МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет водного господарства
та природокористування
Кафедра комп'ютерних наук

04-05-05

Методичні вказівки

до виконання лабораторних робіт з дисципліни
“Системне програмне забезпечення”

для студентів спеціальності
“ Комп’ютерні науки та інформаційні технології”
спеціалізації “Комп’ютерний еколого-економічний моніторинг”

Частина І

Рекомендовано
науково-методичною комісією
зі спеціальності
122 “Комп’ютерні науки
та інформаційні технології”
Протокол № 2 від 25.11.2016 р.

Рівне-2017

Методичні вказівки до виконання лабораторних робіт з дисципліни "Системне програмне забезпечення" для студентів спеціальності "Комп'ютерні науки та інформаційні технології" спеціалізації "Комп'ютерний еколого-економічний моніторинг". Частина І / О.М. Гладка, І.М. Карпович, Л.В. Зубик – Рівне: НУВГП, 2017. – 47 с.

Укладачі:
О.М. Гладка, канд. техн. наук, доцент кафедри комп'ютерних наук;
І.М. Карпович, канд. фіз.-мат. наук, доцент кафедри комп'ютерних наук;
Л.В. Зубик, канд. пед. наук, ст. викл. кафедри комп'ютерних наук.

Відповідальний за випуск:
Ю.Й. Тулашвілі, доктор пед. наук, професор, завідувач кафедри комп'ютерних наук.

ЗМІСТ

ВСТУП

Дисципліна "Системне програмне забезпечення" є однією з основних професійно-орієнтованих дисциплін, за якими ведуть підготовку спеціалістів в галузі сучасних інформаційних технологій. Ця дисципліна вивчає засоби та принципи, які є базовими для сучасної обчислювальної техніки і мають першочергове значення для розробки та експлуатації апаратних та програмних засобів комп'ютерних систем.

До системного програмного забезпечення належить програмне забезпечення найнижчого рівня. Таким програмним забезпеченням є: операційні системи, системи управління файлами, інтерфейсні оболонки для взаємодії користувача з операційними системами, системи програмування, утиліти. Будь-який з компонентів прикладного програмного забезпечення обов'язково працює під управлінням операційної системи.

Метою вивчення дисципліни "Системне програмне забезпечення" є вивчення теорії і дослідження методів розробки та експлуатації операційних систем і систем програмування, а також їх елементів.

Лабораторна робота № 1.

УПРАВЛІННЯ ПРОЦЕСАМИ В ОС WINDOWS

Мета: Вивчення процесів і потоків в операційній системі Windows.

Хід роботи:

1. Ознайомитися із завданням, яке передбачає розробку програми, що запускає додаткові процеси.
2. Використовуючи вивчені механізми, розробити і налагодити:
   а. програму, що реалізовує отримане завдання (запуск процесів);
   б. програму, яка є додатковим процесом.
3. Пояснити роботу вивчених механізмів за кодом.

В ОС типу *Windows NT/2000/XP* створення процесу здійснюється за допомогою виклику функції Win32 API, яка описується мовою Сі наступним чином:

```
BOOL CreateProcess (
   PCTSTR pszApplicationName, // ім'я файлу
   PTSTR pszCommandLine, // командний рядок
   PSECURITY_ATTRIBUTES psaProcess, // атрибути
                                    //захисту процесу
   PSECURITY_ATTRIBUTES psaThread, // атрибути
                                    //захисту потоку
   BOOL bInheritHandles, // успадкування описів
   DWORD fwdCreate, // прапорці процесу
   PVOID pvEnvironment, // змінні оточення
   PCTSTR pszCurDir, // поточний каталог
   PSTARTUPINFO psiStrartInfo,
      // Початкова інформація при створенні процесу
   PPROCESS_INFORMATION ppiProcInfo); //опис процесу
```

Функція `CreateProcess` повертає значення TRUE, якщо системі вдасться створити процес і початковий потік, при цьому створеному об'єкту ядра буде присвоєно унікальний ідентифікатор.

Процес в *Windows NT/2000/XP* можна завершити за допомогою викликів функцій Win32 API під назвою `ExitProcess` і `TerminateProcess`, що описані в такий спосіб:

```
VOID ExitProcess (
    UINT fuExitCode); // код завершення процесу
```

Ця функція не повертає значення.

```
VOID TerminateProcess (
    HANDLE hProcess, // опис процесу, що завершується
    UINT fuExitCode); // код завершення процесу
```

Ця функція також не повертає значення. Вона відрізняється від `ExitProcess` тим, що її може викликати будь-який процес і потік. Обидві функції використовувати не варто, оскільки процес закінчується, коли завершують роботу всі його потоки.

Наступний програмний код просто створить новий процес і запустить калькулятор.

```
#include <windows.h>
void WinMain ()
{
STARTUPINFO start = {sizeof (start)};
PROCESS_INFORMATION procinfo;
TCHAR CommandLine [] = TEXT ("CALC");

CreateProcess (NULL, CommandLine, NULL, NULL,
     FALSE, 0, NULL, NULL, & start, & procinfo);
}
```


Варіанти завдань до лабораторної роботи № 1

1. Розробити дві програми. Перша обчислює суму та добуток чисел від L до U і виводить отримані значення на екран. Друга програма запускає першу як новостворений процес.
2. Розробити дві програми. Перша обчислює число Фібоначчі за номером *n*, введеним користувачем, та формулою $F_i = F_{i-1} + F_{i-2}$, $F_0 = F_1 = 1$ (i=2, …, n) і виводить його на екран. Друга програма запускає першу як новостворений процес.
3. Розробити дві програми. Перша приймає від користувача рядок, що зберігає знакове ціле число, і виводить на екран рядковий еквівалент цього числа прописом (наприклад, введення «-1211» повинно призводити до результату «мінус тисяча двісті

одинадцять»). Друга програма запускає першу як новостворений процес.

4. Розробити дві програми. Перша приймає від користувача рядок, що зберігає число зі знаком і плаваючою крапкою, і виводить на екран рядковий еквівалент цього числа прописом (наприклад, введення «-12.11» повинно призводити до результату «мінус дванадцять цілих одинадцять сотих»). Друга програма запускає першу як новостворений процес.
5. Розробити дві програми. Перша приймає від користувача два рядки. Далі, якщо обидва рядки мвстять цілі числа зі знаком, то на екран виводиться сума чисел, в іншому випадку – конкатенація двох введених рядків. Друга програма запускає першу як новостворений процес.
6. Розробити дві програми. Перша приймає від користувача дві прямокутних матриці, а потім виводить на екран їх суму і добуток. Друга програма запускає першу як новостворений процес.
7. Розробити дві програми. Перша приймає від користувача одновимірний цілочисельний масив, впорядковує його за зростанням і виводить на екран. Друга програма запускає першу як новостворений процес.
8. Розробити дві програми. Перша приймає від користувача одновимірний масив чисел з плаваючою крапкою, впорядковує його за спаданням і виводить на екран. Друга програма запускає першу як новостворений процес.
9. Розробити дві програми. Перша приймає від користувача одновимірний масив рядків, впорядковує його будь-яким із, так званих, «поліпшених алгоритмів» сортування масивів і виводить на екран. Друга програма запускає першу як новостворений процес.
10. Розробити дві програми. Перша приймає від користувача квадратну матрицю, обчислює суму елементів, що лежать на головній та побічній діагоналях, і виводить на екран. Друга програма запускає першу як новостворений процес.

Лабораторна робота № 2.

РОЗРОБКА БАГАТОПОТОКОВИХ ДОДАТКІВ

Мета: Програмна реалізація багатопотокових додатків в операційній системі Windows.

Хід роботи:

1. Ознайомитися із завданням, яке передбачає розпаралелювання роботи програми на кілька потоків.
2. Використовуючи вивчені механізми, розробити програму, яка реалізує отримане завдання.

В сучасних ОС користувачам пропонується кілька способів організації паралельної роботи, основними об'єктами для реалізації яких є процеси і потоки. Процеси – це програми на етапі виконання. Потоки – це складові процесу. Однак, з точки зору розподілу ресурсів, саме потоки є головними, оскільки їм, а не процесам надається на визначений час центральний процесор для виконання будь-якої роботи. Розглянемо кілька функцій Win API, що виконують деякі операції над потоками.

Функція `CreateThread` створює потік для виконання всередині адресного простору процесу, що його викликає:

```
HANDLE CreateThread (
   PSECURITY_ATTRIBUTES lpThreadAttributes,
      // Атрибути захисту потоку
   DWORD dwStackSize, // початковий розмір стека
                      //потоку в байтах
   PTHREAD_START_ROUTINE lpStartAddress,
      // Покажчик на функцію потоку
   PVOID lpParameter, // параметр для нового потоку
   DWORD dwCreationFlags, //прапорці створення потоку
   PDWORD lpThreadId
      // Покажчик, на ідентифікатор, що повертається
);
```

При успішному завершенні функції повертається дескриптор нового потоку. При невдалому завершенні функції повертається `NULL`.

Параметри:

`LpThreadAttributes` – покажчик на структуру `SECURITY_ATTRIBUTES`, яка визначає, чи може повернений дескриптор бути успадкованим дочірніми процесами. Якщо `lpThreadAttributes NULL`, то дескриптор не може бути успадкованим.

`DwStackSize` – визначає розмір в байтах стека для нового потоку. Якщо визначено 0, то розмір стека за замовчуванням дорівнює розміру стека породжуючого потоку. Стек розподіляється автоматично в просторі пам'яті процесу і звільняється при завершенні потоку.

`LpStartAddress` – початкова адреса нового потоку. Це зазвичай адреса функції, оголошеної за замовчуванням про виклики Win32 API, яка приймає означений 32-розрядний покажчик як параметр і повертає 32-розрядний код завершення. Прототип: `DWORD WINAPI ThreadFunc (LPVOID).`

`LpParameter` – визначає єдине 32-розрядне значення параметра, яке передається потоку.

`DwCreationFlags` – визначає додаткові прапорці, які керують створенням потоку. Якщо прапорець визначений `CREATE_SUSPENDED`, то потік створюється в стані очікування. Він не буде виконуватися, поки не буде виклику входу функції `ResumeThread`. Якщо це значення 0, то потік виконується негайно після створення.

`LpThreadId` – покажчик на 32-розрядну змінну, яка отримує значення ідентифікатора потоку.

Потік можна завершити примусово за допомогою виклику наступних функцій Win32 API:

```
VOID ExitThread (
   DWORD ExitCode); // код завершення потоку
і
VOID TerminateThread (
   HANDLE hThread, // потік, який потрібно завершити
   DWORD ExitCode // код завершення потоку
);
```

Для прикладу наведено фрагмент програми, яка обчислює добуток всіх чисел від 1 до 100.

```
// Функція потоку
DWORD WINAPI ThreadFuction (PVOID Parametr)
{Int dob = 1; // Результат добутку
  int ii, * kk;
  kk = (int *) Parametr;
for (ii = * k; ii <(* kk) + 50; ii ++) dob * = ii;
return dob;
}

// Код викликів функцій для створення потоків
DWORD idThread;
int k1 = 1; k2 = 51;
HANDLE h1, h2;
// Створюється два потоки в стані очікування
h1 = CreateThread (NULL, 0, ThreadFunction, &
k1, CREATE_SUSPENDED, & idThread);
h2 = CreateThread (NULL, 0, ThreadFunction, &
k2, CREATE_SUSPENDED, & idThread);
// Виконання потоків
ResumeThread (h1);
ResumeThread (h2);
```

В даному коді зустрілася дуже корисна Win32 API функція `ResumeThread`, яка відновлює виконання припиненого потоку. Її опис:

```
DWORD ResumeThread (
   HANDLE hThread // потік, який потрібно відновити
);
```

Якщо виклик цієї функції успішний, то повертається попереднє значення лічильника простоїв даного потоку, в іншому випадку – `0xFFFFFFFF`.

Виконання окремого потоку можна припиняти кілька разів (точно таке ж число разів він має поновлюватись), а реалізується це

викликом функції SuspendThread, що описується наступним чином:

```
DWORD SuspendThread (
   HANDLE hThread // потік, який потрібно призупинити
);
```

Потік може повідомити ОС, щоб вона не виділяла йому процесор деякий час, вказаний в мілісекундах:

```
VOID Sleep (DWORD MilliSeconds);
```

В ОС *Windows 2000/XP* є також функція, яка знаходить і відкриває потік за ідентифікатором.

```
HANDLE OpenThread (
   DWORD DesiredAccess,
   BOOL InheritHandle,
   DWORD dwThreadId
);
```


Варіанти завдань до лабораторної роботи № 2

1. Розробити програму, яка обчислює суму і добуток чисел від L до U. Обчислення суми і добутку оформити як дві функції потоків. Користувачем вводяться значення меж діапазону, далі – запускаються два відповідні потоки, а потім на екран виводяться отримані значення.
2. Розробити програму, яка обчислює число Фібоначчі за номером n, введеним користувачем, та формулою $F_i = F_{i-1} + F_{i-2}$, $F_0 = F_1 = 1$ (i=2, …, n). Обчислення числа Фібоначчі оформити як функцію потоку. Після завершення функції потоку програма виводить результат.
3. Розробити програму для перекладу цілого числа із знаком в його рядковий еквівалент прописом. Переклад числа оформити як функцію потоку. Введення числа відбувається до запуску потоку, а виведення результату – після його завершення.
4. Розробити програму для перекладу знакового числа з плаваючою крапкою в його рядковий еквівалент прописом.

Переклад числа оформити як функцію потоку. Введення числа відбувається до запуску потоку, а виведення результату – після його завершення.

5. Розробити програму, яка здійснює введення двох рядків, заданих користувачем. Далі, якщо обидві рядки містять цілі числа зі знаком, то на екран виводиться сума чисел, в іншому випадку – конкатенація введених рядків. Перевірку на відповідність рядка цілому числу, обчислення суми чисел та конкатенації рядків оформити як три різних функції потоку. Введення рядків здійснюється до запуску всіх потоків, а виведення результатів – після їх завершення.

6. Розробити програму, яка обчислює суму і добуток двох матриць. Виконання цих операцій оформити як дві функції потоку. Спочатку програма здійснює введення елементів матриць, далі запускає обидва потоки, а потім виводить результати на екран.

7. Розробити програму для упорядкування одновимірного цілочисельного масиву. Сортування масиву оформити як функцію потоку. Спочатку виконується введення елементів масиву, потім запускається потік і далі – виведення упорядкованого масиву.

8. Розробити програму для упорядкування одновимірного масиву чисел з плаваючою крапкою. Сортування масиву оформити як функцію потоку. Спочатку виконується введення елементів масиву, потім запускається потік і далі – виведення упорядкованого масиву.

9. Розробити програму для упорядкування одновимірного масиву рядків. Сортування масиву оформити як функцію потоку. Спочатку виконується введення елементів масиву, потім запускається потік і далі – виведення упорядкованого масиву.

10. Розробити програму для обчислення суми елементів, що лежать на головній і побічній діагоналях квадратної матриці. Виконання цієї операції оформити як функцію потоку. Введення елементів матриці здійснювати до запуску потоку, а виведення результатів – після його завершення.

Лабораторна робота № 3.

УПРАВЛІННЯ ПРІОРИТЕТАМИ ПОТОКІВ

Мета: Визначення і зміна пріоритетів потоків в операційній системі Windows.

Хід роботи:

1. Ознайомитися із завданням, яке передбачає розпаралелювання роботи на кілька потоків з різними рівнями пріоритетів.
2. Використовуючи вивчені механізми, розробити програму, що реалізує отримане завдання з вимірюванням часу роботи кожного потоку.

В основі багатьох алгоритмів планування процесів лежить концепція пріоритетного обслуговування. Потоки мають заздалегідь відому характеристику – пріоритет, на підставі якого визначається порядок виконання потоків. Пріоритет – це число, що характеризує ступінь привілегійованості потоку при використанні ресурсів обчислювальної системи, зокрема, (і в першу чергу) часу процесора: чим вищий пріоритет, тим вищими є привілеї, а значить, потік менше часу буде проводити в чергах. У більшості ОС, що підтримують потоки, їх пріоритет безпосередньо пов'язаний з пріоритетом процесу, в рамках якого виконується потік. Значення иріоритету з опису процесу використовується при призначенні пріоритету потокам цього процесу.

В багатьох ОС передбачається можливість зміни пріоритетів протягом життя потоку. Зміни пріоритету можуть відбуватися з ініціативи самого потоку, коли він звертається з відповідним викликом до ОС, або з ініціативи користувача, коли він використовує відповідну команду. Крім того, ОС сама може змінювати пріоритети потоків в залежності від ситуації, що складається в системі. В останньому випадку пріоритети називаються динамічними, на відміну від незмінних, фіксованих пріоритетів.

В ОС *Windows 2000* процесу можна задати клас пріоритету. Підтримується шість класів пріоритетів:

– простоюючий (`IDLE_PRIORITY_CLASS`) – потоки виконуються, якщо система не зайнята іншою роботою;

– нижчий від нормального (`BELOW_NORMAL_PRIORITY_CLASS`);

– нормальний (NORMAL_PRIORITY_CLASS), якщо у потоків немає особливих потреб;

– вищий від нормального (ABOVE_NORMAL_PRIORITY_CLASS);

– високий (HIGH_PRIORITY_CLASS) – потоки негайно реагують на події;

– реального часу (REALTIME_PRIORITY_CLASS) – коли потоки також негайно реагують на події і можуть витісняти навіть потоки ОС.

Клас пріоритету встановлюється за допомогою функції:

```
BOOL SetPriorityClass (
   HANDLE Process,
   DWORD Priority);
```

Процес може поміняти свій власний клас пріоритету, наприклад:

```
SetProrityClass (GetCurrentProcess (),
HIGH_PRIORITY_CLASS);
```


Щоб дізнатися клас пріоритету, можна скористатися функцією:

```
DWORD GetPriorityClass (
   HANDLE Process);
```

Вибравши клас пріоритету для процесу, не потрібно забувати про потоки. В ОС *Windows 2000* підтримується сім відносних пріоритетів потоків:

– THREAD_PRIORITY_TIME_CRITICAL (рівень пріоритету 31 в класі REALTIME і 15 в інших);

– THREAD_PRIORITY_HIGHEST (на два рівні вище для поточного класу, для NORMAL рівень пріоритету дорівнює 10);

– THREAD_PRIORITY_ABOVE_NORMAL (на один рівень вище, для NORMAL рівень пріоритету дорівнює 9);

– THREAD_PRIORITY_NORMAL (звичайний пріоритет для класу, для NORMAL рівень дорівнює 8);

– THREAD_PRIORITY_BELOW_NORMAL (на один рівень нижче, для NORMAL рівень пріоритету дорівнює 7);

– THREAD_PRIORITY_LOWEST (на два рівні нижче від поточного класу, для NORMAL рівень пріоритету дорівнює 6);

– THREAD_PRIORITY_IDLE (16 – для REALTIME і 1 в інших класах).

Задати відносний пріоритет потоку можна за допомогою функції:

```
BOOL SetThreadPriority (
   HANDLE Thread,
   int Priority);
```

Дізнатися відносний пріоритет потоку дозволяє функція:

```
int GetThreadPriority (
   HANDLE Thread);
```


Наступний фрагмент коду показує, як створити потік з відносим пріоритетом HIGH, якщо за замовчуванням потік створюється з пріоритетом NORMAL.

```
DWORD ThreadId;
HANDLE Thread = CreateThread (NULL, 0,
ThreadFunction, NULL, CREATE_SUSPENDED, &
ThreadId);
SetThreadPriority (Thread,
THREAD_PRIORITY_HIGH);
ResumeThread (Thread);
CloseHandle (Thread);
```

Інколи потрібно знати, скільки часу потік витратив на виконання якихось операцій. В *Windows NT/2000/XP* така функція має вигляд:

```
BOOL GetThreadTimes (
   HANDLE Thread,
   PFILETIME Created,
      // Час з 01.01.1601 в сотнях нс до створення потоку
```

```
PFILETIME Exited,
    // Час з 01.01.1601 в сотнях нс до завершення потоку
PFILETIME Kernel,
    // Час в сотнях нс, витрачений потоком на код ОС
PFILETIME User);
    // Час в сотнях нс, витрачений потоком на код програми
```

Варіанти завдань до лабораторної роботи № 3

1. Розробити програму, яка обчислює суму і добуток чисел від L до U. Обчислення суми і добутку оформити як дві функції потоків з пріоритетами `THREAD_PRIORITY_HIGHEST` і `THREAD_PRIORITY_IDLE` відповідно. Користувачем вводяться значення меж діапазону, далі – запускаються два потоки, а потім на екран виводяться отримані результати із зазначенням часу роботи обох потоків.
2. Розробити програму, яка обчислює число Фібоначчі за номером *n*, введеним користувачем, та формулою $F_i = F_{i-1} + F_{i-2}$, $F_0 = F_1 = 1$ (i=2, …, n). Обчислення числа Фібоначчі оформити як функцію потоку. Запустити програму двічі: з пріоритетом `THREAD_PRIORITY_TIME_CRITICAL` та `THREAD_PRIORITY_NORMAL` і вивести на екран значення часу роботи потоків.
3. Розробити програму для перекладу цілого числа із знаком в його рядковий еквівалент прописом. Переклад числа оформити як функцію потоку. Запустити програму двічі: з пріоритетом `THREAD_PRIORITY_ABOVE_NORMAL` та `THREAD_PRIORITY_LOWEST` і вивести на екран значення часу роботи потоків.
4. Розробити програму для перекладу числа із знаком з плаваючою крапкою в його рядковий еквівалент прописом. Переклад числа оформити як функцію потоку. Запустити програму двічі: з пріоритетом `THREAD_PRIORITY_ABOVE_ NORMAL` та `THREAD_PRIORITY_HIGHEST` і вивести на екран значення часу роботи потоків.
5. Розробити програму, яка здійснює введення двох рядків, заданих користувачем. Далі, якщо обидва рядки містять цілі числа зі знаком, то на екран виводиться сума чисел, в іншому випадку – конкатенація введенних рядків. Перевірку на

відповідність рядка цілому числу, обчислення суми чисел та конкатенації рядків оформити як три функції потоків з пріоритетами THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_LOWEST і THREAD_PRIORITY_IDLE відповідно. Введення рядків здійснюється до запуску всіх потоків, а після їх завершення – виведення результатів із зазначенням часу роботи потоків.

6. Розробити програму, яка обчислює суму і добуток двох матриць. Виконання цих операцій оформити як дві функції потоків з пріоритетами THREAD_PRIORITY_IDLE і THREAD_PRIORITY_TIME_CRITICAL відповідно. Спочатку програма здійснює введення елементів матриць, далі запускає обидва потоки, а потім виводить результати із зазначенням часу роботи потоків.
7. Розробити програму для упорядкування одновимірного цілочисельного масиву. Сортування масиву оформити як функцію потоку. Запустити програму двічі: з пріоритетом THREAD_PRIORITY_ABOVE_NORMAL та THREAD_PRIORITY_LOWEST і вивести на екран значення часу роботи потоків.
8. Розробити програму для упорядкування одновимірного масиву чисел з плаваючою крапкою. Сортування масиву оформити як функцію потоку. Запустити програму двічі: з пріоритетом потоку THREAD_PRIORITY_ABOVE_NORMAL та THREAD_PRIORITY_HIGHEST і вивести на екран значення часу роботи потоків.
9. Розробити програму для упорядкування одновимірного масиву рядків. Сортування масиву оформити як функцію потоку. Запустити програму двічі: з пріоритетом потоку THREAD_PRIORITY_BELOW_NORMAL та THREAD_PRIORITY_ABOVE_NORMAL і вивести на екран значення часу роботи потоків.
10. Розробити програму для обчислення суми елементів, що лежать на головній і побічній діагоналях квадратної матриці. Виконання цієї операції оформити як функцію потоку. Запустити програму двічі: з пріоритетом потоку THREAD_PRIORITY_NORMAL та THREAD_PRIORITY_LOWEST і вивести на екран значення часу роботи потоків.

Лабораторна робота № 4.

СИНХРОНІЗАЦІЯ ПОТОКІВ У СЕРЕДОВИЩІ ОС WINDOWS

Мета: Вивчення об'єктів синхронізації потоків в операційній системі Windows.

Хід роботи:

1. Ознайомитися з вихідними текстами прикладів використання об'єктів синхронізації.
2. Ознайомитися із завданням, яке передбачає використання деякого об'єкту синхронізації.
3. Розробити програму у відповідності з отриманим завданням та прикладами використання відповідних об'єктів.

Існує великий клас засобів ОС, за допомогою яких забезпечується взаємна синхронізація процесів і потоків. Потреба в синхронізації потоків пов'язана зі спільним використанням ресурсів обчислювальної системи. Синхронізація необхідна для виключення змагань і тупиків при обміні даними між потоками, поділі даних, при доступі до центрального процесора і пристроїв введення/виведення. В багатьох ОС ці засоби називаються ***засобами взаємодії між процесами*** (IPC), оскільки зазвичай до них відносяться не тільки засоби міжпроцесної синхронізації, але і засоби міжпроцесного обміну даними.

Синхронізація полягає в узгодженні виконання процесів і потоків шляхом призупинення потоку до настання деякої події і наступній його активації при настанні цієї події. ОС може надавати великий набір засобів синхронізації. Дуже важливим поняттям синхронізації є поняття «критичної секції» програми. **Критична секція** – це частина програми, результат виконання якої може непередбачувано змінюватися, якщо змінні, що відносяться до цієї частини програм, змінюються іншими потоками в той час, коли виконання цього коду ще не закінчено. Критична секція завжди визначається по відношенню до конкретних критичних даних, при неузгодженій зміні яких можуть виникнути небажані ефекти. У всіх потоках, які працюють з критичними даними, повинна бути визначена критична секція, яка, в загальному випадку, складається з різних послідовностей команд. Взаємне виключення дозволяє забезпечити знаходження в критичній секції тільки одного потоку.

Для синхронізації потоків одного процесу програміст може використувати глобальні **блокуючі змінні**. З цими змінними, до яких всі потоки мають прямий доступ, програміст працює, не звертаючись до системних викликів ОС. Кожному набору критичних даних ставиться у відповідність двійкова змінна, якій потік привласнює значення 0, коли він входить в критичну секцію, і значення 1, коли він її залишає. Недолік: під час перебування одного потоку в критичній секції, другий потік, що вимагає той же ресурс, отримавши доступ до процесора, буде із завидною регулярністю опитувати блокуючу змінну, марно витрачаючи процесорний час. Для усунення цього недоліку в багатьох ОС передбачені спеціальні системні виклики для роботи з критичними секціями.

У *Windows NT/2000/XP* перед зміною критичних даних, потік виконує системний виклик `EnterCriticalSection`, в рамках якого спочатку виконується перевірка блокуючої змінної, що відображає стан ресурсу. Якщо ресурс зайнятий (значення блокуючої змінної дорівнює 0), потік блокується і робиться відмітка про те, що потік повинен бути активований, коли відповідний ресурс звільниться. Потік, який в цей час вимагає даний ресурс, після виходу з критичної секції повинен виконати виклик `LeaveCriticalSection`, в результаті якого блокуюча змінна отримує значення 1 (ресурс вільний), а ОС переглядає чергу потоків, що очікують цей ресурс, і переводить перший потік в стан готовності. Ці, а також деякі інші функції Win32 API для роботи з критичними секціями наведені нижче.

```
VOID EnterCriticalSection (PCRITICAL_SECTION
Section);

VOID LeaveCriticalSection (PCRITICAL_SECTION
Section);
```

Коли жоден потік не використовує критичну секцію, її можна вилучити:

```
VOID DeleteCriticalSection (PCRITICAL_SECTION
Section);
```

Коли критична секція є локальною, її потрібно ініціалізувати:

```
VOID InitializeCriticalSection
(PCRITICAL_SECTION Section);
```

Спробувати увійти в критичну секцію без блокування потоку можна за допомогою функції

```
BOOL TryEnterCriticalSection (PCRITICAL_SECTION
Section);
```

Вона повертає FALSE, якщо ресурс зайнятий іншим потоком, і TRUE, якщо потік захопив потрібний ресурс.

Для того, щоб організувати доступ до двох ресурсів, потрібно створити дві критичні секції, що і демонструє наступний фрагмент коду:

```
int Numbers [500]; // Перший ресурс, що розділяється
CRITICAL_SECTION Nums;
double Doubles [500];
CRITICAL_SECTION DoubleNums; // Другий ресурс, що
                             // розділяється

DWORD ThFunction (PVOID Parametr) // функція потоку
{
// Вхід в обидві критичні секції
EnterCriticalSection (& Nums);
EnterCriticalSection (& DoubleNums);
   // У цьому коді потрібно одночасний доступ
   // до обох ресурсів, що розділяються
for (int j = 0; j <500; j ++) Doubles [j] =
Numbers [j] = 500 - j;
   // Залишаємо критичні секції в зворотному порядку
LeaveCriticalSection (& DoubleNums);
LeaveCriticalSection (& Nums);
return 0;
}
```

Ефект взаємного блокування може виникнути, якщо спробувати добавити ще одну функцію потоку, де проводиться використання тих же критичних секцій, але в зворотному порядку.

```
DWORD ThFunction1 (PVOID Parametr) //функція потоку
{// Вхід в обидві критичні секції
EnterCriticalSection (& DoubleNums);
EnterCriticalSection (& Nums);
   // У цьому коді потрібно одночасний доступ
   // до обох ресурсів, що розділяються
for (int j = 0; j <500; j ++) Doubles [j] =
Numbers [j] = 500 - j;
   // Залишаємо критичні секції в зворотному порядку
LeaveCriticalSection (& Nums);
LeaveCriticalSection (& DoubleNums);
return 0;
}
```

Тут існує ймовірність того, що `ThFunction` займає критичну секцію `Nums`, а потік з функцією `ThFunction1` захоплює `DoubleNums`. І тепер, яка б функція не виконувалася, вона не зможе ввійти в іншу, так необхідну їй, критичну секцію.

Чудова властивість критичних секцій полягає в тому, що вони не використовують перехід з режиму користувача в режим ядра. Тому швидкість роботи досить висока, і цей об'єкт може використуватися для більшості програм, що вимагають синхронізації. До недоліків можна віднести те, що їх можна використовувати для синхронізації потоків, що належать тільки одному і тому ж процесу.

Узагальненням блокуючих змінних є так звані **семафори Дейкстри**. Замість двійкових змінних Едсгер Дейкстра запропонував використовувати змінні, які можуть приймати цілі невід'ємні значення. Такі змінні, що використовуються для синхронізації, отримали назву семафорів.

Для роботи з семафора вводяться два примітиви (дії) *P* і *V*. Нехай змінна *S* являє собою семафор, тоді дії *V(S)* і *P(S)* визначаються так:

- *V(S):* змінна *S* збільшується на 1. Вибірка, інкремент і зберігання не можуть бути перервані. До змінної *S* немає доступу іншим потокам під час виконання цієї операції.

- *P(S)*: зменшення *S* на 1, якщо це можливо. Якщо *S* дорівнює 0, то потік, що викликає операцію *P*, поки декремент стане можливим. Перевірка і зменшення є неподільною операцією.

В окремому випадку, коли семафор може приймати тільки значення 0 і 1, він перетворюється в блокуючу змінну, яку часто називають двійковим семафором. Блокуючі змінні і семафори Дейкстри не підходять для синхронізації потоків різних процесів.

ОС повинна надавати потокам системні об'єкти синхронізації, які були б видимі для всіх потоків, навіть якщо вони належать різним процесам і працюють в різних адресних просторах. Набір таких об'єктів залежить від конкретної ОС, яка створює їх за запитами користувачів. Прикладами синхронізуючих об'єктів є **системні семафори, м'ютекси, події, таймери** і ін. Робота з синхронізуючими об'єктами подібна до роботи з файлами: їх можна створювати, відкривати, закривати, знищувати. Крім того, для синхронізації можуть використовуватися файли, процеси і потоки. Всі синхронізуючі об'єкти можуть перебувати в двох станах: сигнальному (вільно) і несигнальному (зайнято). Для кожного об'єкта сенс сигнального стану залежить від типу об'єкта. Потоки за допомогою спеціального системного виклику повідомляють ОС, що вони хочуть синхронізувати своє виконання зі станом деякого об'єкта (`WaitForSingleObject` в *Windows NT/2000/XP*). Інший системний виклик може переводити об'єкт в сигнальний стан (наприклад, `SetEvent` в *Windows NT/2000/XP*). Потік може очікувати установки сигнального стану не одного об'єкта, а кількох (`WaitForMultipleObjects` в *Windows NT/2000/XP*). При цьому він може попросити ОС активувати його при установці або одного зазначеного об'єкта, або всіх. Потік може як аргумент системного виклику очікування вказати також максимальний час, який він буде очікувати переходу об'єкта в сигнальний стан, після чого ОС повинна активувати його в будь-якому випадку. Установки деякого об'єкта в сигнальний стан можуть очікувати відразу кілька потоків. Залежно від об'єкта в стан готовності можуть переводитися або всі потоки, що очікують цю подію, або один з них.

В ОС *Windows NT/2000/XP* є досить великий набір функцій, які використовуються для організації очікування потоками переведення в сигнальний стан одного або декількох об'єктів.

```
DWORD WaitForSingleObject (
   HANDLE Object,
   DWORD Milliseconds); // визначає час очікування
           // або INFINITE – нескінченне очікування
```

У наступному коді потік блокований, поки не виконається інший потік `Th1`.

```
WaitForSingleObject (Th1, INFINITE);
```

Цей приклад демонструє значення тайм-ауту, що не дорівнює `INFINITE`.

```
DWORD dw = WaitForSingleObject (Th1, 10000);
switch (dw) {
   case WAIT_OBJECT_0: // потік завершив роботу
      break;
   case WAIT_TIMEOUT: // потік не завершився через 10с
      break;
   case WAIT_FAILED: // сталася якась помилка
      break;
}
```


Встановлювати відразу кілька очікуваних об'єктів або один із списку, можна за допомогою функції:

```
DWORD WaitForMultipleObjects (
   DWORD Counter, // кількість об'єктів ядра (від 1 до 64)
   HANDLE * Objects, // масив описів об'єктів
   BOOL WaitForAll, // очікувати всі (TRUE) або
                    // один із списку (FALSE)
   DWORD Milliseconds); // визначає час очікування
           // або INFINITE - нескінченне очікування
```

Наступний код демонструє використання цієї функції:

```
HANDLE hh [2];
hh [0] = Th1; hh [1] = Th2;
DWORD = WaitForMultipleObjects (2, hh, FALSE,
       10000);
```

```
switch (dw) {
    case WAIT_FAILED: // сталася якась помилка
       break;
    case WAIT_TIMEOUT: // потік не завершився через 10с
       break;
    case WAIT_OBJECT_0: // потік Th1 завершив роботу
       break;
    case WAIT_OBJECT_0 + 1://потік Th2 завершив роботу
       break;
}
```

Об'єкт-подія використовується для того, щоб повідомити іншим потокам про те, що деякі дії завершені. Події зазвичай використовують в тому випадку, коли якийсь потік виконує ініціалізацію, а потім сигналізує іншому потоку, що він може продовжити роботу. Ініціалізуючий потік переводить об'єкт «подія» в несигнальний стан і приступає до своїх операцій. Закінчивши, він скидає об'єкт «подія» в сигнальне перебування. Тоді інший потік, який чекав переходу події в сигнальний стан, переводиться в стан готовності. В ОС *Windows NT/2000/XP* події містять лічильник кількості користувачів і дві логічні змінні: тип події та стан. Ці об'єкти можуть бути двох типів: з автоскиданням і з ручним скиданням.

Подія створюється функцією:

```
HANDLE CreateEvent (
   PSECURITY_ATTRIBUTES Attributes, // атрибути
                                    // захисту
   BOOL ManualOrAuto, // ручне (TRUE) або
                      // автоматичне (FALSE) скидання
   BOOL Initial, // початковий стан: вільний (TRUE)
                 // або зайнятий (FALSE)
   PCTSTR Name); // символьне ім'я об'єкта
```

Створена подія може відкриватися за допомогою функції:

```
HANDLE OpenEvent (
   DWORD Access, // режим доступу
```

```
BOOL Inherit, // спадкування
PCTSTR Name); // символьне ім'я об'єкта
```

Після створення події можна управляти її станом. Для цього існують дві функції. Переведення в сигнальний стан здійснюється викликом функції:

```
BOOL SetEvent (HANDLE Event);
```

Змінити його на "зайнято" можна за допомогою функції:

```
BOOL ResetEvent (HANDLE Event);
```

Коли потік успішно дочекався події з автоскиданням, він автоматично скидається в зайнятий стан, і для нього, зазвичай, не потрібно викликати функцію `ResetEvent`.

В наступному фрагменті коду показано, як використовуються події:


```
HANDLE Event1;
int WinMain (...)
{
// Створити нагадування з ручним скиданням в
// сигнальному стані
Event1 = CreateEvent (NULL, FALSE, FALSE, NULL);
// Створюються два потоки, причому пропущені всі
// параметри, крім функції потоку
HANDLE Th1 = CreateThread (..., Function1, ...);
HANDLE Th2 = CreateThread (..., Function2, ...);
...
// Далі можна виконувати будь-які дії
...
CloseHandle (Event1);
}

DWORD WINAPI Function1 (PVOID Parametr)
{
WaitForSingleObject (Event1, INFINITE);
...
SetEvent (Event1);
```

```
return 0;
}

DWORD WINAPI Function2 (PVOID Parametr)
{
WaitForSingleObject (Event1, INFINITE);
...
SetEvent (Event1);
return 0;
}
```

Таймер очікування – це об'єкт, який самостійно переходить в сигнальний стан в певний час або через регулярні проміжки часу. Об'єкт створюється функцією:

```
HANDLE CreateWaitableTimer (
   PSECURITY_ATTRIBUTES Attributes, // атрибути
                                    // захисту
   BOOL ManualOrAuto,// ручне (TRUE) або
                // автоматичне (FALSE) скидання стану
   PCTSTR Name); // символьне ім'я об'єкта
```

Таймер очікування завжди створюється в несигнальному стані. Щоб перевести таймер у сигнальний стан досить викликати функцію:

```
HANDLE SetWaitableTimer (
   HANDLE Timer, // потрібний таймер
   const LARGE_INTEGER * DueTime,
      // коли таймер повинен спрацювати вперше
   LONG Period, // скільки разів буде спрацьовувати
             // таймер, 0 - для одного спрацьовування
   PTIMERAPCROUTINE ComplRoutine,
                // Процедура для асинхронного виклику
   PVOID ArgumentsToRoutine,
       // Аргумент для процедури асинхронного виклику
   BOOL Resume);
// Необхідний для комп'ютерів з підтримкою режиму сну
```

Щоб перевести таймер очікуваннння в несигнальний стан, потрібно викликати:

```
HANDLE CancelWaitableTimer (
   HANDLE Timer); // Потрібний таймер
```

В наступному фрагменті коду таймер налаштовується так, щоб спрацювати перший раз 29 лютого 2004 року о 17.30, і після цього – кожні три години, тобто 10800000 мілісекунд.

```
HANDLE Timer1;
SYSTEMTIME Time1;
FILETIME LocalTime, UTC_Time;
LARGE_INTEGER IntUTC;

// Створюється таймер з автоскиданням
Timer1 = CreateWaitableTimer (NULL, FALSE, NULL);
// Задаються параметри для таймера
Time1.wYear = 2004; Time1.wMonth = 2;
Time1.wDay = 29; Time1.wHour = 17;
Time1.wMinute = 30; Time1.wSecond = 0;
Time1.wMilliseconds = 0;

SystemTimeToFileTime (& Time1, & LocalTime);
LocalFileTimeToFileTime (& LocalTime, & UTC_Time);
IntUTC.LowPart = UTC_Time.dwLowDateTime;
IntUTC.HighPart = UTC_Time.dwHighDateTime;

// Встановлюється таймер
SetWaitableTimer (Timer1, & IntUTC, 10800000,
                  NULL, NULL, FALSE);
...
CloseHandle (Timer1);
```

Час спрацьовування можна також встановлювати не в абсолютних одиницях, а у відносних, які розраховуються в блоках по 100 нс (тобто 0.1 сек. дорівнює мільйону таких блоків). Число при цьому має бути від'ємним. В наступному коді показано, як встановити таймер на спрацьовування через 20 секунд після виклику відповідної функції.

```
HANDLE Timer1;
LARGE_INTEGER LargeInt;
// Створюється таймер з автоскиданням
Timer1 = CreateWaitableTimer (NULL, FALSE, NULL);
// Задаються параметри для таймера, який повинен спрацю
// вати через 20 сек. Час береться в блоках по 100 нс.
int UnitsBySeconds = 10000000;
LargeInt = -20 * UnitsBySeconds;
// Встановлюється таймер, який спрацює спочатку через
// 20 сек, а потім кожні три години
SetWaitableTimer (Timer1, & LargeInt, 10800000,
                  NULL, NULL, FALSE);
...
CloseHandle (Timer1);
```

Ще одне зауваження щодо таймерів очікування. Коли звільняється таймер з ручним скиданням, то всі потоки, які очікували даний об'єкт, відновлюють свою роботу, а коли в сигнальний стан переходить таймер з автоскиданням, поновлюється виконання тільки одного потоку.

Семафор в цілому був описаний вище. Що стосується ОС *Windows NT/2000/XP*, то семафор тут є об'єктом ядра, і найчастіше такі об'єкти використовуються для обліку ресурсів. Семафори, крім інших параметрів, характерних для багатьох об'єктів ядра, мають ще два специфічних: один використовується для установки максимально можливого числа ресурсів, а інший – це лічильник справжньої кількості ресурсів.

Для семафорів визначені такі правила роботи:

1. Семафор переходить в сигнальний стан, якщо значення лічильника ресурсів більше, ніж 0.
2. Семафор зайнятий, якщо значення лічильника дорівнює 0.
3. Не допускається установка від'ємного значення лічильника.
4. Лічильник не може мати значення, яке більше від максимальної кількості ресурсів.

Семафор створюється викликом функції:

```
HANDLE CreateSemaphore (
   PSECURITY_ATTRIBUTES Attributes, // атрибути
                                    // захисту
   LONG Initial, // кількість ресурсів, доступних
                 // спочатку
   LONG Maximum, // максимальна кількість ресурсів
   PCTSTR Name); // символьне ім'я об'єкта
```

Отримати опис існуючого семафора можна за допомогою функції:

```
HANDLE OpenSemaphore (
   DWORD Access, // режим доступу
   BOOL Inherit, // спадкування
   PCTSTR Name); // символьне ім'я об'єкта
```

Потік може збільшити лічильник справжньої кількості ресурсів, викликавши функцію:

```
BOOL ReleaseSemaphore (
   HANDLE Semaphore, // описувач семафора
   LONG ReleaseCount, // наскільки збільшити
                  // лічильник ресурсів (зазвичай 1)
   PLONG PreviousCount); // початкове значення
                        // лічильника (зазвичай NULL)
```

Процес може використовувати семафор, щоб обмежити число вікон, які він створить. Спочатку він використовує функцію `CreateSemaphore`, щоб створити семафор і визначити початкове та максимальне значення лічильника.

```
HANDLE Semaphore; LONG Max = 12, PreviousCount;
// Створення семафора з однаковими значеннями
// лічильників, що дорівнюють 12
Semaphore = CreateSemaphore (NULL, cMax, cMax,
                             NULL);
// Безіменний семафор
if (Semaphore == NULL)
{// Перевірка помилок}
```

Перш, ніж будь-який потік створює нове вікно, він викликає функцію WaitForSingleObject, щоб визначити, чи дозволяє поточний лічильник семафора створення додаткових вікон. Параметр блокування часу функції очікування встановлено на 0.

```
DWORD WaitResult;
WaitResult = WaitForSingleObject (Semaphore, 0);
switch (WaitResult) {
    // Семафор вільний
    case WAIT_OBJECT_0:
        // Можна створити наступне вікно
        break;
    // Семафор зайнятий, час минув
    case WAIT_TIMEOUT:
        // Не створювати наступне вікно
        break;
}
```

Коли потік закриває вікно, він викликає функцію ReleaseSemaphore, щоб збільшити лічильник семафора.

```
if (! ReleaseSemaphore (Semaphore, 1, NULL))
{ // Виникла помилка
}
```

Об'єкти ядра **м'ютекси** гарантують потокам взаємовиключний доступ до єдиного ресурсу. Звідси і пішла назва об'єкта. Потік, намагаючись отримати доступ до критичних даних, виконує відповідний системний виклик. Якщо м'ютекс знаходиться в сигнальному стані, то потік тут же стає його власником, встановлюючи його в несигнальний стан, і входить у критичну секцію. Після того, як потік виконав роботи з критичними даними, він віддає м'ютекс, встановлюючи його в сигнальний стан. У цей момент м'ютекс вільний і не належить жодному потоку. Якщо який-небудь потік очікує його звільнення, то він стає таким власником цього мьютекса, одночасно м'ютекс переходить в несигнальний стан. Таким чином, ці об'єкти повинні містити лічильник кількості користувачів, лічильник рекурсії і ідентифікатор потоку. Для м'ютекса визначені наступні правила:

1. Якщо ідентифікатор потоку дорівнює 0, то м'ютекс знаходиться в сигнальному стані і не захоплений жодним потоком.
2. Якщо ідентифікатор потоку не дорівнює 0, то м'ютекс захоплений одним потоком і знаходиться в несигнальному стані.
3. М'ютекси можуть порушувати правила, що діють в ОС.

Процес може створити м'ютекс викликом функції:

```
HANDLE CreateMutex (
   PSECURITY_ATTRIBUTES Attributes, // атрибути
                                    // захисту
   BOOL InitialOwner, // початковий стан мьютекса
   PCTSTR Name);      // символьне ім'я об'єкта
```

Отримати опис існуючого мьютекса можна за допомогою функції:

```
HANDLE OpenMutex (
   DWORD Access, // режим доступу
   BOOL Inherit, // спадкування
   PCTSTR Name); // символьне ім'я об'єкта
```

Параметр `InitialOwner` визначає початковий стан м'ютекса. Якщо передається `FALSE`, то м'ютекс знаходиться в сигнальному стані і не належить жодному потоку, причому ідентифікатор потоку і лічильник рекурсії дорівнюють нулю. Якщо передається `TRUE`, то лічильник рекурсії стає рівним 1, а ідентифікатор потоку в м'ютексі стає рівним ідентифікатору потоку, що його викликав, і тепер м'ютекс знаходиться в несигнальному стані. Потік отримує доступ до ресурсу, викликаючи якусь функцію, що його очікує, з передачею їй опису м'ютекса. Якщо вона визначає, що м'ютекс зайнятий, то викликаючий потік переходить в стан очікування. Це запам'ятовується, і коли ідентифікатор потоку м'ютекса обнуляється, в нього записується ідентифікатор потоку, що чекає. Лічильнику рекурсії присвоюється при цьому значення 1. Після цього очікуючий потік може бути активований. Треба відзначити, що система обов'язково перевірить, чи не збігається ідентифікатор потоку у м'ютекса з ідентифікатором потоку, що

очікує м'ютекса. У разі збігу система виділить потоку процесорний час, хоча м'ютекс ще зайнятий. Лічильник рекурсії збільшується на 1, коли потік захоплює м'ютекс, і значення цього лічильника може бути більшим за 1 тільки в тому випадку, якщо потік захопив один і той же об'єкт кілька разів. Коли очікування м'ютекса завершується, потік отримує монопольний доступ до ресурсу. Всі інші потоки переходять у стан очікування. Після закінчення роботи з ресурсом, потік зобов'язаний звільнити м'ютекс викликом функції:

```
BOOL ReleaseMutex (HANDLE Mutex);
```

Ця функція зменшує лічильник рекурсії на 1, і якщо м'ютекс передавався у володіння потоку кілька разів, він повинен викликати `ReleaseMutex` таке ж число разів, щоб, врешті-решт, обнулити лічильник рекурсії. Коли це станеться, ідентифікатор потоку також стане рівним 0, і м'ютекс стане вільним. Система перевірить, чи немає очікуючих потоків, і вибере один з них, щоб передати тому м'ютекс.

Як видно, цей об'єкт має таку чудову властивість, якої немає в інших синхронізуючих об'єктів: м'ютекс запам'ятовує потік, якому він належить. Якщо сторонній потік спробує викликати `ReleaseMutex`, то ця функція просто поверне `FALSE`. Якщо ж якийсь потік завершиться, не встигнувши звільнити м'ютекс, то вважається, що відбулася відмова від мьютекса, і система сама переведе його в сигнальний стан.

```
HANDLE Thread1, Thread2;
HANDLE Mutex1;
int WinMain (...)
{
Mutex1 = CreateMutex (NULL, FALSE, "Mutex1");
// Створюється мьютекс
// Створюються два потоки, причому пропущені всі
                      параметри, крім функції потоку
HANDLE Th1 = CreateThread (..., Function1, ...);
HANDLE Th2 = CreateThread (..., Function2, ...);
// Далі можна виконувати будь-які дії
...
CloseHandle (Mutex1);
```

```
}
DWORD WINAPI Function1 (PVOID Parametr)
{
WaitForSingleObject (Mutex1, INFINITE);
...
ReleaseMutex (Mutex1);
return 0;
}
DWORD WINAPI Function2 (PVOID Parametr)
{
WaitForSingleObject (Mutex1, INFINITE);
...
ReleaseMutex (Mutex1);
return 0;
}
```

Варіанти завдань до лабораторної роботи № 4

1. У пансіонаті відпочивають 5 філософів. В їдальні розташований круглий стіл, на якому знаходиться одна велика тарілка зі спагеті, яка поповнюється нескінченно, також там розставлені 5 тарілок, в які накладається спагеті, і 5 виделок. Для того, щоб пообідати, філософ входить до їдальні і сідає на стілець. Філософ зможе їсти тільки в тому випадку, коли вільні обидві виделки – праворуч і ліворуч від його тарілки. При виконанні цієї умови філософ піднімає їх одночасно і їсть протягом певного часу. В іншому випадку, філософу доводиться чекати звільнення обох виделок. Пообідавши, філософ кладе обидві виделки на стіл одночасно і йде. Розробити програму моделювання описаного процесу, якщо він відбувається нескінченно. Скористатися об'єктами синхронізації типу «таймер очікування».

2. Розробити програму моделювання взаємодії двох потоків, один з яких пише дані в буферний пул, а інший зчитує їх з пулу. Буферний пул складається з N буферів, кожен містить один запис. В загальному випадку потік-письменник і потік-читач мають різні швидкості і звертаються до пулу зі змінною інтенсивністю. Для правильної роботи потік-письменник призупиняється, коли всі буфери зайняті, і переходить в активний стан при наявності хоча б одного вільного буфера. Потік-читач призупиняється, коли всі буфери

порожні, і активується, коли з'являється, принаймні, один запис. Описаний процес відбувається нескінченно. Скористатися об'єктами синхронізації типу «семафор».

3. В перукарні розташоване єдине крісло, на якому спить перукар, і N стільців для клієнтів. Коли клієнт приходить в перукарню, він будить перукаря, сідає в крісло. Стрижка проводиться протягом заданого часу. Якщо ж крісло зайняте іншим клієнтом, то новоприбулий клієнт займає будь-який вільний стілець і чекає своєї черги. Якщо всі стільці зайняті, то клієнт йде. Коли обслужено всіх клієнтів, перукар сідає в крісло і знову засинає. Розробити програму моделювання описаного процесу, якщо він відбувається нескінченно. Скористатися об'єктами синхронізації типу «подія».

4. За столом сидять господар і троє гостей. Завдання гостя – приготувати собі коктейль, для якого потрібні три складові: сироп, молоко і морозиво. У кожного з гостей є по одному інградієнту, а господар має необмежений запас усіх трьох продуктів. Робочий цикл починається з того, що господар випадковим чином ставить на стіл два з трьох компонентів. Після цього той гість, у якого є відсутній інградієнт, робить собі коктейль і сповіщає про це господаря. Той у відповідь ставить два інших компоненти на стіл, і цикл повторюється. Розробити програму моделювання описаного процесу, якщо він відбувається нескінченно. Скористатися об'єктами синхронізації типу «м'ютекс».

5. Старий міст, рух яким можливий лише в один бік, витримує не більше трьох автомобілів. Автомобілі можуть з'являтися з обох боків мосту. Розробити програму моделювання процесу керування рухом по мосту. Скористатися об'єктами синхронізації типу «критична секція».

6. Розробити програму моделювання процесу, що описаний у варіанті 5. Скористатися об'єктами синхронізації типу «таймер очікування».

7. Розробити програму моделювання процесу, що описаний у варіанті 4. Скористатися об'єктами синхронізації типу «подія».

8. Розробити програму моделювання процесу, що описаний у варіанті 3. Скористатися об'єктами синхронізації типу «м'ютекс».

9. Розробити програму моделювання процесу, що описаний у варіанті 2. Скористатися об'єктами синхронізації типу «критична секція».

10. Розробити програму моделювання процесу, що описаний у варіанті 1. Скористатися об'єктами синхронізації типу «семафор».

Лабораторна робота № 5.

ВИКОРИСТАННЯ МЕХАНІЗМУ ВІРТУАЛЬНОЇ ПАМ'ЯТІ В ОС WINDOWS

Мета: Вивчення віртуальної пам'яті в операційній системі Windows.

Хід роботи:

1. Ознайомитися зі специфікаціями функцій `WinAPI` для роботи з розділами віртуального адресного простору процесів і «купами».
2. Ознайомитися із завданням, яке передбачає розробку програми з використанням «купи» або механізму захоплення і звільнення розділів віртуальної пам'яті, що запускає додаткові процеси.
3. Використовуючи вивчені механізми, розробити і налагодити програмне застосування, що реалізує отримане завдання.
4. Пояснити роботу вивчених механізмів за вихідним програмним кодом розробленого додатку.

Віртуальний адресний простір (ВАП) кожного процесу в ОС *Windows NT/2000/XP* організовано наступним чином. В момент його створення він майже повністю порожній. Для використання якоїсь його частини, необобхідно виділити в ньому певні області за допомогою функції `VirtualAlloc` – ця операція називається **резервуванням**. При цьому ОС повинна вирівнювати початок області відповідно до **гранулярності виділення пам'яті**, яка становить на поточний момент 64 Кб. Система повинна також враховувати, що розмір цієї області повинен бути **кратний розміру сторінки**. Для процесорів *Pentium* розмір сторінки складає 4 КБ. Іншими словами, якщо процес спробує зарезервувати 10 Кб, то буде виділено область розміром 12 Кб. Коли виділена область стає не потрібною, її необхідно звільнити викликом функції `VirtualFree`.

Щоб використовувати виділену область ВАП, для неї необхідно також виділити фізичну пам'ять, спроектувавши її на область. Ця операція називається **передачею фізичної пам'яті** і здійснюється за допомогою функції `VirtualAlloc`. Для фізичної пам'яті також визначають її повернення, що виконується за допомогою функції `VitualFree`.

Для окремих сторінок фізичної пам'яті можна встановлювати

атрибути захисту:

- PAGE_NOACCESS – будь-яка операція спричиняє спотворення доступу;
- PAGE_READONLY – спроби запису або виконання можуть викликати порушення доступу;
- PAGE_READWRITE – спроби виконати вміст сторінки викликають порушення доступу;
- PAGE_EXECUTE – читання і запис можуть викликати порушення доступу;
- PAGE_EXECUTE_READ – порушення доступу при спробі запису;
- PAGE_EXECUTE_READWRITE – можливі будь-які операції;
- PAGE_WRITECOPY – при виконанні цієї сторінки порушення доступу, під час запису процесу дається особиста копія сторінки;
- PAGE_EXECUTE_WRITECOPY – будь-які операції, під час запису процесу дається особиста копія сторінки;
- PAGE_NOCACHE – вимикає кешування сторінки (прапорець);
- PAGE_WRITECOMBINE – об'єднання декількох операцій запису (прапорець);
- PAGE_GUARD – використовується для отримання інформації про записи на будь-яку сторінку (прапорець).

Дізнатися розміри сторінки, гранулярність виділення пам'яті і інші параметри ОС можна за допомогою функції Win32 API:

```
VOID GetSystemInfo (LPSYSTEM_INFO SysInfo);
```

В цю функцію як параметр передається адреса структури даних, що описана в такий спосіб:

```
typedef struct {
   union {
      DWORD Oem; // не використовується
      struct {
         WORD ProcArchitecture;
                        // Тип архітектури процесора
         WORD Reserved; // не використовується
      };
   };
```

```
    DWORD PageSize; // Розмір сторінки в байтах
    LPVOID MinApplnAddress;
                    // Мінімальна адреса доступного ВАП
    LPVOID MaxApplnAddress;
                   // Максимальна адреса доступного ВАП
    DWORD_PTR ActiveProcessors;
                     // Процесори, що виконують потоки
    DWORD NumberOfProc;
                   // Кількість встановлених процесорів
    DWORD ProcType; // Тип процесора для Windows 98/Me
    DWORD Granularity; // гранулярність
    WORD ProcLevel, ProcRevizion;
       // Додаткові параметри для ОС Windows 2000
} SYSTEM_INFO, * LPSYSTEM_INFO;
```

Якщо є необхідність дізнатися параметри, які мають відношення до пам'яті (а їх всього чотири), досить виконати лише два оператори:

```
SYSTEM_INFO SysInfo;
GetSystemInfo (& SysInfo);
```

Після цього можна спокійно переглядати вміст полів структури `SysInfo`, де і будуть перебувати шукані системні параметри.

Наступна функція дозволяє відстежувати стан пам'яті на поточний момент часу:

```
VOID GlobalMemoryStatus (LPMEMORY_STATUS MemStat);
```

Як видно, їй необхідно передати адресу деякої структури, яка описана наступним чином:

```
typedef struct {
    DWORD Length; // Розмір структури в байтах
    DWORD MemLoad; // Зайнятість підсистеми
                     // управління пам'яттю (0 - 100)
    SIZE_T TotalPhysMem; // Обсяг фізичної пам'яті
    SIZE_T AvailablePhysMem;
                    // Обсяг вільної фізичної пам'яті
    SIZE_T TotalPageFile;
                 // Максимальний розмір файлу підкачки
    SIZE_T AvailablePageFile;
              // Розмір вільного місця в файлі підкачки
    SIZE_T TotalVirtual;
```

```
                    // Максимальний розмір ВАП процесу
    SIZE_T AvailableVirtual;
                     // Розмір доступного ВАП процесу
} MEMORY_STATUS, * LPMEMORY_STATUS;
```

Якщо потрібно отримати якісь параметри пам'яті, буде потрібно ще кілька операторів, але попередньо потрібно встановити довжину всієї структури:

```
MEMORY_STATUS MemStat;
MemStat.Length = {sizeof (MemStat)};
GlobalMemoryStatus (& SysInfo);
```

Надалі можна сміливо користуватися значеннями інших полів структури `MemStat`.

В ОС *Windows NT/2000/XP* є функція, яка дозволяє отримувати інформацію про конкретну ділянку пам'яті в межах ВАП процесу:

```
DWORD VirtualQuery (
    LPCVOID Address, // адреса ділянки пам'яті
    PMEMORY_BASIC_INFORMATION MemBase,
        // Адреса структури з інформацією про пам'ять
    DWORD Length); // Розмір цієї структури
```

Ця функція вимагає адресу структури типу `MEMORY_BASIC_INFORMATION`, що описується так:

```
typedef struct {
    PVOID Base; // Адреса Address, округлена до
                    // адреси, кратної розміру сторінки
    PVOID AllocBase; // Базова адреса області, в яку
                            // входить адреса Address
    DWORD AllocProtection;
                        // Атрибут захисту для області
    SIZE_T RegionSize;
        // Розмір сторінок, що мають однакові атрибути
    DWORD State; // Стан всіх суміжних сторінок
    DWORD Protection;
            // Атрибути захисту всіх суміжних сторінок
    DWORD Type;
        // Тип фізичної пам'яті всіх суміжних сторінок
} MEMORY_BASIC_INFORMATION, *
PMEMORY_BASIC_INFORMATION;
```

У тому випадку, коли знадобиться вивести інформацію про

області ВАП, що пов'язані з поточною діяльністю, це можна зробити в такий спосіб, проте трохи «грубо»:

```
// Спочатку необхідно отримати розмір сторінки для даної системи
SYSTEM_INFO SysInfo;
GetSystemInfo (& SysInfo);

PVOID BaseAddr = NULL;
MEMORY_BASIC_INFORMATION MemBase;
for (;;) {
   if (VirtalQuery (BaseAddr, & MemBase, sizeof
        (MemBase))! = sizeof (MemBase)) break;
   printf ( "% Lp \ t \", MemBase.AllocBase);
                                    // Базова адреса
   printf ( "% d \ t", MemBase.RegionSize /
        SysInfo.PageSize); // Сторінок на область
   switch (MemBase.State) {// Стан області
      case MEM_FREE: printf ( "вільно\t"); break;
      case MEM_RESERVE: printf ( "зарезервовано,
           пам'ять не передано\t"); break;
      case MEM_COMMIT: printf ( "зарезервовано,
           пам'ять передано\t"); break;
      }
   switch (MemBase.Protection) {// Атрибути
                                  // захисту області
      case PAGE_NOACCESS: printf ( "----\t"); break;
      case PAGE_READONLY: printf ( "R---\t"); break;
      case PAGE_READWRITE: printf ( "RW--\t"); break;
      case PAGE_EXECUTE: printf ( "--E-\t"); break;
      case PAGE_EXECUTE_READ: printf ( "R-E-
                                \t"); break;
      case PAGE_EXECUTE_ READWRITE: printf (
                                "RWE-\t"); break;
      case PAGE_EXECUTE_ WRITECOPY: printf (
                                "RWEC\t"); break;
      }
   switch (MemBase.Type) {// Тип області
      case MEM_FREE: printf ( "Вільна"); break;
      case MEM_PRIVATE: printf ( "Закрита"); break;
      case MEM_IMAGE: printf ( "Образ файлу"); break;
```

```
      case MEM_MAPPED: printf ( "Відображається
                                файл"); break;
        default: printf ( "Невідомо");
        }
    printf ( "\ n");
    BaseAddr = MemBase.BaseAddress +
                            MemBase.RegionSize;
    }
```

Віртуальна пам'ять зручна для роботи з великими масивами даних. Для малих за розміром об'єктів більше підходять так звані «купи». Якщо є потреба в обміні даними між процесами, то ОС Windows пропонує ще один механізм – відображення файлів на пам'ять. Нижче описані функції для управління віртуальною пам'яттю, що дозволяють резервувати область, віддавати їй фізичну пам'ять і встановлювати параметри захисту.

Для резервування призначена функція:

```
PVOID VirtualAlloc (
    PVOID Address,
        // Адреса, де система має резервувати пам'ять
    SIZE_T Size, // Розмір області, яку треба зарезервувати
    DWORD AllocType, // Тип резервування
                // (резервувати або передати фіз.память)
    DWORD Protect); // Атрибут захисту (PAGE_ *)
```

Функція повертає NULL, якщо не вдалося виділити пам'ять для області. Для резервування досить при виклику вказати тип MEM_RESERVE. Якщо тепер треба передати їй фізичну пам'ять, потрібно ще раз викликати VirtualAlloc з прапорцем доступу MEM_COMMIT. Можна виконати обидві операції одночасно:

```
PVOID Region = VirtualAlloc (NULL, 25 * 1024,
   MEM_RESERVE | MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
```

Тут є запит на виділення області з розміром 25 Кб і передачу їй фізичної пам'яті. Оскільки перший параметр функції дорівнює NULL, ОС спробує знайти підходяще місце, переглядаючи всі ВАП. Область і передана їй пам'ять отримають однаковий атрибут захисту PAGE_EXECUTE_READWRITE.

Для звільнення зарезервованої області чи повернення фізичної пам'яті можна користуватися функцією:

```
BOOL VirtualFree (
    PVOID Address,
          // адреса, де система зарезервувала пам'ять
    SIZE_T Size,
             // розмір області, що була зарезервована
    DWORD FreeType); // тип звільнення
```

Для звільнення області потрібно викликати `VirtualFree` з її адресою, в `Size` вказати 0, оскільки система знає розмір області, а в `FreeType` – `MEM_RELEASE`. Якщо ж виникла необхідність просто повернути частину фізичної пам'яті, то `Address` повинна адресувати першу сторінку, що повертається, `Size` – кількість звільнених байтів, а `FreeType` – ідентифікатор `MEM_COMMIT`.

```
PVOID Region;
VirtualFree (Region, 0, MEM_RELEASE);
```

Атрибути захисту сторінки пам'яті можна змінити викликом функції:

```
PVOID VirtualProtect (
    PVOID Address,
          // адреса, де система зарезервувала пам'ять
    SIZE_T Size,
          // число байтів, для яких змінюється захист
    DWORD NewProtection, // нові атрибути захисту
    PDWORD OldProtecttion); // старі атрибути захисту
```

Для зміни атрибутів захисту у тих сторінок, які належать різним областям, функцію `VirtualProtect` використовувати не можна:

```
PVOID Region = VirtualAlloc (NULL, 25 * 1024,
   MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
DWORD OldProt;
VirtualProtect (Region, 3 * 1024, PAGE_READONLY,
                  & OldProt);
```

Ще один механізм управління пам'яттю – «купа» –також область зарезервованого адресного простору. Цій області велика ділянка фізичної пам'яті не передається. Спеціалізований «менеджер купи» передає їй фізичну пам'ять або повертає сторінки в залежності від того, що робить програма із своїми даними.

В ОС *Windows NT/2000/XP* при ініціалізації процесу в його ВАП створюється стандартна купа, розмір якої 1 Мб. Описати цю

купу можна за допомогою виклику функції:

```
HANDLE GetProcessHeap ();
```

Можна створити і додаткові купи:

```
HANDLE HeapCreate (
   DWORD Options,
                // спосіб виконання операцій над купою
   SIZE_T StartSize,
                 // початкова кількість байтів в купі
   SIZE_T MaxSize);
                // Максимальна кількість байтів в купі
```

Якщо в `Options` вказано 0, то до купи можуть одночасно звертатися кілька потоків. Атрибут `HEAP_NO_SERIALIZE` дозволяє потоку здійснювати доступ до купи монопольно, але користуватися таким способом не рекомендується. Інший прапорець `HEAP_GENERATE_EXCEPTIONS` дає системі можливість повідомляти програми про помилку звертання до купи. Якщо в третьому параметрі `MaxSize` вказати значення більше 0, то буде створена нерозширювана купа саме такого розміру, в іншому випадку – система резервує область і може розширювати її до максимального розміру. Ця функція в разі успіху повертає описувач новоствореної купи.

Для виділення блоку пам'яті з купи необхідно викликати функцію:

```
PVOID HeapAlloc (
   HANDLE Heap, // описувач купи
   DWORD Flags, // прапорці виділення пам'яті
   SIZE_T Bytes); // кількість виділених байтів
```

Якщо як прапорець вказано `HEAP_ZERO_MEMORY`, то функція поверне блок пам'яті, заповнений нулями.

Іноді потрібно змінити розмір виділеного блоку пам'яті: зменшити або збільшити. Для цього викликається функція:

```
PVOID HeapReAlloc (
   HANDLE Heap, // описувач купи
   DWORD Flags, // прапорці зміни пам'яті
   PVOID Memory, // поточна адреса блоку
   SIZE_T Bytes); // новий розмір в байтах
```

Можливі чотири значення прапорця: `HEAP_NO_SERIALIZE`, `HEAP_GENERATE_EXCEPTIONS`, `HEAP_ZERO_MEMORY` і

HEAP_REALLOC_IN_PLACE_ONLY. При використанні HEAP_ZERO_MEMORY нулями заповнюються тільки додаткові байти. Прапорець HEAP_REALLOC_IN_PLACE_ONLY свідчить про те, що блок переміщати всередині купи не можна. Повертає ця функція адресу нового блоку або NULL, якщо не вдалося змінити розмір.

Після того, як було виділено блок пам'яті, можна визначити його розмір:

```
SIZE_T HeapSize (
   HANDLE Heap, // описувач купи
   DWORD Flags,
   // прапорці зміни пам'яті (0 або HEAP_NO_SERIALIZE)
   PVOID Memory); // поточна адреса блоку
```

Коли блок перестав бути потрібним, його звільняють функцією:

```
BOOL HeapFree (
   HANDLE Heap, // описувач купи
   DWORD Flags,
   // прапорці зміни пам'яті (0 або HEAP_NO_SERIALIZE)
   PVOID Memory); // поточна адреса блоку
```

У разі успіху ця функція повертає TRUE. Аналогічно працює функція HeapDestroy, яка звільняє всі блоки пам'яті всередині купи і повертає системі область, зайняту купою:

```
BOOL HeapDestroy (HANDLE Heap); // Описувач купи
```

Невеликий фрагмент коду демонструє використання деяких з цих функцій.

```
// Отримання опису стандартної купи активного процесу
HANDLE SysHeap = GetProcessHeap ();

UINT MAX_ALLOCATIONS = 15;
            // Максимальна кількість виділень пам'яті
UINT NumOfAllocations = 0;
               // Поточна кількість виділень пам'яті

for (;;) {
   // Виділення з купи 2 КБ пам'яті
   if (HeapAlloc (SysHeap, 0, 2 * 1024) == NULL)
               break;
   else ++ NumOfAllocations;
   // Умова переривання циклу
```

```
if (NumOfAllocation == MAX_ALLOCATIONS) break;
    }

// Виведення відповідних повідомлень в залежності від ситуації
if (NumOfAllocations == 0)
     printf ( "Пам'ять з купи не виділялася.");
else printf ( "Пам'ять з купи виділялася% d разів.",
              NumOfAllocations);
```

В ОС *Windows NT/2000/XP* є пара функцій Win32 API, які дозволяють блокувати (або зафіксувати) і розблокувати сторінку в оперативній пам'яті. Функція `VirtualLock` дозволяє запобігти запису пам'яті на диск:

```
BOOL VirtualLock (
   LPVOID Address, // адреса початку пам'яті
   SIZE_T Size); // кількість байтів
```

Якщо фіксація більше не потрібна, то її можна відмінити функцією:

```
BOOL VirtualUnlock (
   LPVOID Address, // адреса початку пам'яті
   SIZE_T Size); // кількість байтів
```

Обидві функції повертають ненульове значення в разі успіху. Наступний фрагмент демонструє їх використання:

```
int MEMSIZE = 4096;
PVOID Mem = NULL;

int num;

Mem = VirtualAlloc (NULL, 4 * 1024, MEM_RESERVE,
                    PAGE_EXECUTE_READWRITE);
if (Mem! = NULL) {
   if (VirtualLock (Mem, MEMSIZE)) printf (
                    "Прив'язка \ n");
   else printf ( "Помилка прив'язки");
   scanf ( "% d", & num);

   if (VirtualUnlock (Mem, MEMSIZE))
        printf ( "Прив'язка знята \ n");
   else printf ( "Помилка зняття прив'язки \ n");

   if (VirtualFree (Mem, 0, MEM_RELEASE))
```

```
        printf ( "Пам'ять звільнена \ n");
    else printf ( "Пам'ять не звільнена \ n");
    }
else printf ( "Пам'ять не виділена \ n");
}
```

Варіанти завдань до лабораторної роботи № 5

1. Розробити програму, яка демонструє управління структурами даних типу «стек», елементи якого займають 10 Кб. Операції, що виконуються над стеком:

- перевірити, стек порожній чи не порожній;
- заштовхнути елемент;
- виштовхнути елемент;
- переглянути вершину стека;
- поміняти значення двох верхніх елементів стека.

2. Розробити програму, яка демонструє управління структурами даних типу «стек», елементи якого займають 15 Кб. Операції, що виконуються над стеком:

- перевірити, стек порожній чи не порожній;
- заштовхнути елемент;
- виштовхнути елемент;
- переглянути вершину стека;
- продублювати вершину стека.

3. Розробити програму, яка демонструє управління структурами даних типу «стек», елементи якого займають 12 Кб. Операції, що виконуються над стеком:

- перевірити, стек порожній чи не порожній;
- заштовхнути елемент;
- виштовхнути елемент;
- переглянути вершину стека;
- поміняти значення другого і третього зверху елементів стека.

4. Розробити програму, яка демонструє управління структурами даних типу «черга», елементи якої займають 10 Кб. Операції, що виконуються над чергою:

- перевірити чергу – порожня чи не порожня;
- додати елемент в хвіст черги;
- видалити елемент з голови черги;
- переглянути голову черги;
- поміняти місцями значення з голови і хвоста черги.

5. Розробити програму, яка демонструє управління структурами даних типу «черга», елементи якої займають 15 Кб. Операції, що виконуються над чергою:

- перевірити чергу – порожня чи не порожня;
- додати елемент в хвіст черги;
- видалити елемент з голови черги;
- переглянути голову черги;
- продублювати хвіст черги.

6. Розробити програму, яка демонструє управління структурами даних типу «ДЕК» (черга з двома кінцями), елементи якого займають 10 Кб. Операції, що виконуються над Деком:

- перевірити Дек – порожній чи не порожній;
- додати елемент в лівий кінець Дека;
- додати елемент в правий кінець Дека;
- видалити елемент ліворуч;
- видалити елемент праворуч;
- переглянути елемент ліворуч;
- переглянути елемент праворуч.

7. Розробити програму, яка демонструє управління структурами даних типу «обмежений ліворуч ДЕК» (черга з двома кінцями), елементи якого займають 15 Кб. Операції, що виконуються над Деком:

- перевірити Дек – порожній чи не порожній;
- додати елемент в лівий кінець Дека;
- додати елемент в правий кінець Дека;
- видалити елемент праворуч;
- переглянути елемент праворуч.

8. Розробити програму, яка демонструє управління структурами даних типу «обмежений праворуч ДЕК» (черга з двома кінцями), елементи якого займають 12 Кб. Операції, що

виконуються над Деком:

- перевірити Дек – порожній чи не порожній;
- додати елемент в лівий кінець Дека;
- додати елемент в правий кінець Дека;
- видалити елемент ліворуч;
- переглянути елемент ліворуч.

9. Розробити програму, яка демонструє управління структурами даних типу «лінійний односпрямований список» (L1-list), елементи якого займають 10 Кб. Операції, що виконуються над списком (при цьому визначається покажчик списку та елемент списку за покажчиком):

- перевірити, список порожній чи не порожній;
- встановити покажчик на початок списку;
- додати елемент за покажчиком;
- видалити елемент за покажчиком;
- переглянути елемент за покажчиком;
- перемістити покажчик вправо.

10. Розробити програму, яка демонструє управління структурами даних типу «динамічний вектор» (одновимірний масив), елементи якого займають 12 Кб. Операції, що виконуються над вектором (при цьому визначаються початок і кінець вектора, індекс елемента вектора):

- перевірити, вектор порожній чи не порожній;
- прочитати елемент із зазначеним індексом;
- змінити значення елемента з вказаним індексом;
- додати елемент в кінець вектора;
- очистити вектор.

СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Джонс Э. Программирование в сетях Microsoft Windows. Мастер-класс / Э. Джонс, Дж. Оланд. – СПб.: Питер, 2002. – 608 с.

2. Иртегов Д.В. Введение в операционные системы / Д.В. Иртегов. – СПб.: БХВ-Петербург, 2002. – 624 с.

3. Ковалев И.В. Операционные системы и системное программное обеспечение: учеб. пособие / И.В. Ковалев, А.С. Кузнецов. – Красноярск: ИПЦ КГТУ, 2008. – 302 с.

4. Копичко С.М. Системне програмне забезпечення / С.М. Копичко, С.М. Макаров – Енциклопедичне видання: Навч.-метод. посіб. – К.: ТОВ Редакція «Комп'ютер», 2008. – 128 с.

5. Майнази М. Windows XP Professional / М. Майнази. – М.: Лори, 2003. – 744 с.

6. Молчанов А.Ю. Системное программное обеспечение / А.Ю. Молчанов. – СПб.: Питер, 2003. – 400 с.

7. Олифер В.Г. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер. – СПб.: Питер, 2001. – 544 с.

8. Прытков В.А. Типовые механизмы синхронизации процессов : учеб.-метод. пособие по дисц. «Системное программное обеспечение ЭВМ» для студ. спец. «Вычислительные машины, системы и сети» / В.А. Прытков, А.А. Уваров, В.А. Супонев – Минск: БГУИР, 2007.

9. Рихтер Д. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Д. Рихтер. – СПб.: Питер, 2001. – 752 с.

10. Соломон Д. Внутреннее устройство MS Windows 2000. Мастер-класс / Д. Соломон, М. Руссинович. – СПб.: Питер, 2001. – 752 с.

11. Танненбаум Э. Современные операционные системы / Э. Танненбаум. – СПб.: Питер, 2002. – 1040 с.

12. Харт Д. Системное программирование в среде Win32 / Д. Харт. – М.: Вильямс, 2001. – 464 с.