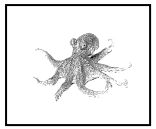

РОЗДІЛ 3.
Взаємодія процесів, що ґрунтується на спільних змінних

3.1. Взаємодія процесів

Взаємодія процесів пов'язана з двома основними завданнями:

- комунікацією процесів;
- синхронізацією процесів.

Комунікація процесів передбачає обмін даними між процесами. При цьому інформація передається від одного процесу до іншого в будь-якому напряму.

Синхронізація включає узгодження поведінки процесів і залежність виконання одного процесу від *подій*, що можуть бути в іншому процесі.

Механізм реалізації взаємодії процесів залежить від виду використовуваної комп'ютерної системи і в загальному випадку ґрунтується або на використанні моделі *спільних змінних* (системи зі спільною пам'яттю), або на використанні моделі *посилання повідомлень* (системи з локальною пам'яттю).

У системах зі спільною пам'яттю взаємодія процесів виконується за допомогою спільної пам'яті. Комунікація процесів і синхронізація процесів виконується через *спільні змінні*. Для передавання даних процес записує ці дані в спільні змінні, звідки їх зчитує інший процес. Синхронізація виконується за допомогою відповідних спільних змінних, які змінюються процесом, у якому відбувається подія, і зчитуються (аналізуються) процесом, що чекає на подію.

Використання спільних змінних, що доступні будь-якому процесу (це обов'язково глобальні змінні), приводить до того, що в програмі може відбуватися одночасне звертання процесів до спільних змінних, що призводить до конфлікту процесів або некоректної роботи програми. Тому програмування з використанням моделі спільних змінних пов'язано з необхідністю вирішення специфічних проблем, які загалом зводяться до розв'язання двох основних завдань – *взаємного виключення* і *синхронізації процесів*.

3.1.1. Завдання взаємного виключення

Завдання взаємного виключення полягає в тому, що під час виконання двох і більше паралельних процесів може виникнути одночасне звернення процесів до одного і того ж *спільного ресурсу* (СР). Таке звернення зазвичай призводить до конфлікту процесів, що полягає або в різкому уповільненні роботи програми, або в некоректній її роботі, якщо, наприклад, один процес зчитає дані, а другий їх змінює в цей же час, або (в крайньому випадку) – до аварійного завершення програми.

Загальна схема вирішення завдання взаємного виключення ґрунтується на тому, що потрібно *призупинити* (*блокувати*) процес, який звертається до спільного ресурсу, який вже використовується в цей момент іншим процесом. *Розблокування* процесу має бути виконано одразу після звільнення спільного ресурсу.

Існують *два підходи* до вирішення завдання взаємного виключення. Перший підхід ґрунтується на *контролі процесів* і пов'язаний з виявленням у процесах ділянок, у яких вони звертаються до спільного ресурсу. Такі ділянки процесів отримали назву *критичних ділянок (КД)*. Для вирішення завдання взаємного виключення необхідно *не допустити* одночасного входження процесів у свої критичні ділянки. Якщо один процес уже знаходиться в критичній ділянці, то будь-який інший процес за намагання входження в свою критичну ділянку має бути блокований доти, доки перший процес не вийде зі своєї критичної ділянки.

Класична схема організації такого контролю потребує використання операцій (примітивів) `ВХІДКД` і `ВИХІДКД`, що розміщуються відповідно перед і після критичної ділянки, тобто обрамляють критичну ділянку, створюючи "огорожу" навколо неї (рис. 3.1).

Алгоритм виконання операції *ВХІДКД*:

1. Перевірити, чи знаходиться будь-який інший процес у своїй критичний ділянці ?
2. Якщо знаходиться, то блокувати процес, що виконує операцію *ВХІДКД*.
3. Якщо не знаходиться, то встановити заборону на входження всіх процесів у свої критичної ділянки і дозволити цьому процесу увійти в його критичну ділянку.

Алгоритм виконання операції *ВИХІДКД*:

1. Зняти заборону на входи процесів в їх критичні ділянки.

Конкретна реалізація примітивів *ВХІДКД* і *ВИХІКД* у мовах та бібліотеках програмування виконана у вигляді механізмів семафорів, мютексів, критичних секцій.

Другий підхід до вирішення завдання взаємного виключення передбачає контроль безпосередньо за спільним ресурсом (рис. 3.2). Такий контроль може бути здійснений за допомогою оголошення належних змінних спільними ресурсами, а також за допомогою моніторів. З цією метою створюється програмний модуль (монітор), що містить в собі спільний ресурс, а також набір процедур для доступу до спільного ресурсу. Тепер доступ до спільного ресурсу з процесів можливий тільки через виклик потрібної процедури монітора.

Процедури монітора мають важливу властивість – вони *взаємно виключають* один одного. Тобто, якщо процес викликав і виконує будь-яку процедуру монітора, то виклик іншим процесом будь-якої процедури цього монітора приведе до блокування цього процесу до того часу, поки не закінчиться виконання вже розпочатої процедури монітора. Таким чином, монітор дозволяє виконання

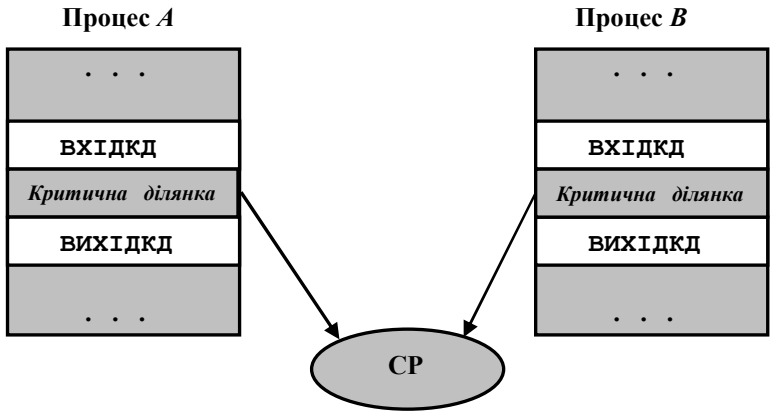

Рис. 3.1. Загальна схема вирішення завдання взаємного виключення, яка ґрунтується на контролі процесів

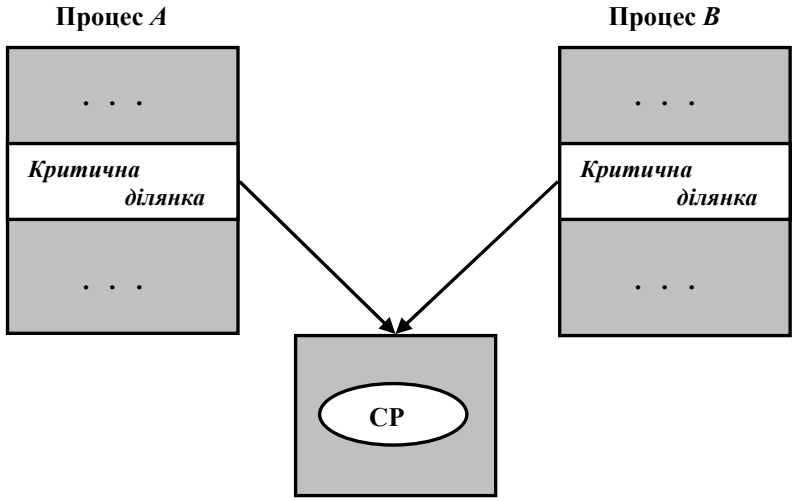

Рис. 3.2. Загальна схема вирішення завдання взаємного виключення, яка ґрунтується на контролі спільного ресурсу

тільки однієї процедури. Якщо ввести термін «процес знаходиться в моніторі», під яким розуміти, що процес виконує будь-яку процедуру монітора, то можна стверджувати, що в моніторі може знаходитись тільки один процес. Механізм моніторів реалізується різними способами: у мові Ада – за допомогою захищених модулів, в мові Java – за допомогою синхронізованих методів.

3.1.2. Завдання синхронізації процесів

Завдання синхронізації двох процесів полягає в тому, що в одному процесі, наприклад `B`, у визначеній точці (*точці події*) виконується подія (обчислення даних, уведення або виведення даних і т.ін.), а другий процес `A` у визначеній точці (*точці очікування події*) блокується доти, доки ця подія не відбудеться і він зможе продовжити своє виконання. Точка події і точка очікування – це *точки синхронізації*. Якщо процес `A` вийшов на точку синхронізації, коли подія вже відбулася, то він не блокується і продовжує виконуватись.

Існує декілька схем синхронізації процесів (рис. 3.3):

- один процес очікує на подію в одному процесі;
- один процес очікує на події в кількох процесах;
- декілька процесів очікують на подію в одному процесі.

Точки `S` і `W` на рис. 3.3 – це точки синхронізації процесів; `S` означає точку посилання сигналу про подію, `W` –точку очікування сигналу про подію.

Для вирішення завдання синхронізації, яке іноді називають *синхронізацією за подіями (event synchronization)*, можна використовувати різні механізми синхронізації процесів, таки як семафори, події, монітори.

3.2. Семафори

Механізм семафорів запропонував математик Е.Дейкстра [9; 35]. У класичній інтерпретації механізм семафорів – це спеціальний захищений тип `Semaphore` та дві неподільні операції над змінною `S` цього типу: `P(S)` і `V(S)`. Неподільність операції означає, що її не можна переривати, поки не завершиться її виконання. Бінарні

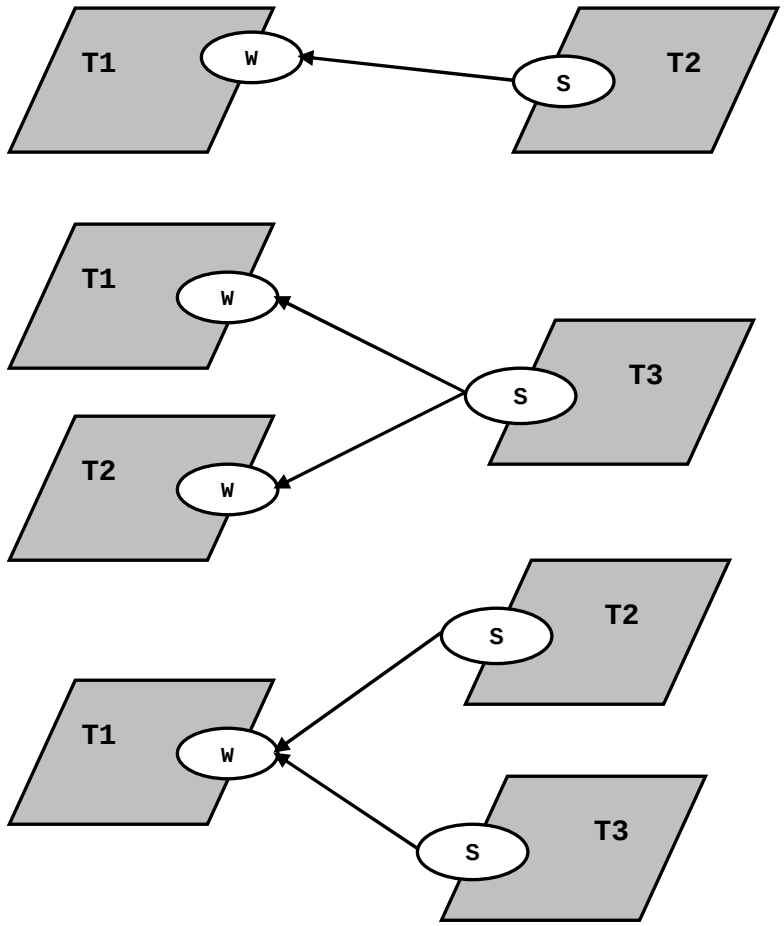

Рис. 3.3. Види синхронізації процесів

семафори набувають значень 0 і 1, багатозначні (множні) семафори – значень 0, 1, .., M.

Алгоритми операцій P(S) і V(S):

Операція P(S): (*ВХІДКД*)

1. Перевірити значення S.
2. Якщо S = 0, то блокувати процес, що виконує цю операцію.
3. Інакше S:= S – 1 і дозволити входження процесу в критичну ділянку.

Операція V(S): (*ВИХІДКД*)

1. S := S – 1

Механізм семафорів є універсальним, який може бути використаний як для вирішення завдання взаємного виключення (рис. 3.4), так і для синхронізації процесів (рис. 3.5).

3.2.1. Семафори в мові Ада

У другому стандарті мови Ада95 механізм семафорів подано у вигляді пакета Synchronous_Task_Control в додатку Annex D: Real-Time Systems. Пакет реалізує механізм семафорів таким чином. Семафорний тип забезпечується приватним типом Suspension_Object, операції P(S) і V(S) реалізовані за допомогою процедур Suspend_Until_True() і Set_True(). Використовується бінарний логічний семафор, тобто семафорні змінні типу Suspension_Object набувають значень false і true. Крім указаних процедур, в пакеті реалізовані допоміжні процедури Set_False() для встановлення значення семафора в false і Current_State() для зчитування поточного значення семафора.

Специфікація пакета:

```
package Ada.Synchronous_Task_Control is
   type Suspension_Object  is limited private;
```


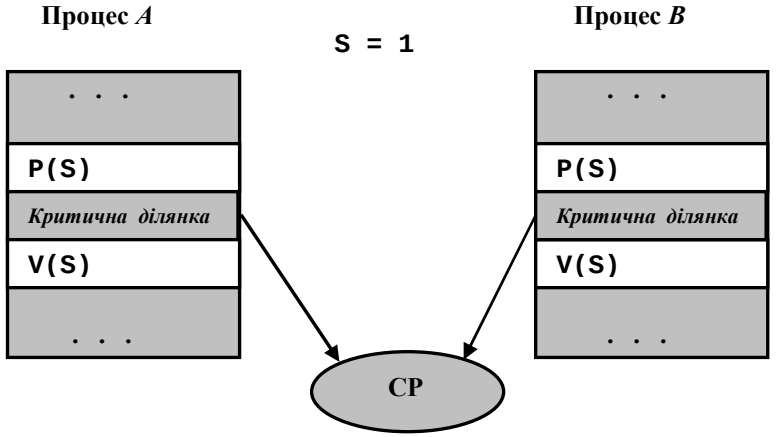

Рис. 3.4. Загальна схема вирішення завдання взаємного виключення з використанням семафорів

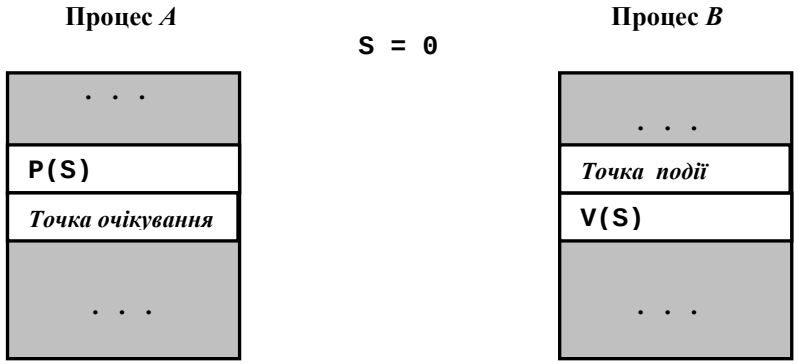

Рис. 3.5. Загальна схема вирішення завдання синхронізації з використанням семафорів

```
    procedure Set_True(S :  in out Suspension_Object);
    procedure Set_False(S : in out Suspension_Object);
    function  Current_State(S : Suspension_Object)
                                        return Boolean;
    procedure Suspend_Until_True(S : in out
                                    Suspension_Object);
private
   .  .  .
end Ada.Synchronous_Task_Control;
```

3.2.2. Семафори в Win32

Семафори в бібліотеці Win32 реалізовані за допомогою спеціального типу, а також кількох функцій. Семафор в реалізації Win32 визначається як лічильник, що набуває значення від нуля до визначеного значення, тобто використовує багатозначний семафор. Якщо семафор дорівнює нулю, то він заборонений, і в разі виклику процесом функції *очікування* цей процес буде блокований доти доки часу, поки інший процес не змінить значення семафора (інкрементує лічильник).

Створення та використання семафорів у Win32 виконується за допомогою типу HANDLE, а також набору функцій:

1.Функція CreateSemaphore() створює об'єкт - семафор:

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
                    // покажчик на  атрибути захисту
  LONG lInitialCount,
                    // початкове значення семафора
  LONG lMaximumCount,
                   // максимальне значення семафора
  LPCTSTR lpName
                   // покажчик на ім'я об'єкта
);
```

Параметри:

- lpSemaphoreAttributes – покажчик на структуру SECURITY_ATTRIBUTES, яка визначає, чи може ідентифікатор, що повертається, бути успадкованим дочірнім процесом. Якщо покажчик установлено в NULL, то покажчик не може бути успадкованим;

- lInitialCount – початкове значення лічильника семафора. Воно не може бути меншим за нуль або більшим за максимальне значення; якщо початкове значення лічильника встановлено в нуль, то семафор знаходиться в забороненому (закритому) стані;
- lMaximumCount – визначає максимальне значення семафора, яке має бути більше нуля;
- lpName – визначає ім'я об'єкта у вигляді рядка, який має закінчуватися нулем. Якщо параметр NULL, то створюється семафор, який не має ім'я.

2. Функція OpenSemaphore()повертає ідентифікатор уже створюваного семафора:

```
HANDLE OpenSemaphore(
   DWORD dwDesiredAccess, // прапор доступу
BOOL bInheritHandle,      // прапор наслідування
   LPCTSTR lpName         // покажчик на ім'я об'єкта
);
```

Параметри:

- dwDesiredAccess – визначає вид доступу до об'єкта – семафора;
- bInheritHandle – визначає, чи буде ідентифікатор успадкованим. Якщо TRUE, створюваний процес, може успадкувати ідентифікатор;
- lpName – ім'я об'єкта в рядку, що закінчується нулем.

3.Функція ReleaseSemaphore() збільшує лічильник семафора на визначене значення.

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,  //ідентифікатор об'єкта семафора
  LONG lReleaseCount, //  число, що додається до
                      //  поточного значення семафора
  LPLONG lpPreviousCount  // адрес попереднього
                          // лічильника
);
```

Параметри:

- `hSemaphore` – ідентифікатор об'єкта - семафора;
- `lReleaseCount` – визначає значення, на яке буде змінюватися значення семафора;
- `lpPreviousCount` – покажчик на змінну для отримання попереднього значення семафора.

4. Функція `WaitForSingleObject()` належить до групи функцій *очікування*. Вона перевіряє значення семафора. Якщо семафор дорівнює нулю, то процес блокується; інакше, значення семафора зменшується на одиницю.

```
DWORD WaitForSingleObject(
  HANDLE hHandle,            // ідентифікатор об'єкта
  DWORD dwMilliseconds,   //  тайм-аут в мілісекундах
  BOOL  bAlertable       //  прапор раннього виконання
);
```

Параметри:

- `hHandle` – ідентифікатор об'єкта - семафора;
- `dwMilliseconds` – визначає час очікування в мілісекундах. Якщо час очікування не визначений (процес буде чекати поки не зміниться значення семафора), то цей параметр встановлюється як `INFINITE;`
- `bAlertable` – визначає можливість раннього виконання функції.

Функції очікування. Існує три типи *функцій очікування*, які дозволяють потоку блокувати своє виконання залежно від результатів виконання перевірки стану об'єкта синхронізації:

- одиночні,
- множинні,
- попереджувальні.

Одиночні функції синхронізації `SignalObjectAndWait`, `WaitForSingleObject`, `WaitForSingleObjectEx` використовують ідентифікатор (типу `HANDLE`) об'єкта синхронізації і блокують процес, якщо об'єкт синхронізації знаходиться у забороненому стані; функції закінчують своє виконання, якщо виконується таке:

- указаний об'єкт знаходиться у дозволеному стані,
- вичерпаний час очікування. Час очікування може бути встановлений в `INFINITE`, що означає необмежений час очікування.

Функція **`SignalObjectAndWait`** дозволяє викличному потоку встановити стан одного об'єкта в дозволений і чекати дозволеного стану іншого об'єкта.

Формати виклику цих функцій:

```
BOOL SignalObjectAndWait(
  HANDLE hObjectToSignal,
                //ідентифікатор об'єкта, що сигналізує
  HANDLE hObjectToWaitOn,
                // ідентифікатор об'єкта-сигналу
  DWORD dwMilliseconds,
                //тайм-аут у мілісекундах
  BOOL bAlertable
                //прапор раннього виконання
);

DWORD WaitForSingleObject(
  HANDLE hHandle,
                  // ідентифікатор об'єкта,
                  // від якого очікується сигнал
  DWORD dwMilliseconds   // тайм-аут у мілісекундах
);


DWORD WaitForSingleObjectEx(
  HANDLE hHandle,  // ідентифікатор об'єкта,
                   // від якого очікується сигнал
  DWORD dwMilliseconds, // тайм-аут в мілісекундах
  BOOL  bAlertable      // прапор раннього виконання
                        //(для операцій В/В)
 );
```

Множинні функції `WaitForMultipleObjects`, `WaitForMultipleObjectsEx`, `sgWaitForMultipleObjects`, `MsgWaitForMultipleObjectsEx` дозволяють викличному потоку визначати масив, який містить один або кілька ідентифікаторів

об'єктів синхронізації; функції закінчують своє виконання, якщо вони реалізують таке:

- стан одного або кількох (усіх) об'єктів дозволений – можна вказувати об'єкти для виклику функцій;
- вичерпаний час очікування. Час очікування може бути встановлений в INFINITE, що означає необмежений час очікування.

Функції MsgWaitForMultipleObjects і MsgWaitForMultipleObjectsEx дозволяють визначити об'єкт події введенням масивів ідентифікаторів об'єктів, для чого потрібно визначити тип уведення в черзі введення потоку. Наприклад, потік може використати MsgWaitForMultipleObjects для власного блокування доти, доки, поки стан об'єкта не буде встановлено і дозволено введення з маніпулятора «миша» в черзі введення потоку. Потік може використати функції GetMessage або PeekMessage для оброблення введення.

Формати виклику цих функцій

```
DWORD WaitForMultipleObjects(
   DWORD nCount,       // кількість об'єктів у масиві
   CONST HANDLE *lpHandles,
              // покажчик на масив ідентифікаторів
   BOOL bWaitAll,      // чекати одного або всіх
   DWORD dwMilliseconds // тайм-аут
);

DWORD WaitForMultipleObjectsEx(
   DWORD nCount, // кількість об'єктів у масиві
   CONST HANDLE *lpHandles,
                // покажчик на масив ідентифікаторів
   BOOL  bWaitAll,         // чекати одного або всіх
   DWORD dwMilliseconds,   // тайм-аут
   BOOL  bAlertable     // прапор раннього виконання
);

DWORD MsgWaitForMultipleObjects(
   DWORD    nCount,      //кількість об'єктів у масиві
   LPHANDLE pHandles,    // покажчик на масив
                         // ідентифікаторів
   BOOL  fWaitAll,       // чекати одного або всіх
   DWORD dwMilliseconds, // тайм-аут
   DWORD dwWakeMask      // тип події уведення
                         // для очікування
);

DWORD MsgWaitForMultipleObjectsEx(
    DWORD    nCount,      //кількість об'єктів у масиві
    LPHANDLE pHandles,    // покажчик на масив
                          // ідентифікаторів
    DWORD dwMilliseconds, // тайм-аут
    DWORD dwWakeMask,     // тип події уведення
                          // для очікування
    DWORD dwFlags         // прапор очікування
);
```


Попереджувальні функції MsgWaitForMultipleObjectsEx, SignalObjectAndWait, WaitForMultipleObjectsEx, WaitForSingleObjectEx можуть додатково виконувати попереджувальні операції. Вони також можуть закінчуватися з досягненням деякої умови, але також і в разі появи в системній черзі введення-виведення інформації, або віддаленого виклику процедур.

Функції очікування можуть змінювати стан об'єктів синхронізації, причому змін зазнають тільки ті об'єкти, від яких залежить вихід з функції очікування і розблокування потоку відповідно. Таким чином, функції змінюють такі типи об'єктів синхронізації:

- лічильник семафора зменшується на одиницю і досягає забороненого стану, якщо дорівнює нулю;
- об'єкти мютексу (Mutex), подія автоскидання встановлюються в заборонені;
- стан таймера синхронізації заборонений.

3.3. Монітори

Ідея монітора, яку запропонував Б. Хансен і розвинув С. Хоар, ґрунтується на об'єднанні змінних, що описують спільний ресурс, і дій, які визначають засоби доступу до спільного ресурсу [9; 35]. *Монітор – програмний модуль*, що містить змінні та процедури для роботи з ними, причому доступ до змінних можливий *тільки через процедури* монітора.

Монітор – засіб розподілу ресурсів і взаємодії процесів. Це призначення монітора реалізується за допомогою властивостей, якими наділені процедури монітора. Характерна особливість про-

цедур монітора – *взаємне виключення* ними одне одного. У будь-який момент часу може виконуватися *тільки одна* процедура монітора. Якщо будь-який процес викликав і виконує процедуру монітора, то жоден процес не може виконувати будь-які процедури цього монітора. За спроби виклику іншим процесом процедури, що виконується, або іншої процедури монітора цей процес блокується і розміщується в черзі блокованих процесів доти, доки активний процес не закінчить виконання процедури монітора. Тобто в моніторі не може "знаходитись" більше одного процесу. Така властивість процедур монітора забезпечує взаємне виключення процесів, які працюють з монітором.

Загальна структура монітора:

```
monitor  Ім'я_Монітора;
       -- Опис локальних даних
       -- Опис процедур для доступу до даних
begin
       -- Ініціалізація локальних даних
end  Ім'я_Монітора;
```

У моніторі декларуються локальні змінні (спільні змінні), які захищені монітором, і процедури монітора. Значення локальних змінних можуть бути встановлені під час створення монітора. Далі значення цих змінних можуть бути прочитані або змінені процесами тільки за допомогою процедур, визначених у моніторі.

Приклад монітора:

```
monitor  Склад;

      Товар: Data;       -- спільний ресурс

      procedure На_Склад (T: in Data);
      procedure Зі_Складу(T: out Data);

begin

      Товар:= 0.0; -- ініціалізація спільного
                   -- ресурсу
end  Склад;
```


Властивості процедур монітора забезпечують вирішення завдання взаємного виключення за доступу до спільних ресурсів, об'явленими в моніторі. При цьому монітор формує чергу процесів, які викликали процедури монітора і є блокованими через зайнятість монітора (тобто спільного ресурсу).

3.3.1. Монітори в мові Ада95

Концепцію моніторів у новому стандарті мови Ада95 реалізовано у вигляді спеціальних програмних модулів – захищених модулів (`protected units`) [16; 18; 45]. Їх призначення – розширення можливості мови для програмування паралельних процесів, зокрема, для вирішення проблеми доступу до спільних ресурсів і синхронізації процесів. Крім того, захищені модулі забезпечують підтримку різних парадигм систем реального часу, для розроблення яких мову Ада використовують в першу чергу.

Спільні дані і операції над ними (захищені операції) об'єднуються в захищеному модулі, аналогічно тому, як це робиться в інших модулях мови Ада – пакетах. Доступ до спільних ресурсів можливий тільки через захищені операції, які мають властивості, що дозволяють вирішити завдання взаємного виключення під час роботи зі спільними ресурсами.

Як і всі модулі в мові, захищені модулі складаються зі специфікації і тіла.

Специфікація захищеного модуля:

```
PROTECTED [TYPE]   Ім'я_Захищеного_Модуля
                                [Дискримінант]    IS
    --  Опис_Захищених_Операцій
[PRIVATE]
    --  Опис_Захищених_Елементів
END  Ім'я_Захищеного_Модуля;
```

Захищенні операції – це:

- захищені функції,
- захищені процедури,
- захищені входи.

Захищені функції забезпечують доступ тільки до *читання* захищених елементів. Але дозволяють робити це *одночасно* всім про-

цесам автоматичним копіюванням елементів, які зчитуються. Це порушує головну властивість процедур монітора, яка дозволяє знаходитися в моніторі тільки одному процесу, але це "порушення" дозволяє скоротити час доступу до захищених елементів і не має будь-яких наслідків, оскільки зміна даних заборонена і не виконується.

Захищені процедури забезпечують ексклюзивний доступ до захищених елементів через читання і запис.

Захищені входи забезпечують ті самі функції, що й захищені процедури, додатково реалізуючи за допомогою *бар'єрів* ексклюзивний (умовний) доступ до тіла захищеного входу. Це дозволяє реалізувати за допомогою входів вирішення завдання синхронізації.

Приватна частина специфікації обмежує видимість захищених елементів: операцій і об'єктів, що описані в ній. Спільні дані, доступ до яких контролюється захищеним модулем, описуються в приватній частині його специфікації.

❖ **Приклад 3.1** Специфікації захищеного модуля

```
------------------------------------------------------------
-- Ада95. Захищений модуль                                 --
------------------------------------------------------------
protected   Контроль  is
   procedure   Включення;       -- захищені підпрограми
   function    Перемкнути(X :  Float);
end  Контроль;

protected   type   Сенсор   is
   entry   Чекати;              -- захищені  входи
   entry   Сигнал;
   procedure Зміна_Стану(X : in  float);
   function  Замір_Стану return float;
private
   Прп: Boolean:= False;     -- Прапор
   Стан: float;
end  Сенсор;

protected   Блок232(Номер: in Positive)  is
   entry  Параметри(X: out  integer);
   procedure  ЗмінаПараметра(Y: in  integer);
private
   -- захищений елемент
```



```
   Об'єкт: array(1 .. Номер) of  integer;
end  Блок232;
```

Тіло захищеного модуля реалізує захищені операції, які об'явлені в його специфікації, використовуючи для цього локальні ресурси, які можуть бути об'явлені в тілі модуля.

```
PROTECTED  BODY   Ім'я_Захищеного_Модуля   IS
   --  Локальні_Описи
BEGIN
   --  Реалізація  захищених операцій і
   --  захищених елементів
END  Ім'я_Захищеного_Модуля;
```

Захищені процедури і функції реалізуються в тілі захищеного модуля, як це робиться в тілі пакета. На відміну від модулів - задач, реалізація захищеного входу в тілі захищеного модуля не пов'язана з оператором приймання accept, а виконується за допомогою тіла входу, у якому обов'язково використовується *бар'єр*.

Описання тіла захищеного входу:

```
ENTRY  Ім'я_Захищеного_Входу  WHEN   Умова  IS
BEGIN
   .  .  .     -- Послідовність_Операторів
END    Ім'я_Защищеного_Входу;
```

Конструкція **When** Умова – це бар'єр, де Умова – логічний вираз, який визначає *відкритий* або *закритий* вхід. Перевірка умови в бар'єрі виконується під час виклику захищеного входу. Якщо значення виразу Умова дорівнює true, то вхід відкритий і виконується тіло захищеного входу, інакше вхід є зачинений і виконання процесу, який викликав цей вхід, блокується до того часу, поки значення виразу Умова в бар'єрі не буде змінено в true іншою задачею за допомогою захищеної процедури або іншого захищеного входу.

В прикладі 3.2 наведено реалізацію тіла захищеного модуля Сенсор, специфікацію якого подано раніше у прикладі 3.1:

❖ **Приклад 3.2**

```
------------------------------------------------------------
-- Ада95. Тіло захищеного модуля                           --
------------------------------------------------------------
   protected  body   Сенсор  is

      -- тіло захищеного  входу з бар'єром
      entry   Чекати  when  Прп is

      begin
         Прп:= False;
      end  Чекати;

      -- тіло захищеного  входу з бар'єром
      entry  Сигнал  when  not  Прп  is
      begin
         Прп:= True;
      end  Сигнал;

      procedure Зміна_Стану(x : in  float) is
      begin
         Стан := x;
      end Зміна_Стану;

      function  Замір_Стану return float is
      begin
         return Стан;
      end  Замір_Стану;

   end  Сенсор;
```

Структуру захищеного модуля Сенсор показано на рис. 3.6. Захищену функцію Замір_Стану() і процедуру Замір_Стану() зображено справа, захищені входи Чекати()і Сигнал() – зліва. Захищені елементи (Стан, Прп) зображено всередині захищеного модуля в овалі.

Виклик захищеної функції дозволяє процесу зчитувати дані із захищеного модуля. Кілька процесів можуть виконувати таке читання *одночасно*, викликаючи потрібні функції. Під час виконання читання в тілі захищеної функції заборонено зміну даних. Тіло захищеної функції може містити виклик іншої захищеної функції, але не виклик захищеної процедури.

Виклик захищеної процедури дозволяє процесу як читати, так і змінювати інформацію в захищеному модулі. На відміну від захищеної функції під час виконання захищеної процедури дозволяється змінювати дані. Якщо кілька процесів виконують виклик захищених процедур, то тільки один з них отримує можливість роботи з викликаною процедурою. У тілі захищеної процедури дозволено виклик як захищеної функції, так і захищеної процедури.

Виклик закритого захищеного входу приводить до блокування процесу до того часу, поки вхід не стане відкритим, тобто умовний вираз в бар'єрі набуде значення True. Це може статися в разі виконання іншим процесом потрібної захищеної операції, пов'язаної зі змінними, використаними в бар'єрі.

Блокований процес розміщується в черзі, яка пов'язана із входом, а також зі змінними, що використовуються в бар'єрі. Якщо вхід відкритий, то виконується тіло входу.

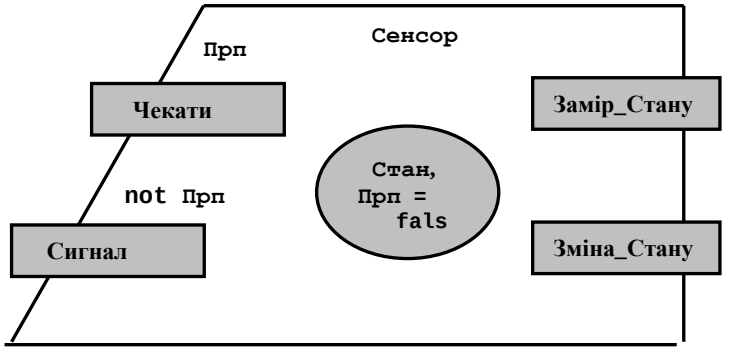

Рис. 3.6. Структура захищеного модуля

❖ **Приклад 3.3**

```
------------------------------------------------------------
-- Ада95.Захищений модуль у завданні взаємного виключення -
------------------------------------------------------------
protected   Буфер    is
    procedure  Додати  (Вклад:  out  Positive);
    procedure  Видалити(Вклад:  out  Positive);
private
```

```
    Лічильник:  Integer:= 0;
end Буфер;

-- тіло захищеного модуля
protected  body  Буфер  is

    procedure  Додати(Вклад: out  Positive) is
    begin
        Лічильник := Лічильник + 1;
        Вклад := Лічильник;
    end Додати;
    procedure  Видалити(Вклад: out Positive) is
    begin
        Лічильник := Лічильник - 1;
        Вклад := Лічильник;
    end  Видалити;
 end   Буфер;
```

Задачі додають або зменшують значення змінної `Лічильник`, викликаючи процедури `Додати` і `Видалити` захищеного модуля `Буфер`:

```
Буфер.Додати(Зарплата);     Буфер.Видалити(Плата);
```

Захищений модуль `Буфер` гарантує синхронізований доступ задач до захищеної змінної `Лічильник`. Черги під час роботи із захищеним модулем не створюються, оскільки використовуються тільки захищені процедури, а не захищені входи.

У прикладі 3.4 захищений модуль `Вклад` виконує роль буфера, куди задачі `Клієнт_А` і `Клієнт_В` записують і звідки зчитують дані. Захищений модуль `Вклад` повинен забезпечити взаємовиключний доступ задач до спільного ресурсу, яким є змінна `Рахунок`, а також синхронізацію процесів залежно від стану ресурсу.

❖ Приклад 3.4

```
------------------------------------------------------------
-- Ада95.Захищений модуль у завданні взаємного виключення -
-- та синхронізації процесів                               --
------------------------------------------------------------
protected  Вклад   is
    entry  В_Банк(М  :  in    Гроші);
```



```
    entry  З_Банку(М :  out   Гроші);
private
    Рахунок: Гроші;           -- спільний ресурс
    Прапор:  Boolean := False;
end  Вклад;
----------
protected  body    Вклад    is
    entry   В_Банк(М:  in   Гроші)
                          when  Прапор = False  is
    begin
       Рахунок:= М;
       Прапор:= True;
    end  В_Банк;
    entry  З_Банку(М:  out   Гроші)
                          when   Прапор = True  is
       М := Рахунок;
       Прапор := False;
    end  З_Банку;
end Вклад ;
---------
task  Клієнт_А;

task  body   Клієнт_А  is
   Дохід: Гроші;
begin
       .  .  .
    --  виклик  входу В_Банк
    Вклад.В_Банк(Дохід);
       .  .  .
end  Клієнт_А;
--------------
task  Клієнт_В;

task  body  Клієнт_В  is
   Сплата: Гроші;
begin
       .  .  .
    --  виклик  входу  З_Банку
    Вклад.З_Банку(Сплата);
       .  .  .
end  Клієнт_В;
```

3.3.2. Монітори в мові Java

Мова Java не має моніторів, подібних до захищених модулів у мові Ада, але вона дозволяє створювати класи, методи яких будуть мати основну властивість процедур монітора – виконуватися в

режимі взаємного виключення. Такі методи в Java повинні мати модифікатор `synchronized`. Інкапсуляція спільного ресурсу в класі-моніторі виконується за допомогою модифікатора `private`.

Метод, описаний як синхронізований, допускає виконання тільки в одному потоці. Якщо інший потік викликає синхронізований метод, який вже виконується, то такий потік буде блокований доти, доки не завершиться виконання синхронізованого методу.

Приклади синхронізованих методів:

```
void   synchronized  Додати(double x);
int    synchronized  Обсяг (int x, int y);
```

❖ **Приклад 3.5** Клас, який виконує функції монітора

```
/*-----------------------------------------------------
--  Реалізація монітора в мові Java                  --
----------------------------------------------------- */
class Монітор{

    private int Буфер;       // спільний ресурс

    // синхронізовані процедури доступу до
    // спільного ресурсу Буфер
    synchronized int Читати(){
       return Буфер;
    }
    synchronized void Писати(int x){
       Буфер = X;
    }
}// Монітор
```

Створення екземпляра класу `Монітор`:

```
Монітор  ZZ = new Монітор();
```

Одночасне виконання методів `ZZ.Читати()` та `ZZ.Писати()` не можливо. Якщо процес (потік) `B` викликав і виконує, наприклад, метод `ZZ.Читати()`, то інший потік `A` буде блокований, якщо він спробує викликати будь-який метод екземпляра `ZZ` класу `Монітор`, і зможе продовжити своє виконання тільки після завершення виконання методу `ZZ.Читати()` в потоці `B`.

3.4. Вирішення завдання взаємного виключення

Розглядаються приклади вирішення задвдання взаємного виключення. При цьому припускається погодження про то, що змінна є спільним ресурсом незалежно від того, читається вона або змінюється. Перевага надається попередньому копіюванню спільних змінних до виконання операцій з ними в критичній ділянці, які зазвичай потребують більше часу, ніж створювання копій. Застосування реальних СМП систем показало, що відмова від попереднього копіювання спільних змінних (особливо це стосується великих масивів), які тільки читаються, призводить до хаотичної конкуренції процесів, які одночасно використовують спільний ресурс, і це пов'язано зі збільшенням часу виконання програми. В цілому питання попереднього копіювання спільних даних треба вирішувати у кожному окремому випадку, виконавши попередній аналіз необхідності створювання копій або відмови від нього у випадку, якщо виконання операцій в критичній ділянці не потребує значного часу.

3.4.1. Вирішення завдання взаємного виключення в мові Ада95

В мові Ада95 рішення завдання взаємного виключення можна здійснити за допомогою прагм `Atomic` і `Volatile`, механізму семафорів або з використанням механізму моніторів (захищених модулів).

Неділимі змінні. Простіший спосіб вирішення завдання взаємного виключення в мові Ада95 ґрунтується на використанні неділимих змінних (атомік об'єктів). Їх описують через спеціальні прагми `Atomic` і `Volatile`, які контролюють використання спільних змінних. Є чотири форми цих прагм:

```
pragma Atomic(Ім'я_Змінної);
pragma Volatile(Ім'я_Змінної);
pragma Atomic_Components(Ім'я_Масиву);
pragma Volatile_Components(Ім'я_Масиву);
```

Операції читання або зміни атомік об'єктів розглядаються і виконуються як неподільні операції. Тобто, якщо один процес уже виконує дії з атомік-об'єктом, то інший може отримати доступ до

нього лише в разі звільнення об'єкта. Отже, всі дії процесів з атомік-об'єктами виконуються лише послідовно і ніколи – одночасно.

❖ Приклад 3.6

```
------------------------------------------------------------
-- Ада95.Pragma Atomic у завданні взаємного виключення  --
------------------------------------------------------------
procedure Lab36 is

   Буфер: integer:= 10;         -- спільний ресурс
   pragma Atomic(Буфер);        -- захищена змінна

   task A;
   task body A is
   begin
      put_line("Process A started");

      -- Операція зі спільним ресурсом
            Буфер := Буфер + 2;  -- критична ділянка

      put_line("Process A finished");
   end A;
   ------
   task B;
   task body B is
   begin
      put_line("     Process B started");

      -- Операція зі спільним ресурсом
         Буфер:= Буфер - 25;    -- критична ділянка

      put_line("     Process B finished");
   end B;

-- основна процедура
begin

   put_line("  Main procedure started ");

end Lab36;
```

Семафори. Задачі `A` і `B` виконують деякі дії над загальним ресурсом – цілою змінною `Буфер`. Ці операції мають виконуватись у взаємовиключному режимі, який можна забезпечити за допомогою семафорів.

Використовуючи загальну схему вирішення завдання взаємного виключення за допомогою семафорів (рис. 3.4) необхідно:

- створити змінну – семафор з початковим значенням 0 або `false`
- розмістити операції `P(S)` і `V(S)` навколо критичної ділянки в кожному процесі

У мові Ада ці дії виконуються за допомогою типу `Suspension_Object` і підпрограм `Suspend_Until_True()` та `Set_True()`.(див. 3.3.1).

У зв'язку з тим, що при створенні семафора (`Sem_Ku`) він автоматично набуває значення `false`, потрібноно змінити його на `true` перед запуском задач, щоб початковий вхід у критичну ділянку був відкритим для обох процесів. Явний запуск задач `A` і `B` виконується через виклик процедури `Старт_Задач`, де описані обидві задачі.

❖ Приклад 3.7

```
------------------------------------------------------------
--  Ада95.Семафори в завданні взаємного виключення      --
------------------------------------------------------------
with Ada.Synchronous_Task_Control,Text_IO,Integer_Text_IO;
use  Ada.Synchronous_Task_Control,Text_IO,Integer_Text_IO;
procedure Lab37 is

   Ресурс: integer:= 10;        -- спільний ресурс
   Sem_Ku: Suspension_Object;   -- семафор

   procedure Старт_Задач is

      task A;

      task body A is
      begin
         put_line("Process A started");

         -- Операція зі спільним ресурсом
         Suspend_Until_True(Sem_Ku);
            Буфер := Буфер + 2;  -- критична ділянка
```

```
         Set_True(Sem_Ku);

         put_line("Process A finished");
      end A;

      task B;

      task body B is
      begin
         put_line("     Process B started");

         -- Операція зі спільним ресурсом
         Suspend_Until_True(Sem_Ku);
            Буфер:= Буфер - 25;     -- критична ділянка
         Set_True(Sem_Ku);

         put_line("     Process B finished");
      end B;

   begin

      null;
   end Старт_Задач;

-- основна процедура
begin

   put_line("  Main procedure started ");

   Set_True(Sem_Ku); -- начальне значення семафора  (true)

   Старт_Задач;      -- запуск задач A і B

end Lab37;
```

Захищені модулі. Розглянемо використання захищеного модуля для вирішення завдання синхронізації процесів, що було розглянуто в попередньому прикладі.

Структуру захищеного модуля `Захист` показано на рис. 3.7. Модуль включає захищений елемент – змінну `Буфер`, що буде описана в приватній частині специфікації захищеного модуля. Для реалізації взаємовиключних дій над спільним ресурсом `Буфер` наведено дві процедури `Операція_A()` і `Операція_B()`.

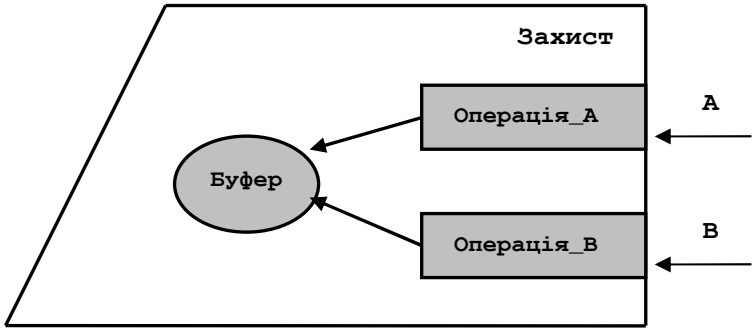

Рис. 3.7. Захищений модуль в завданні взаємного виключення

❖ Приклад 3.8

```
-------------------------------------------------------------
-- Ада95.Захищений модуль в                                 --
-- завданні взаємного виключення                           --
-------------------------------------------------------------
procedure Lab38 is
   -- захищений модуль
   protected Захист is
      procedure Операція_A;
      procedute Операція_B;
   private
      Буфер : integer:= 10;      -- спільний ресурс
   end Захист;

-- тіло захищеного модуля
   protected  body Захист is

      procedure Операція_A is
      begin
         Буфер := Буфер + 2;
      end Операція_A;

      procedure Операція_B is
      begin
         Буфер := Буфер - 25;
      end Операція_B;
```

```
   end Захист;

   -- використання захищеного модуля в задачах
   task A;

   task body A is
   begin
         .  .  .
      -- дії над спільним ресурсом
      Захист.Операція_A;
       .  .  .
   end A;
   ------
   task B;

   task B is
   begin
       .  .  .
      -- дії над спільним ресурсом
      Захист.Операція_B;
       .  .  .
   end B;

-- головна процедура
begin

   null;

end Lab38;
```

3.4.2. Вирішення завдання взаємного виключення в Win32

Бібліотека Win32 містить декілька засобів, які можна використати для вирішення завдання взаємного виключення. Це реалізація механізму семафорів, мютекси та критичні секції (табл. 3.1). У наступних підрозділах розглядаються приклади вирішення завдвння взаємного виключення для трьох задач `T1`, `T2` і `T3` за допомогою засобів Win32. Кожна задача бере участь у формуванні спільної змінної `Ресурс` через додавання значення. Операція над змінною `Ресурс` розглядається в кожному процесі як критична ділянка, яку треба захищати від одночасного входження в неї процесів за допомогою "створювання огорожі" з належних операцій Win32, які виконують функції примітивів `ВХІДКД` і `ВИХІДКД`.

Увага! Під час використання в наступних прикладах функцій Win32 для параметрів, які прямо не стосуються керування процесами, умовно встановлено значення NULL.

Таблиця 3.1. Засоби взаємного виключення Win32

Об'єкт		Опис
Семафор	Semaphore	Містить лічильник від нуля до визначеного значення, які визначають кількість потоків, що мають доступ до спільного ресурсу
Мютекс	Mutex	Цим об'єктом може володіти тільки один процес у поточний момент часу, що дозволяє цьому процесу ексклюзивний доступ до спільного ресурсу
Критична секція	Critical Section	Містить критичну ділянку, вхід у яку дозволяється тільки одному процесу
Функції очікування	WaitForSingleObject, WaitForMultipleObject	Перевіряють умову, залежно від якої процес може увійти в критичну ділянку. Якщо це неможливо, то процес блокується і чекає на її звільнення, після чого входить в критичну ділянку

Семафори. У разі використанні семафорів роль примітивів `ВХІДКД` і `ВИХІДКД` виконують функції `WaitForSingleObject()` та `ReleaseSemaphore()`.

❖ **Приклад 3.9**

```
------------------------------------------------------------
--|Ада95.Використання семафорів бібліотеки  Win32       --
--|в завданні взаємного виключення                       --
------------------------------------------------------------
With Win32, Win32.winbase, Win32.winnt;
Use  Win32, Win32.winbase, Win32.winnt;
procedure Lab39 is

   Ресурс: integer;  -- спільний ресурс
                     -- (глобальна змінна)
```

```
-- HANDLE  змінна для створювання семафора
Sem1: HANDLE;

-- допоміжні змінні
Temp: DWORD;
p1: Boolean;

procedure Запуск_Задач is

   task T1;
   task T2;
   task T3;

   task body T1 is
      X1: integer;     -- локальна змінна
   begin
      put_line("Process T1 started");
          .  .  .
      --  Критична ділянка
      Temp:= WaitForSingleObject(Sem1,Infinite);
         Ресурс := Ресурс + X1;
      p1:= ReleaseSemaphore(Sem1,1,NULL);
          .  .  .
      put_line("Process T1 finished");
   end T1;
   ---------------
   task body T2 is
      X2: integer;     -- локальна змінна
   begin
      put_line("Process T2 started");
          .  .  .
      --  Критична ділянка
      Temp:= WaitForSingleObject(Sem1,Infinite);
         Ресурс := Ресурс + X2;
      p1:= ReleaseSemaphore(Sem1,1, NULL);
          .  .  .
      put_line("Process T2 finished");
   end T2;
   -------------
   task body T3 is
      X3: integer;     -- локальна змінна
   begin
      put_line("Process T3 started");
          .  .  .
      --  Критична ділянка
      Temp:= WaitForSingleObject(Sem1,Infinite);
```



```
         Ресурс := Ресурс + X3;
      p1:= ReleaseSemaphore(Sem1,1,NULL);
          .  .  .
      put_line("Process T3 finished");
   end T3;
begin
   null;
end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");

   --  створення семафора з початковим значенням 1
   Sem1:=  CreateSemaphore(NULL,0,1,NULL);

      -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab39;
```

Мютекси. Мютекс (від *mutual exclusion* – взаємне виключення) – засіб, схожий на семафор, але на відміну від семафора він не має значення [45]. Мютекс належить тільки одному процесу і може передаватися від одного процесу до іншого (бути захопленим процесом). Вхід в критичну ділянку дозволяється тільки тому процесу, який теперь володіє мютексом. Після виходу з критичної ділянки процес звільнює мютекс, який захвачується іншим процесом.

Створення мютексу виконується зі змінною типу `HANDLE` за допомогою операцій `CreateMutex()`, `OpenMutex()`, `InitMutex()`.

У разі використанні мютексів роль примітивів `ВХІДКД` і `ВИХІДКД` можуть виконувати функції `WaitForSingleObject()` та `ReleaseMutex()`.

❖ Приклад 3.10

```
------------------------------------------------------------
--|Ада95.Використання мютексів у бібліотеці  Win32      --
--|в завдвнні взаємного виключення                       --
------------------------------------------------------------
procedure Lab310 is

   Ресурс: integer;  -- спільний ресурс (глобальна змінна)
```

```
-- HANDLE змінна для створення мютексу
Mute: HANDLE;

-- допоміжні змінні
Temp: DWORD;
p1: Boolean;

procedure Запуск_Задач is

   task T1;
   task T2;
   task T3;

   task body T1 is
      X1: integer;      -- локальна змінна
   begin
      put_line("Process T1 started");
          .  .  .
      --  Критична  ділянка
      Temp:= WaitForSingleObject(Mute,Infinite);
         Ресурс := Ресурс + X1;
      p1:= ReleaseMutex(Mute);
          .  .  .
      put_line("Process T1 finished");
   end T1;
   ---------------
   task body T2 is
      X2: integer;      -- локальна  змінна
   begin
      put_line("Process T2 started");
          .  .  .
      --  Критична ділянка
      Temp:= WaitForSingleObject(Mute,Infinite);
         Ресурс := Ресурс + X2;
      p1:= ReleaseMutex(Mute);
          .  .  .
      put_line("Process T2 finished");
   end T2;
   -------------
   task body T3 is
      X3: integer;      -- локальна  змінна
   begin
      put_line("Process T3 started");
          .  .  .
      --  Критична ділянка
      Temp:= WaitForSingleObject(Mute,Infinite);
         Ресурс := Ресурс + X3;
```



```
         p1:= ReleaseMutex(Mute);
             .  .  .
         put_line("Process T3 finished");
      end T3;
   begin

      null;
   end Запуск_Задач;

begin -- основна процедура

   put_line("  Main procedure started ");

   --  створення мютекса
   Mute:= CreateMutex(NULL,0,NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;
end Lab310;
```

Критичні секції. Механізм *критичних секцій* – це засіб, який об'єднує конструкції ВХІДКД і ВИХІДКД в *єдиний* оператор і позбавляє від помилок, які властиві семафорам (мютексам), коли на вході в критичну ділянку перевіряється один семафор, а на виході змінюється інший семафор і це ніяк не контролюється. На жаль, реалізація механізму критичних секцій у Win32 виконано подібно до механізмів семафорів і мютексів у вигляді двох примітивів замість одного. Тобто цей механізм є різновидом механізму семафорів з тією відмінністю, що замість семафора в ньому використовується ім'я критичної ділянки, вхід у яку контролюється і дозволяється тільки одному процесу.

Створення критичної секції виконується зі змінною типу `CRITICALSECTION` за допомогою операцій `InitializeCriticalSection()`, знищення – за допомогою операції `DeleteCriticalSection()`.

У разі використання критичних ділянок роль примітивів ВХІДКД і ВИХІДКД виконують функції `EnterCriticalSection()` та `LeaveCriticalSection()`. На відміну від семафорів і мютексів у механізмі критичних ділянок час очікування в операції входження в критичну ділянку `EnterCriticalSection()` не встановлюється, тобто він завжди є `INFINITE`.

❖ **Приклад 3.11**

```
------------------------------------------------------------
--|Ада95.Використання критичних секцій бібліотеки Win32  --
--|у задачі взаємного виключення                          --
------------------------------------------------------------
procedure Lab311 is

   Ресурс: integer;  -- спільний ресурс
                     -- (глобальна змінна)

   -- створювання критичної секції
   CrCec: CRITICAL_SECTION;

   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
         X1: integer;          -- локальна змінна
      begin
         put_line("Process T1 started");
             .  .  .
         --  Критична ділянка
         EnterCriticalSection(CrSec);
            Ресурс := Ресурс + X1;
         LeaveCriticalSEction(CrSec);
             .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
         X2: integer;            -- локальна змінна
      begin
         put_line("Process T2 started");
             .  .  .
         --  Критична ділянка
         EnterCriticalSection(CrSec);
            Ресурс := Ресурс + X2;
         LeaveCriticalSEction(CrSec);
             .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
```



```
      task body T3 is
         X3: integer;            -- локальна змінна
      begin
         put_line("Process T3 started");
             .  .  .
         --  Критична ділянка
         EnterCriticalSection(CrSec);
            Ресурс := Ресурс + X3;
         LeaveCriticalSEction(CrSec);
             .  .  .
         put_line("Process T3 finished");
      end T3;
   begin

      null;

   end Запуск_Задач;

begin     -- основна процедура

   put_line("  Main procedure started ");

   --  створення критичної секції
   InitializeCriticalSection(CrSec);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab311;
```

3.4.3. Вирішення завдання взаємного виключення в мові Java

Вирішення завдання взаємного виключення в Java програмах виконується за допомогою синхронізованих методів і блоків.

Синхронізовані методи. Синхронізованим є метод, що має модифікатор `synchronized`. Якщо потік виконує синхронізований метод (знаходиться всередині метода), то всі інші потоки, які намагаються викликати будь-який синхронізований метод цього ж екземпляра класу, мають чекати, поки завершиться його виконання. Тобто синхронізований метод має властивості процедури монітора.

Приклад 3.12 ілюструє реалізацію Java монітора, який містить спільний ресурс – змінну Буфер, і два методи Читати() та Писати() для доступу до спільного ресурсу.

❖ Приклад 3.12

```
/*-------------------------------------------------------
-- Java. Реалізація монітора для взаємного виключення   --
-------------------------------------------------------*/
   class Монітор{

       private int Буфер; -- спільний ресурс

       synchronized int Читати(){
          return Буфер;
       }
       synchronized void Писати(int x){
          Буфер = x;
       }
   }// Монітор
```

Одночасне виконання методів Читати() та Писати() одного екземпляра класу Монітор неможливо. Якщо процес (потік) B викликав і виконує, наприклад, з екземпляра Блок класу Монітор метод Блок.Читати(), то інший потік A буде блокований, якщо він спробує викликати будь-який метод класу Монітор, наприклад, Блок.Писати(), і зможе продовжити своє виконання тільки після завершення виконання методу Блок.Читати() у потоці B.

Синхронізовані блоки. Загальна форма оператора synchronized:

```
synchronized(Об'єкт) {
     //  оператори
},
```

де Об'єкт – це посилання на об'єкт, доступ до якого треба синхронізувати.

Синхронізований блок гарантує , що виклик з потоків методу, що є членом об'єкта Об'єкт, буде виконуватися в режимі взаємного виключення.

❖ Приклад 3.13

Застосування синхронізованого блоку для взаємовиключного виклику методу Зміна_Значення з класу Робота в потоках – екземплярах класу Доступ.

```
/*-------------------------------------------------------
-- Java. Застосування синхронізованого блоку            --
-------------------------------------------------------*/
    class Робота{
       double Зміна_Значення(double e){
          return e*e;
       }
    }// Робота

    class  Доступ  implements Thread{

       Робота Блок1;   // посилання на клас Робота

       // конструктор
       Доступ(Блок  o){
          Блок1 = o;
       }

       public void run(){
          double x = 5;
          double y;

          synchronized(Блок1) {
            y = Блок1.Зміна_Значення(x);
       }
    }// Доступ

    class Основний{

       public static void main(String [] args){

          Робота Рб  = new Робота();
          Доступ  Д1 = new Доступ(Рб);
          Доступ  Д2 = new Доступ(Рб);

          Д1.start();
          Д2.start();

       }// main
    }// Основний
```

3.5. Вирішення завдання синхронізації

3.5.1. Вирішення завдання синхронізації в мові Ада

Для вирішення завдання синхронізації в мові Ада95 можна застосувати механізм семафорів, а також захищені модулі.

Застосування семафорів. Розглянемо приклад, у якому задача A чекає на подію, яку має містити задачі B, наприклад, введення даних (змінної x). Для синхронізації процесів з уведення використовуємо семафор СігналПроПодію з початковим значенням false, яке встановлюється автоматично під час створювання семафора. Очікування події в процесі A реалізовано за допомогою операції Suspend_Until_True(СігналПроПодію), а посилання сигналу про подію в задачі B реалізовано за допомогою операції Set_True(СігналПроПодію).

❖ **Приклад 3.14**

```
------------------------------------------------------------
--Ада95.Синхронізація задач                                --
------------------------------------------------------------
procedure Синхронізація is

   X : integer;  --  глобальна  змінна

   СігналПроПодію: Suspension_Object;        -- семафор

    -- задача, що чекає на подію
    task A;
    task body A is
    begin
           .  .  .
      -- точка очікування  події
      Suspend_Until_True(СігналПроПодію);
         .  .  .
    end A;
    -- задача, де  пройде  подія
    task B;
    task body B is
    begin
         .  .  .
      get(X); -- уведення даних(подія, на
              -- яку чекає A)
```



```
      Set_True(СігналПроПодію);     -- сигнал задачі A
           .  .  .
    end B;
begin
    null;
end Синхронізація;
```

Застосування захищеного модуля. Розглянемо застосування захищеного модуля для вирішення завдання синхронізації задач A і B, що поданої у попередньому прикладі. Синхронізація задач реалізується за допомогою захищеного модуля Контроль, у якому визначені дві захищені операції: вхід Чекати_Введення() і процедура СигналПроВведення(), а також допоміжна захищена змінна (прапор) Fl, яка буде використовуватись у бар'єрі захищеного входу. Структуру захищеного модуля Контроль подано на рис. 3.8.

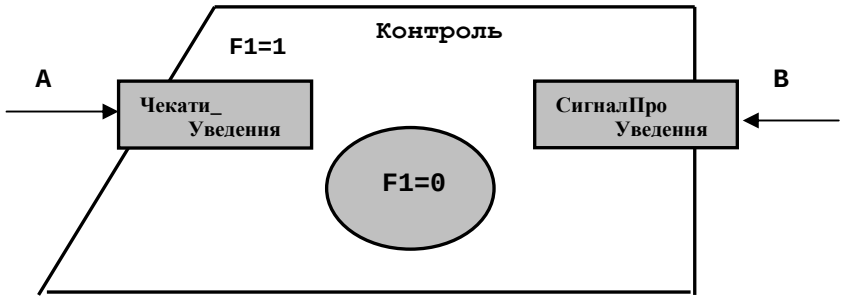

Рис. 3.8. Захищений модуль в завданні синхронізації

Очікування на подію (введення X) у задачі A виконано як виклик входу Чекати_Уведення(), а посилання сигналу про подію в задачі B – як виклик процедури СигналПроУведення(), яка змінює значення прапору Fl на 1. Вхід Чекати_Уведення() має бар'єр, який закритий, якщо значення прапора F1=0, і відкритий, якщо F1=1. Таким чином, початкове значення прапора Fl=0 дозволяє блокувати процес A в операції виклику входу Чекати_Уведення() до того часу, поки процес B не змінить значення

бар'єра за допомогою операції СигналПроУведення() після того, як подія буде мати місце.

❖ **Приклад 3.15**

```
------------------------------------------------------------
--  Ада95.Захищений модуль                                --
------------------------------------------------------------
-- захищений модуль
  protected  Контроль is

     entry  Чекати_Уведення;
     procedure СигналПроУведення;
  private

     Fl : integer:= 0;

  end Контроль;

  -- тіло захищеного модуля
  protected  body Контроль is

     entry  Чекати_Уведення  when  Fl=1 is
     begin
        null;
     end Чекати_Уведення;

     procedure СигналПроУведення is
     begin
        Fl := 1;
     end СигналПроУведення;

  end Контроль;

  -- використання захищеного модуля в задачах
  task A;
  task B;

  task body A is
  begin
        .  .  .
     -- точка синхронізації(очікування уведення)
     Контроль.Чекати_Уведення;
        .  .  .
  end A;

  task body  B is
  begin
```



```
        .  .  .
     -- сигнал про подію
     Контроль.СигналПроУведення;
        .  .  .
  end B;
```

Для синхронізації задачі A з кількома задачами (B,C, D), де виконується введення, необхідна модифікація захищеного модуля Контроль, яка пов'язана з тим, що бар'єр входу Чекати_Уведеня буде мати вигляд when Fl = 3, а процедура СигналПроУведення буде змінювати прапор Fl:= Fl + 1.Тепер задача A під час виклику входу Чекати_Уведення продовжить своє виконання тільки після того, як кожна задача B,C і D викличе процедуру СигналПроУведення і прапор Fl набуде значення 3.

3.5.2 Вирішення завдання синхронізації процесів у Win32.

Завдяня синхронізації в Win32 можна вирішити за допомогою семафорів, подій або таймерів. При цьому використовуються функції очікування (табл. 3.2). Докладний опис функцій бібліотеки Win32 наведено в дод. А.

Таблиця 3.2. Засоби синхронізації у Win32

Об'єкт		Опис
Подія	Event	Повідомляє одному або кільком потокам, що чекають, про подію
Семафор	Semaphore	Містить лічильник від нуля до визначеного значення, який за значення нуль блокує процес, а зі зміною значення розблокує процес
Таймер	Timer	Визначає один або кілька потоків, що очікують настання визначеного часу
Функції очікування	WaitForSingleObject, WaitForMultipleObject	Перевіряють умову залежно від якої процес блокується і чекає на подію. Якщо подія вже відбулася, то процес не блокується

Застосування семафорів. У разі використання семафорів очікування події виконується за допомогою функції `WaitForSingleObject()`, а сигнал про подію, що відбулася, – через функцію `ReleaseSemaphore()`.

У прикладі 3.16 задача `T1` інформує задачу `T2` про подію, що відбулася в задачі `T1` (`Подія1`), а задача `T3` чекає на подію, що відбудеться в задачі `T2` (`Подія2`).

❖ Приклад 3.16

```
------------------------------------------------------------
--|Ада95.Використання семафорів бібліотеки  Win32       --
--|в завданні синхронізації                              --
------------------------------------------------------------
procedure Lab316 is

   -- HANDLE  змінні для створювання семафорів
   Sem12, Sem23: HANDLE;

   -- допоміжні змінні
   Temp: DWORD;
   p1, p2: Boolean;

   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
             .  .  .
         -- Подія1
         --  Сигнал задачі T2 про подію (Подія1)
         --  у задачі T1
         p1:= ReleaseSemaphore(Sem12,1,NULL);
             .  .  .
         put_line("Process T1 finished");
      end T1;
      --------------
      task body T2 is
      begin
         put_line("Process T2 started");
             .  .  .
         --  очікування на подію (Подія1) в T1
         Temp:= WaitForSingleObject(Sem12,Infinite);
                   .  .  .
         -- Подія2
         -- Сигнал задачі T3 про подію (Подія2) в задачі T2
         p1:= ReleaseSemaphore(Sem23,1,NULL);
             .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
             .  .  .
         --  Очікування на подію (Подія2) в T2
         Temp:= WaitForSingleObject(Sem23,Infinite);
             .  .  .
         put_line("Process T3 finished");
      end T3;
   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");

   --  створення семафорів з початковим значенням 0
   Sem12:=  CreateSemaphore(NULL, 0,1,NULL);
   Sem23:=  CreateSemaphore(NULL, 0,1,NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;
end Lab316;
```


У прикладі 3.17 задача `T3` чекає на події, що відбудуться в задачі `T1` (`Подія1`) і задачі `T2` (`Подія2`).

❖ Приклад 3.17

```
------------------------------------------------------------
--|Ада95.Використання семафорів бібліотеки  Win32       --
--|у завданні синхронізації                              --
------------------------------------------------------------
procedure Lab317 is
   -- HANDLE  змінні для створювання семафорів
   Sem1_3, Sem2_3: HANDLE;
```

```
    -- допоміжні змінні
   Temp: DWORD;
   p1, p2: Boolean;
   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
              .  .  .
         -- Подія1
         -- Сигнал задачі T3 про подію (Подія1) в задачі T1
         p1:= ReleaseSemaphore(Sem1_3,1,NULL);
              .  .  .
         put_line("Process T1 finished");
      end T1;
      --------------
      task body T2 is
      begin
         put_line("Process T2 started");
              .  .  .
         -- Подія2
         -- Сигнал задачі T3 про подію (Подія2) в задачі T2
         p1:= ReleaseSemaphore(Sem2_3,1,NULL);
              .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію (Подія1) в T1
         Temp:= WaitForSingleObject(Sem1_3,Infinite);
         --  Очікування на подію (Подія2) в T2
         Temp:= WaitForSingleObject(Sem2_3,Infinite);
              .  .  .
         put_line("Process T3 finished");
      end T3;
   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
```



```
   put_line("  Main procedure started ");
   --  створення семафорів з початковими значеннями 0
   Sem1_3:=  CreateSemaphore(NULL, 0, 1,NULL);
   Sem2_3:=  CreateSemaphore(NULL, 0, 1,NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab317;
```

У прикладі 3.18 задача T1 інформує задачі T2 і T3 про подію, що відбулася в задачі T1. Це можливо виконати через два бінарні семафори, але в прикладі це виконано за допомогою одного множинного семафора Sem1_23, який набуває значення 0 .. 2.

❖ Приклад 3.18

```
-------------------------------------------------------------
--|Ада95.Використання семафорів бібліотеки  Win32          --
--|в завданні синхронізації                                --
-------------------------------------------------------------
procedure Lab318 is

   -- HANDLE  змінна для створювання богатозначного семафора
   Sem1_23: HANDLE;

   -- допоміжні змінні
   Temp: DWORD;
   p1, p2: Boolean;

   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
              .  .  .
         -- Подія
         --  Сигнали задачам T2 і T3 про подію в задачі T1
         p1:= ReleaseSemaphore(Sem1_23,2,NULL);
              .  .  .
         put_line("Process T1 finished");
      end T1;
```

```
      --------------
      task body T2 is
      begin
         put_line("Process T2 started");
              .  .  .
         --  Очікування на подію в T1
         Temp:= WaitForSingleObject(Sem1_23,Infinite);
              .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію в T1
         Temp:= WaitForSingleObject(Sem1_23,Infinite);
              .  .  .
         put_line("Process T3 finished");
      end T3;
   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");
   --  створення семафора зі значеннями (0,1,2)  і
   --  початковим значенням 0
   Sem1_23:=  CreateSemaphore(NULL, 0, 2, NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab318;
```

Застосування подій. Механізм подій призначений виключно для синхронізації процесів.

Створення події виконується зі змінною типу HANDLE за допомогою функції CreateEvent():

```
   HANDLE CreateEvent(
      LPSECURITY_ATTRIBUTES    адреса_атрибутів_захисту
      BOOL     прапор_ручної_установки_події
      BOOL     прапор_початкового_стану
      LPCTSTR  адреса_об'єкта_події
   );
```


У разі використання механізму подій очікування події виконується за допомогою однієї із функцій очікування, наприклад, WaitForSingleObject(), а сигнал про подію, що відбулася, – функцією SetEvent():

```
BOOL SetEvent((
   HANDLE  Подія       // об'єкт - подія
);
```

Для роботи з подіями використовують також функції:

- OpenEvent() – повертає значення існуючого об'єкта-події;
- PulseEvent() – забезпечує установку стану події в сигнальне і наступне перемикання його в несигнальне після реалізації посилання сигналу очікуваним потокам;
- ResetEvent() – виконує скидання події (установлює стан події в несигнальний).

У прикладах 3.19 – 3.21 наведено вирішення за допомогою механізму повідомлень завдання синхронізації, які розглядалися в прикладах 3.16 – 3.18.

У прикладі 3.19 задача T1 інформує задачу T2 про подію, що відбулася в задачі T1 (Подія1), а задача T3 чекає на подію, що відбудеться в задачі T2 (Подія2).

❖ Приклад 3.19

```
------------------------------------------------------------
--|Ада95.Використання подій бібліотеки  Win32           --
--|в завданні синхронізації                             --
------------------------------------------------------------
procedure Lab319 is

   -- HANDLE  змінна для створювання  подій
   Evn12, Evn23: HANDLE;

   -- допоміжні змінні
   Temp  : DWORD;
   p1, p2: BOOL;

   procedure Запуск_Задач is
```

```
      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
              .  .  .
         -- Подія1
         -- Сигнал задачі T2 про подію (Подія1)в задачі T1
         p1:= SetEvent(Evn12);
              .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
      begin
         put_line("Process T2 started");
              .  .  .
         --  очікування на подію (Подію1) в T1
         Temp:= WaitForSingleObject(Evn12,Infinite);
                    .  .  .
         -- Подія2
         -- Сигнал задачі T3 про подію (Подія2)
         p1:= SetEvent(Evn23);
              .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію (Подія2) в T2
         Temp:= WaitForSingleObject(Evn23,Infinite);
              .  .  .
         put_line("Process T3 finished");
      end T3;
   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");

   --  створення подій
```



```
   Evn12:=  CreateEvent(NULL, 0,0, NULL);
   Evn23:=  CreateEvent(NULL, 0,0, NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab319;
```

У прикладі 3.20 задача T3 чекає на події, що відбудуться в задачі T1 (Подія1) і задачі T2 (Подія2). Для сигналізації про ці події застосовані події Evn1_3 та Evn2_3.

❖ Приклад 3.20

```
-------------------------------------------------------------
--|Ада95.Використання подій бібліотеки  Win32              --
--|в завданні синхронізації                                --
-------------------------------------------------------------
procedure Lab320 is

   -- HANDLE  змінна для створювання  подій
  Evn1_3, Evn2_3: HANDLE;

   -- допоміжні змінні
   Temp: DWORD;
   p1, p2: BOOL;

   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
              .  .  .
         -- Подія1
         --  Сигнал задачі T3 про подію (Подія1)
         p1:= SetEvent(Evn1_3);
              .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
      begin
```

```
         put_line("Process T2 started");
             .  .  .
         -- Подія2
         -- Сигнал задачі T3 про подію (Подія2) в задачі T2
         p1:= SetEvent(Evn2_3);
             .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію (Подія1) в T1
         Temp:= WaitForSingleObject(Evn1_3,Infinite);
         --  Очікування на подію (Подія2) в T2
         Temp:= WaitForSingleObject(Evn2_3,Infinite);
              .  .  .
         put_line("Process T3 finished");
      end T3;
   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");
   --  створення подій
   Evn1_3:=  CreateEvent(NULL, 0, 0, NULL);
   Evn2_3:=  CreateEvent(NULL, 0, 0, NULL);
   -- виклик процедури для запуску задач
   Запуск_Задач;
end Lab320;
```

У прикладі 3.21 задача T1 інформує задачі T2 і T3 про подію, що відбулася в задачі T1. Це можливо виконати через одну подію Evn1_23.

❖ Приклад 3.21

```
-------------------------------------------------------
--|Ада95.Використання подій бібліотеки  Win32        --
--|в завданні синхронізації                          --
-------------------------------------------------------
procedure Lab321 is

     -- HANDLE  змінна для створювання події
```



```
   Evn1_23: HANDLE;

   -- допоміжні змінні
   Temp: DWORD;
   p1: BOOL;
   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
             .  .  .
         --  Подія
         --  Сигнал задачам T2 і T3 про подію в задачі T1
         p1:= SetEvent(Evn1_23);
             .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
      begin
         put_line("Process T2 started");
             .  .  .
         --  Очікування на подію в T1
         Temp:= WaitForSingleObject(Evn1_23,Infinite);
             .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію в T1
         Temp:= WaitForSingleObject(Evn1_23,Infinite);
              .  .  .
         put_line("Process T3 finished");
      end T3;

   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
```

```
   put_line("  Main procedure started ");

   --  створення події
  Evn1_23:=  CreateEvent(NULL, 1, 0, NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab321;
```

3.5.3. Вирішення завдання синхронізації процесів в мові Java

Синхронізація процесів в мові Java виконується за допомогою методів `wait()`, `notify()` або `notifyAll()`. У першій версії мови з цією метою також були використані методи `suspend()/resume()`, але згодом від них відмовились, оскілки вони спричинили побічні ефекти.

Методи `wait()`, `notify()` та `notifyAll()` об'явлені в класі `Object`, доступні всім класам і мають форму:

```
final void  wait()  throws  InterruptedException
final void  notify()
final void  notifyAll()
```

Виконання методу `A.wait()` у потоці зумовілює блокування потоку, поки інший процес не виконає метод `A.notify()` або `A.notifyAll()`.

Метод `notify()` розблокує потік, який блоковано методом `wait()`. На жаль, мова не визначає, який потік буде розблоковано, якщо потоків декілька. Метод `notifyAll()` розблокує *усі* потоки, які блоковані методом `wait()` в одному об'єкті. Якщо на момент виконання методів `notify()` і `notifyAll()` не існує блокованих потоків, то методи ігноруються. Тому, якщо треба методом `notify()` обов'язково повідомити процес про подію, яка відбулася до того, як він буде її чекати, необхідно застосувати додаткову загальну змінну, значення якої треба перевіряти поряд з використанням методу `wait()`.

Звернімо увагу, що методи `wait()`, `notify()` і `notifyAll()` взаємодіють тільки тоді, коли вони прив'язані до

одного і того ж об'єкта. Крім того, всі три методи можуть бути викликані тільки всередині синхронізованого методу.

❖ **Приклад 3.22**

Клас `Synchro` містить два синхронізовані методи `ЧекатиНаПодію()` та `СигналПроПодію()`, за допомогою яких виконується синхронізація потоків: у потоці `Потік1` відбувається подія, на котру чекає потік `Потік2`.

```
/* ---------------------------------------------------
-- Java. Синхронізація  двох потоків                --
---------------------------------------------------*/
class Synchro {

   private int Прапор;    // додаткова змінна

   synchronized void ЧекатиНаПодію(){

      if(Прапор = = 0){
        try{
           wait();        // очікування події

        }catch(InterruptedException){
            System.out.println("Потік П1 завершений");
        }
      }
   }

   synchronized void СигналПроПодію(){
      Прапор ++;
      notify();           // розблокування
   }

}//Synchro

class Потік1 extends  Thread {

   Synchro O1;           // Посилання на клас Synchro
   // конструктор
   Потік1(Synchro s){
      O1 = s;
   }
   public  void run(){
         . . .
```

```
      // подія
         . . .
      // сигнал про подію для Потоку2
      O1.СигналПроПодію();
         . . .
      System.out.println("Потік П1 завершений");
    }// run
   }// Потік1

class Потік2 extends  Thread {

   Synchro O2;        // Посилання на клас  Synchro

   // конструктор
   Потік2(Synchro s){
      O1 = s;
   }

   public void run(){
         . . .

      // очікування сигналу про подію від Потоку 1
      O2.ЧекатиНаПодію();

         . . .
      System.out.println("Потік П2 завершений");
   }// run

}// Потік2

class СинхронізаціяПотоків {
   public static void main(String args []){

      // екземпляр монітора
      Synchro Снхр = new Synchro();

      // екземпляри  потоків
      Потік1 П1 = new Потік1(Снхр);
      Потік2 П2 = new Потік2(Снхр);

      // старт потоків
      П1.start();
      П2.start();
   }// main
}// Синхронізація потоків
```


- **Запитання для модульного контролю**
  1. Назвіть види взаємодії процесів ?
  2. Що включає постановка та загальна схема вирішення завдання взаємного виключення ?
  3. Наведіть алгоритми примітивів ВХІДКД і ВИХІДКД ?
  4. У чому полягає головна відмінність двох підходів до рішення завдання взаємного виключення ?
  5. Яке призначення мають багатозначні семафори ?
  6. Як застосують семафори в завданні взаємного виключення і в завданні синхронізації процесів ?
  7. Які процедури в Ада семафорах виконують роль операції `P()` і `V()` ?
  8. Яке призначення типу `HANDLE` в бібліотеці Win32 ?
  9. Як реалізовано механізм критичних секцій в Win32 ?
  10. В чому полягає різниця між семафорами та мютексами в Win32 ?
  11. Яку структуру має захищений модуль ? Види та призначення захищених операцій ?
  12. Яку роль виконує бар'єр у захищеному модулі ?
  13. Як виконується синхронізація за допомогою захищеного модуля ?
  14. Яка особливість використання захищених функцій ?
  15. Які черги формуються при використанні захищеного модуля ?
  16. В чому полягає відмінність в реалізації концепції монітора в мовах Java і Ада95 ?

- **Завдання для самостійної роботи**
  1. Написати програму мовою Java, в якій вирішується завдання взаємного виключення для трьох потоків, що мають доступ до спільної змінної `Ресурс`. Використати:
     а) синхронізовані методи,
     б) синхронізовані блоки.
  2. Написати програму на мові Ада95, в якій вирішується завдання взаємного виключення для чотирьох потоків, які мають доступ до спільної змінної `Ресурс`.

Використати:

а) семафори з пакета `Synchronous_Task_Control`,

б) прагми,

в) засоби бібліотеки Win32.

3. Вирішити за допомогою семафорів Win32 завдання синхронізації одного потоку з кількома іншими (потік чекає на події в трьох інших потоках).
4. Вирішити за допомогою засобів Win32 завдання синхронізації кількох потоків з одним (чотири потоки чекають на подію в одному потоці).
5. Завдання 3 реалізувати за допомогою захищеного модулю мови Ада95.
6. Завдання 4 реалізувати за допомогою захищеного модуля мови Ада95 .
7. Виконати моделювання механізму богатозначних семафорів за допомогою захищеного модуля мови Ада95.
8. Завдання 3 реалізувати за допомогою засобів мови Java.
9. Завдання 4 реалізувати за допомогою засобів мови Java.
10. Провести дослідження часу виконання в ПКС одної і тої ж паралельної програми, де задача взаємного виключення вирішується за допомогою різних засобів синхронізації. Визначити також час виконання цієї програми у випадку, коли задача взаємного виключення взагалі не вирішується.