці під час кожного звертання пристрою сторінкової підтримки до відповідного фрейму;

4. ***прапорець зміни*** (Dirty), який рівний одиниці під час кожної операції запису у відповідний фрейм;

5. ***прапорець читання-запису*** (Read/Write), що задає права доступу до цієї сторінки або таблиці сторінок (для читання і для запису або тільки для читання);

6. ***прапорець привілейованого режиму*** (User/Supervisor), який визначає режим процесора, необхідний для доступу до сторінки. Якщо цей прапорець дорівнює нулю, сторінка може бути адресована тільки із привілейованого режиму, якщо одиниці — доступна також і з режиму користувача;

Прапорці присутності, доступу і зміни можна використовувати ОС для організації віртуальної пам'яті.

3.5 Асоціативна пам'ять

Для підвищення продуктивності у разі сторінкової організації пам'яті запропоновано технологію ***асоціативної пам'яті*** або ***кеша трансляції***. У швидкодіючій пам'яті (швидшій, ніж основна пам'ять) створюють набір із кількох елементів (різні архітектури відводять під асоціативну пам'ять від 8 до 2048 елементів. Кожен елемент кеша трансляції відповідає одному елементу таблиці сторінок.

Під час генерування фізичної адреси спочатку відбувається пошук відповідного елемента таблиці в кеші (за полем каталогу, полем таблиці та зсуву), і якщо він знайдений, стає доступною адреса відповідного фрейму, що негайно можна використати для звертання до пам'яті. Якщо ж у кеші відповідного елемента немає, то доступ до пам'яті здійснюють через таблицю сторінок, а після цього елемент таблиці сторінок зберігають в кеші замість найстарішого елемента.

4 Сторінково-сегментна організація пам'яті

4.1 Базові принципи

Для того щоб об'єднати переваги обох підходів, у деяких апаратних архітектурах (зокрема, в IA-32) використовують комбінацію сегментної та сторінкової організації пам'яті. За такої організації перетворення логічної адреси у фізичну відбувається за три етапи:

1. У програмі задають логічну адресу із використанням сегмента і зсуву.

2. Логічну адресу перетворюють у лінійну (віртуальну) адресу за правилами, заданими для сегментації.

3. Віртуальну адресу перетворюють у фізичну за правилами, заданими для сторінкової організації.

Таку архітектуру називають ***сторінково-сегментною організацією пам'яті.***

4.2 Перетворення адрес

Машинна мова оперує логічними адресами. Логічна адресаскладається із селектора і зсуву.

Лінійна або ***віртуальна адреса*** — це ціле число без знака розміром 32 біти. За його допомогою можна дістати доступ до 4 Гбайт комірок пам'яті. Перетворення логічної адреси в лінійну відбувається всередині ***пристрою сегментації*** за правилами перетворення адреси на базі сегментації.

Фізичну адресу використовують для адресації комірок пам'яті в мікросхемах пам'яті. Її теж зображають 32-бітовим цілим числом без знака. Перетворення лінійної адреси у фізичну відбувається всередині ***пристрою сторінкової підтримки*** за правилами для сторінкової організації пам'яті (лінійну адресу розділяють апаратурою на адресу сторінки і сторінковий зсув, а потім перетворюють у фізичну адресу із використанням таблиць сторінок, кеша трансляції тощо).

5 Керування основною пам'яттю в ОС Windows

5.1 Сторінкова адресація у Windows

Під час роботи з лінійними адресами у Windows XP використовують дворівневі таблиці сторінок. У кожного процесу є свій каталог сторінок, кожен елемент якого вказує на таблицю сторінок. Таблиці сторінок усіх рівнів містять по 1024 елементи таблиці сторінок, кожний такий елемент вказує на фрейм фізичної пам'яті. Фізичну адресу каталогу сторінок зберігають у блоці KPROCESS.

Розмір лінійної адреси, з якою працює система, становить 32 біти. З них 10 біт відповідають адресі в каталозі сторінок, ще 10 — це індекс елемента в таблиці, останні 12 біт адресують конкретний байт сторінки (і є зсувом).

Розмір елемента таблиці сторінок теж становить 32 біти. Перші 20 біт адресують конкретний фрейм, а інші 12 біт описують атрибути сторінки (захист, стан сторінки в пам'яті, який файл підкачування використовує). Якщо сторінка не перебуває у пам'яті, то в перших 20 біт зберігають зсув у файлі підкачування.

Для визначення розміру сторінки у Win32 API використовують функцію отримання інформації про систему ***GetSystemInfo():***

```
void GetSystemInfo(
  [out] LPSYSTEM_INFO lpSystemInfo
);
```

```
SYSTEM_INF0 info;
// структура для отримання інформації про систему

GetSystemlnfo(&info);
pгintf("Розмір сторінки: %d\n", info.dwPageSize);
```

5.2 Особливості адресації процесів і ядра

Лінійний адресний простір процесу поділяється на дві частини: перші 2 Гбайт адрес доступні для процесу в режимі користувача і є його захищеним адресним простором; інші 2 Гбайт адрес доступні тільки в режимі ядра і відображають системний адресний простір. Деякі версії дають можливість задати співвідношення 3 Гбайт/1 Гбайт під час завантаження системи.

5.3 Структура адресного простору процесів і ядра

В адресному просторі процесу можна виділити такі ділянки:

1. перші 64 Кбайт — це спеціальна ділянка, доступ до якої завжди спричиняє помилки;
2. усю пам'ять між першими 64 Кбайт і останніми 136 Кбайт (майже 2 Гбайт) може використовувати процес під час свого виконання;
3. далі розташовані два блоки по 4 Кбайт: блоки оточення потоку (TEB) і процесу (PEB);
4. наступні 4 Кбайт - ділянка пам'яті, куди відображаються різні системні дані (системний час, значення лічильника системних годин, номер версії системи), тому для доступу до них процесу не потрібно перемикатися в режим ядра;
5. останні 64 Кбайт використовують для запобігання спробам доступу за межі адресного простору процесу (спроба доступу до цієї пам'яті дасть помилку).

Системний адресний простір містить такі ділянки:

1. Перші 512 Мбайт системного адресного простору використовують для завантаження ядра системи.
2. 4 Мбайт пам'яті виділяють під каталог сторінок і таблиці сторінок процесу.
3. Спеціальну ділянку пам'яті розміром 4 Мбайт, яку називають гіперпростором, використовують для відображення різних структур даних, специфічних для

процесу, на системний адресний простір.

4. 512 Мбайт виділяють під системний кеш.

5. У системний адресний простір відображаються спеціальні ділянки пам'яті — вивантажуваний пул і невивантажуваний пул.

6. Приблизно 4 Мбайт у кінці системного адресного простору виділяють під структури даних, необхідні для створення аварійного образу пам'яті, а також для структур даних.

Системний адресний простір містить такі ділянки:

1. Перші 512 Мбайт системного адресного простору використовують для завантаження ядра системи.

2. 4 Мбайт пам'яті виділяють під каталог сторінок і таблиці сторінок процесу.

3. Спеціальну ділянку пам'яті розміром 4 Мбайт, яку називають гіперпростором, використовують для відображення різних структур даних, специфічних для процесу, на системний адресний простір.

4. 512 Мбайт виділяють під системний кеш.

5. У системний адресний простір відображаються спеціальні ділянки пам'яті — вивантажуваний пул і невивантажуваний пул.

6. Приблизно 4 Мбайт у кінці системного адресного простору виділяють під структури даних, необхідні для створення аварійного образу пам'яті, а також для структур даних.

Функція розподілу віртуальної пам’яті

```
LPVOID VirtualAlloc(
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

6. Приклади програмної реалізації планування процесів та потоків

Приклад 1. Розподіл віртуальної пам’яті процесу

```
//Розподіл віртуальної памяті процесу
#include <windows.h>
#include <iostream.h>

int main()
{
```

```
    int *a;                          //покажчик на масив цілих чиселconst
    int size = 1000; //розмір масиву
    //розподіляємо віртуальну памятьа =
    (int*) VirtualAlloc(
                NULL,
                size * sizeof(int),
                MEM_COMMIT,
                PAGE_READWRITE);
    if(!a)
    {
            cout << "Virtual allocation failed. " << endl;return
            GetLastError();
    }
            cout << "Virtual memory address: " << a << endl;
//звільняємо віртуальну память
    if (!VirtualFree(a, 0, MEM_RELEASE))
    {
            cout << "Memory release failed. " << endl;return GetLastError();
    }
    cin.get();
    return 0;
}
```

Приклад 2. Резервування та розподіл віртуальної пам’яті

```
#include <vcl.h> #include
<windows.h> #include
<iostream.h>#include
<conio.h>

int main()
{
        LPVOID lpr, lpc; const int
        Kb = 1024; const int size =
        100;

        //резервуємо віртуальну памятьlpr =
        VirtualAlloc(
                    NULL,
                    size * Kb,
                    MEM_RESERVE,
                    PAGE_READWRITE);
        if (!lpr)
        {
```

```
        cout << "Virtual memory reservation failed." <<   endl;return GetLastError();
      }
      cout << "Virtual memory addressed:    " << lpr << endl;
      //розподіляємо віртуальну памятьlpc =
    VirtualAlloc(
            (LPVOID) 0X00880000,
            Kb,
            MEM_COMM
            IT,
            PAGE_READWRITE);
          getch();
    if (!lpc)
    {
        cout << "Virtual memory reservation failed." <<   endl;return GetLastError();
      }
      cout << "Virtual memory addressed:    " << lpc << endl;
      //відмінюємо розподіл if(!VirtualFree(lpc, Kb,
      MEM_DECOMMIT))
      {
        cout << "Memory decommit failed.    " << endl;return
        GetLastError();
      }

        //звільняємо віртуальну память
      if(!VirtualFree(lpr, 0, MEM_RELEASE))
      {
        cout << "Memory release failed.    " << endl;return
        GetLastError();
      }
      getch();
    return 0;
}
```

Приклад 3. Блокування та розблокування віртуальних сторінок

```
#include <windows.h>
#include <iostream.h>
#include <conio.h>

int main()
{
    LPVOID vm;                  // покажчик на віртуальну память
    SIZE_T size = 4096; //розмір памяті
```

```
        //резервуємо віртуальну памятьvm =
        VirtualAlloc(
                NULL,
                size,
                MEM_COMMIT,
                PAGE_READWRITE);
        if (!vm)
        {
                cout << "Virtual allocation failed." <<   endl;return
                GetLastError();
         }
         //блокуємо віртуальну память
         if(!VirtualLock(vm, size))
         {
                cout << "Virtual lock failed.    " << endl;return GetLastError();
         }
         //розблокуємо віртуальну память
         if(!VirtualUnlock(vm, size))
         {
                cout << "Virtual unlock failed.    " << endl;return
                GetLastError();
         }
         //звільняємо віртуальну память if(!VirtualFree(vm,
         0, MEM_RELEASE))
         {
                cout << "Memory release failed.    " << endl;return
                GetLastError();
         }
         cout << "OK!.    " << endl;getch();
         return 0;
}
```

Приклад 4. Читання та зміна множини віртуальних сторінок процесу

```
#include <windows.h>
#include <iostream.h>
#include <conio.h>

int main()
{
        const int size = 4096;//розмір сторінки HANDLE
        hProcess;                       //дескриптор процесу

        SIZE_T min, max; //мін. та макс. розміри множини сторінокSIZE_T *pMin =
        &min; //покажчик на мінімальний розмір SIZE_T *pMax = &max;
```

```
//покажчик на максимальний розмір
// дескриптор процесу
hProcess = GetCurrentProcess();
if(!GetProcessWorkingSetSize(hProcess, pMin, pMax))
{
    cout << "Get process working set size failed. " << endl;
    return GetLastError();
}
else
{
    cout << "Min = " << (min/size) << endl;
    cout << "Max = " << (max/size) << endl;
}
// установити нові межі робочого простору
    if(!SetProcessWorkingSetSize(hProcess, min-10, max-10))
{
    cout << "Set process working set size failed. " << endl;return GetLastError();
}
// прочитати нові межі робочого простору
if(!GetProcessWorkingSetSize(hProcess, pMin, pMax)
{
    cout << "Get process working set size failed. " << endl;return GetLastError();
}
else
{
    cout << "Min = " << (min/size) << endl;
    cout << "Max = " << (max/size) << endl;
}
getch(); return 0;
```

Приклад 5. Визначення стану віртуальної пам’яті

```
#include <vcl.h> #include
<windows.h> #include
<iostream.h>#include
<conio.h>

int main()
{
    BYTE   *a, *b; //базова адреса області та підобластіconst int
    shift = 5000;                    //розмір області
```

```
        const int size = 10000;                 //зміщення для підобласті

        MEMORY_BASIC_INFORMATION mbi; //структура для інформації про
                                                //віртуальну память

        DWORD mbi_size = sizeof(MEMORY_BASIC_INFORMATION);

        //розподіляємо віртуальну памятьa =
        (BYTE*) VirtualAlloc(
                    NULL,
                    size, MEM_COMMIT,
                    PAGE_READWRITE);
        if (!a)
        {
              cout << "Virtual allocation failed." <<   endl;return
              GetLastError();
          }

          //Установлює адреси підобластіb =
          a+shift;

          //Визначаємо інформацію про віртуальну память if(mbi_size
          != VirtualQuery(b, &mbi, mbi_size))
          {
              cout << "Virtual query failed." <<   endl;return GetLastError();
        }

          //Виводимо інформацію про віртуальну память
          cout << "Base address: " << mbi.BaseAddress << endl;
          cout << "Allocation base: " << mbi.AllocationBase << endl;
          cout << "Allocation protect:" << mbi.AllocationProtect<<endl;cout << "Region size: "
          << mbi.RegionSize << endl;
          cout << "State: " << mbi.State << endl; cout << "Protect: "
          << mbi.Protect << endl;cout << "Type: " << mbi.Type <<
          endl;

            //звільняємо віртуальну память
          if(!VirtualFree(a, 0, MEM_RELEASE))
          {
              cout << "Memory release failed.    " << endl;return
              GetLastError();
          }
          getch();
        return 0;
}
```

Завдання для студентів

1. Розробити програми для роботи з віртуальною пам’яттю. Для програмної реалізації використовувати середовище програмування Microsoft Visual Studio, мова програмування – C/C++, інтерфейс консольний.
2. Скласти звіт про виконання лабораторної роботи. Зміст звіту:
   - ➢ опис функцій для роботи з віртуальною пам’яттю;
   - ➢ постановка задачі;
   - ➢ програмний код;
   - ➢ результати виконання програми;
   - ➢ висновок.
3. До захисту лабораторної роботи підготувати відповіді на контрольні питання.

Контрольні питання

1. Визначити основні поняття віртуальної пам’яті.
2. Дати визначення логічної та фізичної адресації пам'яті.
3. Охарактеризувати особливості сегментації пам'яті.
4. Як виконується реалізація сегментації в архітектурі IA-32?
5. Охарактеризувати базові принципи сторінкової організації пам'яті.
6. Навести порівняльний аналіз сторінкової організації пам'яті та сегментації.
7. Охарактеризувати багаторівневі таблиці сторінок.
8. Як виконується реалізація таблиць сторінок в архітектурі IA-32?
9. Дати визначення асоціативної пам'яті.
10. Охарактеризувати базові принципи сторінково-сегментної організації пам'яті.
11. Як виконується перетворення адрес в архітектурі IA-32?
12 Як виконується реалізація сторінкової адресації у Windows?
13. Охарактеризувати особливості адресації процесів і ядра.
14. Навести характеристику структури адресного простору процесів і ядра.

Варіанти завдань:

1. Розробити програму, яка демонструє управління структурою даних типу «стек», елементами якого є значення типу int. Стек реалізувати за допомогою динамічного масиву розміром 10 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над стеком: •перевірити, стек порожній чи не порожній; •заштовхнути елемент; •виштовхнути елемент; •переглянути вершину стека; •поміняти значення двох верхніх елементів стека.
2. Розробити програму, яка демонструє управління структурою даних типу «стек», елементами якого є значення типу short. Стек реалізувати за допомогою динамічного масиву розміром 15 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над стеком: •перевірити, стек порожній чи не порожній;

•заштовхнути елемент; •виштовхнути елемент; •переглянути вершину стека; •продублювати вершину стека.

3. Розробити програму, яка демонструє управління структурою даних типу «стек», елементами якого є значення типу char. Стек реалізувати за допомогою динамічного масиву розміром 12 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над стеком: •перевірити, стек порожній чи не порожній; •заштовхнути елемент; •виштовхнути елемент; •переглянути вершину стека; •визначити скільки елементів у стеку.

4. Розробити програму, яка демонструє управління структурою даних типу «черга», елементами якої є значення типу int. Чергу реалізувати за допомогою динамічного масиву розміром 10 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над чергою: •перевірити чергу – порожня чи не порожня; •додати елемент в хвіст черги; •видалити елемент з голови черги; •переглянути голову черги; •поміняти місцями значення з голови і хвоста черги.

5. Розробити програму, яка демонструє управління структурою даних типу «черга», елементами якої є значення типу short. Чергу реалізувати за допомогою динамічного масиву розміром 12 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над чергою: •перевірити чергу – порожня чи не порожня; •додати елемент в хвіст черги; •видалити елемент з голови черги; •переглянути голову черги; •продублювати хвіст черги.

6. Розробити програму, яка демонструє управління структурою даних типу «черга», елементами якої є значення типу char. Чергу реалізувати за допомогою динамічного масиву розміром 6 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над чергою: •перевірити чергу – порожня чи не порожня; •додати елемент в хвіст черги; •видалити елемент з голови черги; •переглянути голову черги; •визначити скільки елементів у черзі.

7. Розробити програму, яка демонструє управління структурою даних типу «Дек» (черга з двома кінцями), елементами якого є значення типу char. Дек реалізувати за допомогою динамічного масиву розміром 10 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над Деком: •перевірити Дек – порожній чи не порожній; •додати елемент в лівий кінець Дека; •додати елемент в правий кінець Дека; •видалити елемент ліворуч; •видалити елемент праворуч; •переглянути елемент ліворуч; •переглянути елемент праворуч.

8. Розробити програму, яка демонструє управління структурою даних типу «Дек» (черга з двома кінцями), елементами якого є значення типу int. Дек реалізувати за допомогою динамічного масиву розміром 13 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над Деком: •перевірити Дек – порожній чи не порожній; •додати елемент в лівий кінець Дека; •додати елемент в правий кінець Дека; •видалити елемент ліворуч; •видалити елемент праворуч; •переглянути елемент ліворуч; •переглянути елемент праворуч; •визначити скільки елементів у Деку.

9. Розробити програму, яка демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу int. Динамічний вектор реалізувати за допомогою динамічного масиву розміром 13 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над вектором: •перевірити, вектор порожній чи не порожній; •прочитати елемент із зазначеним індексом; •змінити значення

елемента з вказаним індексом; •додати елемент в кінець вектору; •очистити вектор; •видалити з вектору останній елемент.

10. Розробити програму, яка демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу float. Динамічний вектор реалізувати за допомогою динамічного масиву розміром 7 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над вектором: •перевірити, вектор порожній чи не порожній; •прочитати елемент із зазначеним індексом; •змінити значення елемента з вказаним індексом; •додати елемент в кінець вектору; •очистити вектор; •знайти кількість нульових елементів у векторі.
11. Розробити програму, яка демонструє управління структурою даних типу «стек», елементами якого є значення типу long long. Стек реалізувати за допомогою динамічного масиву розміром 16 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над стеком: •перевірити, стек порожній чи не порожній; •заштовхнути елемент; •виштовхнути елемент; •переглянути вершину стека; •вивести на екран усі елементи стеку починаючи з вершини.
12. Розробити програму, яка демонструє управління структурою даних типу «стек», елементами якого є значення типу float. Стек реалізувати за допомогою динамічного масиву розміром 8 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над стеком: •перевірити, стек порожній чи не порожній; •заштовхнути елемент; •виштовхнути елемент; •переглянути вершину стека; •визначити найбільший елемент у стеку.
13. Розробити програму, яка демонструє управління структурою даних типу «стек», елементами якого є значення типу double. Стек реалізувати за допомогою динамічного масиву розміром 14 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над стеком: •перевірити, стек порожній чи не порожній; •заштовхнути елемент; •виштовхнути елемент; •переглянути вершину стека; •визначити найменший елемент у стеку.
14. Розробити програму, яка демонструє управління структурою даних типу «черга», елементами якої є значення типу long long. Чергу реалізувати за допомогою динамічного масиву розміром 16 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над чергою: •перевірити чергу – порожня чи не порожня; •додати елемент в хвіст черги; •видалити елемент з голови черги; •переглянути голову черги; •визначити найбільший елемент у черзі.
15. Розробити програму, яка демонструє управління структурою даних типу «черга», елементами якої є значення типу float. Чергу реалізувати за допомогою динамічного масиву розміром 11 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над чергою: •перевірити чергу – порожня чи не порожня; •додати елемент в хвіст черги; •видалити елемент з голови черги; •переглянути голову черги; •визначити найменший елемент у черзі.
16. Розробити програму, яка демонструє управління структурою даних типу «черга», елементами якої є значення типу double. Чергу реалізувати за допомогою динамічного масиву розміром 20 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над чергою: •перевірити чергу – порожня чи не порожня; •додати елемент в хвіст черги; •видалити елемент з голови черги; •переглянути голову черги; •вивести на екран елементи черги починаючи з першого.

17. Розробити програму, яка демонструє управління структурою даних типу «черга», елементами якої є значення типу unsigned int. Чергу реалізувати за допомогою динамічного масиву розміром 8 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над чергою: •перевірити чергу – порожня чи не порожня; •додати елемент в хвіст черги; •видалити елемент з голови черги; •переглянути голову черги; •вивести на екран елементи черги починаючи з останнього.
18. Розробити програму, яка демонструє управління структурою даних типу «Дек» (черга з двома кінцями), елементами якого є значення типу short. Дек реалізувати за допомогою динамічного масиву розміром 7 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над Деком: •перевірити Дек – порожній чи не порожній; •додати елемент в лівий кінець Дека; •додати елемент в правий кінець Дека; •видалити елемент ліворуч; •видалити елемент праворуч; •переглянути елемент ліворуч; •переглянути елемент праворуч; •вивести елементу Деку на екран.
19. Розробити програму, яка демонструє управління структурою даних типу «Дек» (черга з двома кінцями), елементами якого є значення типу long long. Дек реалізувати за допомогою динамічного масиву розміром 9 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над Деком: •перевірити Дек – порожній чи не порожній; •додати елемент в лівий кінець Дека; •додати елемент в правий кінець Дека; •видалити елемент ліворуч; •видалити елемент праворуч; •переглянути елемент ліворуч; •переглянути елемент праворуч; •визначити найменший елемент у Деку.
20. Розробити програму, яка демонструє управління структурою даних типу «Дек» (черга з двома кінцями), елементами якого є значення типу float. Дек реалізувати за допомогою динамічного масиву розміром 10 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над Деком: •перевірити Дек – порожній чи не порожній; •додати елемент в лівий кінець Дека; •додати елемент в правий кінець Дека; •видалити елемент ліворуч; •видалити елемент праворуч; •переглянути елемент ліворуч; •переглянути елемент праворуч; •визначити найбільший елемент у Деку.
21. Розробити програму, яка демонструє управління структурою даних типу «Дек» (черга з двома кінцями), елементами якого є значення типу double. Дек реалізувати за допомогою динамічного масиву розміром 18 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над Деком: •перевірити Дек – порожній чи не порожній; •додати елемент в лівий кінець Дека; •додати елемент в правий кінець Дека; •видалити елемент ліворуч; •видалити елемент праворуч; •переглянути елемент ліворуч; •переглянути елемент праворуч; •визначити кількість «вільних» комірок у Деку.
22. Розробити програму, яка демонструє управління структурою даних типу «стек», елементами якого є значення типу unsigned int. Стек реалізувати за допомогою динамічного масиву розміром 11 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над стеком: •перевірити, стек порожній чи не порожній; •заштовхнути елемент; •виштовхнути елемент; •переглянути вершину стека; •визначити кількість «вільних» комірок у стеку.
23. Розробити програму, яка демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу char. Динамічний вектор реалізувати за допомогою динамічного масиву розміром 9 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над вектором: •перевірити, вектор порожній чи не порожній; •прочитати елемент із зазначеним індексом; •змінити значення елемента з вказаним індексом; •додати елемент в кінець вектору; •очистити вектор.

24. Розробити програму, яка демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу int. Динамічний вектор реалізувати за допомогою динамічного масиву розміром 11 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над вектором: •перевірити, вектор порожній чи не порожній; •прочитати елемент із зазначеним індексом; •змінити значення елемента з вказаним індексом; •додати елемент в кінець вектору; •очистити вектор; •вивести значення елементів вектору на екран.
25. Розробити програму, яка демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу short. Динамічний вектор реалізувати за допомогою динамічного масиву розміром 10 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над вектором: •перевірити, вектор порожній чи не порожній; •прочитати елемент із зазначеним індексом; •змінити значення елемента з вказаним індексом; •додати елемент в кінець вектору; •очистити вектор; •знайти найбільший елемент вектору.
26. Розробити програму, яка демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу long long. Динамічний вектор реалізувати за допомогою динамічного масиву розміром 15 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над вектором: •перевірити, вектор порожній чи не порожній; •прочитати елемент із зазначеним індексом; •змінити значення елемента з вказаним індексом; •додати елемент в кінець вектору; •очистити вектор; •знайти найменший елемент вектору.
27. Розробити програму, яка демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу float. Динамічний вектор реалізувати за допомогою динамічного масиву розміром 8 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над вектором: •перевірити, вектор порожній чи не порожній; •прочитати елемент із зазначеним індексом; •змінити значення елемента з вказаним індексом; •додати елемент в кінець вектору; •очистити вектор; •відсортувати елементи вектору за зростанням.
28. Розробити програму, яка демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу double. Динамічний вектор реалізувати за допомогою динамічного масиву розміром 16 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над вектором: •перевірити, вектор порожній чи не порожній; •прочитати елемент із зазначеним індексом; •змінити значення елемента з вказаним індексом; •додати елемент в кінець вектору; •очистити вектор; •відсортувати елементи вектору за спаданням.
29. Розробити програму, яка демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу unsigned int. Динамічний вектор реалізувати за допомогою динамічного масиву розміром 12 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над вектором: •перевірити, вектор порожній чи не порожній; •прочитати елемент із зазначеним індексом; •змінити значення елемента з вказаним індексом; •додати елемент в кінець вектору; •очистити вектор; •переписати елементи вектору в зворотному порядку.
30. Розробити програму, яка демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу unsigned short. Динамічний вектор реалізувати за допомогою динамічного масиву розміром 8 Кб., пам’ять під який виділити за допомогою функції VirtualAlloc(). Операції, що виконуються над вектором: •перевірити, вектор порожній чи не порожній; •прочитати елемент із зазначеним індексом; •змінити значення

елемента з вказаним індексом; •додати елемент в кінець вектору; •очистити вектор; •визначити кількість «вільних» комірок у векторі

Приклад виконання:

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>

// Програма демонструє управління структурою даних типу «динамічний вектор», елементами якого є значення типу int.
// Динамічний вектор реалізований за допомогою динамічного масиву розміром 1 Кб
// Операції, що виконуються над вектором:
// •перевірити, вектор порожній чи не порожній;
// •прочитати елемент із зазначеним індексом;
// •змінити значення елемента з вказаним індексом;
// •додати елемент в кінець вектору.

#define VECTOR_CAPACITY_Byte 1024
#define VECTOR_CAPACITY (VECTOR_CAPACITY_Byte / sizeof(int))

typedef struct
{
    int* data;
    size_t size;
    size_t capacity;
} DynamicVector;

void createDynamicVector(DynamicVector *vector)
{

    vector->data = (int*)VirtualAlloc(NULL, VECTOR_CAPACITY_Byte, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    if (vector->data == NULL)
    {
        perror("Error allocating memory for vector data");
        exit(EXIT_FAILURE);
    }
    vector->size = 0;
    vector->capacity = VECTOR_CAPACITY;

}

int isEmpty(DynamicVector *vector)
{
    return vector->size == 0;
}

int getElement(DynamicVector *vector, size_t index)
{
    if (index < vector->size)
        return vector->data[index];
    else
    {
        fprintf(stderr, "Index out of bounds\n");
        exit(EXIT_FAILURE);
    }
}

void setElement(DynamicVector *vector, size_t index, int value)
{
    if (index < vector->size)
        vector->data[index] = value;
    else
    {
```

```
        fprintf(stderr, "Index out of bounds\n");
        exit(EXIT_FAILURE);
    }
}

void addElement(DynamicVector *vector, int value)
{
    if (vector->size < vector->capacity)
    {
        vector->data[vector->size] = value;
        vector->size++;
    }
    else
    {
        fprintf(stderr, "Vector capacity exceeded\n");
        exit(EXIT_FAILURE);
    }
}

void destroyDynamicVector(DynamicVector *vector)
{
    if (vector->data != NULL)
        VirtualFree(vector->data, 0, MEM_RELEASE);
}

int main()
{
    DynamicVector myVector;
    createDynamicVector(&myVector);

    printf("\nIs vector empty? %s\n", isEmpty(&myVector) ? "Yes" : "No");

    for (size_t i = 0; i < 5; ++i)
        addElement(&myVector, i * 2);

    printf("\nIs vector empty? %s\n", isEmpty(&myVector) ? "Yes" : "No");

    printf("\n");
    for (size_t i = 0; i < myVector.size; ++i)
        printf("Element at index %zu: %d\n", i, getElement(&myVector, i));
    printf("\n");

    size_t indexToModify = 2;
    int newValue = 42;
    setElement(&myVector, indexToModify, newValue);
    printf("Element at index %zu after modification: %d\n", indexToModify,
getElement(&myVector, indexToModify));

    // Звільнення пам'яті, виділеної для динамічного вектора
    destroyDynamicVector(&myVector);

    return 0;
}
```