Лабораторна робота №5

Тема. Багатопоточність в операційній сисемі WINDOWS. Створення, керування та синхронізація потоків.

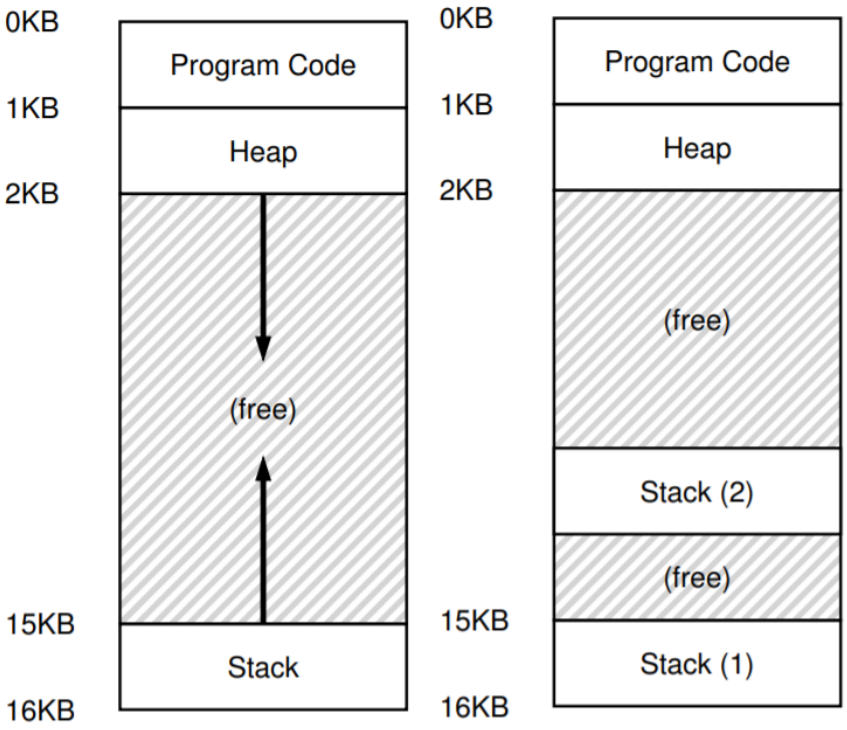
Мета. Ознайомитися з багатопоточністю в ОС Windows. Навчитись реалізовувати розпаралелення алгоритмів за допомогою багатопоточності в ОС Windows з використанням функцій WinAPI. Навчитись використовувати різні механізми синхронізації потоків.

Теоретичні відомості

Розглядаючи поняття процесу, визначають ще одну абстракцію для запущеного процесу: потік. У класичному уявленні існує єдина точка виконання в рамках програми (тобто єдиний потік контролю, на якому збираються та виконуються інструкції), багатопотокова програма має більш ніж одну точку виконання (тобто кілька потоків контролю, кожен з яких який отримується та виконується).

Кожен потік дуже схожий на окремий процес, за винятком однієї відмінності: вони мають спільний адресний простір і, отже, мають доступ до одних і тих же даних. Таким чином, стан одного потоку дуже подібний до стану процесу. Він має лічильник програм (PC), який відстежує, звідки програма отримує інструкції. Кожен потік має свій власний приватний набір реєстрів, який він використовує для обчислень; таким чином, якщо на одному процесорі працюють два потоки, при переході від запуску одного (T1) до запуску іншого (T2) має відбутися перемикання контексту. Контекстний перемикач між потоками дуже подібний до перемикання контекстів між процесами, оскільки перед запуском T2 необхідно зберегти регістр стану T1 і відновити стан реєстру T2. За допомогою процесів ми зберегли стан до блоку управління процесами (PCB); тепер нам знадобиться один або кілька блоків управління потоками (TCB) для збереження стану кожного потоку процесу. Однак є одна істотна відмінність у перемиканні контексту, який ми виконуємо між потоками порівняно з процесами: адресний простір залишається незмінним (тобто немає необхідності змінювати, яку таблицю сторінок ми використовуємо).

Ще одна істотна відмінність між потоками та процесами стосується стека. У простій моделі адресного простору класичного процесу (однопотокового) є єдиний стек, який зазвичай знаходиться внизу адресного простору. Однак у багатопотоковому процесі кожен потік працює окремо і, звичайно, може залучати різні підпрограми для виконання будь -якої роботи. Замість одного стека в адресному просторі буде по одному на кожен потік.



На цьому малюнку можна побачити два стеки, розповсюджені по адресному простору процесу. Таким чином, будь-які змінні, параметри, повернені значення, що виділяються стеком, та інші речі, які розміщуємо у стеку, будуть розміщені у тому, що іноді називають локальним сховищем потоків, тобто стеком відповідного потоку. Раніше стек і купа могли зростати незалежно, і проблеми виникали лише тоді, коли в адресному просторі вичерпалося місце. Тут немає такої приємної ситуації, оскільки стеки, як правило, не повинні бути дуже великими (виняток становлять програми, які дуже часто використовують рекурсію).

Потоки можуть виконувати один і той же код і маніпулювати одними і тими ж даними, а також спільно використовувати дескриптори об'єктів ядра, оскільки таблиця дескрипторів створюється не в окремих потоках, а в процесах.

Кожен потік починає виконання з якоїсь вхідної функції. У первинному потоці такою є main, wmain, WinMain або wWinMain. Якщо Ви хочете створити вторинний потік, в ньому теж має бути вхідна функція, яка виглядає приблизно так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam)
{ DWORD dwResult = 0;
...
 return(dwResult); }
```

Функція потоку може виконувати будь-які завдання. Після того як вона закінчить свою роботу, поверне управління. У цей момент потік зупиниться, пам'ять, відведена під його стек, буде звільнена, а лічильник користувачів його

об'єкта ядра «потік» зменшиться на 1. Коли лічильник обнулится, цей об'єкт ядра буде зруйнований.

Функцію потоку можна назвати як завгодно. Однак, якщо в програмі декілька функцій потоків, то потрібно присвоїти їм різні імена, інакше компілятор вирішить, що це кілька реалізацій єдиної функції.

Функцій потоків передається єдиний параметр, сенс якого визначається розробником, а не операційною системою. Тому тут немає проблем з ANSI / Unicode.

Функція потоку повинна повертати значення, яке буде використовуватися як код завершення потоку. Тут повна аналогія з бібліотекою C / C ++: код завершення первинного потоку стає кодом завершення процесу.

Функції потоків повинні у міру можливості обходитися своїми параметрами і локальними змінними. Так як до статичної або глобальної змінної можуть одночасно звернутися декілька потоків, то є ризик пошкодити її вміст.

Якщо треба створити додаткові потоки, потрібно викликати з первинного потоку функцію CreateThread:

```
HANDLE CreateThread(
     PSECURITY_ATTRIBUTES psa;
     DWORD cbStack;
     PTHREAD_START_ROUTINE pfnStartAddr;
     PVOID pvParam;
     DWORD fdwCreate;
     PDWORD pdwThreadID);
```

При кожному виклику цієї функції система створює об'єкт ядра «потік». Це не сам потік, а компактна структура даних, яка використовується операційною системою для управління потоком і зберігає статистичну інформацію про потік. Система виділяє пам'ять під стек потоку з адресного простору процесу. Новий потік виконується в контексті того ж процесу, що і батьківський потік. Тому він отримує доступ до всіх дескрипторів об'єктів ядра, всієї пам'яті і стека всіх потоків в процесі. За рахунок цього потоки в рамках одного процесу можуть легко взаємодіяти один з одним.

Параметри функції **CreateThread:**

- **psa** вказівник на структуру SECURITY_ATTRIBUTES. Щоб об'єкту ядра «потік» присвоїти атрибути захисту за замовчуванням вкажіть NULL. А щоб дочірні процеси змогли наслідувати дескриптор цього об'єкта, визначте структуру SECURITY_ATTRIBUTES і ініціалізуйте її елемент bInheritHandle значенням TRUE.
- **cbStack** Цей параметр визначає, яку частину адресного простору потік зможе використовувати під свій стек. Кожному потоку виділяється окремий стек. Функція CreateProcess, запускаючи додаток, викликає

CreateThread, і та ініціалізує головний потік процесу. При цьому CreateProcess заносить в параметр cbStack значення, що зберігається в самому виконуваному файлі. Керувати цим значенням дозволяє ключ / STACK компонувальника:

/ STACK: [reserve] [, commit]

Аргумент reserve визначає обсяг адресного простору, який система повинна зарезервувати під стек потоку (за замовчуванням - 1 Мб). Аргумент commit задає обсяг фізичної пам'яті, який спочатку передається області, зарезервованої під стек (за замовчуванням - 1 сторінка). У міру виконання коду в потоці може знадобитися відвести під стек більше однієї сторінки пам'яті. При переповненні стека виникне виняткова ситуація. Перехопивши це виняток, система передасть зарезервувати простору ще одну сторінку (або стільки, скільки вказано в аргументі commit). Такий механізм дозволяє динамічно збільшувати розмір стека лише в разі потреби.

Якщо передати в параметрі cbStack ненульове значення, функція резервує всю зазначену пам'ять. Її обсяг визначається або значенням параметра cbStack, або значенням, заданим в ключі / STACK компонувальника (вибирається більше з них). Але передається стеку лише той обсяг пам'яті, який відповідає значенню в cbStack. Якщо передати в параметрі cbStack нульове значення, CreateThread створює стек для нового потоку, використовуючи інформацію, вбудовану компонувальником в EXE-файл.

Значення аргументу reserve встановлює верхню межу для стека, і це обмеження дозволяє припиняти діяльність функцій з нескінченної рекурсією.

- **`pfnStartAddr`** визначає адресу функції потоку, з якої повинен почати роботу створений потік
- **`pvParam`** ідентичний параметру pvParam функції потоку. CreateThread лише передає цей параметр. Таким чином, даний параметр дозволяє передавати функції потоку якесь ініціалізоване значення. Воно може бути або просто числовим значенням, або вказівником на структуру даних з додатковою інформацією. При створенні кожного потоку у одну і ту ж функцію може передаватися своє значення pvParam.
- **`fdwCreate`** визначає додаткові прапорці, що керують створенням потоку. Він приймає одне з двох значень: 0 (виконання потоку починається негайно) або CREATE_SUSPENDED. В останньому випадку система створює потік, ініціалізує його і призупиняє до наступних вказівок. Прапорець CREATE_SUSPENDED дозволяє програмі змінити будь-які властивості потоку перед тим, як він почне виконувати код.
- **`pdwThreadID`** це адреса змінної типу DWORD, в якій функція повертає ідентифікатор нового потоку.

Приклад 1.

```
#include <Windows.h>
#include <stdio.h>
DWORD WINAPI mythread(__in LPVOID lpParameter)
{
printf("Thread inside %d \n", GetCurrentThreadId());
return 0;
 }
 int main(int argc, char* argv[])
{
HANDLE myhandle;
DWORD mythreadid;
 myhandle = CreateThread(0, 0, mythread, 0, 0, &mythreadid;);
printf("Thread after %d \n", mythreadid);
getchar();
return 0;
}
```

Потік можна завершити чотирма способами:

- функція потоку повертає управління (рекомендований спосіб);
  Це єдиний спосіб, який гарантує коректну очистку всіх ресурсів, що належали потоку. При цьому:
  - об'єкти, створені з використанням даного потоком, знищуються відповідними деструкторами;
  - система коректно звільняє пам'ять, яку займав стек потоку;
  - система встановлює код завершення даного потоку;
  - лічильник користувачів даного об'єкта ядра «потік» зменшується на 1
- потік самознищується викликом функції ExitThread (небажаний спосіб);
  VOID ExitThread(DWORD dwExitCode);
  При цьому звільняються всі ресурси операційної системи, виділені цього потоку, але C / C ++ - ресурси (наприклад, об'єкти, створені з C ++ - класів) не очищаються.
- один з потоків даного або стороннього процесу викликає функцію TerminateThread (небажаний спосіб);
  BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
  На відміну від ExitThread, яка знищує тільки потік, що її викликає, ця функція завершує потік, вказаний в параметрі hThread. У параметрі dwExitCode ставите значення, яке система розглядає як код завершення потоку. Після того як потік буде знищений, лічильник користувачів його об'єкта ядра «потік» зменшиться на 1.
- завершується процес, що містить даний потік (теж небажано).

Коли потік завершився інші потоки можуть викликати функцію GetExitCodeThread, і перевірити, чи завершено потік, ідентифікований дескриптором hThread, і, якщо так, визначити його код завершення

```
BOOL GetExitCodeThread(
 HANDLE hThread,
PDWORD pdwExitCode);
```

Призупинення потоків

```
DWORD ResumeThread(HANDLE hThread);
```

Відновлення роботи потоків

```
DWORD SuspendThread(HANDLE hThread);
```

Потік може повідомити системі не виділяти йому процесорний час на визначений ний період, викликавши:

```
VOID Sleep (DWORD dwMilliseconds);
```

Ця функція призупиняє потік на dwMilliseconds мілісекунд. Викликаючи Sleep, потік добровільно відмовляється від залишку виділеного йому кванта часу.

Функція, яка повідомляє час, витрачений процесором на обробку даного потоку

```
BOOL GetThreadTimes(
HANDLE hThread,
 PFILETIME pftCreationTime,
PFILETIME pftExitTime,
 PFILETIME pftKernelTime,
 PFILETIME pftUserTime);
```

GetThreadTimes повертає чотири тимчасових параметра:

Час створення (creation time) - Абсолютна величина, виражена в інтервалах по 100 нс. Відраховується з півночі 1 січня 1601 року за Гринвічем до моменту створення потоку.

Час завершення (exit time) Абсолютна величина, виражена в інтервалах по 100 нс. Відраховується з півночі 1 січня 1601 року за Гринвічем до моменту завершення потоку. Якщо потік все ще виконується, цей показник має невизначене значення.

Час виконання ядра (kernel time) Відносна величина, виражена в інтервалах по 100 нс. Повідомляє час, витрачений цим потоком на виконання коду операційної системи.

Час виконання User (User time) Відносна величина, виражена в інтервалі лах по 100 нс. Повідомляє час, витрачений потоком на виконання коду програми

Задання пріоритетів

```
BOOL SetPriorityClass( HANDLE hProcess, DWORD fdwPriority);
```

Дізнатися пріоритет

```
DWORD GetPriorityClass(HANDLE hProcess);
```

Існує три основні проблеми організації міжпоточної взаємодії:

1. Обмін даними.
2. Доступ до загальних ресурсів.

3. Узгодження дій.

Розглянемо міжпоточної взаємодія на простому прикладі: спулер. Коли потоку потрібно надрукувати файл, він поміщає його ім'я до спеціального каталогу спулера. Інший потік - демон друку періодично перевіряє наявність файлів, друкує їх і видаляє імена з каталогу. Нехай каталог спулера складається з великої кількості сегментів, пронумерованих 0, 1, 2,:, в кожному з яких може зберігатися ім'я файлу. Також існує спільно використовувана змінна in, яка вказує на наступний вільний сегмент. Припустимо, сегменти з 0 по 6 зайняті - файли чекають своєї черги на друк, отже in = 7. Більш-менш одночасно два потоку А і В вирішили поставити свої файли в чергу на друк. Можливе виникнення наступної ситуації. Потік А зчитує значення in (7) у своїй локальній змінній, після чого закінчується його квант часу і операційна система перемикає процесор на виконання потоку В. В свою чергу, потік В теж зчитує і локально зберігає значення змінної in, теж рівне 7, тобто обидва потоки вважають, що вільний сегмент - 7. Потік В зберігає в 7 сегменті спулера ім'я файлу, і змінює значення змінної in на 8, після чого продовжує займатися своїми справами, не пов'язаними з друком. Нарешті управління переходить до потоку А, і він продовжує з того місця, на якому зупинився. Потік А зберігає ім'я свого файлу в сегменті спулера, з номером, збереженому в локальній змінній, тобто теж в сьомому сегменті. Природно, заміщаючи ім'я файлу, збереженого потоком В. Потім, потік збільшує на одиницю значення своєї внутрішньої змінної (7 + 1 = 8) і записує в змінну in значення 8. Структура каталогу спулера не порушена, тому демон друку нічого не запідозрить, однак файл, відправлений на друк потоком В ніколи надрукований не буде. Ситуація, при якій кілька потоків борються за доступ до загального ресурсу, а результат залежить від того, хто цей доступ отримав першим, називається станом перегонів (race condition). Налагодження програм з такими ситуаціями складна, оскільки більшість прогонів будуть хорошими, але зрідка будуть відбуватися дивні й незрозумілі речі.

КРИТИЧНІ СЕКЦІЇ. Основним способом запобігання проблем, пов'язаних з спільним доступом до загального ресурсу, є ЗАБОРОНА одночасного доступу до нього більш ніж одним потоком. Це означає, що в той момент, коли один потік використовує колективні дані, іншому потоку це робити заборонено. Проблема зі спулер виникла через те, що потік В розпочав роботу з спільно використовуваної змінної in до того, як потік А її закінчив. Дейкстра ввів поняття критичного інтервалу (критичної області, секції) - це фрагмент програми, в якому є звернення до спільно використовуваних даних. Для забезпечення правильної та ефективної спільної роботи декількох паралельних потоків, що оперують загальним ресурсом, необхідне виконання пяти умов: 1. У кожен момент часу тільки один потік може знаходитися в своїй критичній області. 2. Потік, що знаходиться поза критичною областю не може блокувати інші потоки. 3. Будь-який потік, готовий увійти в свою критичну область, має увійти в неї за кінцевий час. 4. У програмі не повинно бути припущень, щодо швидкості та кількості

потоків. 5. Реалізація взаємодії повинна розглядати атомарними лише примітивні операції load, store, test.

Розглянемо різні способи реалізації завдання взаємного виключення

Заборона переривань. Найпростіше рішення полягає в забороні переривань при вході потоку в критичну область, і дозвіл їх при виході з неї. Оскільки процесор перемикається з потоку на потік тільки по перериванню від таймера, відключення переривань виключить передачу процесора іншому потоку на час знаходження потоку в критичному інтервалі. Однак некоректно написане користувальницький додаток може привести до краху операційної системи, наприклад, користувальницький потік відключив переривання, і, в результаті якого-небудь збою, не включив їх назад. Таким чином, заборона переривань буває корисним в самій операційній системі, але не прийнятний в якості механізму взаємного виключення для користувацьких потоків.

Активне очікування Постійна перевірка значення змінної в очікуванні деякого значення називається активним очікуванням. Цього способу слід уникати, оскільки він є безцільної тратою процесорного часу. Активне очікування використовується тільки тоді, коли є впевненість в невеликому терміні очікування. Блокування, що використовує активне очікування називається спін-блокуванням.

Активне очікування із суворим чергуванням. Розглянемо два паралельно виконуваних циклічних потоки. Змінна turn відстежує, чия черга входити в критичну область. Turn може приймати тільки два значення 0 або 1, спочатку дорівнює 0. Потік 0 перевіряє значення turn, зчитує 0 і входить в критичну область. Потік 1 також перевіряє змінну turn, зчитує 0 і входить в цикл, безперервно перевіряючи, коли turn стане рівним 1 Коли потік 0 виходить з критичного інтервалу, він міняє значення turn на 1, дозволяючи потоку 1 потрапити в критичну область. Однак це рішення має недолік: потоки надто пов'язані між собою. Для того щоб один з потоків повторно увійшов у свій критичний інтервал, необхідно, щоб другий потік тривав, тільки так він зможе змінити значення turn. Якщо другий потік, після виходу з критичної області затримався в залишку циклу, то він затримає і перший потік, що є порушенням прогресу.

Активне очікування без жорсткого чергування. Рішення, що задовольняє умовам взаємного виключення, прогресу і очікування запропонував Дейкстра. Дейкстра показав, що якщо не застосовувати спеціальних додаткових засобів синхронізації, то вирішити проблему взаємного виключення хоча і можна, але дуже складно навіть для двох потоків, і по-друге виникає проблема активного очікування, при якій потік чекає дозволу увійти в свій критичний інтервал, займаючи процесор, що призводить до неефективного функціонування всієї системи.

Атомарний доступ: сімейство Interlocked-функцій. Розглянемо приклад. Оголосимо глобальну змінну, присвоїмо їй 0, створимо два потоки - перший

виконує функцію Thread1, другий - Thread2. Код функцій ідентичний - обидві збільшують значення глобальної змінної на 1.

```
// Визначаємо глобальну змінну
 long x = 0;
DWORD WINAPI Thread1 (PVOID a)
{ x ++;
 return (0); }
DWORD WINAPI Thread2 (PVOID a)
{ x ++;
 return (0); }
```

Для забезпечення коректної роботи декількох потоків із загальним ресурсом, необхідно використовувати Interlocked функції, які гарантують, що потік отримає монопольний доступ до змінної, і що ніякий іншої потік не зможе одночасно змінити її.

```
LONG InterlockedIncrement (PLONG a);
LONG InterlockedDecrement (PLONG a);
 LONG InterlockedExchange (PLONG a, LONG b);
LONG InterlockedExchangeAdd (PLONG a, LONG b);
PVOID InterlockedCompareExchange (PVOID a, PVOID b, PVOID c);
```

Функції InterlockedIncrement і InterlockedDecrement, відповідно збільшують і зменшують значення змінної типу LONG на 1, її адресу передається в параметрі a. Функції не повертають нового значення змінної, а повертають код порівняння з нулем.

InterlockedExchange дозволяє замінити поточне значення змінної a на нове значення, що передається в параметрі b. Функція повертає вихідне значення змінної. Функція корисна при організації спін-блокування:

```
// Глобальна змінна, використовувана як індикатор того,
// чи зайнятий поділюваний ресурс
BOOL ResourseInUse = FALSE;
... void MyFunc () {
// Очікуємо доступу до ресурсу
while (InterlockedExchange (& ResourseInUse, TRUE) == TRUE)
sleep (0);
// Отримуємо монопольний доступ …
 // Поступаємося доступом
 InterlockedExchange (& ResourseInUse, FALSE); }
```

У функції MyFunc постійно крутиться цикл while, в якому змінної ResourseInUse присвоюється значення TRUE і перевіряється її попереднє значення. Якщо воно дорівнювало FALSE, значить ресурс не був зайнятий, але викликаючий потік щойно зайняв його - на цьому цикл завершується. В іншому випадку (значення дорівнювало TRUE) ресурс займав інший потік, і цикл повторюється.

InterlockedExchangeAdd - дозволяє змінювати значення змінної a на величину значення b:

LONG a = 5;

LONG old = InterlockedExchangeAdd (& a, -2);

// Тепер old = 5, a = 3 InterlockedExchange і InterlockedExchangeAdd повертають вихідне значення змінної.

InterlockedCompareExchange виконує операцію порівняння і привласнення на атомарному рівні. Вона порівнює поточне значення змінної a зі значенням, переданим у параметрі c. Якщо значення збігаються, в параметр a заноситься значення b, в іншому випадку, значення a не змінюється:

LONG a = 5;

LONG old = InterlockedCompareExchenge ((PVOID) & a, (PVOID) 500, (PVOID) 5);

// Тепер a = 500, old = 5

Немає жодної interlocked функції, що дозволяє отримати значення змінної в той момент, коли інший намагається її змінити - якщо один потік викликав InterlockedIncrement в той час, як інший читає вміст тієї ж змінної, її значення, читане другим потоком буде завжди достовірним. Він отримає значення до або після зміни змінної - головне значення завжди буде коректним, а не довільною величиною, отриманою в результаті втручання в операцію. Принцип роботи функцій на процесорах Intel базується на видачі по шині апаратного сигналу, що не дає іншим процесорам звернутися за тією ж адресою пам'яті. Функції виконуються дуже швидко - зазвичай не більше 50 тактів процесора, при цьому не відбувається переходу з режиму користувача в режим ядра (він віднімає не менше 1000 тактів). Функції можуть застосовуватися для захисту змінних тільки всередині одного процесу.

Критичні секції

1) Необхідно створити структуру CRITICAL_SECTION. Структура повинна бути глобальною, щоб до неї могли звернутися різні потоки. Функції, керуючі критичними секціями, самі ініціалізують і модифікують елементи даної структури.

CRITICAL_SECTION cs;

2) Ініціалізувати критичну секцію, передавши адресу структури CRITICAL_SECTION.

VOID InitializeCriticalSection (LPCRITICAL_SECTION cs);

3) Вхід в критичну секцію:

VOID EnterCriticalSection (LPCRITICAL_SECTION cs);

Функція перевіряє значення елементів структури LPCRITICAL_SECTION і виконує наступні дії:

- Якщо ресурс вільний, модифікуються елементи структури, вказуючи, що викликаний потік займає ресурс, після чого повертає управління, і потік, отримавши доступ до ресурсу, продовжує свою роботу.

- Якщо ресурс вже захоплений якимось потоком, функція оновлює елементи структури, відзначаючи, скільки разів поспіль цей потік захопив ресурс, і повертає управління.
- Якщо ресурс зайнятий іншим потоком - переводить і викликає потік в режим очікування. Потік, очікуючи входу в критичний інтервал, спить і не витрачає процесорного часу. Як тільки потік, який займає ресурс викличе LeaveCriticalSection, очікуваний потік буде пробуджений і отримає доступ до ресурсу.

4) Вихід із критичної секції:

VOID LeaveCriticalSection (LPCRITICAL_SECTION cs);

5) Якщо критична секція більше не потрібна, її слід видалити:

VOID DeleteCriticalSection (LPCRITICAL_SECTION cs);

Замість EnterCriticalSection можна скористатися функцією

VOID TryEnterCriticalSection (LPCRITICAL_SECTION cs);

Вона ніколи не призупиняє виконання виклику потоку. Якщо при її виклику зазначена критична секція не зайнята ні яким потоком, включаючи виклик, функція передає права на критичну секцію викликаному потоку і повертає TRUE. В іншому випадку вона лише тестує стан і негайно повертає FALSE. За допомогою критичних секцій не можна синхронізувати потоки з різних процесів. При спробі потоку увійти в критичну секцію, зайняту іншим потоком, він негайно призупиняється. Це значить, що потік переходить з режиму користувача в режим ядра, на що витрачається не менше 1000 тактів процесора. На багатопроцесорній машині потік, який володіє ресурсом, може виконуватися на іншому процесорі і може дуже швидко звільняти ресурс. Тоді існує ймовірність, що ресурс буде звільнений ще до переходу виклику потоку в режим ядра, тобто багато процесорного часу буде витрачено даремно. Для підвищення швидкодії критичних секцій в них інтегроване спінблокування: викликаючи EnterCriticalSection, функція виконує задану кількість циклів спін-блокування, намагаючись отримати доступ до ресурсу. Якщо всі спроби виявилися невдалими, функція переводить потік в режим ядра, де він перебуватиме в стані очікування.

Семафор – універсальний механізм для організації взаємодії процесів та потоків. Визначення семафору зробив нідерландський вчений Едсегр Дейкстра в1965 році. Семафор – об’єкт ядра ОС, який можна розглядати як лічильник, що містить ціле число в діапазоні від 0 до заданого при його створенні максимального значення. При досягненні семафором значення 0, він переходить у несигнальний стан, а при будь яких інших значеннях – у сигнальний.

Над семафором можна виконати такі операції:

Ініціалізація – встановлення початкового значення.

Перевірка стану семафору. Якщо семафор не рівний нулю, лічильник зменшується на 1, інакше процес блокується доти, поки лічильник не буде рівний 0.

Операція збільшення лічильника на 1.

Семафор, в залежності від значень, які він може приймати буває двійковий (0 та 1), трійковий (0, 1, 2) і т.д. Лічильник зменшується при кожному успішному завершенні функції очікування, і збільшується за допомогою виклику функції WINAPI.

З кожним семафором зв'язується список (черга) процесів, які очікують дозволи пройти семафор.

Для створення семафора служить функція CreateSemaphore:

HANDLE WINAPI CreateSemaphore(
LPSECURITY_ATTRIBUTESlpSemaphoreAttributes, //атрибути доступу
LONGlInitialCount, // початкове значення лічильника
LONGlMaximumCount, // максимальне значення лічильника
LPCTSTRlpName); // ім'я об'єкта

Функція повертає дескриптор створеного семафора, або 0, якщо створити його не вдалося. Якщо в системі вже є семафор з таким ім'ям, то новий не створюється, а повертається дескриптор існуючого. При використанні семафора в середині одного процесу можна його створити без імені, передавши в якості lpName значення null. Ім'я семафора не повинне збігатися з іменем уже існуючого об'єкта типу Event, Mutex, WaitableTimer, Job, або FileMapping

Дескриптор раніше створеного семафора може бути отриманий функцією OpenSemaphore:

HANDLE WINAPI OpenSemaphore(
DWORD dwDesiredAccess, //задає права доступу до об'єкта
BOOLbInheritHandle, //вказує, чи може семафор успадковуватись
LPCTSTRPlpName); //ім'я об'єкта

Для збільшення лічильника семафора використовується функція ReleaseSemaphore:

BOOL WINAPI ReleaseSemaphore(
HANDLE hSemaphore,// дескриптор
LONG lReleaseCount, // величина простору семафора
LPLONGlpPreviousCount); // адреса змінної, яка прийме попереднє значення

Якщо значення лічильника після виконання функції перевищить заданий для нього максимум, то функція ReleaseSemaphore поверне false і значення семафора не зміниться. Параметром lpPreviousCount можна передати null, якщо це значення програмі не потрібно.

На відміну від м'ютекса, семафор не можна захопити у володіння, оскільки відсутнє саме поняття прав власності над ним. Крім цього, семафор може бути звільнений будь-яким потоком.

М'ютекс (mutex, від mutualexclusion – взаємне виключення) – об'єкт взаємного виключення, призначений для захисту певного об'єкта у потоці від доступу інших потоків. М'ютекси є одним із варіантів семафорних механізмів для організації взаємного виключення. Їхнє основне призначення – організація

взаємного виключення для потоків з одного або різних процесів. М’ютекси – це прості двійкові семафори, які можуть перебувати в одному з двох станів – сигнальному або несигнальному (відкритий і закритий). Коли потік стає власником м’ютекса, він переводиться у несигнальний стан. Критична секція у значній мірі виконує ті ж завдання, що і м'ютекс. У операційних системах Microsoft Windows ГОЛОВНА ВІДМІННІСТЬ МІЖ М'ЮТЕКСОМ І КРИТИЧНОЮ СЕКЦІЄЮ в тому, що м'ютекс є об'єктом ядра і може бути використаний кількома процесами одночасно, критична секція ж належить процесу і служить для синхронізації тільки його потоків. Процедура входу і виходу критичних секцій зазвичай займає менший час, ніж аналогічні операції м'ютекса. Між м'ютексом і критичною секцією є й термінологічні відмінності. Процедура, аналогічна захопленню м'ютекса, називається входом у критичну секцію, зняття блокування м'ютекса — виходом з критичної секції. М’ютекс перебуває в сигнальному стані тільки тоді, коли він не належить жодному із процесів. Як тільки хоча б один процес запитує володіння м’ютексом, він переходить у несигнальний стан і залишається в ньому доти, поки не буде звільнений власником. На відміну від критичних секцій, м’ютекси дозволяють задавати точний інтервал очікування, а м’ютекси, залишені закінченими процесами, автоматично переходять у сигнальний стан.

Для створення м’ютекса використовується функція CreateMutex:

HANDLE WINAPI CreateMutex(
LPSECURITY_ATTRIBUTES lpMutexAttributes, // атрибути доступу BOOLbInitialOwner, // задає, чи буде процес володіти м’ютексом після створення LPCTSTRlpName); // ім’я м’ютекса

Функція повертає дескриптор створеного м’ютекса (або 0 при невдалому завершенні). Якщо м’ютекс із заданим ім'ям вже існує, повертається його дескриптор. У цьому випадку функція API GetLastError поверне код помилки ERROR_ALREDY_EXISTS. Ім'я не повинно збігатися з ім'ям уже існуючого об'єкта типу Semaphore, Event, Job, WaitableTimer або FileMapping. Ім’я може мати префікс "Global\" або "Local\", які відповідно вказують на створення об’єкта у глобальному або локальному просторі імен. Якщо невідомо, чи існує вже м’ютекс із заданим ім'ям, програма не повинна запитувати володіння об'єктом при створенні (тобто повинна передати в якості bInitialOwner значення false). Якщо м’ютекс уже існує, програма може отримати його дескриптор функцією OpenMutex:

HANDLE WINAPI OpenMutex(
DWORD dwDesiredAccess, // права доступу до об’єкта
BOOLbInheritHandle, // чи може успадковуватись дочірніми процесами
LPCTSTR lpName);// ім’я

Функція повертає дескриптор відкритого м’ютекса, або 0 у випадку помилки. Потік захоплює володіння м’ютексом через виклик однієї з функцій очікування

(WaitForSingleObject, WaitForMultipleObjects тощо) відносно дескриптора м’ютекса.
Параметр dwDesiredAccess може набувати одне з таких значень: MUTEX_ALL_ACCESS – програма отримує повний доступ до об'єкта; MUTEX_MODIFY_STATE – право змінювати стан м’ютекса (потрібне для звільнення м’ютекса функцією ReleaseMutex);
SYNCHRONIZE – програма може використовувати об'єкт тільки у функціях очікування й функції ReleaseMutex.
Вивільняється м’ютекс за допомогою функції ReleaseMutex:
BOOL WINAPI ReleaseMutex(HANDLEhMutex);

Функції очікування дозволяють потоку блокувати власне виконання. Функції очікування не повертаються, поки не будуть виконані вказані критерії. Тип функції очікування визначає набір використаних критеріїв. Коли викликається функція очікування, вона перевіряє, чи дотримані критерії очікування. Якщо критерії не були виконані, викличний потік переходить у стан очікування, поки не будуть виконані умови критеріїв очікування або не закінчиться вказаний інтервал очікування. Існує чотири типи функцій очікування:
1. Single-object
2. multiple-object
3. Alertable
4. Registered

Single-object Wait Functions
Функції SignalObjectAndWait(), WaitForSingleObject() та WaitForSingleObjectEx() вимагають дескриптора для одного об'єкта синхронізації. Ці функції повертаються, коли відбувається одне з наступного:
1. Вказаний об'єкт знаходиться в сигналізованому стані.
2. Інтервал тайм-ауту закінчується. Інтервал тайм-ауту можна встановити на INFINITE, щоб вказати, що очікування не закінчувалось.
Функція SignalObjectAndWait () дозволяє потоку, що викликає її, атомарно (атомна операція - викликається операція, яка не може бути перервана) встановити стан об’єкта для сигналізації та чекати, поки стан іншого об’єкта буде сигналізованим.
Multiple-object Wait Functions
Функції WaitForMultipleObjects(), WaitForMultipleObjectsEx(), MsgWaitForMultipleObjects() та MsgWaitForMultipleObjectsEx() дозволяють потоку, що викликає, вказати масив, що містить один або кілька дескрипторів об'єкта синхронізації. Ці функції повертаються, коли відбувається одне з наступного:
1. Стан будь-якого із зазначених об'єктів встановлюється як сигналізований або стан усіх об'єктів встановлюється як сигналізований. Ви визначаєте, чи використовуватиметься один або всі стани у виклику функції.

2. Інтервал тайм-ауту закінчується. Інтервал тайм-ауту можна встановити на INFINITE, щоб вказати, що очікування не закінчується.

Функції MsgWaitForMultipleObjects() та MsgWaitForMultipleObjectsEx() дозволяють вказати об'єкти вхідних подій у масиві обробника об'єктів. Це робиться, коли ви вказуєте тип вводу, який слід чекати у черзі вводу потоку. Наприклад, потік може використовувати MsgWaitForMultipleObjects(), щоб заблокувати його виконання, поки стан вказаного об'єкта не буде встановлений як сигналізований, і в черзі введення потоку доступне введення миші. Потік може використовувати функцію GetMessage() або PeekMessage() для отримання вхідних даних.

Під час очікування, коли стану всіх об’єктів буде встановлено сигналізацію, ці функції декількох об’єктів не модифікують стану зазначених об’єктів, поки стану всіх об’єктів не буде сигналізовано. Наприклад, стан мютексного об'єкта може сигналізуватися, але викличний потік не отримує права власності, поки для станів інших об'єктів, зазначених у масиві, також не встановлено сигналізацію. Тим часом, якийсь інший потік може отримати право власності на об'єкт мютексу, тим самим встановлюючи його стан без позначення.

Під час очікування сигналу стану окремого об’єкта ці функції з декількома об’єктами перевіряють дескриптори масиву в порядку, починаючи з індексу 0, доки один із об’єктів не буде сигналізований. Якщо сигналізують декілька об’єктів, функція повертає індекс першого дескриптора масиву, об’єкт якого був сигналізований.

Події (Events)

Події є об'єктом синхронізації ядра. Події можуть сигналізувати іншим потокам, вказуючи на те, що зараз виконується якась умова, наприклад доступне повідомлення. Важлива додаткова можливість, яку пропонують події, полягає в тому, що кілька потоків можуть бути звільнені від очікування одночасно, коли сигналізується одна подія.

Події класифікуються як події ручного скидання та події автоматичного скидання, і ця властивість події встановлюється викликом CreateEvent.

• Подія скидання вручну може сигналізувати декільком потоків, які очікують на подію одночасно, і це очікування може бути скинуто.

• Подія автоматичного скидання сигналізує один потік, який чекає на подію, і скидається автоматично.

Події використовують наступні функції: CreateEvent, CreateEventEx, OpenEvent, SetEvent, ResetEvent, PulseEvent.

```
HANDLE CreateEvent(
LPSECURITY_ATTRIBUTES lpsa,
BOOL bManualReset,
BOOL bInitialState,
LPTCSTR lpEventName)
```

Вкажіть подію скидання вручну, встановивши значення bManualReset як TRUE. Подібним чином подія спочатку встановлюється у сигнальний стан, якщо bInitialState є TRUE. Відкрити іменовану подію, можливо, з іншого процесу, за допомогою OpenEvent.

Функції які контролюють події

BOOL SetEvent (HANDLE hEvent),

BOOL ResetEvent (HANDLE hEvent),

BOOL PulseEvent (HANDLE hEvent).

Потік може сигналізувати про подію за допомогою SetEvent. Якщо подію буде автоматично скинуто, звільняється один потік очікування, можливо, один з багатьох, і подія автоматично повертається у несигнальний стан. Якщо жоден потік не чекає на подію, подія залишається в сигналізованому стані, поки потік не почне на неї не чекати, і потік негайно звільняється. Зауважте, що семафор з максимальною кількістю 1 має такий самий ефект.

Якщо, навпаки, подія скидається вручну, вона залишається сигнальною, доки потік не викликає ResetEvent для цієї події. За цей час усі потоки очікування звільняються, і, можливо, інші потоки чекатимуть і будуть звільнені перед скиданням.

PulseEvent звільняє всі потоки, які зараз очікують події ручного скидання, але подія потім автоматично скидається. У разі події автоматичного скидання вивільняє одну нитку очікування, якщо така є.

Зверніть увагу, що ResetEvent є корисною лише після сигналізації про подію вручну скидання SetEvent. Будьте обережні під час використання WaitForMultipleObjects, щоб дочекатися всіх сигнальних подій. Потік очікування буде звільнено лише тоді, коли всі події одночасно знаходяться у сигналізованому стані, а деякі сигналізовані події можуть бути скинуті до звільнення потоку.

Завдання.

1. Реалізувати заданий алгоритм в окремому потоці.
2. Виконати розпаралелення заданого алгоритму на 2, 4, 8, 16 потоків.
3. Реалізувати можливість зупинку роботи і відновлення, зміни пріоритету певного потоку.
4. Реалізувати можливість завершення потоку.
5. Застосувати різні механізми синхронізації потоків. (Згідно запропонованих варіантів)
6. Зобразити залежність час виконання – кількість потоків (для випадку без синхронізації і зі синхронізацією кожного виду).
7. Результати виконання роботи відобразити у звіті.

Варіанти методів синхронізації:

1) Мютекс
2) Семафор
3) Інтерлок-функції
4) Критична секція
5) Події
6) Функції очікування
7) Алгоритм активного очікування
8) Алгоритм монітору
9) Таймери очікування

При необхідності, до запропонованих методів можна додати інші методи. Наприклад, для реалізації активного очікування використати інтерлок – функції.

Варіанти завдань (згідно номеру у журналі підгрупи):

1. Реалізувати циклічний посимвольний вивід: прізвище ім'я, номер студентського. (кількість ітерацій >10000). Синхронізація: 9, 4.

2. Обчислити суму елементів заданого масиву (кількість елементів >10000, елементи рандомні). Синхронізація: 3, 2.

3. Бінарний пошук максимального елементу масиву (кількість елементів >10000, елементи рандомні). Вивести значення та індекс. Синхронізація: 2, 8

4. Бінарний пошук мінімального елементу масиву (кількість елементів >10000, елементи рандомні). Вивести значення та індекс. Синхронізація: 1, 6.

5. Сортування заданого масиву методом quick sort (кількість елементів >10000, елементи рандомні). Вивести посортований масив. Синхронізація: 3, 5.

6. Обчислити інтеграл функції f(x) = x*sin(x/2) на проміжку [0, 100] правилом трапеції (кількість розбитів 10000). Синхронізація: 5, 7.

7. Пошук заданого слова у файлі і вивід рядка з тим словом (кількість рядків у файлі > 1000, текст довільна наукова стаття). Синхронізація: 1, 4

8. Підрахунок кількості заданих символів у файлі. (кількість рядків у файлі > 1000, текст довільна наукова стаття). Синхронізація: 2, 3.

9. Вивід слів з файлу, що розпочинаються на задану літеру (кількість рядків у файлі > 1000, текст довільна наукова стаття). Синхронізація: 4, 9.

10. Обчислити суму елементів заданого масиву (кількість елементів >10000, елементи масиву задаються формулою a(0)=2, a(i)=a(i-1)*i+exp(i), i -індекс елементу). Синхронізація: 3, 8

11. Пошук найдовшого слова у файлі (кількість рядків у файлі > 1000, текст довільна наукова стаття). Вивести слово і кількість символів. Синхронізація: 1, 7

12. Обчислення n-го елементу ряду чисел Фібоначі, що розпочинається з f(0)=13, через рекурентні формули (n>10). Синхронізація: 1, 3

13. Вивід найменшого слова в кожному рядку. (кількість рядків у файлі > 1000, текст довільна наукова стаття) Синхронізація: 2, 4

14. Піднести задану квадратну матрицю до n-го ступеню (елементи матриці випадкові, n -ввести з клавіатури) Синхронізація: 5, 8

Полегшений варіант завдання і синхронізації: використання варіанту завдання 1 і синхронізації 1 і 4 - оцінюється на 1 бал менше.