7.3 ВЗАЄМНІ ВИКЛЮЧЕННЯ З ВИКОРИСТАННЯМ СИСТЕМНИХ ФУНКЦІЙ

Для усунення недоліків, які були описані в попередніх алгоритмах, у багатьох ОС передбачаються спеціальні системні виклики для роботи з критичними секціями. На рис. 7.5 показано, як за допомогою таких функцій реалізовано взаємне виключення в операційній системі Windows 2000.

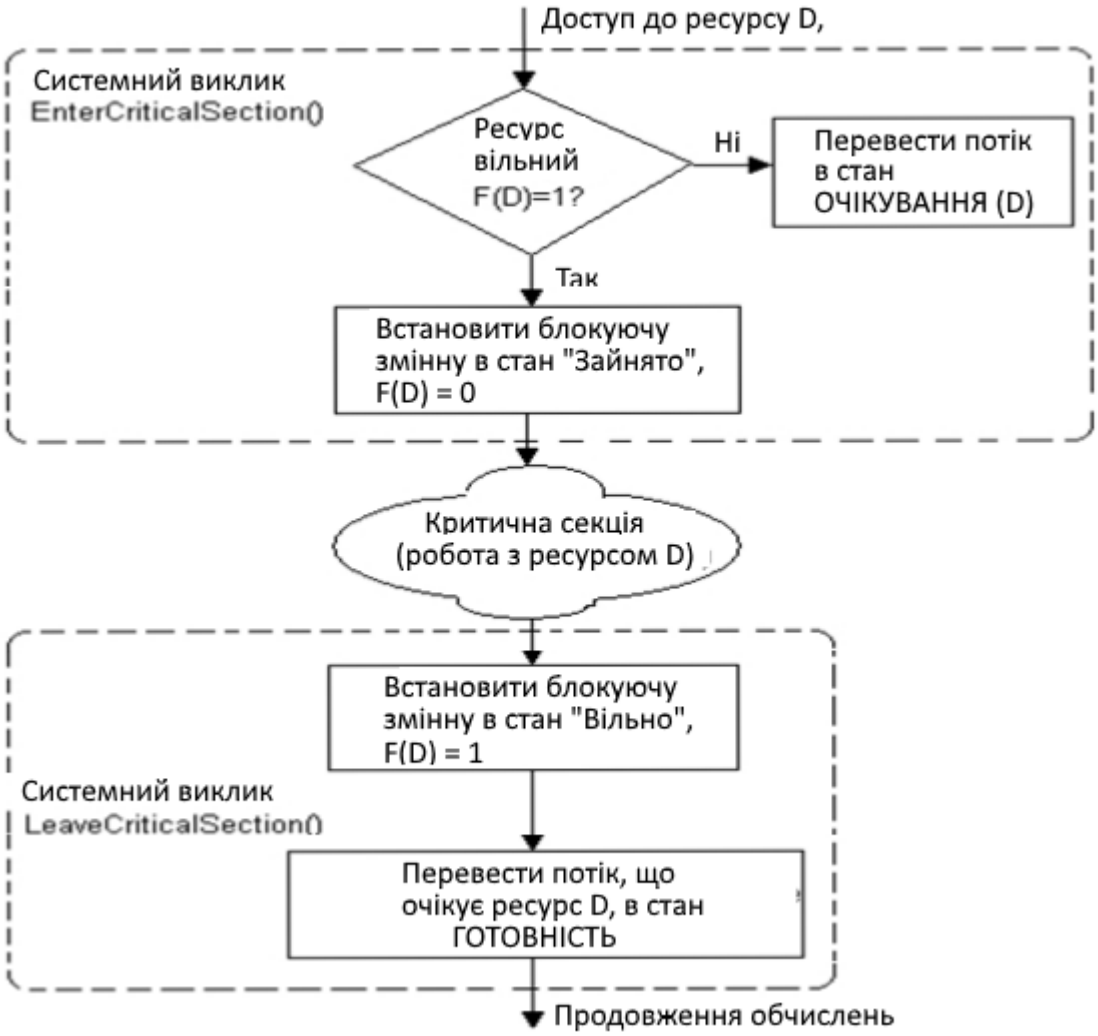

Рисунок 7.5 – Реалізація взаємного виключення з використанням системних функцій входу в критичну секцію і виходу з неї

Таким чином виключається непродуктивна втрата процесорного часу на циклічну перевірку звільнення зайнятого ресурсу.

7.3.1 Взаємовиключення: програмний підхід

Програмний підхід може бути реалізований для паралельних процесів, які виконуються як в однопроцесорній, так і в багатопроцесорній системі із загальною основною пам'яттю. Такі підходи припускають елементарні взаємовиключення на рівні доступу до пам'яті. Тобто, одночасний доступ (читання і/або запис) до одного і того ж елементу пам'яті упорядковується за допомогою деякого механізму. Ніякої іншої підтримки з боку апаратного забезпечення ОС або мови програмування не передбачається.

Алгоритм Деккера

Дейкстра представив алгоритм взаємних виключень для двох процесів, запропонований голландським математиком Деккером (1965 р.). Алгоритм Деккера є першим відомим точним рішенням взаємного виключення без заборони переривань. Назва алгоритму пов'язана з голландським математиком Теодором Деккером, який розв'язав цю проблему. Алгоритм дозволяє двом потокам спільно використати одноразовий ресурс без конфліктів, використовуючи для зв'язку лише загальну пам'ять (лістинг 7.3). Детально з алгоритмом можна познайомитися в роботі [12].

Лістинг 7.3 – Алгоритм Деккера для двох процесів

```
1    static int right = 0;
2    static char wish[2] = { 0,0 };
3    void csBegin ( int proc ) {
4    int competitor;
5    competitor = other ( proc );
6    while (1) {
7    wish[proc] = 1;
8    do {
9    if ( ! wish[competitor] ) return; 10        }
11   while ( right != competitor );
12   wish[proc] = 0;
13   while ( right == competitor ); 14  }
15 }
16   void csEnd ( int proc ) {
17   right = other ( proc );
18   wish[proc] = 0; 19 }
```

Алгоритм передбачає, по-перше, загальну змінну right для представлення номера процесу, який має переважне (але не абсолютне) право на вхід в критичну секцію. По-друге, масив wish, кожен елемент якого відповідає одному з процесів і представляє «бажання» процесу увійти до критичної секції. Процес заявляє про своє «бажання» увійти до секції (рядок 7). Якщо при цьому з'ясовується, що процес-конкурент не виставив свого «бажання» (рядок 9), то відбувається повернення з функції, тобто, процес входить в критичну секцію незалежно від того, кому належало переважне право на вхід.

Якщо ж в рядку 9 з'ясовується, що конкурент теж виставив «бажання», то перевіряється право на вхід (рядок 10). Якщо право належить нашому процесу, то повторюється перевірка «бажання» конкурента (рядки 8-10), поки воно не буде скасовано. Конкурент змушений буде відмінити своє «бажання», тому що він в цій ситуації перейде до рядка 11, де процес, що не має переважного права, повинен це зробити. Після відміни свого «бажання» процес чекає, поки переважне право не повернеться до нього (рядок 12), а потім знову повторює заяву «бажання» і так далі (рядки 6-13). Таким чином, процес у функції csBegin або повторює цикл 7-14, або виходить з функції і входить в критичну секцію (10). При виході з критичної секції (функція csEnd) процес передає переважне право входу конкурентові (рядок 16) і відмовляється від свого «бажання» (рядок 17).

Алгоритм Петерсона

Алгоритм Деккера розв'язує задачу взаємних виключень, але досить складно, і на додаток важко довести його коректність. У 1981 році Петерсон (Peterson) запропонував витонченіше і простіше вирішення проблеми взаємних виключень, яке перевело алгоритм Деккера в розряд застарілих.

Узагальнений алгоритм Петерсона [15] для двох процесів наведений на лістингу 7.4.

Листинг 7.4 – Алгоритм Петерсона для двох процесів

```
1    static int right;
2    static char wish[2] = { 0,0 };
3    void csBegin ( int proc ) {
4    int competitor;
```

```
5    if ( proc == 0 ) competitor = 1;
6    else competitor = 0;
7    wish[proc] = 1;
8    right = competitor;
9    while (wish[competitor] && (right == competitor); 10 }
11   void csEnd ( int proc ) {
12   wish[proc] = 0; 13 }
```

При вході в критичну секцію процес заявляє про своє «бажання» (рядок 7) і відмовляється від свого переважного права (рядок 8). Процес чекатиме, якщо його конкурент заявив своє «бажання» і має переважне право (рядок 9). Якщо немає інтересу конкурента або якщо незалежно від інтересу конкурента наш процес має переважне право, то процес входить в критичну секцію.

Якщо наш процес відмовився від свого права в рядку 8, то право нашого процесу може бути відновлене конкурентом, коли останній теж увійде до функції csBegin свого коду і виконає рядок 8. При виході з критичної секції процес просто знімає свій признак входу, і тоді його конкурент, очікуючий в рядку 8, одержує можливість виходу з циклу рядка 9 за першою частиною умови.

Узагальнення алгоритму Деккера для N процесів наведено в роботі [13].

Загальні позитивні властивості алгоритмів, що грунтуються на неальтернативних перемикачах (Деккера і Петерсона), такі:

- вони коректні як для одно-, так і для багатопроцесорних систем;
- вони ліберальні, оскільки дозволяють швидшим процесам входити у свої критичні секції частіше, ніж повільним;
- вони не обмежують кількість обслуговуваних ними процесів;
- вони дозволяють процесам скільки завгодно довго затримуватися поза своєю критичною секцією.

Але, запропоновані алгоритми, мають і недоліки:

- рішення непрості для розуміння і помилитися в їх реалізації дуже легко;
- процеси використовують зайняте очікування при вході в критичну секцію.

Алгоритм Лемпорта

Алгоритм Лемпорта (алгоритм булочної, Lamport's bakery algorithm) розв'язує задачу взаємного виключення для N процесів як для багатопроцесорних, так і розподілених систем обробки даних. Він був розроблений Леслі Лемпортом і вперше описаний в статті [16]. Остаточна редакція алгоритму вийшла в 2001 році в роботі [12].

Основна ідея алгоритму запозичена з принципу роботи магазину. Процеси (по аналогії з покупцями) вибирають собі номери, а потім процес, що має найменший номер, входить в критичну секцію.

Розглянемо цей метод детальніше. Уявимо собі булочну, де при вході стоїть автомат, який кожному клієнтові, що приходить, видає листочок з написаним на ньому номером. При цьому номер збільшується з приходом кожного нового клієнта. Кожного разу, коли продавець виявляється вільним, він обслуговує клієнта з найменшим номером. Наведемо приклад можливої послідовності дій.

Новий клієнт (A) заходить у булочну. Автомат видає йому номер 1. Новий клієнт (B) заходить у булочну. Автомат видає йому номер 2.

Продавець звільняється і обслуговує клієнта A (у цей момент у булочній знаходяться клієнти з номерами 1 і 2, найменший з цих номерів - один).

Новий клієнт (C) заходить у булочну. Автомат видає йому номер 3.

Продавець звільняється і обслуговує клієнта B (у цей момент у булочній знаходяться клієнти з номерами 2 і 3, найменший з цих номерів - два).

Продавець звільняється і обслуговує клієнта C (у цей момент у булочній знаходиться тільки клієнт з номером 3).

Новий клієнт (D) заходить у булочну. Автомат видає йому номер 4.

Продавець звільняється і обслуговує клієнта D (у цей момент у булочній знаходиться тільки клієнт з номером 4).

Новий клієнт (E) заходить у булочну. Автомат видає йому номер 5. Новий клієнт (F) заходить у булочну. Автомат видає йому номер 6. Новий клієнт (G) заходить у булочну. Автомат видає йому номер 7. Продавець звільняється і обслуговує клієнта E.

Продавець звільняється і обслуговує клієнта F.

Новий клієнт (H) заходить у булочну. Автомат видає йому номер 8. Продавець звільняється і обслуговує клієнта G.

Продавець звільняється і обслуговує клієнта H.

У цій моделі усі дії відбувалися послідовно. У реальності ж може статися так, що два клієнти прийдуть одночасно. Які номери їм видати в цьому випадку? Одне з можливих рішень цієї проблеми може бути таким: видамо їм однакові номери, і дамо продавцеві вказівку при виборі з двох клієнтів з однаковим номером обслуговувати того клієнта, в якого, наприклад, менший номер паспорта. Таким чином, вважається, що усі клієнти якось впорядковані за пріоритетом.

Алгоритм Лемпорта діє аналогічним чином. При цьому роль клієнтів грають потоки, а обслуговування продавцем відповідає діям процесу в критичній секції. Аналогом номера паспорта в разі взаємодії потоків служитиме ідентифікатор потоку. Наведемо реалізацію цього алгоритму на псевдокоді (лістинг 7.5), в тому вигляді, як він наведений в книзі з паралельного і розподіленого програмування [4].

Лістинг 7.5 – Алгоритм Лемпорта перевірки взаємних виключень

```
1    void lock(int i) {
2    choosing[i] = true;
3    for (int j = 0; j < n; j++) {
4    if (number[j] >= number[i])
5    number[i] = number[j]; 6 }
7    number[i]++;
8    choosing[i] = false;
9    for (int j = 0; j < n; j++) {
10   while (choosing[j]);
11   while ((number[j] <> 0) && ((number[j] < number[i])
12   || ((number[j] == number[i]) && (j < i))));
13   }
14 }
1 void unlock(int i) {
2 number[i] = 0;
3 }
```

Наведений запис алгоритму важкий для розуміння. З детальнішим описом алгоритму можна познайомитися в роботах [12; 18].

7.3.2 Взаємне виключення: апаратна підтримка

Взаємне виключення за допомогою змінних-перемикачів базується на атомарності звернень до пам'яті. Як було показано вище, це робить рішення універсальним як для одно-, так і для багатопроцесорних систем. Але більшість архітектур комп'ютерів мають у складі своєї системи спеціальні команди з розширеною атомарністю звернень до пам'яті, за допомогою яких можна реалізувати взаємне виключення.

Якщо в системі є тільки один процесор, то паралельні процеси не можуть перекриватися, а здатні тільки чергуватися. Крім того, процес триватиме до тих пір, поки не буде викликаний сервіс операційної системи або доки процес не буде перерваний. Тому, для того щоб гарантувати взаємне виключення, досить захистити процес від переривання. Процес у такому разі може забезпечити взаємне виключення таким чином:

```
While (true) {
/* Заборона переривань */
/* Критичний розділ */
/* Дозвіл переривань */
/* Інший код */
}
```

Оскільки робота програми в критичному розділі не може бути перервана (неможливе переривання навіть по таймеру), виконання взаємного виключення гарантується. Проте ціна такого підходу висока. Ефективність роботи може помітно знизитися, оскільки при цьому обмежена можливість процесора з чергування програм. Було б також безрозсудно давати призначеному для користувача процесу можливість дозволу і заборони переривань в усій обчислювальній системі. Уявіть собі, що буде, якщо процес заборонив усі переривання і в результаті якого-небудь збою не включив їх назад. ОС на цьому може закінчити своє існування. Інша проблема полягає в тому, що такий підхід не працюватиме в багатопроцесорній

архітектурі, оскільки одночасно може працювати декілька процесів. У цьому випадку заборона переривань не гарантує виконання взаємовиключень.

Як уже згадувалося, на рівні апаратного забезпечення звернення до елементу пам'яті виключає будь-які інші звернення до цього елементу. Грунтуючись на цьому принципі, розробники процесорів пропонують ряд машинних команд, які за один цикл вибірки команди атомарно виконують над елементом пам'яті дві дії, такі як читання і запис, або читання і перевірка значення. Оскільки ці дії виконуються в одному циклі, на них не в змозі вплинути ніякі інші інструкції.

Головним недоліком описаної технології є те, що потік повинен постійно перевіряти в циклі, чи не зняли блокування. Таку ситуацію називають активним очікуванням, а таке блокування – спін-блокуванням (spinlock). Розглянемо дві з інструкцій (команд), що найчастіше реалізовуються.

Команда TestAndSet

Інструкцію перевірки і установки значення можна визначити як:

```
Boolean TestAndSet(int I) { If I == 0) {
I = 1;
Return true;
}
else return alse;
}
```

Інструкція (команда) TestAndSet (перевірити і присвоїти 1) перевіряє значення свого аргументу I. Якщо він дорівнює 0, функція замінює його на 1 і повертає true. Інакше значення змінної не міняється і повертається false. Функція TestAndSet виконується атомарно, тобто її виконання не може бути перерване.

У лістингу 7.6 показаний протокол взаємних виключень, який ґрунтується на використанні описаної інструкції.

Лістинг 7.6 – Апаратна перевірка взаємних виключень. Інструкція перевірки і установки:

```
const int n = /* Кількість процесів */; int bolt;
```

```
void P(int I) { while(true)  {
while(!testset(bolt)); /* чекати */
/* Критичний розділ */ bolt = 0;
/* інша частина коду */
}
}
void main() {
bolt = 0; parbegin(P(1),P(2),…,P(n));
}
```

Змінна bolt, що розділяється, ініціалізувалася нульовим значенням. Тільки процес, який може увійти до критичного розділу, знаходить, що значення змінної bolt дорівнює 0. Усі інші процеси при спробі входу в критичний розділ переходять в режим очікування. Вийшовши із критичного розділу, процес встановлює заново значення bolt рівним 0, після чого знову тільки один процес з очікуючих входу в критичний розділ отримає потрібний йому доступ. Вибір цього процесу залежить від того, якому з процесів вдалося виконати інструкцію testset першим.

Команда обміну Swap

Виконання команди Swap (обміняти значення регістра і елементу пам'яті), що обмінює два значення, які знаходяться в пам'яті, можна проілюструвати такою функцією:

```
Void Swap (int register, int memory) { int temp;
temp = memory; memory = register; register = temp;
}
```

У процесі її виконання доступ до елементу пам'яті для всіх інших процесів блокується.

У лістингу 7.7 показаний протокол взаємних виключень, заснований на використанні описаної інструкції. Змінна bolt, що розділяється, ініціалізувалася нульовим значенням. У кожного процесу є локальна змінна key, що ініціалізувалася значенням 1. У критичний розділ може увійти тільки один процес, який виявляє, що значення змінної bolt дорівнює 0. Цей процес забороняє вхід в критичний розділ усім

іншим процесам шляхом установки значення bolt, рівним 1. Вийшовши з критичного розділу процес заново встановлює значення bolt, рівним 0.

Лістинг 7.7 – Апаратна перевірка взаємних виключень. Інструкція обміну

```
const int n = /* Кількість процесів */;
int bolt; /* змінна, що розділяється, */ void P (int I) {
int keyi; while(true){
keyi = 1;
while (keyi != 0)
Swap (keyi, bolt);
/* Критичний розділ */ Swap (keyi, bolt);
/* інша частина коду */
}
}
void main () { bolt = 0;
parbegin(P(1),P(2),…,P(n));
}
```

Якщо bolt = 0, то в критичному розділі немає жодного процесу. Якщо bolt = 1, то в критичному розділі знаходиться рівно один процес, а саме той, змінна key якого має нульове значення.