МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерних наук

ОПЕРАЦІЙНІ СИСТЕМИ

Методичні вказівки до виконання лабораторних робіт для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Комп'ютерні науки» галузі знань F Інформаційні технології спеціальності F3 Комп'ютерні науки денної та заочної форм навчання

Частина 2

Луцьк – 2026

УДК 004.02

До друку
Голова вченої ради ФКІТ _____І. С. Кондіус
(підпис)

Електронна копія друкованого видання передана для внесення в репозитарій ЛНТУ
директор бібліотеки _____Н. П. Поліщук
(підпис)

Затверджено вченою радою ФКІТ,
протокол №__ від «__» _____ 2026 року.

Розглянуто та схвалено на засіданні кафедри комп'ютерних наук ЛНТУ,
протокол №__ від «__» _____2026 року.

Завідувач кафедри КН_____В. О. Ліщина

Укладач: _____ Р. А. Хиць, асистент кафедри комп'ютерних наук ЛНТУ
(підпис)

_____ Ю. Й. Тулашвілі, доктор педагогічних наук, професор кафедри
(підпис) комп'ютерних наук ЛНТУ

Рецензент: _____ Н. М. Ліщина, кандидат технічних наук, доцент, доцент кафедри
(підпис) інженерії програмного забезпечення ЛНТУ

Операційні системи: методичні вказівки до виконання лабораторних робіт для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Комп'ютерні науки» галузі знань F Інформаційні технології спеціальності F3 Комп'ютерні науки денної та заочної форм навчання Частина 2/ уклад. Р. А. Хиць, Ю. Й. Тулашвілі. Луцьк: ЛНТУ. 2026. 78 с.

У методичних вказівках наведені лабораторні роботи з дисципліни.
Призначені для студентів спеціальності F3 «Комп'ютерні науки» денної та заочної форми навчання.

ЗМІСТ

ВСТУП

Курс «Операційні системи» є однією із навчально-професійних дисциплін спеціальності, яка формує професійні знання майбутніх програмістів, спеціалістів в галузі комп'ютерних технологій. Вивчення курсу забезпечує ознайомлення з особливостями роботи з операційними системами Windows та Linux, розуміння основ взаємодії та файлового середовища систем, способів взаємодії з нею.

Друга частина методичних вказівок є продовженням першої частини і присвячена практичним аспектам системного програмування в середовищі операційної системи Windows засобами інтерфейсу Win32 API. Вона охоплює керування оперативною пам'яттю процесу, структуру Windows-додатків та механізм завантаження операційної системи, логічну і фізичну організацію файлових систем, керування процесами та потоками, їх паралельне виконання і планування, засоби синхронізації потоків (семафори, м'ютекси), мережні засоби операційних систем на основі програмного інтерфейсу сокетів, а також засоби захисту інформації в операційній системі.

Головною метою даної дисципліни є: знайомство з основними поняттями операційної системи, вироблення навиків взаємодії з системами, вивчення особливостей та аналіз відмінностей систем як у взаємодії, так і організації подальшої діяльності у створенні програм для тих чи інших операційних систем. Освоєння створення скриптів як базовий функціонал систем. Розвиток навичок користування командним рядком та графічними оболонками.

Завдання вивчення курсу «Операційні системи» є теоретична та практична підготовка студентів спеціальності підходу до розв'язання технічних проблем, знання загальних принципів та методів взаємодії з операційними системами сімейства Windows та Linux, а також вміти їх використовувати при вдосконаленні та створенні програмних продуктів, і роботи в системах.

У результаті виконання лабораторних робіт, представлених у цій частині методичних вказівок, студенти набудуть практичних навичок розробки, налагодження і тестування системних застосувань у середовищі Microsoft Visual Studio мовою програмування C++; навчаться використовувати функції Win32 API для керування пам'яттю, процесами, потоками та мережною взаємодією; набудуть умінь виявляти й усувати типові помилки багатопотокового програмування (стани змагання, взаємні блокування), а також ознайомляться з механізмами захисту інформації в операційній системі Windows

.

ЛАБОРАТОРНА РОБОТА №6
Керування оперативною пам'яттю

Мета роботи: навчитися керувати оперативною пам'яттю засобами операційної системи, розробляти програми керування пам'яттю.

Зміст теоретичних відомостей:
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

Віртуальна пам'ять – технологія, у якій вводиться рівень додаткових перетворень між адресами пам'яті, які використовує процес, і адресами фізичної пам'яті комп'ютера. Ці перетворення забезпечують захист пам'яті та відсутність прив'язки процесу до конкретних 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 лише ті розділи адресного простору процесу, які використовуються в конкретний момент.

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

Завдяки такому підходу можна збільшити розмір адресного простору процесу і забезпечити виконання процесів, що за розміром перевищують обсяг основної пам'яті.

Логічна, або віртуальна адреса – адреса, яку генерує програма, що виконується на певному процесорі. Адреси, з якими працюють 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 працює з фізичними адресами безпосередньо. За перетворення логічних адрес у фізичні відповідає спеціальний апаратний пристрій MMU (Memory Management Unit – пристрій керування пам'яттю). Сукупність усіх доступних фізичних адрес становить фізичний адресний простір.

2 Сегментація пам'яті

Сегментація пам'яті відображає 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 та зсуву всередині сегмента; саме 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 формується таблиця дескрипторів сегментів процесу, кожен елемент якої відповідає одному сегменту і складається з 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 права доступу процесу не відповідають правам, заданим для сегмента), апаратне забезпечення генерує помилку. Якщо все гаразд, сума бази і зсуву дає в 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:

- можливість організувати кілька незалежних сегментів пам'яті для процесу і використати їх для зберігання даних різної природи; права доступу до кожного сегмента можуть бути задані по-різному;
- окремі сегменти можуть спільно використовуватися різними процесами – їхні таблиці дескрипторів сегментів повинні містити однакові елементи, що описують такий сегмент;
- 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 потреби, забезпечуючи виконання процесів будь-якого розміру.

Недоліки сегментації пам'яті:

- необхідність введення додаткового рівня перетворення адрес знижує продуктивність; для ефективної реалізації сегментації потрібна апаратна підтримка;
- керування блоками пам'яті змінної довжини з урахуванням необхідності їхнього збереження на диску може бути складним завданням;
- вимога, щоб кожному сегменту відповідав неперервний блок фізичної пам'яті відповідного розміру, спричиняє зовнішню фрагментацію пам'яті (внутрішньої фрагментації при цьому не виникає, оскільки сегменти мають змінну довжину).

Через фрагментацію та складність реалізації ефективного звільнення пам'яті й

обміну з диском сегментацію застосовують обмежено.

В архітектурі IA-32 логічні адреси в програмі формуються з використанням сегментації і мають вигляд “селектор:зсув”. Значення селектора завантажують у спеціальний регістр процесора (сегментний регістр) і використовують як індекс у таблиці дескрипторів сегментів, що перебуває в пам'яті та є аналогом таблиці сегментів. В архітектурі IA-32 підтримується шість сегментних регістрів.

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 в IA-32 мають довжину 64 біти. Вони визначають властивості програмних об'єктів (наприклад, сегментів пам'яті або таблиць дескрипторів). Дескриптор містить значення бази, що відповідає адресі об'єкта (початок сегмента), значення межі, тип об'єкта (сегмент, таблиця дескрипторів тощо) та 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 неможливо – так забезпечується захист пам'яті. Для архітектури IA-32 внаслідок перетворення логічної адреси отримують не фізичну адресу, а лінійну адресу.

3 Сторінкова організація пам'яті

До основних технологій реалізації віртуальної пам'яті належить сторінкова організація пам'яті. Її головна ідея – поділ пам'яті на блоки фіксованої довжини, які називають сторінками.

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 чи 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 – структура даних, що містить набір елементів, кожен з яких зберігає номер сторінки, номер відповідного їй фрейму фізичної пам'яті (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 степенем числа 2; у сучасних ОС використовують сторінки розміром від 2 до 8 Кбайт (у спеціальних режимах адресації можливі сторінки більшого розміру).

Для кожного процесу створюють власну таблицю сторінок. Коли процес починає виконання, ОС обчислює 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 відображається неперервним блоком і належить тільки цьому процесу, 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 реалізовано захист пам'яті.

ОС повинна мати інформацію про поточний стан фізичної пам'яті (зайнятість і незайнятість фреймів, їхню кількість тощо). Цю інформацію зазвичай зберігають у таблиці фреймів. Кожен її елемент відповідає фрейму і містить усі відомості про нього.

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 за потреби виділяти першу сторінку зі списку, а після звільнення повертати її до списку. Із сегментами так чинити не можна, оскільки кожен сегмент використовується лише за своїм призначенням;

2. реалізація обміну даними з диском спрощується. Для організації обміну ділянка на диску, яку використовують для зберігання вивантажених з пам'яті сторінок, теж може бути розбита на блоки фіксованого розміру, рівного розміру фрейму.

Сторінкова організація пам'яті має недоліки:

1. цей підхід спричиняє внутрішню фрагментацію, пов'язану з тим, що розмір сторінки завжди фіксований, і за потреби 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; цю фрагментацію можна знизити зменшенням розміру сторінок;

2. таблиці сторінок мають бути більшими за розміром, ніж таблиці сегментів.

Щоб адресувати логічний адресний простір значного обсягу за допомогою однієї таблиці сторінок, її потрібно зробити дуже великою. Щоб уникнути великих таблиць, застосовують технологію багаторівневих таблиць сторінок: таблиці сторінок самі розбиваються на сторінки, інформацію про які зберігають у таблиці сторінок верхнього рівня. Кількість рівнів рідко перевищує два, але може досягати чотирьох.

Логічну адресу розбивають на індекс у таблиці верхнього рівня, індекс у таблиці нижнього рівня та зсув. Це дає дві основні переваги: таблиці сторінок стають меншими за розміром, тому пошук у них виконується швидше; крім того, не всі таблиці сторінок мають перебувати в пам'яті в конкретний момент часу – якщо процес не використовує якийсь блок пам'яті, вміст відповідних сторінок нижнього рівня

може бути тимчасово збережений на диску.

Таблицю верхнього рівня називають каталогом сторінок; для кожної задачі задають окремий каталог сторінок, фізичну адресу якого зберігають у спеціальному керуючому регістрі cr3, і куди він автоматично завантажується апаратним забезпеченням під час перемикання 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 в поєднанні з адресою фрейму формує фізичну адресу.

Розмір полів каталогу і таблиці становить 10 біт, що дає таблиці сторінок із 1024 елементами; розмір поля зсуву – 12 біт, що відповідає сторінкам і фреймам розміром 4 Кбайт. Одна таблиця сторінок нижнього рівня адресує 4 Мбайт пам'яті (1024 фрейми), а весь каталог сторінок – 4 Гбайт.

Елементи таблиць сторінок усіх рівнів мають однакову структуру. *Поля елемента:*

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

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

3. прапорець доступу (Accessed) – стає рівним одиниці під час кожного звертання пристрою сторінкової підтримки до відповідного фрейму;

4. прапорець зміни (Dirty) – стає рівним одиниці під час кожної операції запису у відповідний фрейм;

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

6. прапорець привілейованого режиму (User/Supervisor) – визначає режим процесора, потрібний для доступу до сторінки: якщо прапорець дорівнює нулю, сторінка доступна лише з привілейованого режиму, якщо одиниці – доступна і з режиму користувача.

Прапорці присутності, доступу і зміни операційна система може використовувати для організації віртуальної пам'яті.

Для підвищення продуктивності в разі сторінкової організації пам'яті застосовують технологію асоціативної пам'яті, або кеша трансляції (TLB). У швидкодіючій пам'яті, швидшій, ніж основна, створюють набір 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 зсувом), і якщо він знайдений, стає доступною адреса відповідного фрейму, яку негайно можна використати для звертання до пам'яті. Якщо ж у кеші відповідного елемента немає, доступ до пам'яті здійснюють через таблицю сторінок, після чого знайдений елемент таблиці сторінок зберігають у кеші замість найстарішого елемента.

4 Сторінково-сегментна організація пам'яті

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 в три етапи*:

1. у програмі задають логічну адресу з 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.

Машинна мова архітектури IA-32 оперує логічними адресами. Логічна адреса складається із селектора і зсуву.

Лінійна, або віртуальна адреса – 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 апаратура розділяє на адресу сторінки і сторінковий зсув, а потім перетворює у фізичну адресу з використанням таблиць сторінок, кеша трансляції тощо).

5 Керування основною пам'яттю в ОС Windows

Під час роботи з лінійними адресами в сучасних версіях Windows використовують дворівневі (а в 64-розрядних системах – багаторівневі) таблиці сторінок. У кожного процесу є власний каталог сторінок, кожен елемент якого вказує на таблицю сторінок. Таблиці сторінок усіх рівнів містять по 1024 елементи, кожен такий елемент вказує на фрейм фізичної пам'яті. Фізичну адресу каталогу сторінок зберігають у керуючому блоці процесу.

Розмір лінійної адреси, з якою працює 32-розрядна система, становить 32 біти: 10 бітів відповідають індексу в каталозі сторінок, ще 10 – індексу елемента в таблиці

сторінок, а останні 12 бітів адресують конкретний байт сторінки (і є зсувом).

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

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

```
SYSTEM_INFO info;           // структура для отримання інформації про систему
GetSystemInfo(&info);
std::cout << "Розмір сторінки: " << info.dwPageSize << std::endl;
```

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 й відображають системний адресний простір. Деякі версії Windows дають можливість задати інше співвідношення (наприклад, 3 Гбайт / 1 Гбайт) під час завантаження системи.

В адресному просторі процесу можна виділити такі ділянки:

1. перші 64 Кбайт – це спеціальна ділянка, доступ до якої завжди спричиняє помилку;

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: блок оточення потоку (TEB) і блок оточення 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 дає 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, потрібні для створення аварійного образу пам'яті, а також під інші системні структури даних.

Приклади програмної реалізації керування пам'яттю

Усі приклади оформлені як консольні застосунки Win32 для Microsoft Visual Studio. Для створення проєкту: File → New → Project → Console App (C++); у властивостях проєкту переконатися, що Character Set встановлено як “Not Set” або “Multi-Byte”, якщо потрібні функції з суфіксом A. Для виведення в консоль використовується <iostream> та простір імен std замість застарілих заголовків <iostream.h> і <conio.h>.

Увага: за стандартним шаблоном Console App Visual Studio створює проєкт із передкомпільованим заголовком pch.h, і згенерований файл main.cpp/ConsoleApplication.cpp вже містить рядок #include "pch.h". Якщо просто вставити код прикладу замість усього вмісту файлу, потрібно або залишити цей рядок першим (а нижче дописати наведений код), або вимкнути передкомпільовані заголовки (Властивості проєкту → C/C++ → Precompiled Headers → Not Using Precompiled Headers), інакше компілятор видасть помилку C1010. Найпростіший варіант – створювати проєкт як Empty Project і додавати файл .cpp вручну: тоді pch.h не використовується.

Приклад 1 – Розподіл віртуальної пам'яті процесу

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

int main()
{
    const int size = 1000;     // розмір масиву (кількість елементів)
    int* a = nullptr;          // покажчик на масив цілих чисел

    // розподіляємо віртуальну пам'ять (резервування + фізичне виділення)
    a = (int*)VirtualAlloc(
        NULL,
        size * sizeof(int),
        MEM_COMMIT,
        PAGE_READWRITE);
    if (!a)
    {
        std::cout << "Virtual allocation failed." << std::endl;
        return GetLastError();
    }
    std::cout << "Virtual memory address: " << a << std::endl;

    // звільняємо віртуальну пам'ять
    if (!VirtualFree(a, 0, MEM_RELEASE))
    {
        std::cout << "Memory release failed." << std::endl;
        return GetLastError();
    }
    std::cin.get();
    return 0;
}
```

кінець прикладу 1

На відміну від оригінальної версії прикладу (де для commit використовувалась жорстко задана адреса 0x00880000, що в сучасних версіях Windows із увімкненим ASLR майже завжди недоступна і спричиняє помилку), у новій версії для commit передається адреса lpr, повернута на етапі резервування – це коректний і портативний спосіб роботи з MEM_RESERVE/MEM_COMMIT.

Приклад 2 – Резервування та розподіл віртуальної пам'яті

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

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

    // резервуємо віртуальну пам'ять lpr (без фізичного виділення)
    lpr = VirtualAlloc(
        NULL,
        size * Kb,
        MEM_RESERVE,
        PAGE_READWRITE);

    if (!lpr)
    {
        std::cout << "Virtual memory reservation failed." << std::endl;
        return GetLastError();
    }
    std::cout << "Virtual memory reserved at: " << lpr << std::endl;

    // розподіляємо (commit) частину зарезервованої пам'яті lpc
    lpc = VirtualAlloc(
        lpr,
        Kb,
        MEM_COMMIT,
        PAGE_READWRITE);

    if (!lpc)
    {
        std::cout << "Virtual memory commit failed." << std::endl;
        return GetLastError();
    }
    std::cout << "Virtual memory committed at: " << lpc << std::endl;

    std::cin.get();

    // відміняємо розподіл (decommit)
    if (!VirtualFree(lpc, Kb, MEM_DECOMMIT))
    {
        std::cout << "Memory decommit failed." << std::endl;
        return GetLastError();
    }
```

```
    // звільняємо зарезервовану віртуальну пам'ять
    if (!VirtualFree(lpr, 0, MEM_RELEASE))
    {
        std::cout << "Memory release failed." << std::endl;
        return GetLastError();
    }

    std::cin.get();
    return 0;
}
```

кінець прикладу 2

Приклад 3 – Блокування та розблокування віртуальних сторінок

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

int main()
{
    LPVOID vm;                  // покажчик на віртуальну пам'ять
    SIZE_T size = 4096;         // розмір пам'яті (одна сторінка)
    // резервуємо і одразу розподіляємо віртуальну пам'ять vm
    vm = VirtualAlloc(
        NULL,
        size,
        MEM_COMMIT,
        PAGE_READWRITE);
    if (!vm)
    {
        std::cout << "Virtual allocation failed." << std::endl;
        return GetLastError();
    }
    // блокуємо віртуальну пам'ять (забороняємо вивантаження сторінок на диск)
    if (!VirtualLock(vm, size))
    {
        std::cout << "Virtual lock failed." << std::endl;
        return GetLastError();
    }
    // розблоковуємо віртуальну пам'ять
    if (!VirtualUnlock(vm, size))
    {
        std::cout << "Virtual unlock failed." << std::endl;
        return GetLastError();
    }
    // звільняємо віртуальну пам'ять
    if (!VirtualFree(vm, 0, MEM_RELEASE))
    {
        std::cout << "Memory release failed." << std::endl;
        return GetLastError();
    }
    std::cout << "OK!" << std::endl;
    std::cin.get();
    return 0;
}
```

кінець прикладу 3

Функції GetProcessWorkingSetSize та SetProcessWorkingSetSize оголошені не в windows.h, а в окремому заголовку psapi.h, тому його обов'язково потрібно підключити окремо.

Приклад 4 – Читання та зміна меж робочого набору сторінок процесу

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

int main()
{
    const int size = 4096;          // розмір сторінки
    HANDLE hProcess;                // дескриптор процесу
    SIZE_T minSize, maxSize;        // мін. та макс. розміри робочого набору
    SIZE_T* pMin = &minSize;
    SIZE_T* pMax = &maxSize;

    hProcess = GetCurrentProcess();

    // прочитати поточні межі робочого набору
    if (!GetProcessWorkingSetSize(hProcess, pMin, pMax))
    {
        std::cout << "Get process working set size failed." << std::endl;
        return GetLastError();
    }
    std::cout << "Min = " << (minSize / size) << " pages" << std::endl;
    std::cout << "Max = " << (maxSize / size) << " pages" << std::endl;

    // установити нові межі робочого набору
    if (!SetProcessWorkingSetSize(hProcess, minSize - 10, maxSize - 10))
    {
        std::cout << "Set process working set size failed." << std::endl;
        return GetLastError();
    }

    // прочитати нові межі робочого набору
    if (!GetProcessWorkingSetSize(hProcess, pMin, pMax))
    {
        std::cout << "Get process working set size failed." << std::endl;
        return GetLastError();
    }
    std::cout << "New Min = " << (minSize / size) << " pages" << std::endl;
    std::cout << "New Max = " << (maxSize / size) << " pages" << std::endl;

    std::cin.get();
    return 0;
}
```

кінець прикладу 4

Приклад 5 – Визначення стану ділянки віртуальної пам'яті

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

int main()
{
    BYTE* a = nullptr;             // базова адреса області
    BYTE* b = nullptr;             // адреса підобласті
    const int shift = 5000;        // зміщення для підобласті
    const int size = 10000;        // розмір області

    MEMORY_BASIC_INFORMATION mbi;  // структура для інформації про віртуальну пам'ять
    SIZE_T mbiSize = sizeof(MEMORY_BASIC_INFORMATION);
    // розподіляємо віртуальну пам'ять
    a = (BYTE*)VirtualAlloc(
        NULL,
        size,
        MEM_COMMIT,
        PAGE_READWRITE);

    if (!a)
    {
        std::cout << "Virtual allocation failed." << std::endl;
        return GetLastError();
    }
    // встановлюємо адресу підобласті
    b = a + shift;

    // визначаємо інформацію про віртуальну пам'ять підобласті
    if (mbiSize != VirtualQuery(b, &mbi, mbiSize))
    {
        std::cout << "Virtual query failed." << std::endl;
        return GetLastError();
    }

    // виводимо інформацію про віртуальну пам'ять
    std::cout << "Base address: " << mbi.BaseAddress << std::endl;
    std::cout << "Allocation base: " << mbi.AllocationBase << std::endl;
    std::cout << "Allocation protect: " << mbi.AllocationProtect << std::endl;
    std::cout << "Region size: " << mbi.RegionSize << std::endl;
    std::cout << "State: " << mbi.State << std::endl;
    std::cout << "Protect: " << mbi.Protect << std::endl;
    std::cout << "Type: " << mbi.Type << std::endl;

    // звільняємо віртуальну пам'ять
    if (!VirtualFree(a, 0, MEM_RELEASE))
    {
        std::cout << "Memory release failed." << std::endl;
        return GetLastError();
    }

    std::cin.get();
    return 0;
}
```

кінець прикладу 5

Завдання для самостійного виконання

1. Розробити програми керування оперативною пам'яттю. Для програмної реалізації задач використовувати середовище розробки Microsoft Visual Studio. Програми розробляти у вигляді консольних застосунків Win32 (C++).

2. Скласти звіт про виконання лабораторної роботи. Зміст звіту: опис функцій для роботи з оперативною пам'яттю; постановка задачі; програмний код; результати виконання програм (скриншоти консолі).

3. При захисті лабораторної роботи підготувати відповіді на контрольні питання.

Контрольні питання

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

Лабораторна робота №7
Структура Windows-додатків. Завантаження операційної системи

Мета роботи: ознайомитися зі структурою Windows-додатків і організацією їхньої взаємодії з операційною системою, набути практичних навичок написання і налагодження програм, що містять опис класу вікна, віконну процедуру, обробку повідомлень; вивчити етапи завантаження операційної системи Windows та способи перегляду параметрів завантаження.

Зміст теоретичних відомостей:

1. Точка входу Windows-додатку. Функція WinMain.
2. Реєстрація класу вікна. Створення і відображення вікна.
3. Повідомлення та цикл їх обробки.
4. Віконна процедура. Основні повідомлення Windows.
5. Способи надсилання повідомлень: SendMessage і PostMessage.
6. Етапи завантаження операційної системи Windows.
7. Додаткові варіанти запуску (безпечний режим та інші).

Теоретичні відомості

1. Точка входу Windows-додатку. Функція WinMain.

Точкою входу Windows-додатку є функція WinMain, яка завжди має такий прототип:

```
int WINAPI WinMain(
    HINSTANCE hInstance,      // дескриптор поточного екземпляра додатку
    HINSTANCE hPrevInstance,  // застарілий параметр, завжди NULL
    LPSTR lpCmdLine,          // рядок параметрів командного рядка
    int nCmdShow);            // спосіб відображення вікна на екрані
```

Параметр hInstance – унікальне число, яке ідентифікує запущену копію (екземпляр) програми; якщо користувач запустить кілька копій однієї програми, кожна матиме своє значення hInstance.

Параметр nCmdShow визначає, у якому вигляді вікно повинне з'явитися на екрані спочатку (звичайний розмір, згорнуте, розгорнуте); для цього використовують іменовані константи, наприклад SW_SHOWNORMAL, а не числа.

2. Реєстрація класу вікна. Створення і відображення вікна.

Перед створенням вікна потрібно зареєструвати клас вікна – заповнити структуру WNDCLASSEX, яка описує загальні властивості всіх вікон цього класу (стиль, покажчик на віконну процедуру, значок, курсор, колір фону, ім'я класу), і передати її функції RegisterClassEx. Після успішної реєстрації класу створюють вікно функцією CreateWindow (або CreateWindowEx), яка повертає дескриптор створеного

вікна (HWND) – унікальне значення, за яким Windows надалі ідентифікує це вікно в усіх функціях, що працюють із ним.

На момент повернення з CreateWindow вікно вже існує всередині Windows, але ще не відображається на екрані. Для його показу викликають функцію ShowWindow (вказує, у якому вигляді показати вікно), а потім UpdateWindow (примушує вікно негайно перемалювати свою робочу область, надсилаючи йому повідомлення WM_PAINT).

3. Повідомлення та цикл їх обробки.

Після відображення вікна додаток входить у цикл обробки повідомлень – послідовно отримує події (натискання клавіш, рух миші, перемальовування тощо) у вигляді повідомлень із черги, яку для додатку формує Windows:

```
MSG msg;
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);  // перетворення кодів клавіш у символи
    DispatchMessage(&msg);   // передача повідомлення віконній процедурі
}
```

Кожне повідомлення описується структурою MSG, яка містить дескриптор вікна-отримувача (hwnd), номер повідомлення (message) та два додаткових параметри wParam і lParam, конкретний зміст яких залежить від типу повідомлення. Функція GetMessage повертає 0, коли з черги вилучене повідомлення WM_QUIT – це завершує цикл і програму.

4. Віконна процедура. Основні повідомлення Windows.

Функція DispatchMessage передає кожне повідомлення віконній процедурі (WndProc) – функції зворотного виклику, ім'я якої вказане в полі lpfnWndProc структури WNDCLASSEX. У ній за допомогою конструкції switch/case визначають, яке повідомлення надійшло, і обробляють його; усі необроблені повідомлення обов'язково передають функції DefWindowProc, яка виконує для них стандартні дії Windows:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{   switch (message)
    {
    case WM_CREATE:  /* обробка створення вікна */      return 0;
    case WM_PAINT:   /* обробка перемальовування */     return 0;
    case WM_DESTROY: /* обробка закриття вікна */       return 0;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);}
}
```

Найчастіше обробляють такі повідомлення:

- WM_CREATE – надходить одразу після створення вікна функцією

CreateWindow; зазвичай тут виконують одноразову ініціалізацію (наприклад, створення дочірніх елементів керування);

- WM_SIZE – надходить під час зміни розмірів вікна; нову ширину й висоту робочої області можна отримати макросами LOWORD(lParam) і HIWORD(lParam);
- WM_PAINT – сигналізує, що робочу область вікна (або її частину) потрібно перемалювати; обробку завжди починають викликом BeginPaint і завершують викликом EndPaint;
- WM_DESTROY – надходить, коли вікно видаляється з екрана (наприклад, користувач закрив його); стандартна реакція – викликати PostQuitMessage(0), що поміщає в черг повідомлення WM_QUIT і завершує цикл обробки повідомлень.

5. Способи надсилання повідомлень: SendMessage і PostMessage.

Повідомлення можуть надходити до віконної процедури двома способами. *Функція SendMessage викликає віконну процедуру безпосередньо і чекає, поки та обробить повідомлення (синхронний, блокувальний викликпотоку-відправника). Функція PostMessage, навпаки, лише ставить повідомлення в чергу повідомлень потоку-отримувача і відразу повертає керування, не чекаючи його обробки* (асинхронний, неблокувальний виклик). Більшість повідомлень, що генеруються самою системою в результаті дій користувача (клавіатура, миша, перемальовування, таймер), система ставить у черги повідомлень відповідних вікон.

Приклад 1 демонструє реєстрацію класу вікна, створення і відображення головного вікна, цикл обробки повідомлень та обробку повідомлень WM_PAINT і WM_DESTROY у віконній процедурі. Проєкт у Visual Studio створюється як Empty Project, до якого додається один файл .cpp; наявність у коді функції WinMain (а не main) лінкер визначає автоматично і збирає застосунок як звичайний Windows-додаток (GUI), а не як консольний.

Приклад 1 – Найпростіший Windows-додаток

```
#include <windows.h>
#include <cstring>

// Прототип віконної процедури
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    const char ClassName[] = "MyWindowClass";
    const char Title[] = "First Win32 Application";

    WNDCLASSEXA wc = {};
    wc.cbSize = sizeof(WNDCLASSEXA);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WndProc;
    wc.hInstance = hInstance;
```

```
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = ClassName;
    wc.hIconSm = NULL;

    if (!RegisterClassExA(&wc))
    {
        MessageBoxA(NULL,    "Window    class    registration    failed.",    "Error",
MB_ICONERROR);
        return 0;
    }

    HWND hWnd = CreateWindowExA(
        0, ClassName, Title, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, 500, 300,
        NULL, NULL, hInstance, NULL);

    if (!hWnd)
    {
        MessageBoxA(NULL, "Window creation failed.", "Error", MB_ICONERROR);
        return 0;
    }

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    MSG msg = {};
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return (int)msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_CREATE:
        return 0;

    case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hWnd, &ps);
        const char* text = "Hello, Win32!";
        TextOutA(hdc, 20, 20, text, (int)strlen(text));
        EndPaint(hWnd, &ps);
        return 0;
    }

    case WM_DESTROY:
```

```
        PostQuitMessage(0);
        return 0;

    default:
        return DefWindowProcA(hWnd, message, wParam, lParam);
    }
}
```

кінець прикладу 1

У прикладі всюди явно використано функції та структури з суфіксом A (WNDCLASSEXA, RegisterClassExA, CreateWindowExA, DefWindowProcA, MessageBoxA, TextOutA) – це гарантує коректну компіляцію з рядками типу char незалежно від налаштування Character Set проєкту (Unicode чи Multi-Byte), яке за замовчуванням у Visual Studio встановлене як Unicode.

6. Етапи завантаження операційної системи Windows.

Завантаження сучасної версії Windows (8/10/11) на комп'ютері з UEFI відбувається в кілька послідовних етапів:

1. самотестування обладнання (POST – Power-On Self-Test), яке виконує програмний код мікросхеми UEFI/BIOS;

2. мікропрограма UEFI зчитує завантажувач Windows (bootmgfw.efi) з системного розділу EFI (на застарілих системах з MBR/Legacy BIOS – bootmgr) і передає йому керування;

3. завантажувач Windows (Boot Manager) зчитує сховище конфігурації завантаження BCD (Boot Configuration Data) – реєстрову базу даних, яка визначає, яку операційну систему завантажувати за замовчуванням, і з яким тайм-аутом показувати меню вибору, якщо встановлено кілька систем;

4. запускається завантажувач операційної системи (winload.efi), який завантажує в пам'ять ядро (ntoskrnl.exe), рівень абстракції від обладнання (hal.dll), розділ реєстру SYSTEM та драйвери, позначені як такі, що запускаються на етапі завантаження (boot-start drivers);

5. ядро завершує власну ініціалізацію, запускається диспетчер сеансів (smss.exe), який готує підсистему Windows та запускає процес входу в систему (winlogon.exe) і диспетчер служб (services.exe);

6. після успішної автентифікації користувача завантажується його робочий стіл, і запускаються призначені для автозапуску програми.

Сховище BCD замінило застарілий текстовий файл boot.ini, що використовувався у Windows 2000/XP/2003; переглянути та редагувати його можна командою bcdedit, яку потрібно виконувати з командного рядка, запущеного з правами адміністратора.

7. Додаткові варіанти запуску (безпечний режим та інші).

Якщо звичайне завантаження неможливе або потрібна діагностика, Windows надає середовище відновлення (Windows RE) з додатковими варіантами запуску. Потрапити в нього можна через Параметри → Система → Відновлення → Особливі варіанти завантаження → Перезапустити зараз, або утримуючи клавішу Shift під час вибору Перезавантаження в меню Пуск. У розділі Startup Settings доступні, серед інших, такі режими:

- Безпечний режим (Safe Mode) – завантаження тільки з базовими драйверами (відеоадаптер, клавіатура, миша, накопичувачі), потрібними для роботи Windows;
- Безпечний режим із завантаженням мережних драйверів – те саме, але додатково запускаються мережні драйвери і служби;
- Безпечний режим із підтримкою командного рядка – замість графічного інтерфейсу Windows після завантаження відображається лише командний рядок;
- Увімкнути протоколювання завантаження – усі етапи завантаження драйверів записуються у файл %SystemRoot%\Ntbtlog.txt;
- Вимкнути автоматичний перезапуск після збою – дозволяє побачити на екрані повідомлення про критичну помилку (BSOD) замість автоматичного перезавантаження.

Завдання для самостійного виконання

Частина 1:

1. Розробити Windows-додаток засобами Win32 API (без використання MFC чи інших бібліотек-обгорток), що виконує реєстрацію класу вікна, створення головного вікна та цикл обробки повідомлень. У заголовку вікна вивести прізвище студента та номер групи. Обробити щонайменше повідомлення WM_CREATE, WM_PAINT (вивести в робочу область вікна текст або прості фігури засобами GDI відповідно до індивідуального варіанта) і WM_DESTROY, а також одне додаткове повідомлення на власний вибір (наприклад, WM_LBUTTONDOWN – змінити колір фону при клацанні мишею, або WM_KEYDOWN – виводити в заголовок вікна код натиснутої клавіші).
2. У звіті навести повний листинг програми з поясненням призначення кожного обробленого повідомлення та скриншот результату роботи програми.

Частина 2:

1. Відкрити командний рядок із правами адміністратора, виконати команду bcdedit /enum, навести в звіті отримані записи (ідентифікатор завантажувача, шлях, опис, тайм-аут) і пояснити призначення кожного поля.
2. Зайти в Особливі варіанти завантаження (Параметри → Система → Відновлення → Перезапустити зараз → Усунення несправностей → Додаткові параметри → Параметри запуску) і перерахувати в звіті всі доступні варіанти запуску з коротким поясненням призначення кожного.
3. У Журналі подій Windows (Просмотр событий → Журналы Windows →

Система) знайти подію, що відповідає останньому завантаженню операційної системи, і зафіксувати в звіті її час та джерело.

Контрольні питання*:*

1. Поясніть призначення параметрів функції WinMain.
2. Для чого призначена структура WNDCLASSEX і функція RegisterClassEx?
3. Що повертає функція CreateWindow і для чого використовується це значення?
4. З яких функцій складається відображення вже створеного вікна на екрані?
5. З яких полів складається структура повідомлення MSG?
6. Що таке віконна процедура і яке поле структури класу вікна на неї вказує?
7. За яких умов цикл обробки повідомлень завершує свою роботу?
8. Яка функція виконує стандартну обробку повідомлень, не оброблених віконною процедурою?
9. Чим відрізняється надсилання повідомлення функцією SendMessage від PostMessage?
10. Перелічіть основні етапи завантаження операційної системи Windows.
11. Яке призначення сховища конфігурації завантаження BCD і чим воно замінило файл boot.ini?
12. Які компоненти завантажуються на етапі роботи завантажувача операційної системи (winload)?
13. Які додаткові варіанти запуску Windows ви знаєте та для чого вони призначені?
14. Як отримати доступ до додаткових варіантів запуску в сучасних версіях Windows?

Лабораторна робота №8
Логічна і фізична організація файлових систем

Мета роботи: навчитися розробляти програми реалізації операцій з файлами та каталогами, а також програми міжпроцесової взаємодії через файлову систему (блокування файлів, відображувані у пам'ять файли, поіменовані канали).

Зміст теоретичних відомостей:

1. Основні файлові операції та їх реалізація засобами Win32 API.
2. Операції з каталогами.
3. Міжпроцесова взаємодія через файлову систему

Теоретичні відомості

1. Основні файлові операції та їх реалізація засобами Win32 API.

Робота прикладної програми з файлом передбачає такі основні операції: відкриття файлу (завантаження в пам'ять дескриптора файлу – структури даних з атрибутами та місцем розташування файлу на диску); створення файлу (новий файл нульової довжини, який автоматично відкривається); закриття файлу (вивільнення дескриптора, запис незбережених змін на диск); читання і записування даних, починаючи з поточної позиції у файлі; переміщення вказівника поточної позиції; отримання й задавання атрибутів файлу; вилучення файлу (недопустиме для відкритих файлів).

Відкриття і створення файлу. Виконує функція CreateFile, яка повертає дескриптор файлу (HANDLE) або значення INVALID_HANDLE_VALUE у разі помилки. Параметр режиму відкриття (amode) набуває значень GENERIC_READ і/або GENERIC_WRITE; параметр cmode визначає поведінку щодо наявного файлу: CREATE_NEW – створити, повернути помилку, якщо файл уже є; CREATE_ALWAYS – створити, перезаписавши наявний; OPEN_EXISTING – відкрити наявний, повернути помилку, якщо немає; OPEN_ALWAYS – відкрити наявний або створити новий.

```
HANDLE hFile = CreateFileA(
    "file.txt",          // ім'я файлу
    GENERIC_READ | GENERIC_WRITE,
    0,                    // спільний доступ заборонено
    NULL,                 // атрибути захисту за замовчуванням
    OPEN_ALWAYS,          // відкрити наявний або створити новий
    FILE_ATTRIBUTE_NORMAL,
    NULL);
```

Закриття, читання і записування. Дескриптор файлу закривають функцією CloseHandle. Функції ReadFile і WriteFile пересилають дані між файлом (починаючи

з поточної позиції) і буфером у пам'яті процесу, повертаючи кількість фактично оброблених байтів через вихідний параметр; обидві повертають TRUE в разі успіху.

```
DWORD bytesWritten;
WriteFile(hFile, buf, sizeof(buf), &bytesWritten, NULL);

DWORD bytesRead;
ReadFile(hFile, buf, sizeof(buf), &bytesRead, NULL);
// якщо bytesRead менше за запитаний розмір – досягнуто кінця файлу
```

Прямий доступ. Функція SetFilePointer переміщує вказівник поточної позиції на offset байтів від початку (FILE_BEGIN), поточної позиції (FILE_CURRENT) або кінця файлу (FILE_END), що дає змогу читати чи записувати довільний запис файлу фіксованої довжини без послідовного перебору попередніх.

Копіювання, переміщення, атрибути. Функції CopyFile і MoveFileEx виконують копіювання та перейменування/переміщення файлу засобами самої ОС (без явного читання-записування даних прикладною програмою). Атрибути файлу (розмір, час створення й останньої модифікації, ознака каталогу) отримують функцією GetFileAttributesEx, яка заповнює структуру WIN32_FILE_ATTRIBUTE_DATA.

2 Операції з каталогами.

Створення нового каталогу виконує функція CreateDirectory, вилучення порожнього каталогу – RemoveDirectory (для непорожнього каталогу повертає помилку). Обхід вмісту каталогу реалізують функціями FindFirstFile (відкриває пошук за маскою імені та повертає дескриптор пошуку, заповнюючи структуру WIN32_FIND_DATA даними про перший знайдений елемент) і FindNextFile (повертає наступний елемент, FALSE – якщо елементів більше немає); після завершення обходу дескриптор пошуку закривають функцією FindClose. Поточний робочий каталог процесу визначають і змінюють функціями GetCurrentDirectory та SetCurrentDirectory.

```
WIN32_FIND_DATAA fd;
HANDLE hFind = FindFirstFileA("c:\\mydir\\*.*", &fd);
do
{
    std::cout << fd.cFileName << std::endl;  // ім'я елемента каталогу
} while (FindNextFileA(hFind, &fd));
FindClose(hFind);
```

3. Міжпроцесова взаємодія через файлову систему.

Файлове блокування – засіб синхронізації процесів, які звертаються до одного файлу: поки блокування встановлене, інші процеси не можуть отримати доступ до заблокованої ділянки. У Win32 для цього використовують функцію LockFileEx (прапорець LOCKFILE_EXCLUSIVE_LOCK – блокування для запису; без нього – для читання; LOCKFILE_FAIL_IMMEDIATELY – не чекати звільнення, а відразу

повернути помилку, якщо ділянка вже заблокована) і відповідну їй UnlockFileEx для знімання блокування.

Технологія відображення файлу у пам'ять дає змогу працювати з його вмістом так, ніби він є частиною адресного простору процесу – через звичайні покажчики, без явних викликів ReadFile/WriteFile. Для цього спочатку функцією CreateFileMapping створюють об'єкт відображення для відкритого файлу, а потім функцією MapViewOfFile відображають файл (або його частину) на адресний простір процесу – функція повертає покажчик на початок відображеної ділянки. Після завершення роботи ділянку звільняють функцією UnmapViewOfFile, а дескриптор об'єкта відображення закривають функцією CloseHandle.

Переваги такого підходу: для доступу до даних достатньо одного системного викликання MapViewOfFile, а далі дані читають і змінюють напряму через покажчик, без копіювання між системними буферами і буфером режиму користувача; кілька процесів можуть відображати той самий файл, отримуючи спільний доступ до пам'яті. Головний недолік – розмір відображеного файлу не можна збільшити простим записом за його межі: підсистема віртуальної пам'яті в цьому разі не може визначити нову довжину файлу.

Поіменований канал (named pipe) – засіб двобічного обміну повідомленнями між процесами, у тому числі на різних комп'ютерах мережі, без використання звичайних файлів файлової системи. Сервер створює екземпляр каналу функцією CreateNamedPipe і чекає підключення клієнта функцією ConnectNamedPipe; після підключення обмін даними виконують уже знайомими функціями ReadFile і WriteFile. Завершивши роботу, сервер розриває з'єднання функцією DisconnectNamedPipe, а дескриптор каналу закриває функцією CloseHandle.

Приклади програмної реалізації

Приклад 1 – Записування і читання файлу

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

int main()
{
    const char* fileName = "demo_file.txt";

    HANDLE hFile = CreateFileA(fileName, GENERIC_WRITE, 0, NULL,
                               CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE)
    {
        std::cout << "Create file failed. Error: " << GetLastError() << std::endl;
        return 1;
    }

    for (int i = 0; i < 10; ++i)
    {
```

```
        DWORD bytesWritten;
        WriteFile(hFile, &i, sizeof(i), &bytesWritten, NULL);
    }
    CloseHandle(hFile);
    std::cout << "File written successfully." << std::endl;

    hFile = CreateFileA(fileName, GENERIC_READ, 0, NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE)
    {
        std::cout << "Open file failed. Error: " << GetLastError() << std::endl;
        return 1;
    }

    int value;
    DWORD bytesRead;
    std::cout << "File contents: ";
    while (ReadFile(hFile, &value, sizeof(value), &bytesRead, NULL) && bytesRead ==
sizeof(value))
    {
        std::cout << value << ' ';
    }
    std::cout << std::endl;

    CloseHandle(hFile);
    std::cin.get();
    return 0;
}
```

кінець прикладу 1

Приклад 2 – Копіювання і переміщення файлу

```
#include <windows.h>
#include <iostream>
int main()
{
if (!CopyFileA("demo_file.txt", "demo_file_copy.txt", FALSE))
{
std::cout << "Copy file failed. Error: " << GetLastError() << std::endl;
return 1;
}
std::cout << "File copied successfully." << std::endl;

if             (!MoveFileExA("demo_file_copy.txt",             "demo_file_moved.txt",
MOVEFILE_REPLACE_EXISTING))
{
std::cout << "Move file failed. Error: " << GetLastError() << std::endl;
return 1;
}
std::cout << "File moved successfully." << std::endl;

std::cin.get();
return 0;
}
```

кінець прикладу 2

Приклад 3 – Прямий доступ до файлу

```
#include <windows.h>
#include <iostream>
int main()
{
    HANDLE hFile = CreateFileA("demo_file.txt", GENERIC_READ, 0, NULL,
                               OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE)
    {
        std::cout << "Open file failed. Error: " << GetLastError() << std::endl;
        return 1;
    }

    int recordNumber;
    std::cout << "Enter record number (0-9): ";
    std::cin >> recordNumber;

    LONG offset = recordNumber * (LONG)sizeof(int);
    if (SetFilePointer(hFile, offset, NULL, FILE_BEGIN) == INVALID_SET_FILE_POINTER)
    {
        std::cout << "Set file pointer failed. Error: " << GetLastError() << std::endl;
        CloseHandle(hFile);
        return 1;
    }

    int value;
    DWORD bytesRead;
    ReadFile(hFile, &value, sizeof(value), &bytesRead, NULL);
    std::cout << "Record " << recordNumber << " = " << value << std::endl;

    CloseHandle(hFile);
    std::cin.get();
    return 0;
}
```

кінець прикладу 3

Приклад 4 – Створення каталогу, обхід вмісту та вилучення

```
#include <windows.h>
#include <iostream>
int main()
{
    const char* dirName = "demo_dir";

    if (!CreateDirectoryA(dirName, NULL))
    {
        std::cout << "Create directory failed. Error: " << GetLastError() <<
std::endl;
        return 1;
    }
    std::cout << "Directory created." << std::endl;

    WIN32_FIND_DATAA fd;
    HANDLE hFind = FindFirstFileA("*.*", &fd);
    if (hFind == INVALID_HANDLE_VALUE)
```

```
    {
        std::cout << "Find first file failed. Error: " << GetLastError() << std::endl;
        return 1;
    }

    do
    {
        if (!(fd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
        {
            std::cout << fd.cFileName << " (" << fd.nFileSizeLow << " bytes)" <<
std::endl;
        }
    } while (FindNextFileA(hFind, &fd));

    FindClose(hFind);

    if (!RemoveDirectoryA(dirName))
    {
        std::cout << "Remove directory failed. Error: " << GetLastError() <<
std::endl;
        return 1;
    }
    std::cout << "Directory removed." << std::endl;

    std::cin.get();
    return 0;
}
```

кінець прикладу 4

Приклад 5 – Відображення файлу у пам'ять

```
#include <windows.h>
#include <iostream>
int main()
{
    const DWORD size = 4096;

    HANDLE hFile = CreateFileA("mapped_file.txt", GENERIC_READ | GENERIC_WRITE, 0,
NULL,
                               CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE)
    {
        std::cout << "Create file failed. Error: " << GetLastError() << std::endl;
        return 1;
    }

    HANDLE hMap = CreateFileMappingA(hFile, NULL, PAGE_READWRITE, 0, size, NULL);
    if (!hMap)
    {
        std::cout << "Create file mapping failed. Error: " << GetLastError() <<
std::endl;
        CloseHandle(hFile);
        return 1;
    }

    int* data = (int*)MapViewOfFile(hMap, FILE_MAP_WRITE, 0, 0, size);
```

```
    if (!data)
    {
        std::cout << "Map view of file failed. Error: " << GetLastError() << std::endl;
        CloseHandle(hMap);
        CloseHandle(hFile);
        return 1;
    }

    // записуємо дані безпосередньо через відображену ділянку пам'яті
    data[0] = 100;
    data[1] = data[0] * 2;
    std::cout << "data[0] = " << data[0] << ", data[1] = " << data[1] << std::endl;

    UnmapViewOfFile(data);
    CloseHandle(hMap);
    CloseHandle(hFile);

    std::cin.get();
    return 0;
}
```

кінець прикладу 5

Приклад 6 – Файлове блокування

```
#include <windows.h>
#include <iostream>
int main()
{
    HANDLE hFile = CreateFileA("lock_demo.txt", GENERIC_READ | GENERIC_WRITE, 0, NULL,
                               OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE)
    {
        std::cout << "Create file failed. Error: " << GetLastError() << std::endl;
        return 1;
    }

    DWORD lockSize = 100;
    OVERLAPPED ov = { 0 };

    if (!LockFileEx(hFile, LOCKFILE_EXCLUSIVE_LOCK, 0, lockSize, 0, &ov))
    {
        std::cout << "Lock file failed. Error: " << GetLastError() << std::endl;
        CloseHandle(hFile);
        return 1;
    }
    std::cout << "File region locked. Press Enter to unlock..." << std::endl;
    std::cin.get();

    UnlockFileEx(hFile, 0, lockSize, 0, &ov);
    std::cout << "File region unlocked." << std::endl;

    CloseHandle(hFile);
    return 0;
}
```

кінець прикладу 6

Завдання для самостійного виконання*:*

Кожен студент виконує індивідуальне завдання – комбінацію з чотирьох завдань нижче відповідно **до номера свого варіанта (таблиця варіантів наведена після списку завдань)**.

1. Написати програму, яка створює новий текстовий файл і записує в нього послідовність із 10 цілих чисел, уведених користувачем.

2. Написати програму, яка читає послідовність цілих чисел зі створеного файлу та виводить їхню суму і середнє арифметичне.

3. Написати програму, яка копіює вказаний файл в інший каталог, виводячи повідомлення про успіх або код помилки GetLastError().

4. Написати програму, яка перейменовує (переміщує) файл у межах одного диска, перевіряючи код помилки.

5. Написати програму, яка визначає розмір файлу та час останньої модифікації засобами GetFileAttributesEx і виводить ці дані на екран.

6. Написати програму, яка створює новий каталог і підкаталог у ньому.

7. Написати програму, яка виводить список усіх файлів вказаного каталогу (ім'я, розмір) засобами FindFirstFile/FindNextFile.

8. Написати програму, яка вилучає з вказаного каталогу всі файли з певним розширенням, не змінюючи підкаталоги.

9. Написати програму, яка вилучає вказаний каталог разом з усіма файлами в ньому.

10. Написати програму, яка визначає поточний робочий каталог процесу, встановлює інший каталог поточним і виводить обидва значення.

11. Написати програму прямого читання довільного запису файлу фіксованої довжини за його номером (SetFilePointer).

12. Написати програму, яка встановлює виключне блокування (LockFileEx) на частину файлу, очікує введення з клавіатури, а потім знімає блокування (UnlockFileEx).

13. Написати програму, яка відображає файл у пам'ять (CreateFileMapping/MapViewOfFile), записує дані через відображену ділянку пам'яті та коректно звільняє ресурси.

Варіанти завдань (номери відповідають списку вище):

- Варіант 1, 11, 21: завдання 1, 6, 7, 13
- Варіант 2, 12, 22: завдання 2, 4, 8, 11
- Варіант 3, 13, 23: завдання 3, 5, 9, 10
- Варіант 4, 14, 24: завдання 1, 2, 7, 9
- Варіант 5, 15, 25: завдання 3, 4, 8, 11
- Варіант 6, 16, 26: завдання 4, 6, 10, 13
- Варіант 7, 17, 27: завдання 5, 6, 9, 10

- Варіант 8, 18, 28: завдання 2, 4, 7, 11
- Варіант 9, 19, 29: завдання 3, 5, 7, 13
- Варіант 10, 20, 30: завдання 1, 2, 8, 11

Контрольні питання*:*

1. Назвіть основні файлові операції та поясніть призначення кожної.
2. Які параметри приймає функція CreateFile? Що означають значення CREATE_NEW, CREATE_ALWAYS, OPEN_EXISTING, OPEN_ALWAYS?
3. Як виконується читання і записування даних у файл засобами Win32 API?
4. Як визначити, що під час читання досягнуто кінця файлу?
5. Як реалізувати прямий доступ до довільного запису файлу фіксованої довжини?
6. Якими функціями виконуються копіювання та переміщення файлу? Чим MoveFile відрізняється від MoveFileEx?
7. Як отримати атрибути файлу (розмір, час модифікації)?
8. Як створити, вилучити та обійти вміст каталогу засобами Win32 API?
9. Що таке файлове блокування і чим консультативне блокування відрізняється від обов'язкового?
10. Які функції використовують для встановлення та знімання файлового блокування у Win32?
11. У чому полягає ідея файлів, що відображаються у пам'ять? Які функції для цього використовують?
12. Назвіть переваги та недоліки роботи з відображуваними у пам'ять файлами.
13. Що таке поіменовані канали і для чого вони призначені?
14. Якими функціями реалізується створення поіменованого каналу, очікування підключення клієнта та обмін даними?

Лабораторна робота №9
Керування процесами та потоками

Мета роботи: навчитися керувати процесами і потоками засобами операційної системи, розробляти програми керування процесами і потоками.

Зміст теоретичних відомостей:

1. Складові елементи та структури даних процесу.
2. Створення і завершення процесів у Win32 API.
3. Потоки.

Теоретичні відомості

1. Складові елементи та структури даних процесу.

Процес – це контейнер ресурсів, потрібних для виконання програми. Адресний простір процесу складається з набору адрес віртуальної пам'яті, які може використовувати тільки цей процес; адресний простір недоступний іншим процесам. Процес також володіє системними ресурсами (відкритими файлами, мережними з'єднаннями, об'єктами синхронізації тощо) та містить стартову інформацію для потоків, які в ньому створюватимуться. Процес обов'язково містить хоча б один потік – без потоків виконання процесу неможливе.

Для виконавчої системи Windows процес зображується об'єктом-процесом (керуючим блоком процесу, EPROCESS), що містить ідентифікаційну інформацію (ідентифікатор процесу – pid; ідентифікатор процесу-предка; ім'я завантаженого програмного файлу), інформацію про адресний простір і доступні ресурси, а також блок оточення процесу (PEB), доступний у режимі користувача.

2. Створення і завершення процесів у Win32 API.

Створення нового процесу виконує функція CreateProcess, якій передають шлях до виконуваного файлу або повний командний рядок, атрибути безпеки, прапорці створення, та два покажчики на структури STARTUPINFO (параметри нового процесу) і PROCESS_INFORMATION (результат – дескриптори та ідентифікатори створеного процесу і його головного потоку). Структура PROCESS_INFORMATION містить поля hProcess, hThread, dwProcessId, dwThreadId.

```
STARTUPINFOA si;
PROCESS_INFORMATION pi;
ZeroMemory(&si, sizeof(si)); si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

char cmdLine[] = "notepad.exe";  // буфер має бути доступним для запису
CreateProcessA(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
```

Поточний процес завершує сам себе функцією ExitProcess(exitcode); інший процес (за його дескриптором) можна примусово завершити функцією TerminateProcess(hProcess, exitcode) – на відміну від ExitProcess, вона не дає процесу-жертві коректно звільнити власні ресурси, тому застосовується лише в аварійних випадках. Код завершення процесу, що вже завершився, читають функцією GetExitCodeProcess. Дескриптор і ідентифікатор поточного процесу повертають відповідно функції GetCurrentProcess() і GetCurrentProcessId().

3. Потоки

Потік виконується в межах адресного простору свого процесу і має: вміст набору регістрів процесора, що визначає його поточний стан; два стеки – один для роботи в режимі користувача, інший для режиму ядра (обидва розміщені в адресному просторі процесу-власника); локальну пам'ять потоку (TLS); власний унікальний ідентифікатор (tid), узятий з того самого простору імен, що й ідентифікатори процесів.

Потік створює функція CreateThread, якій передають покажчик на функцію потоку (вона обов'язково повинна мати сигнатуру DWORD WINAPI ім'я(LPVOID параметр)), додаткові дані для потоку та (за потреби) розмір стека. Функція повертає дескриптор створеного потоку, а через вихідний параметр – його ідентифікатор.

```
DWORD WINAPI MyThreadProc(LPVOID lpParam)
{
    // тіло потоку – використовує параметр lpParam
    return 0;
}

DWORD threadId;
HANDLE hThread = CreateThread(NULL, 0, MyThreadProc, lpParam, 0, &threadId);
```

Якщо тіло потоку викликає функції стандартної бібліотеки мови C (наприклад, malloc, strtok), замість CreateThread рекомендують використовувати функцію _beginthreadex з заголовка <process.h> – вона додатково готує для потоку окремий блок даних бібліотеки CRT; результат потрібно привести до типу HANDLE. Завершують такий потік функцією _endthreadex замість простого return.

Потік може завершитися самостійно, виконавши оператор return у своїй функції, або викликом _endthreadex (для потоків, створених через _beginthreadex). Інший потік можна тимчасово зупинити функцією SuspendThread і відновити функцією ResumeThread (обидві повертають попередній рахівник зупинок потоку); примусове завершення виконує функція TerminateThread – як і у випадку процесів, вона не дає потоку звільнити власні ресурси, тому має застосовуватися лише як крайній засіб. Дескриптор поточного потоку повертає функція GetCurrentThread(). Очікування завершення потоку (або процесу) реалізують функцією WaitForSingleObject(дескриптор, час_очікування), де час очікування INFINITE

означає необмежене чекання.

Приклади програмної реалізації

Приклад 1 – Створення потоку функцією CreateThread

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

DWORD WINAPI SumThread(LPVOID lpParam)
{
    int n = (int)(intptr_t)lpParam;
    long long sum = 0;
    for (int i = 1; i <= n; ++i)
        sum += i;
    std::cout << "Sum of 1.." << n << " = " << sum << std::endl;
    return 0;
}

int main()
{
    DWORD threadId;
    HANDLE hThread = CreateThread(NULL, 0, SumThread, (LPVOID)(intptr_t)1000, 0,
&threadId);
    if (!hThread)
    {
        std::cout << "Create thread failed. Error: " << GetLastError() << std::endl;
        return 1;
    }

    std::cout << "Waiting for thread to finish..." << std::endl;
    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hThread);

    std::cin.get();
    return 0;
}
```

кінець прикладу 1

Приклад 2 – Зупинка, відновлення та завершення потоку

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

volatile long counter = 0;

DWORD WINAPI CounterThread(LPVOID)
{
    while (true)
    {
        InterlockedIncrement(&counter);
        Sleep(100);
    }
    return 0;
```

```
}

int main()
{
    DWORD threadId;
    HANDLE hThread = CreateThread(NULL, 0, CounterThread, NULL, 0, &threadId);
    if (!hThread)
    {
        std::cout << "Create thread failed. Error: " << GetLastError() << std::endl;
        return 1;
    }

    char cmd;
    do
    {
        std::cout << "Command (c-count, s-suspend, r-resume, x-exit): ";
        std::cin >> cmd;
        switch (cmd)
        {
        case 'c':
            std::cout << "counter = " << counter << std::endl;
            break;
        case 's':
            SuspendThread(hThread);
            std::cout << "Thread suspended." << std::endl;
            break;
        case 'r':
            ResumeThread(hThread);
            std::cout << "Thread resumed." << std::endl;
            break;
        }
    } while (cmd != 'x');

    TerminateThread(hThread, 0);
    CloseHandle(hThread);
    return 0;
}
```

кінець прикладу 2

Приклад 3 – Запуск процесу та очікування його завершення

```
#include <windows.h>
#include <iostream>
int main()
{
    STARTUPINFOA si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    char cmdLine[] = "notepad.exe";
    if (!CreateProcessA(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
    {
        std::cout << "Create process failed. Error: " << GetLastError() <<
std::endl;
        return 1;
```

```
    }
     std::cout << "Process started, pid = " << pi.dwProcessId << std::endl;
    std::cout << "Waiting for it to close..." << std::endl;
    WaitForSingleObject(pi.hProcess, INFINITE);
    DWORD exitCode;
    GetExitCodeProcess(pi.hProcess, &exitCode);
    std::cout << "Process exit code: " << exitCode << std::endl;
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
    std::cin.get();
    return 0;
}
```

кінець прикладу 3

Приклад 4 – Створення потоку функцією _beginthreadex

```
#include <windows.h>
#include <process.h>
#include <iostream>
unsigned WINAPI ThreadFunc(void* param)
{
    int n = (int)(intptr_t)param;
    std::cout << "Thread started, param = " << n << std::endl;
    Sleep(500);
    std::cout << "Thread finished." << std::endl;
    _endthreadex(0);
    return 0;
}
int main()
{
    unsigned threadId;
    HANDLE hThread = (HANDLE)_beginthreadex(NULL, 0, ThreadFunc,
(void*)(intptr_t)42, 0, &threadId);
    if (!hThread)
    {
        std::cout << "Begin thread failed." << std::endl;
        return 1;
    }
     WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hThread);
    std::cin.get();
    return 0;
}
```

кінець прикладу 4

Завдання для самостійного виконання:

1. Написати програму, яка створює потік функцією CreateThread; потік повинен обчислювати суму чисел від 1 до N (N – параметр, переданий у потік) і виводити результат, а головний потік – очікувати завершення дочірнього потоку функцією WaitForSingleObject перед завершенням програми.

2. Написати програму з потоком, що в нескінченному циклі збільшує загальний рахівник із певним інтервалом (Sleep); головний потік має забезпечити інтерактивне

меню для перегляду поточного значення рахівника, зупинки (SuspendThread), відновлення (ResumeThread) і завершення (TerminateThread) цього потоку.

3. Написати програму, яка запускає новий процес (на вибір студента – стандартний застосунок Windows або власну скомпільовану програму) функцією CreateProcess, очікує його завершення та виводить код завершення процесу, отриманий функцією GetExitCodeProcess.

4. Написати програму, яка створює потік функцією _beginthreadex (замість CreateThread), передає йому довільний параметр і коректно завершує роботу через _endthreadex.

5. Скласти звіт про виконання лабораторної роботи: опис використаних функцій керування процесами і потоками, постановка задачі, повний листинг програм, результати виконання (скриншоти консолі).

6. При захисті лабораторної роботи підготувати відповіді на контрольні питання.

Контрольні питання:

1. Назвіть складові елементи процесу.

2. Які структури даних використовує операційна система для представлення процесу?

3. Як виконується створення процесу функцією CreateProcess? Охарактеризуйте її основні параметри.

4. Яка структура заповнюється результатом виклику CreateProcess і які поля вона містить?

5. Яка різниця між функціями ExitProcess і TerminateProcess?

6. Як отримати дескриптор і ідентифікатор поточного процесу?

7. Як отримати код завершення вже завершеного процесу?

8. Назвіть складові елементи потоку.

9. Як виконується створення потоку функцією CreateThread? Якою має бути сигнатура функції потоку?

10. Чим відрізняється створення потоку функцією _beginthreadex від CreateThread і коли варто використовувати кожну з них?

11. Якими способами потік може завершити власне виконання?

12. Як виконується зупинка та відновлення потоку? Охарактеризуйте відповідні функції.

13. Чим небезпечне примусове завершення потоку чи процесу функціями TerminateThread/TerminateProcess?

14. Як організувати очікування завершення потоку чи процесу з боку іншого потоку?

Лабораторна робота №10
Створення паралельних процесів та потоків засобами Windows API

Мета роботи: навчитися використовувати засоби Windows API для створення кількох паралельно виконуваних процесів і потоків, а також для синхронізації їх завершення.

Зміст теоретичних відомостей:

1. Поняття процесу і потоку, їх співвідношення.
2. Створення кількох паралельних потоків та очікування їх завершення.
3. Контроль завершення процесу без блокування викликаючого потоку.
4. Пошук даних у текстовому файлі засобами стандартної бібліотеки C++.

Теоретичні відомості

1. Поняття процесу і потоку, їх співвідношення.

Будь-яка робота обчислювальної системи полягає у виконанні певної програми, для чого потрібно завантажити її код у пам'ять, виділити ресурси введення-виведення та надати процесорний час. В операційних системах процес розглядається як заявка на споживання всіх видів ресурсів, окрім процесорного часу, який розподіляється між меншими одиницями роботи – потоками.

Розпаралелити обчислення в межах однієї програми можна було б, створюючи для кожної паралельної гілки роботи окремий процес. Однак стандартні засоби створення процесів не враховують, що ці процеси розв'язують одну спільну задачу: операційна система ізолює кожен процес власним адресним простором і витрачає системні ресурси на дублювання того, що насправді мало б бути спільним (дані, сегмент коду, права доступу). Потоки вирішують цю проблему: усі потоки одного процесу спільно використовують його адресний простір, файли, таймери та інші ресурси, а керівна структура – лише власний контекст регістрів, стек і локальну пам'ять потоку (TLS). Тому створення потоку обходиться операційній системі значно дешевше, ніж створення процесу.

Найбільший ефект багатопотокова обробка дає на багатопроцесорних / багатоядерних системах, де потоки одного процесу можуть виконуватися дійсно паралельно (а не лише псевдопаралельно завдяки перемиканню контексту на одному процесорі), завдяки підтримці симетричної багатопроцесорної обробки (Symmetric Multiprocessing, SMP).

2. Створення кількох паралельних потоків та очікування їх завершення.

Створення окремого потоку функцією CreateThread детально розглянуто в лабораторній роботі №9. Якщо потрібно запустити одразу кілька потоків, що виконуються паралельно, дескриптори всіх створених потоків зберігають у масиві, а

для одночасного очікування завершення всіх (або хоча б одного) з них використовують функцію WaitForMultipleObjects замість послідовних викликів WaitForSingleObject для кожного потоку окремо.

```
DWORD WaitForMultipleObjects(
    DWORD nCount,        // кількість дескрипторів у масиві
    const HANDLE* lpHandles,
    BOOL bWaitAll,       // TRUE – чекати на всі; FALSE – на будь-який перший
    DWORD dwMilliseconds);
```

Якщо bWaitAll дорівнює TRUE, функція повертає керування лише тоді, коли всі вказані об'єкти перейшли в сигнальний стан (для потоків і процесів це означає завершення); якщо FALSE – як тільки сигнальним стане хоча б один із них.

3. Контроль завершення процесу без блокування викликаючого потоку.

Очікування завершення процесу функцією WaitForSingleObject (розглянуте в лабораторній роботі №9) блокує потік, що чекає, до самого завершення процесу-нащадка. Якщо потрібно водночас виконувати іншу роботу, замість блокувального очікування використовують періодичну перевірку коду завершення функцією GetExitCodeProcess: поки процес ще виконується, ця функція повертає в lpExitCode спеціальне значення STILL_ACTIVE; після завершення процесу – його справжній код завершення.

4 Пошук слова у текстовому файлі

Для самостійного завдання потрібно реалізувати пошук заданого слова в текстовому файлі, створеному програмою Notepad. Найпростіший спосіб – почитати файл символ за символом і виконати пошук підрядка функцією std::string::find – але такий пошук знайде слово "фон" навіть у тексті "телефон", оскільки шукає підрядок, а не окреме слово. Щоб коректно знаходити саме слово (відокремлене пробілами чи розділовими знаками), файл розбивають на окремі токени за допомогою потокового виділення (operator>> для std::string або std::istream_iterator<std::string>) і кожен токен порівнюють із шуканим словом повністю, а не як підрядок.

Приклади програмної реалізації

Приклад 1 – Запуск кількох паралельних потоків з різними параметрами

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

DWORD WINAPI Worker(LPVOID lpParam)
{
    int id = (int)(intptr_t)lpParam;
    std::cout << "Thread " << id << " started." << std::endl;
    Sleep(300 + id * 100);
    std::cout << "Thread " << id << " finished." << std::endl;
```

```
    return 0;
}

int main()
{
    const int N = 4;
    HANDLE threads[N];

    for (int i = 0; i < N; ++i)
    {
        DWORD threadId;
        threads[i] = CreateThread(NULL, 0, Worker, (LPVOID)(intptr_t)i, 0,
&threadId);
        if (!threads[i])
        {
            std::cout << "Create thread failed. Error: " << GetLastError() <<
std::endl;
            return 1;
        }
    }

    std::cout << "Waiting for all threads to finish..." << std::endl;
    WaitForMultipleObjects(N, threads, TRUE, INFINITE);

    for (int i = 0; i < N; ++i)
        CloseHandle(threads[i]);

    std::cout << "All threads finished." << std::endl;
    std::cin.get();
    return 0;
}
```

кінець прикладу 1

Приклад запускає Notepad, який відкриває файл first.txt (файл має існувати в каталозі програми, або вкажіть повний шлях до нього), і кожну секунду перевіряє, чи цей процес ще виконується, замість того, щоб заблокувати головний потік до його завершення.

Приклад 2 – Запуск процесу з періодичною перевіркою стану завершення

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

int main()
{
    STARTUPINFOA si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    char cmdLine[] = "notepad.exe first.txt";
    if (!CreateProcessA(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
    {
        std::cout << "Create process failed. Error: " << GetLastError() <<
```

```
std::endl;
        return 1;
    }

    std::cout << "Process started, pid = " << pi.dwProcessId << std::endl;

    DWORD exitCode;
    do
    {
        GetExitCodeProcess(pi.hProcess, &exitCode);
        if (exitCode == STILL_ACTIVE)
        {
            std::cout << "Process is still running..." << std::endl;
            Sleep(1000);
        }
    } while (exitCode == STILL_ACTIVE);

    std::cout << "Process finished with exit code: " << exitCode << std::endl;

    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
    return 0;
}
```

кінець прикладу 2

На відміну від поширеного в старих підручниках варіанта з пошуком підрядка функцією find (який помилково знаходить "фон" усередині "телефон"), цей приклад розбиває вміст файлу на окремі слова й порівнює їх цілком.

Приклад 3 – Пошук слова у текстовому файлі

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

int main()
{
    std::ifstream file("first.txt");
    if (!file)
    {
        std::cout << "Cannot open file." << std::endl;
        return 1;
    }

    std::stringstream buffer;
    buffer << file.rdbuf();
    file.close();

    std::string word;
    std::cout << "Enter a word to find: ";
    std::cin >> word;

    std::istringstream stream(buffer.str());
    std::string token;
```

```
    bool found = false;
    int position = 0;

    while (stream >> token)
    {
        if (token == word)
        {
            found = true;
            break;
        }
        ++position;
    }

    if (found)
        std::cout << "Word found, token number " << position << std::endl;
    else
        std::cout << "Word not found." << std::endl;

    return 0;
}
```

кінець прикладу 3

Завдання для самостійного виконання:

Кожен студент виконує завдання відповідно до номера свого варіанта (N – номер варіанта).

1. Модифікувати приклад 1 так, щоб запускалося N+2 паралельних потоків (а не фіксовані 4), кожен з яких отримує власний унікальний параметр і виводить індивідуальне повідомлення зі своїм номером на початку та в кінці виконання; переконатися, що головний потік коректно дочікується завершення всіх потоків через WaitForMultipleObjects.

2. Модифікувати приклад 2 так, щоб замість Notepad запускався інший застосунок, встановлений у системі (на вибір студента), і вивести час (у секундах), протягом якого цей процес виконувався, до моменту його завершення.

3. За допомогою програми Notepad створити текстовий файл first.txt, що містить щонайменше 30 слів на тему, обрану студентом. Модифікувати приклад 3 так, щоб програма виводила не лише факт знаходження слова, а й загальну кількість його входжень у файл.

4. Скласти звіт про виконання лабораторної роботи: тема, мета, аналіз завдання згідно з варіантом, повний текст програм та результати їх роботи, висновки.

5. При захисті лабораторної роботи підготувати відповіді на контрольні питання.

Контрольні питання:

1. Чому для розпаралелювання обчислень в межах однієї програми потоки ефективніші за створення кількох окремих процесів?

2. Які ресурси потоки одного процесу використовують спільно, а які має кожен

потік власні?

3. Що таке симетрична багатопроцесорна обробка (SMP) і як вона впливає на ефективність багатопотокових застосунків?

4. Як організувати одночасне очікування завершення кількох потоків чи процесів? Охарактеризуйте функцію WaitForMultipleObjects та її параметри.

5. Чим відрізняється параметр bWaitAll, рівний TRUE, від значення FALSE у функції WaitForMultipleObjects?

6. Як перевірити, чи процес ще виконується, без блокування потоку, що його запустив?

7. Яке значення повертає GetExitCodeProcess, якщо процес ще не завершився? Що це значення означає?

8. У чому недолік пошуку слова у файлі за допомогою пошуку підрядка (substring) і як цього недоліку уникнути?

9. Як розбити вміст текстового файлу на окремі слова засобами стандартної бібліотеки C++?

10. Які прапорці способу створення процесу (dwCreationFlags) ви знаєте? Назвіть призначення хоча б трьох із них.

Лабораторна робота №11
Планування процесів та потоків

Мета роботи: навчитися планувати виконання процесів і потоків засобами операційної системи, розробляти програми керування пріоритетами процесів і потоків.

Зміст теоретичних відомостей:

1. Алгоритми планування процесів і потоків.
2. Реалізація планування потоків у Windows (пріоритети, динамічна зміна пріоритету).
3. Функції Win32 API для роботи з пріоритетами процесів і потоків.

Теоретичні відомості

1. Алгоритми планування процесів і потоків.

Виконання потоку – це цикл чергування інтервалів використання процесора (CPU burst) та інтервалів очікування введення-виведення (I/O burst). Коли черговий інтервал використання процесора завершується (потік заблокувався на операції введення-виведення, вичерпав квант часу або завершив роботу), планувальник повинен обрати, який із потоків у черзі готових отримає процесор далі. Розглянемо основні класичні алгоритми такого вибору.

Планування за принципом FIFO. Чергу готових потоків організовують за принципом «першим прийшов – першим обслужений» (First In, First Out). Коли в системі створюється новий потік, його керуючий блок додається у хвіст черги; коли процесор звільняється, його надають потоку з голови черги, і цей потік виконується доти, доки сам не заблокується або не завершиться (тобто алгоритм непереривний, без витіснення за часом).

Наприклад, якщо потоки A, B, C надійшли в чергу готовності в такому порядку і кожному потрібно по 5 одиниць часу процесора, A виконуватиметься перші 5 одиниць часу, далі B – наступні 5, і лише потім C. Головний недолік FIFO – «ефект конвою» (convoy effect): якщо потік A, що виконується першим, має дуже довгий інтервал використання процесора, усі наступні потоки (навіть дуже короткі) будуть змушені довго чекати в черзі.

Кругове планування (Round Robin). Кожному потокові виділяють інтервал часу (квант часу), протягом якого йому дозволено виконуватися. Коли потік усе ще виконується після вичерпання кванта часу, його перериває таймер, і процесор передають наступному потоку в черзі (а перерваний потік повертається у хвіст черги); коли потік блокується або завершує виконання раніше вичерпання кванта, процесор теж негайно передають наступному потокові. Довжина кванта часу однакова для всієї системи.

Кругове планування усуває «ефект конвою», властивий FIFO, забезпечуючи прийнятний час відгуку для всіх потоків. Розмір кванта часу – важливий параметр: занадто малий квант спричиняє надто часте перемикання контексту (а воно саме коштує процесорного часу), занадто великий – кругове планування фактично перетворюється на FIFO.

Планування з пріоритетами. Кожному потокові надають пріоритет, і на виконання завжди ставиться потік із найвищим пріоритетом серед готових до виконання. Пріоритети можуть надаватися потокам статично (заданим один раз під час створення) або динамічно (змінюватися під час виконання залежно від поведінки потоку, наприклад, від частки часу, яку він витрачає на очікування введення-виведення).

Розподіл пріоритетів може призвести до того, що потоки з низьким пріоритетом чекатимуть на процесор дуже довго, якщо в системі завжди є потоки з вищим пріоритетом, готові до виконання. Таку ситуацію називають голодуванням (starvation). Типове рішення – динамічне підвищення пріоритету потоків, які довго очікують на процесор (детальніше розглянуто в розділі 2 для випадку Windows).

Планування на підставі характеристик подальшого виконання:

− Алгоритм «перший – із найкоротшим часом виконання» (Shortest-Time-to-Completion-First, STCF). З кожним потоком пов'язують (оцінену або відому) тривалість наступного інтервалу використання процесора, і для виконання щоразу вибирають потік, у якого ця тривалість найкоротша. У результаті потоки, що захоплюють процесор на короткий час, отримують перевагу під час планування й швидше виходять із системи, що в середньому мінімізує час очікування в черзі готовності для всіх потоків.

− Алгоритм «перший – із найкоротшим часом виконання, що залишився» (SRTCF) відрізняється від STCF тим, що є витіснювальним: коли в чергу готових потоків додають новий потік, наступний інтервал використання процесора якого коротший, ніж час, що залишився до завершення виконання поточного потоку, поточний потік переривається, і на його місце негайно стає новий, коротший потік.

Багаторівневі черги зі зворотним зв'язком. Замість однієї черги готових потоків підтримують кілька черг із різним пріоритетом; черги з нижчим пріоритетом обслуговують лише тоді, коли всі черги вищого рівня порожні. Усередині кожної черги застосовують кругове планування (а в черзі найнижчого рівня – звичайний FIFO). Потокам дозволено переходити з черги в чергу: якщо потік вичерпав свій квант часу, не заблокувавшись, його переміщують у чергу з нижчим пріоритетом (це типова ознака обчислювально інтенсивного потоку); якщо ж потік систематично блокується на введенні-виведенні, не встигаючи вичерпати квант, він залишається в черзі з високим пріоритетом. Таким чином інтерактивні потоки (які часто чекають на введення користувача) природним чином отримують вищий пріоритет за обчислювальні. Щоб уникнути голодування потоків у нижніх черг, передбачають

періодичне автоматичне підвищення пріоритету потоків, які давно не отримували керування.

Лотерейне планування. Кожному потокові видають певну кількість «лотерейних квитків», що дають право час від часу користуватися процесором. Через рівні проміжки часу планувальник випадковим чином обирає один квиток-переможець; потік, якому належить цей квиток, отримує керування до наступного розіграшу. Чим більше квитків має потік, тим вища ймовірність, що саме він виграє розіграш, а отже й більша частка процесорного часу, яку він отримає в середньому – це дає змогу гнучко й пропорційно розподіляти процесорний час між потоками, просто змінюючи кількість виданих їм квитків, без потреби обчислювати точні пріоритети чи квоти.

2. Планування потоків у Windows.

Базовою одиницею планування в ядрі Windows є потік: під час вибору наступного потоку для виконання ядро не розрізняє, якому процесу належить потік, а оперує лише пріоритетами потоків, готових до виконання в певний момент часу. Кожному потоку присвоюють числовий пріоритет від 1 до 31 (більше число – вищий пріоритет); рівні 16-31 – пріоритети реального часу, зарезервовані системою для дій, час виконання яких є критичним чинником; рівні 1-15 – динамічні пріоритети, які можуть бути присвоєні потокам застосунків користувача.

Спочатку процесу присвоюють клас пріоритету: реального часу (real-time, відповідає базовому пріоритету потоку 24), високий (high, 13), нормальний (normal, 8), невикористовуваний (idle, 4). Потім кожному потоку цього процесу присвоюють відносний пріоритет, який відраховується від класу пріоритету процесу (базового пріоритету): найвищий (+2 до базового), вище за нормальний (+1), нормальний (дорівнює базовому), нижче за нормальний (-1), найнижчий (-2). Під час виконання відносний пріоритет потоку може динамічно змінюватися в межах динамічного діапазону (1-15).

Пошук потоку для виконання. Планувальник підтримує спеціальну структуру даних – список готових потоків, що складається з 31 елемента (по одному для кожного рівня пріоритету); з кожним елементом пов'язана черга готових потоків того самого рівня. Новий потік для виконання вибирають, коли: минув квант часу поточного потоку (запускається алгоритм пошуку готового потоку); поточний потік перейшов у стан очікування події (він віддає квант часу, і планувальник негайно запускає пошук готового потоку); чи коли потік перейшов у стан готовності до виконання (запускається алгоритм розміщення готового потоку).

Алгоритм пошуку готового потоку переглядає черги списку готових потоків, починаючи з черги найвищого пріоритету (31) і рухаючись до нижчих рівнів; щойно під час перегляду трапляється непорожня черга, з її голови вибирають потік для виконання. Алгоритм розміщення готового потоку, навпаки, спрацьовує щоразу, коли

якийсь потік переходить у стан готовності: спочатку перевіряють, чи новий потік не має вищого пріоритету за той, що виконується в цей момент; якщо так – новий потік негайно починає виконуватися, а потік, що виконувався, повертається в чергу готовності свого рівня; якщо ні – новий потік просто додається в чергу готовності, що відповідає його пріоритету.

Динамічна зміна пріоритету і кванта часу (підтримка). Підтримка (boost) – тимчасове підвищення динамічного пріоритету потоку. Коли потік переходить у стан готовності внаслідок настання очікуваної ним події (наприклад, завершення операції введення-виведення), виконують операцію підтримки; конкретна величина підвищення залежить від типу події (наприклад, завершення операції введення-виведення з диска підвищує пріоритет на меншу величину, ніж завершення операції введення з клавіатури чи миші, що критично для відчутної інтерактивності). Вихід з будь-якого стану очікування для потоків інтерактивних застосунків (тих, що пов'язані з відображенням інтерфейсу користувача і отримали фокус) підвищує пріоритет на 2.

Унаслідок операцій підтримки динамічний пріоритет потоку не може перевищити значення 15 (тобто вийти за межі діапазону динамічних пріоритетів у зону пріоритетів реального часу). Після закінчення кожного кванта часу поточний пріоритет потоку зменшують на одиницю, поки він не повернеться до базового рівня, після чого пріоритет залишають на одному рівні до наступної операції підтримки. Окремим видом підтримки є тимчасове збільшення самого кванта часу для потоків інтерактивного застосунку, що отримав фокус уведення – це дає такому застосунку змогу довше утримувати процесor без додаткових перемикань контексту, поліпшуючи відчуття швидкодії для користувача.

Якщо в системі постійно є готові до виконання потоки з високим пріоритетом, потоки з нижчим пріоритетом можуть голодувати – нескінченно довго очікувати на процесор. Щоб уникнути цього, спеціальний потік ядра один раз за секунду обходить чергу готових потоків у пошуках тих, що перебували в стані готовності досить довго (понад приблизно 3 с), жодного разу не отримавши шансу на виконання. Коли такий потік знайдено, йому тимчасово присвоюють пріоритет 15 (тобто максимально можливий серед динамічних – це дає змогу йому негайно отримати процесор) і подвоюють довжину його кванта часу. Після того як два кванти часу минуть, пріоритет потоку і довжина його кванта повертаються до вихідних значень.

3. Функції Win32 API для роботи з пріоритетами.

Клас пріоритету процесу читають і змінюють функціями GetPriorityClass/SetPriorityClass; відносний пріоритет потоку – функціями GetThreadPriority/SetThreadPriority (приймають іменовані константи THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_NORMAL, THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST). Режимом динамічної підтримки пріоритету

керують окремо для процесу (Get/SetProcessPriorityBoost) і для потоку (Get/SetThreadPriorityBoost) – значення TRUE вимикає динамічну підтримку, FALSE – увімкнена (типова поведінка).

```
BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriorityClass);
DWORD GetPriorityClass(HANDLE hProcess);

BOOL SetThreadPriority(HANDLE hThread, int nPriority);
int GetThreadPriority(HANDLE hThread);

BOOL SetProcessPriorityBoost(HANDLE hProcess, BOOL bDisablePriorityBoost);
BOOL GetProcessPriorityBoost(HANDLE hProcess, PBOOL pDisablePriorityBoost);
BOOL SetThreadPriorityBoost(HANDLE hThread, BOOL bDisablePriorityBoost);
BOOL GetThreadPriorityBoost(HANDLE hThread, PBOOL pDisablePriorityBoost);
```

Приклади програмної реалізації

Приклад 1 – Перегляд і зміна класу пріоритету процесу

```
#include <windows.h>
#include <iostream>
int main()
{
    HANDLE hProcess = GetCurrentProcess();

    DWORD priorityClass = GetPriorityClass(hProcess);
    std::cout << "Current priority class: " << priorityClass << std::endl;

    if (!SetPriorityClass(hProcess, IDLE_PRIORITY_CLASS))
    {
        std::cout << "Set priority class failed. Error: " << GetLastError() <<
std::endl;
        return 1;
    }
    priorityClass = GetPriorityClass(hProcess);
    std::cout << "New priority class (IDLE): " << priorityClass << std::endl;

    if (!SetPriorityClass(hProcess, NORMAL_PRIORITY_CLASS))
    {
        std::cout << "Set priority class failed. Error: " << GetLastError() <<
std::endl;
        return 1;
    }
    priorityClass = GetPriorityClass(hProcess);
    std::cout << "Restored priority class (NORMAL): " << priorityClass << std::endl;

    return 0;
}
```

кінець прикладу 1

Приклад 2 – Перегляд і зміна пріоритету потоку

```
#include <windows.h>
#include <iostream>
int main()
```

```
{
    HANDLE hThread = GetCurrentThread();

    int priority = GetThreadPriority(hThread);
    std::cout << "Current thread priority: " << priority << std::endl;

    if (!SetThreadPriority(hThread, THREAD_PRIORITY_LOWEST))
    {
        std::cout << "Set thread priority failed. Error: " << GetLastError() <<
std::endl;
        return 1;
    }
    priority = GetThreadPriority(hThread);
    std::cout << "New thread priority (LOWEST): " << priority << std::endl;

    if (!SetThreadPriority(hThread, THREAD_PRIORITY_HIGHEST))
    {
        std::cout << "Set thread priority failed. Error: " << GetLastError() <<
std::endl;
        return 1;
    }
    priority = GetThreadPriority(hThread);
    std::cout << "New thread priority (HIGHEST): " << priority << std::endl;

    return 0;
}
```

кінець прикладу 2

Приклад 3 – Керування динамічною підтримкою пріоритету

```
#include <windows.h>
#include <iostream>
int main()
{
    HANDLE hProcess = GetCurrentProcess();
    HANDLE hThread = GetCurrentThread();
    BOOL boost;

    GetProcessPriorityBoost(hProcess, &boost);
    std::cout << "Process priority boost disabled: " << boost << std::endl;

    SetProcessPriorityBoost(hProcess, TRUE);   // TRUE - вимкнути підтримку
    GetProcessPriorityBoost(hProcess, &boost);
    std::cout << "After disabling, value = " << boost << std::endl;

    SetProcessPriorityBoost(hProcess, FALSE);  // FALSE - увімкнути підтримку
(типово)
    GetProcessPriorityBoost(hProcess, &boost);
    std::cout << "After enabling, value = " << boost << std::endl;

    GetThreadPriorityBoost(hThread, &boost);
    std::cout << "Thread priority boost disabled: " << boost << std::endl;
    return 0;
}
```

кінець прикладу 3

Приклад створює два потоки з різними пріоритетами (найнижчим і найвищим), обидва протягом 2 секунд у нескінченному циклі збільшують власний рахівник, після чого порівнюються підсумкові значення рахівників – потік із вищим пріоритетом має встигнути виконати більше ітерацій. Зверніть увагу: на багатоядерній непослідовно завантаженій системі різниця може бути малою або відсутньою, оскільки обидва потоки можуть виконуватися одночасно на різних ядрах без конкуренції за процесор; найпомітніший ефект пріоритету спостерігається на однопроцесорній системі або під значним навантаженням.

Приклад 4 – Вплив пріоритету потоку на розподіл процесорного часу

```
#include <windows.h>
#include <iostream>
volatile long lowCounter = 0;
volatile long highCounter = 0;
volatile bool running = true;
DWORD WINAPI LowPriorityWorker(LPVOID)
{
    while (running)
        InterlockedIncrement(&lowCounter);
    return 0;
}
DWORD WINAPI HighPriorityWorker(LPVOID)
{
    while (running)
        InterlockedIncrement(&highCounter);
    return 0;
}
int main()
{
    DWORD id1, id2;
    HANDLE h1 = CreateThread(NULL, 0, LowPriorityWorker, NULL, CREATE_SUSPENDED,
&id1);
    HANDLE h2 = CreateThread(NULL, 0, HighPriorityWorker, NULL, CREATE_SUSPENDED,
&id2);
    SetThreadPriority(h1, THREAD_PRIORITY_LOWEST);
    SetThreadPriority(h2, THREAD_PRIORITY_HIGHEST);

    ResumeThread(h1);
    ResumeThread(h2);

    Sleep(2000);
    running = false;
    WaitForSingleObject(h1, INFINITE);
    WaitForSingleObject(h2, INFINITE);
    std::cout << "Low priority thread iterations:  " << lowCounter << std::endl;
    std::cout << "High priority thread iterations: " << highCounter << std::endl;
    CloseHandle(h1);
    CloseHandle(h2);

    return 0;
}
```

кінець прикладу 4

Завдання для самостійного виконання:

1. Написати програму, яка визначає поточний клас пріоритету процесу, встановлює клас HIGH_PRIORITY_CLASS, виводить підтвердження зміни, а потім повертає клас NORMAL_PRIORITY_CLASS.

2. Написати програму, яка визначає поточний пріоритет потоку, послідовно встановлює і виводить усі п'ять значень відносного пріоритету потоку (від THREAD_PRIORITY_LOWEST до THREAD_PRIORITY_HIGHEST).

3. Модифікувати приклад 4 так, щоб створювалося три потоки з різними пріоритетами (наприклад, THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_NORMAL, THREAD_PRIORITY_HIGHEST); вивести підсумкові значення рахівників усіх трьох потоків та зробити висновок про вплив пріоритету на розподіл процесорного часу на конкретному комп'ютері.

4. Скласти звіт про виконання лабораторної роботи: опис використаних функцій керування пріоритетами, постановка задачі, повний листинг програм, результати виконання (скриншоти консолі) і висновки щодо спостереженого впливу пріоритету на черговість виконання потоків.

5. При захисті лабораторної роботи підготувати відповіді на контрольні питання.

Контрольні питання:

1. Охарактеризуйте алгоритми планування: планування за принципом FIFO, кругове планування, планування з пріоритетами.

2. Охарактеризуйте алгоритми планування на підставі характеристик подальшого виконання (STCF, SRTCF), багаторівневі черги зі зворотним зв'язком, лотерейне планування.

3. Як виконується планування потоків у ядрі Windows? Охарактеризуйте систему пріоритетів потоків і процесів.

4. Що таке клас пріоритету процесу і відносний пріоритет потоку? Як вони пов'язані між собою?

5. Що таке динамічна підтримка (boost) пріоритету і коли вона застосовується?

6. Як у Windows запобігають голодуванню потоків з низьким пріоритетом?

7. Як виконується зміна та визначення класу пріоритету процесу? Охарактеризуйте відповідні функції Win32 API.

8. Як виконується завдання та визначення відносного пріоритету потоку? Охарактеризуйте відповідні функції Win32 API.

9. Якими функціями керують режимом динамічної підтримки пріоритету окремо для процесу і для потоку?

10. За яких умов вплив пріоритету потоку на швидкість його виконання буде найпомітнішим, а за яких – малопомітним?

Лабораторна робота №12
Використання семафорів. М'ютекси

Мета роботи: навчитися керувати взаємодією потоків в середовищі операційної системи, застосовуючи семафори та м'ютекси, розробляти програми синхронізації потоків.

Зміст теоретичних відомостей:

1. Поняття семафора, операції down і up.
2. Задача «виробник-споживач» та її розв'язання за допомогою семафорів.
3. Поняття м'ютекса, відмінність від семафора.

Теоретичні відомості

1. Поняття семафора, операції down і up.

У 1965 році Е. Дейкстра (E. W. Dijkstra) запропонував використовувати для синхронізації процесів цілу змінну особливого типу – семафор, значення якої може бути нулем (за відсутності збережених сигналів активізації) або додатним числом, що відповідає кількості відкладених (ще не використаних) сигналів активізації. Над семафором визначені дві атомарні операції – down і up (узагальнення відомих операцій sleep і wakeup).

Операція down порівнює значення семафора з нулем. Якщо воно більше нуля, операція зменшує його (витрачає один зі збережених сигналів активізації) і повертає керування процесу негайно. Якщо значення семафора дорівнює нулю, процес, що викликав down, переводиться в стан очікування – керування йому не повертається, доки інший процес не виконає над цим семафором операцію up. Операція up збільшує значення семафора; якщо з семафором пов'язані один або кілька процесів, що очікують (заблоковані на down), один з них (обраний системою) дістає змогу завершити свою операцію down і продовжити виконання.

Те, що перевірка значення семафора, його зміна і можливий перехід процесу в стан очікування виконуються як єдина неподільна (атомарна) дія, принципово важливо: це гарантує, що жоден інший процес не зможе звернутися до того самого семафора, доки операція не завершиться чи не заблокує процес, що її виконує – інакше можливі ситуації втрати сигналів активізації через паралельне виконання перевірки і зміни значення семафора двома процесами одночасно.

2. Задача «виробник-споживач» та її розв'язання за допомогою семафорів.

Класична задача синхронізації: один або кілька процесів-виробників додають елементи даних у буфер обмеженого розміру, а один або кілька процесів-споживачів вилучають їх із буфера; виробник має зупинитися, коли буфер заповнений, а споживач – коли буфер порожній. Розв'язання використовує три семафори: empty –

кількість вільних місць у буфері (початково дорівнює розміру буфера); full – кількість заповнених місць (початково 0); mutex – двійковий семафор (початкове значення 1) для взаємного виключення доступу до самого буфера.

Нижче наведено псевдокод класичного розв'язання (за Е. Таненбаумом): виробник перед поміщенням елемента в буфер виконує down(empty) і down(mutex), а після поміщення – up(mutex) і up(full); споживач перед вилученням елемента виконує down(full) і down(mutex), а після вилучення – up(mutex) і up(empty).

```
// Псевдокод (ілюстрація, не призначений для компіляції)
semaphore mutex = 1;   // взаємне виключення доступу до буфера
semaphore empty = N;   // кількість вільних місць у буфері
semaphore full  = 0;   // кількість заповнених місць у буфері

producer:                          consumer:
  item = produce_item();             down(full);
  down(empty);                       down(mutex);
  down(mutex);                       item = remove_item();
  insert_item(item);                 up(mutex);
  up(mutex);                         up(empty);
  up(full);                          consume_item(item);
```

Семафори empty і full тут використовуються для синхронізації (гарантують, що виробник зупиниться, коли буфер заповнений, а споживач – коли він порожній), а семафор mutex – суто для взаємного виключення (виключає одночасний доступ двох процесів до спільного буфера); семафор, початкове значення якого дорівнює 1 і який використовується лише для взаємного виключення, називають двійковим семафором – за призначенням він дуже близький до м'ютекса (розділ 2).

У Win32 API семафор створюють функцією CreateSemaphore, яка задає початкове і максимальне значення рахівника семафора. Операцію down виконує WaitForSingleObject (зменшує рахівник семафора на 1, очікуючи, якщо він дорівнює 0), а операцію up – функція ReleaseSemaphore (збільшує рахівник на задану величину).

```
HANDLE CreateSemaphoreA(
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    LONG lInitialCount,   // початкове значення рахівника
    LONG lMaximumCount,   // максимальне значення рахівника
    LPCSTR lpName);       // ім'я семафора (NULL – без імені)

BOOL ReleaseSemaphore(
    HANDLE hSemaphore,
    LONG lReleaseCount,        // на скільки збільшити рахівник (зазвичай 1)
    LPLONG lpPreviousCount);   // попереднє значення рахівника (можна NULL)
```

3. Поняття м'ютекса, відмінність від семафора.

М'ютекс (mutex, від mutual exclusion – взаємне виключення) – спрощена версія семафора, яка може перебувати лише в одному з двох станів: вільному (відсутній прапорець блокування) або зайнятому (захопленому одним із потоків). На відміну від

семафора, м'ютекс не вміє підраховувати – він лише гарантує, що в критичній області коду в будь-який момент часу перебуває не більше одного потоку. Завдяки простоті реалізації м'ютекси особливо ефективні для синхронізації потоків у межах одного процесу.

Якщо потік хоче ввійти в критичну область, він викликає процедуру захоплення м'ютекса (mutex_lock); якщо м'ютекс вільний, вхід дозволяється негайно, а м'ютекс переходить у зайнятий стан. Якщо м'ютекс уже зайнятий, потік, що його викликав, блокується доти, поки потік-власник не звільнить м'ютекс, викликавши процедуру mutex_unlock; якщо м'ютекс блокує кілька потоків, після звільнення з них випадковим чином вибирається один. У просторі користувача м'ютекси легко реалізувати за допомогою нероздільної команди процесора (Test-and-Set Lock, TSL): захоплення м'ютекса, якому не вдається ввійти в критичну область, не займається активним очікуванням (на відміну від наївнішого підходу), а добровільно передає процесорний час іншому потоку викликом функції планувальника потоків (наприклад, thread_yield) і повторює спробу пізніше.

У Win32 API м'ютекс створюють функцією CreateMutex, захоплюють – функцією WaitForSingleObject (як і для семафора), а звільняють – функцією ReleaseMutex. М'ютекс може бути анонімним (синхронізує лише потоки одного процесу, що мають спільний дескриптор м'ютекса) або поіменованим: якщо кілька процесів викликають CreateMutex з однаковим ім'ям, перший процес дійсно створює новий об'єкт, а кожен наступний отримує дескриптор уже існуючого об'єкта (GetLastError поверне ERROR_ALREADY_EXISTS) – так м'ютекс дає змогу синхронізувати потоки різних процесів, а не лише потоки в межах одного процесу.

```
HANDLE CreateMutexA(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    BOOL bInitialOwner,   // TRUE - м'ютекс одразу захоплює потік, що його створив
    LPCSTR lpName);       // ім'я м'ютекса (NULL - анонімний)

BOOL ReleaseMutex(HANDLE hMutex);
```

Якщо потоки синхронізуються лише в межах одного процесу, ефективнішою альтернативою м'ютексу є критична секція (CRITICAL_SECTION, функції InitializeCriticalSection, EnterCriticalSection, LeaveCriticalSection): вона не є об'єктом ядра, тому в типовому випадку (коли критична секція вільна) захоплення не потребує переходу в режим ядра і виконується значно швидше, ніж через об'єкт-м'ютекс; платою за швидкість є те, що критичну секцію не можна використати для синхронізації потоків різних процесів – на відміну від поіменованого м'ютекса.

Приклади програмної реалізації

Реалізація обмеженого циклічного буфера на 5 елементів засобами реальних семафорів Win32 API (а не псевдокоду): семафор hSemaphoreEmpty відповідає

семафору empty, hSemaphoreFull – семафору full; для взаємного виключення доступу до самого буфера тут застосовано критичну секцію (як ефективнішу альтернативу третьому, двійковому семафору mutex, оскільки виробник і споживач – потоки одного процесу).

Приклад 1 – Задача «виробник-споживач» з використанням семафорів

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

HANDLE hSemaphoreEmpty;
HANDLE hSemaphoreFull;
CRITICAL_SECTION cs;

const int BUFFER_SIZE = 5;
int buffer[BUFFER_SIZE];
int writeIndex = 0;
int readIndex = 0;

DWORD WINAPI Producer(LPVOID)
{
    for (int i = 1; i <= 10; ++i)
    {
        WaitForSingleObject(hSemaphoreEmpty, INFINITE);  // down(empty)
        EnterCriticalSection(&cs);
        buffer[writeIndex] = i;
        std::cout << "Produced: " << i << " at index " << writeIndex << std::endl;
        writeIndex = (writeIndex + 1) % BUFFER_SIZE;
        LeaveCriticalSection(&cs);
        ReleaseSemaphore(hSemaphoreFull, 1, NULL);       // up(full)
        Sleep(50);
    }
    return 0;
}

DWORD WINAPI Consumer(LPVOID)
{
    for (int i = 1; i <= 10; ++i)
    {
        WaitForSingleObject(hSemaphoreFull, INFINITE);   // down(full)
        EnterCriticalSection(&cs);
        int value = buffer[readIndex];
        std::cout << "Consumed: " << value << " at index " << readIndex <<
std::endl;
        readIndex = (readIndex + 1) % BUFFER_SIZE;
        LeaveCriticalSection(&cs);
        ReleaseSemaphore(hSemaphoreEmpty, 1, NULL);      // up(empty)
        Sleep(70);
    }
    return 0;
}

int main()
{
    hSemaphoreEmpty = CreateSemaphoreA(NULL, BUFFER_SIZE, BUFFER_SIZE, NULL);
    hSemaphoreFull = CreateSemaphoreA(NULL, 0, BUFFER_SIZE, NULL);
```

```
    InitializeCriticalSection(&cs);

    DWORD id1, id2;
    HANDLE h1 = CreateThread(NULL, 0, Producer, NULL, 0, &id1);
    HANDLE h2 = CreateThread(NULL, 0, Consumer, NULL, 0, &id2);

    HANDLE handles[2] = { h1, h2 };
    WaitForMultipleObjects(2, handles, TRUE, INFINITE);

    CloseHandle(h1);
    CloseHandle(h2);
    CloseHandle(hSemaphoreEmpty);
    CloseHandle(hSemaphoreFull);
    DeleteCriticalSection(&cs);

    std::cout << "Done." << std::endl;
    std::cin.get();
    return 0;
}
```

кінець прикладу 1

Два потоки записують у спільний файл числа: один – парні, інший – непарні; м'ютекс гарантує, що операції запису не переплутаються між собою. Використано поіменований м'ютекс "mutexos" – у реальному застосуванні саме ім'я дало б змогу синхронізувати так само й потоки, що належать різним процесам, а не тільки двом потокам цієї самої програми.

Приклад 2 – Синхронізація запису у файл за допомогою поіменованого м'ютекса

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

HANDLE ghMutex;

DWORD WINAPI WriteEven(LPVOID)
{
    std::ofstream f("numbers.txt", std::ios::app);
    for (int i = 0; i < 10; i += 2)
    {
        WaitForSingleObject(ghMutex, INFINITE);
        f << i << " ";
        f.flush();
        ReleaseMutex(ghMutex);
        Sleep(50);
    }
    return 0;
}

DWORD WINAPI WriteOdd(LPVOID)
{
    std::ofstream f("numbers.txt", std::ios::app);
    for (int i = 1; i < 10; i += 2)
    {
        WaitForSingleObject(ghMutex, INFINITE);
```

```
        f << i << " ";
        f.flush();
        ReleaseMutex(ghMutex);
        Sleep(50);
    }
    return 0;
}

int main()
{
    ghMutex = CreateMutexA(NULL, FALSE, "mutexos");
    if (!ghMutex)
    {
        std::cout << "Create mutex failed. Error: " << GetLastError() << std::endl;
        return 1;
    }

    DWORD id1, id2;
    HANDLE h1 = CreateThread(NULL, 0, WriteEven, NULL, 0, &id1);
    HANDLE h2 = CreateThread(NULL, 0, WriteOdd, NULL, 0, &id2);

    HANDLE handles[2] = { h1, h2 };
    WaitForMultipleObjects(2, handles, TRUE, INFINITE);

    CloseHandle(h1);
    CloseHandle(h2);
    CloseHandle(ghMutex);

    std::cout << "Done, see numbers.txt" << std::endl;
    std::cin.get();
    return 0;
}
```

кінець прикладу 2

Головний потік і один додатковий потік (створений функцією _beginthreadex) спільно звертаються до масиву sharedData: додатковий потік періодично записує в усі 5 елементів те саме число, а головний потік виводить увесь масив на екран. М'ютекс гарантує, що головний потік завжди побачить узгоджений стан масиву (усі 5 елементів з однаковим числом), а не «розірваний» – тобто частину елементів зі старим значенням, а частину з новим, як могло би статися без синхронізації.

Приклад 3 – Захист спільного масиву анонімним м'ютексом

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

HANDLE hMutex;
int sharedData[5];

unsigned WINAPI Worker(void*)
{
    for (int num = 0; num < 20; ++num)
    {
```

```
        WaitForSingleObject(hMutex, INFINITE);
        for (int i = 0; i < 5; ++i)
            sharedData[i] = num;
        ReleaseMutex(hMutex);
        Sleep(20);
    }
    return 0;
}

int main()
{
    hMutex = CreateMutexA(NULL, FALSE, NULL);  // анонімний м'ютекс

    unsigned threadId;
    HANDLE hThread = (HANDLE)_beginthreadex(NULL, 0, Worker, NULL, 0, &threadId);

    for (int i = 0; i < 10; ++i)
    {
        WaitForSingleObject(hMutex, INFINITE);
        std::cout << sharedData[0] << " " << sharedData[1] << " " << sharedData[2]
                  << " " << sharedData[3] << " " << sharedData[4] << std::endl;
        ReleaseMutex(hMutex);
        Sleep(30);
    }

    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hThread);
    CloseHandle(hMutex);
    return 0;
}
```

кінець прикладу 3

Завдання для самостійного виконання:

1. Модифікувати приклад 1 так, щоб розмір буфера, кількість елементів і час очікування (Sleep) відповідали індивідуальному варіанту студента; додати другого споживача (третій потік), який обробляє ту саму чергу full спільно з першим споживачем, переконавшись, що жоден елемент не буде обробленим двічі.

2. Запустити одночасно дві копії скомпільованої програми з прикладу 2 (два окремих процеси). Оскільки м'ютекс поіменований, обидва процеси повинні отримати дескриптор одного й того самого об'єкта синхронізації. Переконатися (наприклад, тимчасово прибравши захоплення м'ютекса), що без синхронізації між процесами записи у файл переплутуються, а з нею – ні.

3. Модифікувати приклад 3 так, щоб замість одного додаткового потоку працювали два, кожен зі своїм власним діапазоном значень для запису в масив; переконатися, що головний потік і надалі завжди бачить узгоджений (не «розірваний») стан масиву.

4. Скласти звіт про виконання лабораторної роботи: опис семафорів і м'ютексів та функцій для роботи з ними, постановка задачі, повний листинг програм, результати виконання (скриншоти консолі) і висновки.

5. При захисті лабораторної роботи підготувати відповіді на контрольні питання.

Контрольні питання:

1. Що таке семафор? Які операції визначені над семафором і що вони виконують?

2. Чому перевірка значення семафора, його зміна і можливий перехід процесу в стан очікування повинні виконуватися як єдина неподільна дія?

3. Опишіть розв'язання задачі «виробник-споживач» за допомогою семафорів: яку роль виконує кожен з трьох семафорів?

4. Що таке двійковий семафор і чим він подібний до м'ютекса?

5. Якими функціями Win32 API створюють семафор та виконують операції down і up над ним?

6. Що таке м'ютекс? Чим він принципово відрізняється від семафора?

7. Як реалізувати м'ютекс у просторі користувача за допомогою команди TSL? Чому це не є активним очікуванням?

8. Якими функціями Win32 API створюють, захоплюють і звільняють м'ютекс?

9. Чим відрізняється анонімний м'ютекс від поіменованого? Для чого використовується ім'я м'ютекса?

10. Чим критична секція (CRITICAL_SECTION) відрізняється від м'ютекса і коли доцільніше використовувати кожен з цих засобів?

Лабораторна робота №13
Мережні засоби операційних систем

Мета роботи: навчитися розробляти програми керування мережними засобами операційних систем, реалізувати найпростіші TCP-клієнт і TCP-сервер засобами Win32 API (Windows Sockets).

Зміст теоретичних відомостей:

1. Загальні принципи мережної підтримки. Рівні мережної архітектури.
2. Стек протоколів TCP/IP.
3. Система імен DNS.
4. Програмний інтерфейс сокетів (Berkeley Sockets, Windows Sockets).

Теоретичні відомості

1. Загальні принципи мережної підтримки. Рівні мережної архітектури.

Мережа – набір вузлів (комп'ютерів чи інших пристроїв), пов'язаних каналами зв'язку, якими передається інформація. Сукупність мереж, що використовують спільний набір мережних протоколів, утворює інтернет (з малої літери); різнорідні мережі пов'язують між собою маршрутизатори, які переадресовують пакети з однієї мережі в іншу. Мережний протокол – набір правил, що задає формат повідомлень і порядок обміну ними; набір протоколів різних рівнів, що разом реалізують мережну архітектуру, називають стеком протоколів.

Складність реалізації мережної взаємодії приховують за допомогою багаторівневого підходу: кожен рівень надає вищому рівню певний набір операцій – мережний сервіс, приховуючи деталі власної реалізації. Розрізняють сервіси, орієнтовані на з'єднання (передбачають встановлення з'єднання, передавання даних безперервним потоком і його розрив – так працює TCP), і дейтаграмні сервіси (кожне повідомлення передається як незалежний пакет, що може прибути в іншому порядку, ніж було надіслано, – так працює UDP).

2 Стек протоколів TCP/IP

Мережна архітектура TCP/IP, на якій ґрунтується Інтернет, має чотири рівні:

- канальний рівень – передавання кадрів даних фізичними каналами (Ethernet, Wi-Fi тощо) між сусідніми вузлами мережі;
- мережний рівень – доставка пакетів (IP-дейтаграм) між будь-якими вузлами неоднорідної мережі з довільною топологією; основний протокол – IP (Internet Protocol); кожен мережний інтерфейс має унікальну IP-адресу (4 байти для IPv4, записують у крапково-десятковому форматі, наприклад 192.168.1.1; через обмежену кількість таких адрес дедалі більше використовують IPv6 з 128-бітовою адресою);

- транспортний рівень – організація зв'язку між процесами на віддалених хостах; протокол TCP (Transmission Control Protocol) забезпечує надійну доставку даних через з'єднання (з підтвердженнями, повторним пересиланням і впорядкуванням сегментів), протокол UDP (User Datagram Protocol) пересилає дейтаграми без гарантії доставки чи порядку, але з меншими накладними витратами; для розрізнення процесів на одному хості використовують номери портів (від 0 до 65535; наприклад, порт 80 зарезервовано за замовчуванням для HTTP, порт 25 – для SMTP);
- прикладний рівень – протоколи конкретних мережних служб, призначені для кінцевих застосувань (HTTP, SMTP тощо).

Під час передавання дані послідовно «обгортають» заголовками кожного рівня (HTTP-запит → TCP-сегмент → IP-дейтаграма → Ethernet-фрейм), а на приймальній стороні – навпаки, послідовно «розгортають», піднімаючись по стеку протоколів від канального рівня до прикладного, поки дані не потраплять у застосування-адресат, визначене номером порту.

3. Система імен DNS.

Доменна система імен (DNS) – розподілена ієрархічна база даних, що відображає символьні (доменні) імена хостів на їхні IP-адреси; процес такого перетворення називають розв'язанням доменних імен. Жоден окремий хост не зберігає всю базу відображень – кожна група хостів підтримує власну зону, відповідальність за яку можна делегувати організаціям нижчого рівня (наприклад, зона .ua делегує піддомени конкретним установам). Найважливіший тип запису в базі DNS – A-запис, що зв'язує повне доменне ім'я з IP-адресою; запис типу CNAME задає псевдонім (аліас) для вже наявного імені. Щоб зменшити навантаження на мережу, отримані відображення кешують на обмежений час на проміжних серверах імен.

4. Програмний інтерфейс сокетів (Berkeley Sockets, Windows Sockets).

Сокет – абстракція, що зв'язує код прикладної програми (рівень застосувань) з реалізацією стека мережних протоколів операційної системи (транспортний рівень); програмний інтерфейс сокетів, розроблений у Каліфорнійському університеті в Берклі (Berkeley Sockets), є фактичним стандартом, на основі якого побудований і програмний інтерфейс Windows Sockets (Winsock) – той самий набір функцій (socket, bind, listen, accept, connect, send, recv, closesocket) із незначними платформенними відмінностями.

Основні системні виклики (однакові за призначенням у Berkeley Sockets і Winsock):

- socket() – створює дескриптор сокета заданого типу (SOCK_STREAM – потоковий, для TCP; SOCK_DGRAM – дейтаграмний, для UDP);
- bind() – пов'язує сокет із конкретною локальною адресою і номером порту

(потрібно серверу, щоб клієнти знали, куди підключатися);

- listen() – переводить сокет сервера в пасивний режим очікування запитів на з'єднання, задаючи довжину черги таких запитів;
- accept() – приймає з черги перший запит на з'єднання і повертає новий дескриптор сокета, призначений для обміну даними саме з цим клієнтом (прослуховувальний сокет після цього продовжує приймати нові запити);
- connect() – з боку клієнта встановлює з'єднання із сервером за вказаною адресою і портом;
- send() / recv() – надсилають і приймають дані через установлене з'єднання;
- closesocket() – закриває сокет після завершення роботи з ним (в інтерфейсі Berkeley Sockets для цього використовують звичайний виклик close()).

На відміну від Berkeley Sockets, перед використанням будь-яких функцій Windows Sockets необхідно ініціалізувати бібліотеку викликом WSAStartup (а після завершення роботи – звільнити ресурси викликом WSACleanup); для компонування потрібно підключити бібліотеку ws2_32.lib і заголовний файл <winsock2.h> (та <ws2tcpip.h> для сучасніших функцій роботи з адресами, наприклад inet_ntop і getaddrinfo). У режимі користувача Winsock API реалізує системна бібліотека ws2_32.dll, яка для фактичного передавання даних мережею звертається до драйвера файлової системи afd.sys, що, у свою чергу, користується драйвером підтримки TCP/IP (tcpip.sys).

```
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "ws2_32.lib")   // автоматичне підключення бібліотеки (тільки в MSVC)

WSADATA wsaData;
WSAStartup(MAKEWORD(2, 2), &wsaData);   // ініціалізація Winsock версії 2.2
// ... робота із сокетами ...
WSACleanup();                            // звільнення ресурсів бібліотеки
```

Приклади програмної реалізації

Сервер очікує одне підключення на порту 5150, приймає від клієнта повідомлення, виводить його на екран і надсилає клієнту відповідь.

Приклад 1 – TCP-сервер

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iostream>
#include <cstring>

#pragma comment(lib, "ws2_32.lib")

int main()
{
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
```

```
        std::cout << "WSAStartup failed." << std::endl;
        return 1;
    }

    SOCKET listeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listeningSocket == INVALID_SOCKET)
    {
        std::cout << "socket() failed. Error: " << WSAGetLastError() << std::endl;
        WSACleanup();
        return 1;
    }

    const int port = 5150;
    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(port);
    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(listeningSocket, (sockaddr*)&serverAddr, sizeof(serverAddr)) ==
SOCKET_ERROR)
    {
        std::cout << "bind() failed. Error: " << WSAGetLastError() << std::endl;
        closesocket(listeningSocket);
        WSACleanup();
        return 1;
    }

    if (listen(listeningSocket, 5) == SOCKET_ERROR)
    {
        std::cout << "listen() failed. Error: " << WSAGetLastError() << std::endl;
        closesocket(listeningSocket);
        WSACleanup();
        return 1;
    }

    std::cout << "Waiting for connection on port " << port << "..." << std::endl;

    sockaddr_in clientAddr;
    int clientAddrLen = sizeof(clientAddr);
    SOCKET clientSocket = accept(listeningSocket, (sockaddr*)&clientAddr,
&clientAddrLen);
    if (clientSocket == INVALID_SOCKET)
    {
        std::cout << "accept() failed. Error: " << WSAGetLastError() << std::endl;
        closesocket(listeningSocket);
        WSACleanup();
        return 1;
    }

    char clientIp[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &clientAddr.sin_addr, clientIp, sizeof(clientIp));
    std::cout << "Connected to " << clientIp << ":" << ntohs(clientAddr.sin_port) <<
std::endl;

    closesocket(listeningSocket);   // подальші нові підключення вже не приймаємо
```

```
    char buffer[1024];
    int bytesReceived = recv(clientSocket, buffer, sizeof(buffer) - 1, 0);
    if (bytesReceived == SOCKET_ERROR)
    {
        std::cout << "recv() failed. Error: " << WSAGetLastError() << std::endl;
        closesocket(clientSocket);
        WSACleanup();
        return 1;
    }
    buffer[bytesReceived] = '\0';
    std::cout << "Received " << bytesReceived << " bytes: " << buffer << std::endl;

    const char* reply = "Hello from server";
    send(clientSocket, reply, (int)strlen(reply), 0);

    closesocket(clientSocket);
    WSACleanup();

    std::cout << "Done." << std::endl;
    return 0;
}
```

кінець прикладу 1

Клієнт приймає IP-адресу сервера як параметр командного рядка, підключається до нього на порт 5150, надсилає повідомлення і виводить отриману відповідь.

Приклад 2 – TCP-клієнт

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iostream>
#include <cstring>

#pragma comment(lib, "ws2_32.lib")

int main(int argc, char** argv)
{
    if (argc <= 1)
    {
        std::cout << "Usage: tcpclient <Server IP address>" << std::endl;
        return 1;
    }

    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        std::cout << "WSAStartup failed." << std::endl;
        return 1;
    }

    SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (s == INVALID_SOCKET)
    {
        std::cout << "socket() failed. Error: " << WSAGetLastError() << std::endl;
```

```
        WSACleanup();
        return 1;
    }

    const int port = 5150;
    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(port);
    serverAddr.sin_addr.s_addr = inet_addr(argv[1]);

    std::cout << "Connecting to " << argv[1] << ":" << port << "..." << std::endl;
    if (connect(s, (sockaddr*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR)
    {
        std::cout << "connect() failed. Error: " << WSAGetLastError() << std::endl;
        closesocket(s);
        WSACleanup();
        return 1;
    }
    std::cout << "Connected." << std::endl;

    const char* msg = "Hello";
    send(s, msg, (int)strlen(msg), 0);
    std::cout << "Sent: " << msg << std::endl;

    char buffer[1024];
    int bytesReceived = recv(s, buffer, sizeof(buffer) - 1, 0);
    if (bytesReceived > 0)
    {
        buffer[bytesReceived] = '\0';
        std::cout << "Received: " << buffer << std::endl;
    }

    closesocket(s);
    WSACleanup();
    return 0;
}
```

кінець прикладу 2

Щоб перевірити роботу прикладів 1 і 2, спочатку запустіть скомпільований сервер, а потім, у другому вікні командного рядка, запустіть клієнт із параметром 127.0.0.1 (підключення до того самого комп'ютера) – наприклад: tcpclient.exe 127.0.0.1.

На відміну від оригінальної версії цього прикладу, де використовувалася застаріла функція gethostbyname (яка в сучасних версіях Windows SDK офіційно не рекомендована – deprecated – і не підтримує IPv6), тут застосовано сучасну функцію getaddrinfo.

Приклад 3 – Визначення IP-адрес за доменним іменем

```
#include <winsock2.h>
#include <ws2tcpip.h>
```

```
#include <iostream>

#pragma comment(lib, "ws2_32.lib")

int main()
{
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    const char* hostname = "localhost";   // змініть на будь-яке доменне ім'я

    addrinfo hints = {};
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    addrinfo* result = nullptr;
    int err = getaddrinfo(hostname, NULL, &hints, &result);
    if (err != 0)
    {
        std::cout << "getaddrinfo() failed. Error: " << err << std::endl;
        WSACleanup();
        return 1;
    }

    std::cout << "IP addresses for " << hostname << ":" << std::endl;
    for (addrinfo* p = result; p != NULL; p = p->ai_next)
    {
        sockaddr_in* addr = (sockaddr_in*)p->ai_addr;
        char ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &addr->sin_addr, ip, sizeof(ip));
        std::cout << "  " << ip << std::endl;
    }

    freeaddrinfo(result);
    WSACleanup();
    return 0;
}
```

кінець прикладу 3

Завдання для самостійного виконання:

1. Модифікувати приклад 1 (сервер) так, щоб після отримання повідомлення від клієнта він не завершував роботу, а в циклі очікував нові підключення (тобто міг послідовно обслужити кількох клієнтів один за одним).

2. Модифікувати приклад 2 (клієнт) так, щоб повідомлення для надсилання серверу вводилося з клавіатури, а не було жорстко задане в коді.

3. Модифікувати приклади 1 і 2 так, щоб замість фіксованого порту 5150 використовувався порт, заданий індивідуальним варіантом студента (наприклад, 5150 + номер варіанта).

4. Модифікувати приклад 3 так, щоб доменне ім'я для розв'язання вводилося з клавіатури або передавалося параметром командного рядка, а не було жорстко задане в коді.

5. Скласти звіт про виконання лабораторної роботи: опис використаних функцій Windows Sockets, постановка задачі, повний листинг програм, результати спільного запуску сервера і клієнта (скриншоти двох консольних вікон) і висновки.

6. При захисті лабораторної роботи підготувати відповіді на контрольні питання.

Контрольні питання:

1. Що таке мережа, інтернет (з малої літери) та Інтернет (з великої)? Яку роль виконують маршрутизатори?

2. Що таке мережний протокол і стек протоколів? Чим відрізняються сервіси, орієнтовані на з'єднання, від дейтаграмних?

3. Назвіть рівні мережної архітектури TCP/IP та призначення кожного з них.

4. Чим відрізняються протоколи TCP і UDP транспортного рівня?

5. Що таке порт і для чого він потрібен? Які номери портів зарезервовано за деякими відомими службами?

6. Як відбувається перетворення (інкапсуляція) даних під час проходження мережних рівнів від прикладного до канального і навпаки?

7. Що таке система імен DNS? Як організована ієрархія доменних імен?

8. Чим відрізняється A-запис від CNAME-запису в базі даних DNS?

9. Що таке сокет? Яке місце інтерфейс сокетів посідає в мережній архітектурі?

10. Охарактеризуйте призначення системних викликів socket, bind, listen, accept, connect.

11. Якими функціями обмінюються даними через установлене з'єднання в інтерфейсі сокетів?

12. Чим відрізняється використання Windows Sockets від класичного інтерфейсу сокетів Берклі? Яке призначення функцій WSAStartup і WSACleanup?

13. Які бібліотеки і заголовні файли потрібні для роботи з Windows Sockets у Visual Studio?

14. Якою сучасною функцією рекомендують замінювати застарілу gethostbyname для визначення IP-адрес за доменним іменем?

Лабораторна робота №14
Захист інформації в операційних системах

Мета роботи: навчитися розробляти програми захисту інформації в операційних системах: визначати ідентифікатори безпеки користувачів, переглядати та змінювати облікові записи й локальні групи користувачів.

Зміст теоретичних відомостей:

1. Основні завдання забезпечення безпеки: аутентифікація, авторизація, аудит.
2. Принципи керування доступом: матриця доступу, списки контролю доступу.
3. Архітектура безпеки Windows: SID, маркери доступу.

Теоретичні відомості

1. Основні завдання забезпечення безпеки: аутентифікація, авторизація, аудит.

Аутентифікація – процес, за допомогою якого система засвідчує, що користувач є тим, за кого себе видає (типово – за паролем, відомим обом сторонам; альтернативно – за володінням фізичним предметом, наприклад смарт-картою, або за біометричними параметрами). Після успішної аутентифікації для доступу до конкретних ресурсів потрібна авторизація (керування доступом) – перевірка, чи має аутентифікований користувач право виконати запитану дію з конкретним ресурсом. Аудит – збирання та збереження інформації про події в системі, важливі для її безпеки (спроби аутентифікації, спроби доступу до об'єктів тощо), у спеціальному журналі для подальшого аналізу.

До основних властивостей, які повинна забезпечувати система безпеки, належать: конфіденційність (можливість приховання даних від стороннього доступу, типово – за допомогою шифрування); цілісність (спроможність захистити дані від несанкціонованого вилучення чи зміни, наприклад за допомогою цифрових підписів); доступність (гарантія того, що легітимний користувач після аутентифікації зможе отримати доступ до ресурсу, на який має право; порушення доступності називають відмовою в обслуговуванні).

2. Принципи керування доступом: матриця доступу, списки контролю доступу.

Розподіл прав доступу відображають матрицею доступу, рядки якої відповідають суб'єктам авторизації (користувачам, групам), а стовпці – ресурсам (файлам, пристроям тощо); елемент матриці визначає, які дії дозволені даному суб'єкту з даним ресурсом. Зберігати матрицю доступу повністю непрактично через її розмір і розрідженість, тому застосовують два основні підходи:

- списки контролю доступу (Access Control Lists, ACL) – для кожного ресурсу

зберігають стовпець матриці: перелік суб'єктів, яким дозволено ним користуватися, і дозволений набір дій. Елемент такого списку (пара «суб'єкт – набір дій») називають елементом контролю доступу (Access Control Element, ACE); цей підхід застосовано в Windows;

- можливості (capabilities) – для кожного суб'єкта зберігають рядок матриці: перелік ресурсів, якими йому дозволено користуватися.

Загальна концепція реалізації ACL передбачає можливість задавати права не лише для окремих користувачів, а й для груп користувачів і для всіх користувачів системи загалом; у системах лінії Windows NT списки контролю доступу можуть містити необмежену кількість елементів із різними комбінаціями прав.

3. Архітектура безпеки Windows: SID, маркери доступу.

Архітектура безпеки Windows складається з кількох компонентів: процес реєстрації користувачів (winlogon) обробляє запити на вхід у систему; менеджер аутентифікації (Local Security Authority Subsystem, LSASS) безпосередньо здійснює аутентифікацію та контролює політику аудиту; менеджер облікових записів (Security Accounts Manager, SAM) підтримує базу даних облікових записів локальних користувачів і груп; довідковий монітор захисту (Security Reference Monitor, SRM) перевіряє права користувача на доступ до конкретного об'єкта та виконує дію за наявності цих прав.

З кожним користувачем і групою у Windows пов'язаний унікальний ідентифікатор безпеки (Security Identifier, SID) – підсистема безпеки звертається до користувачів і груп лише за їхнім SID, а не за іменем. Після успішної аутентифікації користувача LSASS створює для нього маркер доступу (access token), який пов'язують з усіма процесами цього користувача; маркер містить SID користувача, перелік груп, до яких він належить, перелік привілеїв, якими він володіє, а також типові права доступу для об'єктів, які створюватимуть процеси цього користувача. Під час спроби доступу до об'єкта SRM звіряє SID і список груп з маркера доступу процесу з елементами списку контролю доступу (ACL) цього об'єкта.

Win32 API надає функцію LookupAccountName для отримання SID користувача чи групи за їхнім ім'ям; для роботи з маркером доступу процесу використовують функцію OpenProcessToken (повертає дескриптор маркера) та GetTokenInformation (повертає конкретні відомості з маркера, наприклад SID користувача – клас TokenUser, або список його груп – клас TokenGroups); рядкове подання SID отримують функцією ConvertSidToStringSid.

```
BOOL OpenProcessToken(HANDLE ph, DWORD access, PHANDLE ptoken);
BOOL GetTokenInformation(HANDLE token, TOKEN_INFORMATION_CLASS tclass,
                         LPVOID tibuf, DWORD tisize, PDWORD realsize);
BOOL ConvertSidToStringSidA(PSID sid, LPSTR *stringSid);
```

Облікові записи користувачів та локальні групи на Windows-хості можна

створювати, переглядати, змінювати і вилучати засобами мережного програмного інтерфейсу NetAPI (заголовний файл <lm.h>, бібліотека netapi32.lib): функції NetUserAdd, NetUserGetInfo, NetUserChangePassword, NetUserDel – для облікових записів користувачів; NetLocalGroupAdd, NetLocalGroupGetInfo, NetLocalGroupAddMembers, NetLocalGroupDelMembers, NetLocalGroupDel – для локальних груп. На відміну від функцій читання інформації (GetInfo), які може викликати будь-який користувач, функції створення, зміни та вилучення облікових записів і груп (Add, Del) потребують прав адміністратора.

Механізми керування доступом на основі ACL не захищають дані від крадіжки самого носія: зловмисник може підключити викрадений жорсткий диск до іншого комп'ютера із сумісною ОС, де матиме права адміністратора, і отримати доступ до всіх файлів незалежно від заданих для них прав – ця проблема особливо актуальна для портативних комп'ютерів. Для захисту найціннішої інформації застосовують шифрування на рівні файлової системи: у Windows ця технологія називається EFS (Encrypting File System) – спеціальний драйвер файлової системи перехоплює спроби доступу до позначеного файлу і прозоро шифрує та дешифрує його вміст «на льоту», використовуючи ключ, пов'язаний з обліковим записом користувача.

Для захисту даних під час передавання мережею застосовують шифрування каналів зв'язку на різних рівнях стека протоколів: на канальному рівні апаратні засоби автоматично шифрують кожен пакет перед передаванням через незахищений фізичний канал і дешифрують після отримання; на мережному рівні виконують наскрізне шифрування пакетів IP (заголовки залишаються незашифрованими, тіло пакета – зашифроване на всьому шляху від відправника до одержувача); на прикладному рівні шифрування може бути вбудоване безпосередньо в застосування, яке тоді використовує не звичайний інтерфейс сокетів, а інтерфейс спеціалізованої бібліотеки, що забезпечує шифрування даних перед їх передаванням.

Приклади програмної реалізації

Усі приклади – консольні застосунки Win32 (Empty Project у Visual Studio). Приклади 1 і 2 лише читають інформацію і безпечні для запуску без особливих застережень; приклад 3 створює і вилучає реальний об'єкт системи (локальну групу) і вимагає прав адміністратора – детальніше дивіться примітку після нього.

Приклад 1 – Визначення SID поточного користувача

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

#pragma comment(lib, "advapi32.lib")

int main()
{
    HANDLE hToken;
```

```
    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken))
    {
        std::cout << "OpenProcessToken failed. Error: " << GetLastError() <<
std::endl;
        return 1;
    }

    // спочатку дізнаємося потрібний розмір буфера (перший виклик)
    DWORD size = 0;
    GetTokenInformation(hToken, TokenUser, NULL, 0, &size);

    BYTE* buf = new BYTE[size];
    if (!GetTokenInformation(hToken, TokenUser, buf, size, &size))
    {
        std::cout << "GetTokenInformation failed. Error: " << GetLastError() <<
std::endl;
        delete[] buf;
        CloseHandle(hToken);
        return 1;
    }

    PTOKEN_USER tokenUser = (PTOKEN_USER)buf;
    LPSTR sidString = NULL;
    if (ConvertSidToStringSidA(tokenUser->User.Sid, &sidString))
    {
        std::cout << "Current user SID: " << sidString << std::endl;
        LocalFree(sidString);
    }

    delete[] buf;
    CloseHandle(hToken);
    return 0;
}
```

кінець прикладу 1

Примітка: розмір структури TOKEN_USER заздалегідь невідомий (він залежить від довжини конкретного SID), тому GetTokenInformation викликають двічі за стандартною ідіомою Win32 API: перший виклик з нульовим розміром буфера лише повертає потрібний розмір через параметр realsize, а другий виклик (із буфером уже потрібного розміру) повертає самі дані.

Приклад 2 – Перегляд інформації про обліковий запис поточного користувача

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

#pragma comment(lib, "netapi32.lib")

int main()
{
    wchar_t userName[UNLEN + 1];
    DWORD size = UNLEN + 1;
```

```
    if (!GetUserNameW(userName, &size))
    {
        std::cout << "GetUserNameW failed. Error: " << GetLastError() << std::endl;
        return 1;
    }

    USER_INFO_1* ui = NULL;
    NET_API_STATUS status = NetUserGetInfo(NULL, userName, 1, (LPBYTE*)&ui);
    if (status != NERR_Success)
    {
        std::wcout << L"NetUserGetInfo failed for user " << userName
                   << L". Status: " << status << std::endl;
        return 1;
    }

    std::wcout << L"User name: " << ui->usri1_name << std::endl;
    std::wcout << L"Comment: " << (ui->usri1_comment ? ui->usri1_comment :
L"(none)") << std::endl;

    NetApiBufferFree(ui);
    return 0;
}
```

кінець прикладу 2

Створення та вилучення локальної групи користувачів. Програма створює тестову локальну групу, очікує натискання Enter, а потім сама ж вилучає цю групу – такий «самоочисний» порядок дій зменшує ризик випадково залишити в системі тестові об'єкти. Запускати приклад потрібно від імені адміністратора (Run as administrator); без цих прав виклик NetLocalGroupAdd завершиться помилкою з кодом, що відповідає недостатнім правам доступу.

Приклад 3 – Створення та вилучення локальної групи користувачів

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

#pragma comment(lib, "netapi32.lib")

int main()
{
    const wchar_t* groupName = L"TestGroupOS";

    LOCALGROUP_INFO_1 groupInfo;
    groupInfo.lgrpi1_name = (LPWSTR)groupName;
    groupInfo.lgrpi1_comment = (LPWSTR)L"Test group for OS lab";

    NET_API_STATUS status = NetLocalGroupAdd(NULL, 1, (LPBYTE)&groupInfo, NULL);
    if (status != NERR_Success)
    {
        std::cout << "NetLocalGroupAdd failed. Status: " << status << std::endl;
        std::cout << "(Administrator privileges are required.)" << std::endl;
        return 1;
```

```
    }
    std::wcout << L"Group \"" << groupName << L"\" created successfully." <<
std::endl;

    std::cout << "Press Enter to delete the test group..." << std::endl;
    std::cin.get();

    status = NetLocalGroupDel(NULL, groupName);
    if (status != NERR_Success)
    {
        std::cout << "NetLocalGroupDel failed. Status: " << status << std::endl;
        return 1;
    }
    std::cout << "Group deleted successfully." << std::endl;

    return 0;
}
```

кінець прикладу 3

Застереження: ця програма вносить реальні зміни в базу облікових записів операційної системи (хай і тимчасово). Перед виконанням лабораторної роботи на власному чи навчальному комп'ютері переконайтеся, що це дозволено політикою використання обладнання; **за можливості тестуйте подібні приклади у віртуальній машині, а не на основній робочій системі.**

Завдання для самостійного виконання:

1. Модифікувати приклад 1 так, щоб програма додатково виводила список SID усіх груп, до яких належить поточний користувач (клас TokenGroups функції GetTokenInformation).

2. Модифікувати приклад 2 так, щоб ім'я користувача, інформацію про якого потрібно отримати, вводилося з клавіатури (а не визначалося автоматично функцією GetUserNameW), і коректно обробити випадок, коли вказаного користувача не існує.

3. Написати програму, яка визначає SID за іменем користувача чи групи, заданим з клавіатури, за допомогою функції LookupAccountName (а не через маркер доступу поточного процесу, як у прикладі 1).

4. Скласти звіт про виконання лабораторної роботи: опис використаних функцій керування безпекою, постановка задачі, повний листинг програм, результати виконання (скриншоти консолі) і висновки.

5. При захисті лабораторної роботи підготувати відповіді на контрольні питання.

Контрольні питання:

1. Охарактеризуйте основні завдання забезпечення безпеки: аутентифікацію, авторизацію, аудит.

2. Що таке конфіденційність, цілісність і доступність даних? Що означає термін «відмова в обслуговуванні»?

3. Що таке матриця доступу? Чим списки контролю доступу (ACL) відрізняються від можливостей (capabilities)?

4. Що таке елемент контролю доступу (ACE)? З яких полів він складається?

5. Охарактеризуйте архітектуру безпеки Windows: призначення компонентів winlogon, LSASS, SAM, SRM.

6. Що таке ідентифікатор безпеки (SID)? Якою функцією Win32 API визначають SID за іменем користувача?

7. Що таке маркер доступу (access token)? Яку інформацію він містить?

8. Якими функціями отримують маркер доступу процесу та інформацію з нього?

9. Якими функціями NetAPI керують обліковими записами користувачів? Чим відрізняються права, потрібні для читання інформації, від прав, потрібних для створення чи вилучення облікового запису?

10. Якими функціями NetAPI керують локальними групами користувачів?

11. Чому механізми керування доступом на основі ACL не захищають дані в разі викрадення фізичного носія?

12. Охарактеризуйте принципи роботи технології EFS (Encrypting File System).

13. На яких рівнях стека протоколів TCP/IP можна реалізувати шифрування каналів зв'язку? Чим відрізняються ці підходи?

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Tanenbaum A. S., Bos H. Modern Operating Systems. 5th ed. Hoboken : Pearson, 2022.

2. Silberschatz A., Galvin P. B., Gagne G. Operating System Concepts. 10th ed. Hoboken : Wiley, 2021. 1040 p.

3. Allievi A., Ionescu A., Russinovich M. E., Solomon D. A. Windows Internals, Part 2. 7th ed. Redmond : Microsoft Press, 2021. 881 p.

4. Forshaw J. Windows Security Internals: A Deep Dive into Windows Authentication, Authorization, and Auditing. San Francisco : No Starch Press, 2024. 608 p.

5. Stroustrup B. A Tour of C++. 3rd ed. Boston : Addison-Wesley, 2022. 320 p.

6. About Processes and Threads. Win32 apps. Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads (дата звернення: 21.06.2026).

7. Synchronization Objects. Win32 apps. Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/win32/sync/synchronization-objects (дата звернення: 21.06.2026).

8. About Memory Management. Win32 apps. Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/win32/memory/about-memory-management (дата звернення: 21.06.2026).

9. Getting Started With Winsock. Win32 apps. Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/win32/winsock/getting-started-with-winsock (дата звернення: 21.06.2026).

10. File Management (Local File Systems). Win32 apps. Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/win32/fileio/file-management (дата звернення: 21.06.2026).

Операційні системи: конспект лекцій з навчальної дисципліни для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Комп'ютерні науки» галузі знань F Інформаційні технології спеціальності F3 Комп'ютерні науки денної та заочної форм навчання Частина 2/ уклад. Р.А. Хиць, Ю.Й. Тулашвілі. Луцьк: ЛНТУ. 2026. 78 с.

Комп'ютерний набір і верстка: Р.А. Хиць

Підписано до друку 2026 р.
Формат 60х 84/16. Гарнітура Times New Roman.
Папір офсетний 80 г/м2. Друк офсетний.
Ум. друк. арк. 7,5. Обл.-вид. арк. 75.
Наклад 50 прим. Зам. №

Інформаційно-видавничий відділ
Луцького національного технічного університету
43018, Луцьк-18, вул. Львівська,75.
Друк – ІВВ ЛНТУ